Select AI for Python

Release 1.5.0

Oracle

Oct 26, 2026

CONTENTS

1 Getting Started **3**
- 1.1 Introduction to Select AI for Python . . . 3
  - 1.1.1 What you can build . . . 3
  - 1.1.2 Core concepts . . . 3
  - 1.1.3 Synchronous and asynchronous APIs . . . 4
  - 1.1.4 Supported Python versions . . . 4
- 1.2 Installing `select_ai` . . . 5
  - 1.2.1 Installation requirements . . . 5
  - 1.2.2 `select_ai` installation . . . 5
  - 1.2.3 Connection smoke test . . . 6
  - 1.2.4 Install documentation dependencies . . . 6
- 1.3 Connecting to Oracle Database . . . 7
  - 1.3.1 Synchronous connection . . . 7
  - 1.3.2 Asynchronous connection . . . 7
  - 1.3.3 Connection Pool . . . 8
  - 1.3.4 Connection health . . . 9
  - 1.3.5 Wallet connections . . . 9
- 1.4 Run your first SQL query . . . 9
  - 1.4.1 Install and configure Python . . . 9
  - 1.4.2 Prepare the database account . . . 10
  - 1.4.3 Run the example . . . 10
  - 1.4.4 Waiting and timeouts . . . 13
- 1.5 Compatibility and requirements . . . 13
- 1.6 What's new in 1.5.0 . . . 14
  - 1.6.1 New and expanded APIs . . . 14
  - 1.6.2 Upgrade considerations . . . 15

2 Actions **17**
- 2.1 Supported Actions . . . 17
  - 2.1.1 Action methods . . . 17
  - 2.1.2 Choosing an action . . . 18
  - 2.1.3 Examples . . . 18
  - 2.1.4 Streaming . . . 19

3 Privileges **21**
- 3.1 Grant privilege . . . 21
- 3.2 Revoke privilege . . . 23
- 3.3 Grant network access . . . 24
- 3.4 Revoke network access . . . 26

4 Provider **29**
4.1 Provider . . . 29
4.1.1 Examples . . . 30
4.1.2 `Provider` . . . 32
4.1.3 `AnthropicProvider` . . . 33
4.1.4 `AzureProvider` . . . 34
4.1.5 `AWSProvider` . . . 35
4.1.6 `CohereProvider` . . . 36
4.1.7 `OpenAIProvider` . . . 37
4.1.8 `OCIGenAIProvider` . . . 38
4.1.9 `GoogleProvider` . . . 39
4.1.10 `HuggingFaceProvider` . . . 40
4.1.11 Enable AI service provider . . . 41
4.1.12 Cloud provider profile samples . . . 43
4.1.13 Disable AI service provider . . . 50

5 Credential **53**
5.1 Create credential . . . 55
5.1.1 Create Sync API . . . 55
5.1.2 Create Async API . . . 57
5.2 Delete credential . . . 58
5.2.1 Delete Sync API . . . 58
5.2.2 Delete Async API . . . 58
5.3 Credential access . . . 60

6 Sharing and ownership **61**
6.1 Sharing and ownership . . . 61
6.1.1 Connect before running an example . . . 61
6.1.2 Profiles . . . 62
6.1.3 Vector indexes . . . 63
6.1.4 Agent teams . . . 64
6.1.5 Credentials . . . 65
6.1.6 Runnable sharing scripts . . . 65

7 Profile Attributes **67**
7.1 `ProfileAttributes` . . . 67
7.1.1 Common required attributes . . . 67
7.1.2 Attribute groups . . . 68
7.1.3 Object list examples . . . 68
7.1.4 Generation controls . . . 69
7.1.5 Additional instructions . . . 69

8 Profile **73**
8.1 Profile lifecycle . . . 73
8.2 Profile actions . . . 74
8.2.1 Profile Object Model . . . 74
8.2.2 Base `Profile` API . . . 75
8.2.3 `Profile` API . . . 76
8.2.4 Create Profile . . . 82
8.2.5 Reuse Profile . . . 84
8.2.6 Enable and disable Profile . . . 85
8.2.7 Update Profile . . . 86
8.2.8 Delete Profile . . . 87
8.2.9 Narrate . . . 88
8.2.10 Show SQL . . . 89

8.2.11 Explain SQL . . . 90
8.2.12 Show Prompt . . . 91
8.2.13 Run SQL . . . 92
8.2.14 Chat . . . 93
8.2.15 Streaming chat . . . 94
8.2.16 Request-level profile attribute overrides . . . 94
8.2.17 Summarize . . . 97
8.2.18 Translate . . . 99
8.2.19 Share a profile . . . 100
8.2.20 Give feedback on generated SQL . . . 100
8.2.21 List profiles . . . 101

9 Async Profile **103**
9.1 Async profile lifecycle . . . 103
9.2 Async profile actions . . . 104
9.2.1 `AsyncProfile` API . . . 104
9.2.2 Async Profile creation . . . 111
9.2.3 Reuse Async Profile . . . 113
9.2.4 Enable and disable AsyncProfile . . . 114
9.2.5 Update Async Profile . . . 115
9.2.6 Delete Async Profile . . . 116
9.2.7 Async explain SQL . . . 117
9.2.8 Async run SQL . . . 118
9.2.9 Async show SQL . . . 119
9.2.10 Async show prompt . . . 120
9.2.11 Async concurrent SQL . . . 121
9.2.12 Async chat . . . 123
9.2.13 Async streaming chat . . . 124
9.2.14 Async request-level profile attribute overrides . . . 124
9.2.15 Summarize . . . 127
9.2.16 Translate . . . 129
9.2.17 Async pipeline . . . 130
9.2.18 Share a profile asynchronously . . . 132
9.2.19 Give async feedback on generated SQL . . . 132
9.2.20 List profiles asynchronously . . . 133

10 Conversation **135**
10.1 `Conversation Object model` . . . 135
10.2 `ConversationAttributes` . . . 137
10.3 `Conversation` API . . . 138
10.3.1 Create conversation . . . 140
10.3.2 Chat session . . . 141
10.3.3 Prompt history and tags . . . 143
10.3.4 List conversations . . . 145
10.3.5 Delete conversation . . . 146
10.4 `AsyncConversation` API . . . 147
10.4.1 Async chat session . . . 149
10.4.2 Async prompt history and tags . . . 151
10.4.3 Async list conversations . . . 152

11 Vector Index **153**
11.1 Vector Index . . . 153
11.2 `VectorIndex` Object Model . . . 153
11.3 `VectorIndexAttributes` . . . 155

11.3.1 OracleVectorIndexAttributes . . . 157
11.4 VectorIndex API . . . 159
11.4.1 Create vector index . . . 163
11.4.2 List vector index . . . 165
11.4.3 Fetch vector index . . . 166
11.4.4 Update vector index attributes . . . 167
11.4.5 RAG using vector index . . . 168
11.4.6 Delete vector index . . . 169
11.5 AsyncVectorIndex API . . . 170
11.5.1 Async create vector index . . . 173
11.5.2 Async list vector index . . . 175
11.5.3 Async fetch vector index . . . 176
11.5.4 Async update vector index attributes . . . 177
11.5.5 Async RAG using vector index . . . 178

12 Synthetic Data **181**
12.1 Generation modes . . . 181
12.2 Generation parameters . . . 182
12.3 Sync and async APIs . . . 182
12.3.1 SyntheticDataAttributes . . . 183
12.3.2 SyntheticDataParams . . . 184
12.3.3 Single table synthetic data . . . 185
12.3.4 Multi table synthetic data . . . 187

13 Summary **191**
13.1 Inline content . . . 191
13.2 Content from a URI . . . 191
13.3 Content from object storage . . . 191
13.4 Summary parameters . . . 192
13.5 Async summary . . . 192
13.6 Validation . . . 192
13.6.1 SummaryParams . . . 193
13.6.2 ChunkProcessingMethod . . . 194
13.6.3 ExtractivenessLevel . . . 200
13.6.4 SummaryStyle . . . 206

14 AI Agent **213**
14.1 Tool . . . 214
14.1.1 Supported Tools . . . 214
14.1.2 Tool selection . . . 214
14.1.3 Create Tool . . . 221
14.1.4 List Tools . . . 223
14.1.5 Inspect and run a tool . . . 224
14.2 Task . . . 226
14.2.1 Create Task . . . 229
14.2.2 List Tasks . . . 230
14.3 Agent . . . 231
14.3.1 Create Agent . . . 234
14.4 Team . . . 235
14.4.1 Supervised teams . . . 235
14.4.2 Share a team . . . 242
14.4.3 Run Team . . . 244
14.4.4 Export and Import Team . . . 246
14.4.5 Inspect a team . . . 248

14.4.6 Lifecycle helpers . . . 248
14.5 Agent history . . . 250
14.6 AI agent examples . . . 252
14.6.1 Web Search Agent using OpenAI's GPT model . . . 252

15 Async AI Agent **255**
15.1 `AsyncTool` . . . 256
15.1.1 Create Tool . . . 260
15.1.2 List Tools . . . 262
15.1.3 Inspect and run a tool . . . 263
15.2 `AsyncTask` . . . 265
15.2.1 Create Task . . . 267
15.2.2 List Tasks . . . 268
15.3 `AsyncAgent` . . . 269
15.3.1 Create Agent . . . 271
15.3.2 List Agents . . . 271
15.4 AsyncTeam . . . 273
15.4.1 Share an async team . . . 276
15.4.2 Supervised teams . . . 276
15.4.3 Run Team . . . 281
15.4.4 Export and Import Team . . . 283
15.4.5 Inspect a team . . . 285
15.4.6 Lifecycle helpers . . . 286
15.5 Async agent history . . . 288
15.5.1 List Teams . . . 290
15.6 Async AI agent examples . . . 291
15.6.1 Web Search Agent using OpenAI's GPT model . . . 291

16 A2A Integration **295**
16.1 Agent2Agent (A2A) . . . 295
16.1.1 Choose a topology . . . 295
16.1.2 Why clustered deployments use Consul . . . 295
16.1.3 Choose how connections are configured . . . 296
16.1.4 CLI-based deployment on any cloud . . . 296
16.1.5 Installation and prerequisites . . . 297
16.1.6 Protocol compatibility . . . 298
16.1.7 Standalone A2A server . . . 298
16.1.8 Persistent Oracle-backed state . . . 303
16.1.9 Wallet-based database connections . . . 304
16.1.10 Private database CA certificates . . . 304
16.1.11 Database call timeout . . . 304
16.1.12 Clustered A2A gateway and worker . . . 305
16.1.13 Google Cloud deployment . . . 311
16.1.14 Troubleshooting and security notes . . . 313

17 Command Line Interface **315**
17.1 Command line interface . . . 315
17.1.1 Connection options . . . 316
17.1.2 Interactive chat . . . 316
17.1.3 SQL commands . . . 317
17.1.4 Profile commands . . . 317
17.1.5 A2A commands . . . 318
17.1.6 Command summary . . . 322

18 Web Frameworks **323**

18.1 Using `select_ai` with Python web frameworks . . . 323
18.1.1 Framework patterns . . . 323
18.1.2 Install dependencies . . . 324
18.1.3 FastAPI synchronous endpoints . . . 325
18.1.4 FastAPI asynchronous endpoints . . . 327
18.1.5 Flask example . . . 328
18.1.6 Django example . . . 329
18.1.7 Pool sizing . . . 330
18.1.8 Pool wait behavior . . . 330
18.1.9 Request handling . . . 331

19 Concurrent Prompt Processing **333**
19.1 Concurrent prompt processing . . . 333
19.1.1 Recipe summary . . . 333
19.1.2 Environment variables . . . 333
19.1.3 `sync_thread_pool.py` . . . 335
19.1.4 `sync_ordered_results.py` . . . 337
19.1.5 `sync_queue_workers.py` . . . 339
19.1.6 `async_gather.py` . . . 341
19.1.7 `async_as_completed.py` . . . 343
19.1.8 `async_pipeline.py` . . . 345
19.1.9 `async_queue_workers.py` . . . 347
19.1.10 Pool sizing . . . 348

20 API Reference **349**
20.1 Errors, sessions, and result objects . . . 349
20.1.1 Error handling . . . 349
20.1.2 Session objects . . . 352
20.1.3 Agent history results . . . 355
20.1.4 Enumerations . . . 357

Python Module Index **383**

Index **385**

`select_ai` is a Python module which enables integrating DBMS_CLOUD_AI `PL/SQL` package into Python workflows. It bridges the gap between `PL/SQL` package's AI capabilities and Python's rich ecosystem.

CHAPTER

ONE

GETTING STARTED

1.1 Introduction to Select AI for Python

`select_ai` is a Python module that helps you invoke DBMS_CLOUD_AI using Python. It supports text-to-SQL generation, retrieval augmented generation (RAG), synthetic data generation, and several other features using Oracle-based and third-party AI providers.

Select AI for Python bridges Oracle Database's Select AI capabilities and the Python ecosystem. It gives Python applications a higher-level API for working with AI providers, credentials, profiles, natural language prompts, vector indexes, conversations, summarization, synthetic data, and AI agent workflows.

For a complete setup-to-query example, start with *Run your first SQL query*. Existing users can review *What's new in 1.5.0* and *Compatibility*.

1.1.1 What you can build

Start with the workflow you want to build:

- **Natural language data apps:** generate SQL, return query results as pandas DataFrames, or explain results in plain language with *Profiles* and *Async Profiles*.
- **RAG and chat assistants:** ground answers in your documents with *Vector Indexes* and keep chat context with *Conversations*.
- **AI agent workflows:** compose tools, tasks, and agents into *Teams*, then expose a team to other AI applications through an *Agent2Agent (A2A) server*.
- **Python web services:** add Select AI endpoints to FastAPI, Flask, or Django using the pool lifecycle patterns in *Web Frameworks*.
- **Concurrent prompt processing:** use threads, async tasks, queues, or a database pipeline with the *Concurrency Guides*.
- **Content and test-data tools:** *summarize content*, *translate text*, or generate *Synthetic Data* for database tables.
- **Terminal workflows:** chat, generate SQL, and run an A2A server with the optional *Command Line Interface*.

1.1.2 Core concepts

Most workflows use the same building blocks:

Concept	Purpose	Start here
Connection	Connect to Oracle Database using a standalone connection or a pool.	*Connection*
Privileges	Grant package privileges and network ACLs required for Select AI calls.	*Privileges*
Provider	Describe the AI service, model, endpoint, region, or provider-specific options.	*Provider*
Credential	Store provider and service secrets securely in Oracle Database.	*Credential*
Sharing and ownership	Share profiles, vector indexes, teams, and credentials with users or roles and address objects owned by another schema.	*Sharing and ownership*
Profile	Combine provider, credential, database object scope, and generation options into a reusable Select AI profile.	*Profile*
Actions	Choose what Select AI should do with a prompt, such as show SQL, run SQL, chat, narrate, summarize, or translate.	*Actions*
Conversation	Keep prompt history for context-aware chat sessions.	*Conversation*
Vector index	Index document content for RAG over trusted source material.	*Vector Index*
Agent	Define tools, tasks, agents, and teams for multi-step AI workflows.	*Agent*
Agent2Agent (A2A)	Expose an agent team as a discoverable service for A2A clients. Choose standalone or clustered deployment and fixed or dynamic connections.	*A2A server*

1.1.3 Synchronous and asynchronous APIs

`select_ai` supports both synchronous and asynchronous programming styles. Use the synchronous APIs for scripts, notebooks, command-line tools, and simple services. Use the asynchronous APIs with `asyncio` applications, async web frameworks, and workloads that need to run many prompts concurrently.

For long-running services, create a connection pool once during application startup and close it during shutdown. See *Connection*, *Web Frameworks*, and *Concurrent Prompt Processing* for patterns.

1.1.4 Supported Python versions

The Select AI Python API supports Python versions 3.11, 3.12, 3.13, and 3.14.

1.2 Installing `select_ai`

1.2.1 Installation requirements

To use `select_ai` you need:

- Python 3.11, 3.12, 3.13, or 3.14.
- Access to an Oracle Database environment where Select AI is available.
- A database user with the required Select AI package privileges. See *Privileges*.
- Network access from the database to any AI provider endpoints you plan to use.
- `python-oracledb` and `pandas`. These packages are installed automatically as dependencies.

Using a virtual environment is recommended so the package and its dependencies are isolated from your system Python installation.

1.2.2 `select_ai` installation

`select_ai` can be installed from the Python Package Index PyPI using pip.

1. Install Python 3 if it is not already available. Use any version from Python 3.11 through 3.14.
2. Create and activate a virtual environment:

   ```
   python3 -m venv .venv
   source .venv/bin/activate
   ```

   On Windows PowerShell:

   ```
   py -3 -m venv .venv
   .venv\Scripts\Activate.ps1
   ```

3. Upgrade `pip`:

   ```
   python -m pip install --upgrade pip
   ```

4. Install `select_ai`:

   ```
   python -m pip install --upgrade select_ai
   ```

5. If you want the optional command line interface, install the `cli` extra:

   ```
   python -m pip install --upgrade "select_ai[cli]"
   ```

   This installs the `select-ai` command with profile, SQL, and chat commands. To use the A2A server and worker commands, install the separate `a2a` extra:

   ```
   python -m pip install --upgrade "select_ai[a2a]"
   ```

   See *Command Line Interface* and *A2A Integration*.

6. If you are behind a proxy, use the `--proxy` option. For example:

   ```
   python -m pip install --upgrade select_ai --proxy=http://proxy.example.com:80
   ```

1.2.3 Connection smoke test

After installation, verify that Python can import `select_ai` and connect to Oracle Database.

1. Set database connection environment variables:

```
export SELECT_AI_USER=<select_ai_db_user>
export SELECT_AI_PASSWORD=<select_ai_db_password>
export SELECT_AI_DB_CONNECT_STRING=<db_connect_string>
```

2. Create a file `select_ai_connection_test.py`:

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
print("Connected to the Database")
```

3. Run the script:

```
python select_ai_connection_test.py
```

   If the connection succeeds, the script prints:

```
Connected to the Database
```

Once the connection smoke test succeeds, follow *Run your first SQL query* to create a credential and profile and query sample data.

1.2.4 Install documentation dependencies

If you are building this documentation locally from the repository, install the documentation dependencies:

```
python -m pip install -r doc/requirements.txt
```

Then build the docs with the project's Sphinx command or Makefile target.

1.3 Connecting to Oracle Database

`select_ai` uses the Python thin driver i.e. `python-oracledb` to connect to the database and execute PL/SQL subprograms.

The library keeps the active connection or connection pool for the current process so profile, credential, provider, vector index, and agent APIs can use it without passing a connection object to each call. Standalone connections are local to the creating thread; pools are shared within a process. Do not configure both a standalone connection and a pool for the same API style in one thread: disconnect before switching. Use a standalone connection for scripts and notebooks. Use a connection pool for applications that handle concurrent work, such as web services or worker processes.

Most samples read connection values from environment variables:

```
export SELECT_AI_USER=<db_user>
export SELECT_AI_PASSWORD=<db_password>
export SELECT_AI_DB_CONNECT_STRING=<db_connect_string>
```

Then the Python code can load those values:

```
import os
import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
```

1.3.1 Synchronous connection

To connect to an Oracle Database synchronously, use `select_ai.connect()` as shown below:

```
import select_ai

user = "<your_db_user>"
password = "<your_db_password>"
dsn = "<your_db_dsn>"
select_ai.connect(user=user, password=password, dsn=dsn)
```

Close a standalone synchronous connection with `select_ai.disconnect()`:

```
select_ai.disconnect()
```

1.3.2 Asynchronous connection

Asynchronous applications should use `select_ai.async_connect()` along with `await` keyword:

```
import select_ai

user = "<your_db_user>"
password = "<your_db_password>"
dsn = "<your_db_dsn>"
await select_ai.async_connect(user=user, password=password, dsn=dsn)
```

Close a standalone asynchronous connection with `await select_ai.async_disconnect()`:

```
await select_ai.async_disconnect()
```

1.3.3 Connection Pool

You can create a connection pool using the `select_ai.create_pool` and `select_ai.create_pool_async` methods. After a pool is created, these methods configure Select AI operations to acquire and release connections from the pool for each operation.

```
import select_ai

user = "<your_db_user>"
password = "<your_db_password>"
dsn = "<your_db_dsn>"

# for sync pool
select_ai.create_pool(
    user=user,
    password=password,
    dsn=dsn,
    min_size=5,
    max_size=10,
    increment=5
)

# for async pool
select_ai.create_pool_async(
    user=user,
    password=password,
    dsn=dsn,
    min_size=5,
    max_size=10,
    increment=5
)
```

Close a synchronous pool with `select_ai.disconnect()` and an asynchronous pool with `await select_ai.async_disconnect()`.

Create one pool per process. In multi-process deployments, each process creates its own pool, so total database connections can grow quickly. Size pools based on request concurrency and database capacity.

Use pooling for:

- Web applications and API services.
- Worker processes that handle multiple prompts.
- Concurrent prompt processing.
- Long-running applications that should avoid opening a new database connection for every request.

Use a standalone connection for:

- Short scripts.
- Local examples.
- One-off administration tasks.

Check this blog which shows the benefit of connection pooling with a FastAPI service.

1.3.4 Connection health

Use `select_ai.is_connected()` or `await select_ai.async_is_connected()` to check whether the current connection or pool is available:

```
if not select_ai.is_connected():
    select_ai.connect(user=user, password=password, dsn=dsn)
```

```
if not await select_ai.async_is_connected():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
```

1.3.5 Wallet connections

> **Note**
>
> For m-TLS (wallet) based connections, additional parameters like `wallet_location`, `wallet_password`, `config_dir`, `https_proxy`, and `https_proxy_port` can be passed to `connect`, `async_connect`, `create_pool`, and `create_pool_async`.

For example:

```
select_ai.connect(
    user=user,
    password=password,
    dsn=dsn,
    wallet_location="/path/to/wallet",
    config_dir="/path/to/wallet",
    wallet_password="<wallet_password>",
)
```

The same keyword arguments can be used with `async_connect` and the pool creation APIs.

1.4 Run your first SQL query

This walkthrough creates a temporary table with three rows, stores an OpenAI API key in a temporary database credential, creates a Select AI profile, and queries the table using natural language. It removes its table, credential, and profile when finished. Use a development database account.

Requests use the API key in `OPENAI_API_KEY` on each run. Existing database credentials are not used or replaced. OpenAI API usage is billed separately from ChatGPT and Codex subscriptions.

1.4.1 Install and configure Python

Use Python 3.11 through 3.14 and an Oracle Database environment with `DBMS_CLOUD_AI` available. See *Compatibility* for feature-specific requirements.

```
python3 -m venv .venv
source .venv/bin/activate
python -m pip install --upgrade pip
python -m pip install 'select_ai==1.5.0'
```

On Windows, use the virtual-environment activation command in *Installation*.

Set the following values in your shell. Choose an OpenAI model supported by your database integration and available to your API project:

```
export SELECT_AI_USER='your_database_user'
export SELECT_AI_PASSWORD='your_database_password'
export SELECT_AI_DB_CONNECT_STRING='your_database_connect_string'
export OPENAI_API_KEY='your_api_key'
export SELECT_AI_MODEL='gpt-4.1'
```

To try a different key, replace `OPENAI_API_KEY` and rerun the script. Keep keys out of source files and version control. OCI-specific environment variables are not read by this sample.

For wallet connections, add the parameters described in *Wallet connections* to the sample's `connect()` call. Do not commit secrets to the repository.

1.4.2 Prepare the database account

A database administrator must grant the application user Select AI package privileges and outbound HTTP access to `api.openai.com`. Pass `--setup` to perform both steps before running the example. Set the administrator variables in addition to the application and OpenAI variables above:

```
export SELECT_AI_ADMIN_USER='your_database_admin_user'
export SELECT_AI_ADMIN_PASSWORD='your_database_admin_password'
python samples/quickstart.py --setup
```

The script connects as the administrator, calls `grant_privileges()` and `grant_network_access(users=..., host="api.openai.com", privileges="http")`, commits the setup, and disconnects. It then connects as the application user and runs the example. The administrator must be allowed to grant package privileges and manage network ACLs. Administrator and application connections use the same DSN; add required wallet arguments to both connection calls.

Run setup once for each application user/database. For later runs, omit the flag; administrator credentials are then unnecessary:

```
python samples/quickstart.py
```

The grants remain after the temporary sample objects are removed. See *Privileges* to revoke package or network privileges later.

The application account must also have `CREATE SESSION`, `CREATE TABLE`, and sufficient tablespace quota to create the temporary table. Have the administrator assign an appropriate quota in the account's default tablespace. The helper does not grant these privileges or quota. The sample does not create a public credential synonym and does not require `CREATE PUBLIC SYNONYM`.

1.4.3 Run the example

Save the following as `quickstart.py`, or run `python samples/quickstart.py` from a repository checkout:

```
import argparse
import os
import uuid

import select_ai

parser = argparse.ArgumentParser(description=__doc__)
parser.add_argument(
```

(continues on next page)

(continued from previous page)

```
    "--setup",
    action="store_true",
    help="Grant package privileges and OpenAI HTTP access as administrator",
)
args = parser.parse_args()

user = os.environ["SELECT_AI_USER"]
password = os.environ["SELECT_AI_PASSWORD"]
dsn = os.environ["SELECT_AI_DB_CONNECT_STRING"]
api_key = os.environ["OPENAI_API_KEY"]
model = os.environ["SELECT_AI_MODEL"]

if args.setup:
    print("Connecting as administrator...", flush=True)
    select_ai.connect(
        user=os.environ["SELECT_AI_ADMIN_USER"],
        password=os.environ["SELECT_AI_ADMIN_PASSWORD"],
        dsn=dsn,
        tcp_connect_timeout=15,
    )
    try:
        with select_ai.cursor() as cursor:
            cursor.connection.call_timeout = 120_000
            print(f"Granting Select AI package privileges to {user}...", flush=True)
            select_ai.grant_privileges(users=user)
            print(f"Granting {user} HTTP access to api.openai.com...", flush=True)
            select_ai.grant_network_access(
                users=user, host="api.openai.com", privileges="http"
            )
            cursor.connection.commit()
    finally:
        select_ai.disconnect()

suffix = uuid.uuid4().hex[:12].upper()
table_name = f"SDK_DEMO_{suffix}"
credential_name = f"SDK_CRED_{suffix}"
profile_name = f"SDK_PROFILE_{suffix}"

print(f"Connecting as application user {user}...", flush=True)
select_ai.connect(user=user, password=password, dsn=dsn, tcp_connect_timeout=15)

table_created = credential_created = profile_created = False
try:
    # This identifier is generated locally, rather than supplied by a caller.
    with select_ai.cursor() as cursor:
        cursor.connection.call_timeout = 120_000
        cursor.execute("SELECT SYS_CONTEXT('USERENV', 'CURRENT_SCHEMA') FROM DUAL")
        owner = cursor.fetchone()[0]
        print("Creating the sample table...", flush=True)
        cursor.execute(
            f"CREATE TABLE {table_name} (id NUMBER PRIMARY KEY, name VARCHAR2(40))"
        )
```

(continues on next page)

(continued from previous page)

```
        table_created = True
        cursor.executemany(
            f"INSERT INTO {table_name} (id, name) VALUES (:1, :2)",
            [(1, "Notebook"), (2, "Keyboard"), (3, "Monitor")],
        )
        cursor.connection.commit()

    print("Creating the OpenAI credential and profile...", flush=True)
    select_ai.create_credential(
        {
            "credential_name": credential_name,
            "username": "openai",
            "password": api_key,
        }
    )
    credential_created = True
    profile = select_ai.Profile(
        profile_name=profile_name,
        attributes=select_ai.ProfileAttributes(
            provider=select_ai.OpenAIProvider(model=model),
            credential_name=credential_name,
            object_list=[{"owner": owner, "name": table_name}],
        ),
    )
    profile_created = True

    prompt = f"How many rows are in {table_name}? Return the count as ITEM_COUNT."
    print("Generating SQL with OpenAI...", flush=True)
    print("Generated SQL:", profile.show_sql(prompt), flush=True)
    # run_sql generates and executes a fresh response to the prompt.
    print("Generating and executing the query with OpenAI...", flush=True)
    print(profile.run_sql(prompt))
finally:
    print("Removing temporary objects and closing the connection...", flush=True)
    # Attempt each cleanup even if an earlier cleanup fails.
    try:
        if profile_created:
            select_ai.Profile.delete_profile(profile_name, force=True)
    finally:
        try:
            if credential_created:
                select_ai.delete_credential(credential_name, force=True)
        finally:
            try:
                if table_created:
                    with select_ai.cursor() as cursor:
                        cursor.execute(f"DROP TABLE {table_name} PURGE")
            finally:
                select_ai.disconnect()
```

The script prints generated SQL and a DataFrame containing a count of three. Table names include a generated suffix. SQL formatting and the model's chosen column label may vary; representative result output is:

```
   ITEM_COUNT
0           3
```

The profile's `object_list` includes the sample table using the schema name resolved from the database connection. Other profile attributes use their defaults, including object-list enforcement. The object list supplies metadata for SQL generation; database privileges determine which objects the user can access. See *Profile attributes* for explicit enforcement settings.

`show_sql()` generates SQL without executing it. `run_sql()` makes a separate generation request and executes that response; it does not execute the SQL string returned by the earlier call. In applications that require approval of an exact statement, review the generated statement and execute that approved statement through your database API.

If connection fails, check the DSN, password, and wallet settings. For package or network authorization failures, return to the administrator setup. For model or provider errors, check the API key, model access, API billing, and project limits. See *Error handling* for Python exception handling. The sample's cleanup removes only objects it successfully created; administrator grants remain in place for subsequent workflows.

1.4.4 Waiting and timeouts

The script prints progress before connecting, creating objects, and requesting SQL from OpenAI. If it waits at `Generating SQL with OpenAI...`, the database is processing the model request. Missing network ACL privileges usually raise an authorization error; waiting alone does not prove that the ACL is missing. Check database egress, provider availability, API billing, and project limits. HTTP 429 can indicate quota or rate limits; retries may extend the wait.

The sample sets `tcp_connect_timeout=15` for connection establishment and `connection.call_timeout=120_000` for database calls. The latter is in milliseconds and limits each database round trip, rather than total script runtime. Increase it deliberately for slower requests. See the driver timeout documentation.

1.5 Compatibility and requirements

Select AI for Python 1.5.0 supports Python 3.11, 3.12, 3.13, and 3.14. Installation requires `oracledb>=2.0.0` and `pandas==3.0.3`. Some optional features require driver APIs introduced after the minimum installable driver version; upgrade `oracledb` if those APIs are unavailable.

Installing the SDK does not install database PL/SQL packages or enable their features. Availability depends on the Oracle Database service, version, and package updates. Consult Oracle's Select AI and Select AI Agent capability matrix for your environment before choosing a workflow.

Table 1: Feature requirements

Feature	Python installation	Database and deployment requirements
Profiles, SQL generation, and chat	`select_ai`	`DBMS_CLOUD_AI` and `DBMS_CLOUD`; provider credential and outbound database network access. SQL execution also requires object privileges.
Async APIs	`select_ai` with an async-capable `oracledb` driver	Thin mode; use an async connection or async pool. Use a pool for independent concurrent database operations.
Pipelines	`oracledb>=2.4.0` with `create_pipeline()` and `AsyncConnection.run_pipeline()`	True pipelining requires Oracle AI Database 26ai or later. Older databases execute individually. See *Pipeline compatibility and results*.
RAG vector indexes	`select_ai`	Vector-index support in `DBMS_CLOUD_AI`, `DBMS_CLOUD_PIPELINE`, an embedding model, source access, and database storage capacity.
Agent teams and history	`select_ai`	`DBMS_CLOUD_AI_AGENT` and the corresponding agent history views.
Profile, index, team, and credential sharing	`select_ai`	Database packages providing the relevant grant/revoke APIs. The recipient still needs privileges on workflow dependencies.
CLI profile, SQL, and chat commands	`select_ai[cli]`	Same requirements as the corresponding SDK operation.
A2A fixed standalone	`select_ai[a2a]`	Existing agent team; A2A storage privileges and quota. Supports streaming and wallet authentication.
A2A dynamic standalone	`select_ai[a2a]`	One Uvicorn process; A2UI-capable client. Polling supported; streaming and Oracle wallet authentication are unavailable.
A2A clustered	`select_ai[a2a]` on gateway and workers	Consul and at least one worker. Polling supported; streaming and Oracle wallet authentication are unavailable.

The OCI credential-loading examples also require `oci` and a configured OCI API-signing key. This is a sample dependency, rather than an SDK runtime requirement. See *Credential*.

Provider models, regions, and account permissions are independent of Python package compatibility. Replace example model identifiers and storage URIs with values supported by your provider account and database integration.

1.6 What's new in 1.5.0

Version 1.5.0 adds A2A integration, object sharing, request-level profile overrides, and typed agent history APIs. The changes below are relative to 1.4.1.

1.6.1 New and expanded APIs

- Expose an agent team with the optional *A2A server*. Choose fixed or dynamic connections and standalone or clustered deployment. A2A includes Oracle-backed task/context storage, task polling, A2UI connection forms, and A2A 1.0/v0.3 JSON-RPC compatibility.
- *Share profiles, vector indexes, teams, and credentials* with database users or roles. Profile and vector-index fetch/list APIs accept `owner` and expose `qualified_name`.

- Pass an `attributes` mapping to profile and session actions for *request-only overrides*. Async pipelines can apply the same override mapping to every operation.
- Inspect agent execution with *TeamHistory, TaskHistory, and ToolHistory* and their async equivalents.
- Enable and disable profiles without deleting them. Configure `additional_instructions`, integer `seed`, and translation language defaults with `ProfileAttributes`. Per-call translation languages are optional and can use the saved defaults.
- Configure supervised teams with `AgentAttributes.supervisor` and `TeamAttributes.supervisor_agent`; inspect the database-generated `supervisor_task` after creation.

1.6.2 Upgrade considerations

- **Python 3.11 is now the minimum.** Python 3.9 and 3.10 are no longer supported. Supported versions are 3.11 through 3.14.
- **pandas is now pinned to 3.0.3**, replacing 2.2.3. Check applications that consume DataFrames for pandas compatibility when upgrading.
- `ProfileAttributes.seed` is now typed as an integer. Supply an integer rather than a numeric string.
- Unspecified `SyntheticDataParams` values are omitted so the database can apply its defaults. Set values explicitly if your application relies on particular generation settings.
- Install `select_ai[a2a]` to use A2A commands; `select_ai[cli]` covers the other CLI commands. Prepare A2A database storage privileges as described in *A2A prerequisites*.
- `--require-oauth` requires a bearer token and derives an owner scope. Token signatures and claims must be validated by a trusted upstream service; the SDK flag alone does not verify identity. See *A2A authentication*.

Upgrade in your application's virtual environment:

```
python -m pip install --upgrade 'select_ai==1.5.0'
# If you use A2A:
python -m pip install --upgrade 'select_ai[a2a]==1.5.0'
```

See *Compatibility* for the feature requirements and *Run your first SQL query* for a complete example.

CHAPTER

TWO

ACTIONS

An action in Select AI is a keyword that instructs Select AI to perform different behavior when acting on the prompt.

Most applications use the convenience methods on `Profile` or `AsyncProfile`, such as `show_sql()`, `run_sql()`, `narrate()`, and `chat()`. Use `generate(..., action=...)` when you want to choose the action dynamically at runtime.

The default action for `generate()` is `select_ai.Action.RUNSQL`.

2.1 Supported Actions

The following actions can be performed using `select_ai`:

Table 1: Select AI Actions

Action	Enum	Description
chat	`select_ai.Action.CHAT`	Enables general conversations with the LLM, potentially for clarifying prompts, exploring data, or generating content.
explainsql	`select_ai.Action.EXPLAINSQL`	Explains the generated SQL query.
narrate	`select_ai.Action.NARRATE`	Executes generated SQL and explains the output in natural language.
runsql	`select_ai.Action.RUNSQL`	Executes SQL generated from a natural language prompt. This is the default action for `generate()`.
showprompt	`select_ai.Action.SHOWPROMPT`	Shows the prompt sent to the LLM.
showsql	`select_ai.Action.SHOWSQL`	Displays the generated SQL statement without executing it.
summarize	`select_ai.Action.SUMMARIZE`	Generates a summary of inline content or content referenced by a URI.
feedback	`select_ai.Action.FEEDBACK`	Provides feedback to improve the accuracy of generated SQL.
translate	`select_ai.Action.TRANSLATE`	Translates text from one language to another.

2.1.1 Action methods

Request-specific profile attributes can be supplied with the optional `attributes` mapping. For example, override `additional_instructions` for a single chat request:

```
response = profile.chat(
    prompt="Write sample Python code to build and test a regression model",
    attributes={
        "additional_instructions": (
            "Return executable code as plain text without Markdown fences."
        )
    },
)
```

Table 2: Action to method mapping

Action	Convenience method	Return type
RUNSQL	profile.run_sql(prompt)	pandas.DataFrame
SHOWSQL	profile.show_sql(prompt)	str
EXPLAINSQL	profile.explain_sql(prompt)	str
NARRATE	profile.narrate(prompt)	str
CHAT	profile.chat(prompt)	str
SHOWPROMPT	profile.show_prompt(prompt)	str
SUMMARIZE	profile.summarize(...)	str
TRANSLATE	profile.translate(...)	str

2.1.2 Choosing an action

Use `show_sql` before `run_sql` when you want to inspect generated SQL before executing it. Use `run_sql` when the application should return a tabular result. Use `narrate` when users need a natural language answer instead of a table. Use `explain_sql` and `show_prompt` when tuning profile attributes, object lists, comments, constraints, or feedback.

2.1.3 Examples

Use a convenience method:

```
profile = select_ai.Profile(profile_name="oci_ai_profile")
sql = profile.show_sql(prompt="How many promotions?")
```

Use `generate` with an explicit action:

```
profile = select_ai.Profile(profile_name="oci_ai_profile")

sql = profile.generate(
    prompt="How many promotions?",
    action=select_ai.Action.SHOWSQL,
)

df = profile.generate(
    prompt="How many promotions?",
    action=select_ai.Action.RUNSQL,
)
```

Use an action selected at runtime:

```
action = select_ai.Action("showsql")
result = profile.generate(
```

(continues on next page)

(continued from previous page)

```
    prompt="How many promotions?",
    action=action,
)
```

2.1.4 Streaming

Streaming is supported for text-returning generation actions: `CHAT`, `NARRATE`, `EXPLAINSQL`, `SHOWSQL`, and `SHOWPROMPT`. Streaming is not supported for `RUNSQL` because it returns a `pandas.DataFrame`.

```
for chunk in profile.generate(
    prompt="What is OCI?",
    action=select_ai.Action.CHAT,
    stream=True,
    chunk_size=4096,
):
    print(chunk, end="")
```

CHAPTER

THREE

PRIVILEGES

An admin user should grant execute privilege to Select AI database users on the packages `DBMS_CLOUD`, `DBMS_CLOUD_AI`, `DBMS_CLOUD_AI_AGENT`, and `DBMS_CLOUD_PIPELINE`.

The privilege helper APIs are intended for database administrators who need to prepare one or more database schemas for Select AI workloads. These operations should be run from a connection that has permission to grant package execute privileges and manage database network ACLs.

There are two separate setup steps:

- Package privileges allow a Select AI database user to call the Oracle Database PL/SQL packages used by this library.
- Network access allows the database user to make outbound calls to specific hosts, such as AI provider endpoints or SMTP servers.

The `users` argument accepts either a single database user name or a list of database user names.

Note

All sample scripts in this documentation read Oracle database connection details from the environment. Create a dotenv file `.env`, export the following environment variables and source it before running the scripts.

```
export SELECT_AI_ADMIN_USER=<db_admin>
export SELECT_AI_ADMIN_PASSWORD=<db_admin_password>
export SELECT_AI_USER=<select_ai_db_user>
export SELECT_AI_PASSWORD=<select_ai_db_password>
export SELECT_AI_DB_CONNECT_STRING=<db_connect_string>
export TNS_ADMIN=<path/to/dir_containing_tnsnames.ora>
```

3.1 Grant privilege

Connect as an admin user and run `select_ai.grant_privileges(users=select_ai_user)` to grant the package execute privileges required by Select AI. This grants execute access on `DBMS_CLOUD`, `DBMS_CLOUD_AI`, `DBMS_CLOUD_AI_AGENT`, and `DBMS_CLOUD_PIPELINE`.

```
import os

import select_ai

admin_user = os.getenv("SELECT_AI_ADMIN_USER")
password = os.getenv("SELECT_AI_ADMIN_PASSWORD")
```

(continues on next page)

(continued from previous page)

```
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
select_ai_user = os.getenv("SELECT_AI_USER")


select_ai.connect(user=admin_user, password=password, dsn=dsn)
select_ai.grant_privileges(users=select_ai_user)
print("Granted privileges to: ", select_ai_user)
```

output:

```
Granted privileges to: <select_ai_db_user>
```

3.2 Revoke privilege

Similarly, to revoke use the method `select_ai.revoke_privileges(users=select_ai_user)`

```
import os

import select_ai

admin_user = os.getenv("SELECT_AI_ADMIN_USER")
password = os.getenv("SELECT_AI_ADMIN_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
select_ai_user = os.getenv("SELECT_AI_USER")


select_ai.connect(user=admin_user, password=password, dsn=dsn)
select_ai.revoke_privileges(users=select_ai_user)
print("Revoked privileges from: ", select_ai_user)
```

output:

```
Revoked privileges from: <select_ai_db_user>
```

3.3 Grant network access

Connect as admin and run `select_ai.grant_network_access(...)` to add a network ACL entry for host access. This wraps `DBMS_NETWORK_ACL_ADMIN.APPEND_HOST_ACE` and can be used for hosts that require privileges such as `connect`, `http`, or `smtp`.

Network ACLs are required when the database needs to reach an external host. For example, use `http` access for AI provider endpoints and `smtp` access for mail servers. Include `connect` with protocol-specific privileges when the host requires it.

When granting access, specify the target host and, when applicable, the port range. When revoking access, use the same host, privileges, and port range that were used for the grant.

```
import os

import select_ai

admin_user = os.getenv("SELECT_AI_ADMIN_USER")
password = os.getenv("SELECT_AI_ADMIN_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
select_ai_user = os.getenv("SELECT_AI_USER")

select_ai.connect(user=admin_user, password=password, dsn=dsn)
select_ai.grant_network_access(
    users=select_ai_user,
    host="smtp.example.com",
    privileges=["connect", "smtp"],
    lower_port=587,
    upper_port=587,
)
print("Granted network access to: ", select_ai_user)
```

output:

```
Granted network access to: <select_ai_db_user>
```

The async API is `select_ai.async_grant_network_access(...)`.

```
import asyncio
import os

import select_ai

admin_user = os.getenv("SELECT_AI_ADMIN_USER")
password = os.getenv("SELECT_AI_ADMIN_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
select_ai_user = os.getenv("SELECT_AI_USER")


async def main():
    await select_ai.async_connect(user=admin_user, password=password, dsn=dsn)
    await select_ai.async_grant_network_access(
        users=select_ai_user,
        host="smtp.example.com",
        privileges=["connect", "smtp"],
        lower_port=587,
        upper_port=587,
    )
    print("Granted network access to: ", select_ai_user)


asyncio.run(main())
```

3.4 Revoke network access

Connect as admin and run `select_ai.revoke_network_access(...)` to remove a network ACL entry for host access. This wraps `DBMS_NETWORK_ACL_ADMIN.REMOVE_HOST_ACE` and should use the same host, privileges, and port range that were used to grant access.

```
import os

import select_ai

admin_user = os.getenv("SELECT_AI_ADMIN_USER")
password = os.getenv("SELECT_AI_ADMIN_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
select_ai_user = os.getenv("SELECT_AI_USER")

select_ai.connect(user=admin_user, password=password, dsn=dsn)
select_ai.revoke_network_access(
    users=select_ai_user,
    host="smtp.example.com",
    privileges=["connect", "smtp"],
    lower_port=587,
    upper_port=587,
)
print("Revoked network access from: ", select_ai_user)
```

output:

```
Revoked network access from: <select_ai_db_user>
```

The async API is `select_ai.async_revoke_network_access(...)`.

```
import asyncio
import os

import select_ai

admin_user = os.getenv("SELECT_AI_ADMIN_USER")
password = os.getenv("SELECT_AI_ADMIN_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
select_ai_user = os.getenv("SELECT_AI_USER")


async def main():
    await select_ai.async_connect(user=admin_user, password=password, dsn=dsn)
    await select_ai.async_revoke_network_access(
        users=select_ai_user,
        host="smtp.example.com",
        privileges=["connect", "smtp"],
        lower_port=587,
        upper_port=587,
    )
    print("Revoked network access from: ", select_ai_user)


asyncio.run(main())
```

CHAPTER

FOUR

PROVIDER

4.1 Provider

An AI Provider in Select AI refers to the service provider of the LLM, transformer or both for processing and generating responses to natural language prompts. These providers offer models that can interpret and convert natural language for the use cases highlighted under the LLM concept.

See Select your AI Provider for the supported providers.

A provider object describes the AI service that a Select AI profile, vector index, or agent tool should call. The provider object is separate from the credential object: the provider selects the service, model, endpoint, region, and provider-specific options, while the credential stores authentication details.

Most applications should instantiate one of the concrete provider classes instead of using `Provider` directly. Use the base `Provider` class when you need to call a compatible provider endpoint that does not have a dedicated class in this library.

Table 1: Provider classes

Provider class	Provider name	Default endpoint behavior
`AnthropicProvider`	`anthropic`	Uses `api.anthropic.com`.
`AWSProvider`	`aws`	Builds `bedrock-runtime.<region>.amazonaws.com` from `region`.
`AzureProvider`	`azure`	Builds `<azure_resource_name>.openai.azure.com`.
`CohereProvider`	`cohere`	Uses `api.cohere.ai`.
`GoogleProvider`	`google`	Uses `generativelanguage.googleapis.com`.
`HuggingFaceProvider`	`huggingface`	Uses `api-inference.huggingface.co`.
`OCIGenAIProvider`	`oci`	Uses OCI region and OCI Gen AI attributes.
`OpenAIProvider`	`openai`	Keeps `api.openai.com` for network ACL setup; omits the default endpoint from database profiles so Select AI selects its native URL.

4.1.1 Examples

OCI Gen AI provider:

```
provider = select_ai.OCIGenAIProvider(
    region="us-chicago-1",
    oci_apiformat="GENERIC",
    model="cohere.command-r-plus",
)
```

OpenAI provider:

```
provider = select_ai.OpenAIProvider(
    model="gpt-4.1",
)
```

For the public OpenAI service, leave `provider_endpoint` at its default. The SDK sends `provider="openai"` and the model name, without an endpoint override. `OpenAIProvider` omits `provider_endpoint` from database profile payloads. Use the base `Provider` class for a custom OpenAI-compatible endpoint, as shown below. Do not append `/v1/chat/completions`; the database constructs the request path.

Azure OpenAI provider:

```
provider = select_ai.AzureProvider(
    azure_resource_name="my-azure-openai-resource",
    azure_deployment_name="gpt-4o-deployment",
    azure_embedding_deployment_name="text-embedding-deployment",
)
```

AWS Bedrock provider:

```
provider = select_ai.AWSProvider(
    region="us-east-1",
    aws_apiformat="ANTHROPIC",
    model="anthropic.claude-3-5-sonnet-20240620-v1:0",
)
```

Custom provider endpoint:

```
select_ai.create_credential(
    credential={
        "credential_name": "xai_credential",
        "username": "xai",
        "password": "<xai_api_key>",
    },
    replace=True,
)

xai_profile = select_ai.Profile(
    profile_name="xai",
    attributes=select_ai.ProfileAttributes(
        provider=select_ai.Provider(
            provider_endpoint="https://api.x.ai",
            model="grok-4-1-fast-reasoning",
        ),
        credential_name="xai_credential",
        object_list=[
            {"owner": "SH", "name": "CUSTOMERS"},
            {"owner": "SH", "name": "SALES"},
            {"owner": "SH", "name": "PRODUCTS"},
            {"owner": "SH", "name": "COUNTRIES"},
        ],
    ),
    replace=True,
)

sql = xai_profile.show_sql(
    prompt="How many customers do I have?",
)
```

4.1.2 Provider

class select_ai.**Provider**(*embedding_model: str | None = None*, *model: str | None = None*, *provider_name: str | None = None*, *provider_endpoint: str | None = None*, *region: str | None = None*)

Base class for AI Provider

To create an object of Provider class, use any one of the concrete AI provider implementations

Parameters

- **embedding_model** (*str*) – The embedding model, also known as a transformer. Depending on the AI provider, the supported embedding models vary
- **model** (*str*) – The name of the LLM being used to generate responses
- **provider_name** (*str*) – The name of the provider being used
- **provider_endpoint** (*str*) – Endpoint URL of the AI provider being used
- **region** (*str*) – The cloud region of the Gen AI cluster

classmethod create(** (Keyword-only parameters separator (PEP 3102))*, *provider_name: str | None = None*, ***kwargs*)

dict(*exclude_null=True*)

embedding_model: str | None = None

json(*exclude_null=True*)

classmethod key_alias(*k*)

classmethod keys()

model: str | None = None

profile_dict(*exclude_null=True*)

Return provider attributes suitable for a DBMS_CLOUD_AI profile.

The result contains only values held by this provider instance. In particular, native provider endpoints remain available for network access configuration but are omitted from database profile payloads. Endpoint-only custom providers retain provider_endpoint.

provider_endpoint: str | None = None

provider_name: str | None = None

region: str | None = None

should_include_provider_endpoint() → bool

Whether to include provider_endpoint in a DBMS_CLOUD_AI profile.

4.1.3 AnthropicProvider

class select_ai.**AnthropicProvider**(*embedding_model: str | None = None*, *model: str | None = None*, *provider_name: str = 'anthropic'*, *provider_endpoint: str | None = 'api.anthropic.com'*, *region: str | None = None*)

Anthropic specific attributes

classmethod create(*, *provider_name: str | None = None*, ***kwargs*)

dict(*exclude_null=True*)

embedding_model: str | None = None

json(*exclude_null=True*)

classmethod key_alias(*k*)

classmethod keys()

model: str | None = None

profile_dict(*exclude_null=True*)

Return provider attributes suitable for a DBMS_CLOUD_AI profile.

The result contains only values held by this provider instance. In particular, native provider endpoints remain available for network access configuration but are omitted from database profile payloads. Endpoint-only custom providers retain provider_endpoint.

provider_endpoint: str | None = 'api.anthropic.com'

provider_name: str = 'anthropic'

region: str | None = None

should_include_provider_endpoint() → bool

Whether to include provider_endpoint in a DBMS_CLOUD_AI profile.

4.1.4 AzureProvider

class select_ai.**AzureProvider**(*embedding_model: str | None = None*, *model: str | None = None*, *provider_name: str = 'azure'*, *provider_endpoint: str | None = None*, *region: str | None = None*, *azure_deployment_name: str | None = None*, *azure_embedding_deployment_name: str | None = None*, *azure_resource_name: str | None = None*)

Azure specific attributes

Parameters

- **azure_deployment_name** (*str*) – Name of the Azure OpenAI Service deployed model.
- **azure_embedding_deployment_name** (*str*) – Name of the Azure OpenAI deployed embedding model.
- **azure_resource_name** (*str*) – Name of the Azure OpenAI Service resource

azure_deployment_name: str | None = None

azure_embedding_deployment_name: str | None = None

azure_resource_name: str | None = None

classmethod create(**, provider_name: str | None = None, **kwargs*)

dict(*exclude_null=True*)

embedding_model: str | None = None

json(*exclude_null=True*)

classmethod key_alias(*k*)

classmethod keys()

model: str | None = None

profile_dict(*exclude_null=True*)

Return provider attributes suitable for a DBMS_CLOUD_AI profile.

The result contains only values held by this provider instance. In particular, native provider endpoints remain available for network access configuration but are omitted from database profile payloads. Endpoint-only custom providers retain provider_endpoint.

provider_endpoint: str | None = None

provider_name: str = 'azure'

region: str | None = None

should_include_provider_endpoint() → bool

Whether to include provider_endpoint in a DBMS_CLOUD_AI profile.

4.1.5 AWSProvider

class select_ai.**AWSProvider**(*embedding_model: str | None = None*, *model: str | None = None*, *provider_name: str = 'aws'*, *provider_endpoint: str | None = None*, *region: str | None = None*, *aws_apiformat: str | None = None*)

AWS specific attributes

aws_apiformat: str | None = None

classmethod create(*, *provider_name: str | None = None*, ***kwargs*)

dict(*exclude_null=True*)

embedding_model: str | None = None

json(*exclude_null=True*)

classmethod key_alias(*k*)

classmethod keys()

model: str | None = None

profile_dict(*exclude_null=True*)

Return provider attributes suitable for a DBMS_CLOUD_AI profile.

The result contains only values held by this provider instance. In particular, native provider endpoints remain available for network access configuration but are omitted from database profile payloads. Endpoint-only custom providers retain provider_endpoint.

provider_endpoint: str | None = None

provider_name: str = 'aws'

region: str | None = None

should_include_provider_endpoint() → bool

Whether to include provider_endpoint in a DBMS_CLOUD_AI profile.

4.1.6 CohereProvider

class select_ai.**CohereProvider**(*embedding_model: str | None = None*, *model: str | None = None*, *provider_name: str = 'cohere'*, *provider_endpoint: str | None = 'api.cohere.ai'*, *region: str | None = None*)

Cohere AI specific attributes

classmethod create(*, *provider_name: str | None = None*, ***kwargs*)

dict(*exclude_null=True*)

embedding_model: str | None = None

json(*exclude_null=True*)

classmethod key_alias(*k*)

classmethod keys()

model: str | None = None

profile_dict(*exclude_null=True*)

Return provider attributes suitable for a DBMS_CLOUD_AI profile.

The result contains only values held by this provider instance. In particular, native provider endpoints remain available for network access configuration but are omitted from database profile payloads. Endpoint-only custom providers retain provider_endpoint.

provider_endpoint: str | None = 'api.cohere.ai'

provider_name: str = 'cohere'

region: str | None = None

should_include_provider_endpoint() → bool

Whether to include provider_endpoint in a DBMS_CLOUD_AI profile.

4.1.7 OpenAIProvider

class select_ai.**OpenAIProvider**(*embedding_model: str | None = None*, *model: str | None = None*, *provider_name: str = 'openai'*, *provider_endpoint: str | None = 'api.openai.com'*, *region: str | None = None*)

OpenAI specific attributes.

The default hostname is retained for network ACL configuration and omitted from database profiles so DBMS_CLOUD_AI selects its native public endpoint. Use Provider for a custom OpenAI-compatible endpoint.

classmethod create(**, provider_name: str | None = None, **kwargs*)

dict(*exclude_null=True*)

embedding_model: str | None = None

json(*exclude_null=True*)

classmethod key_alias(*k*)

classmethod keys()

model: str | None = None

profile_dict(*exclude_null=True*)

Return provider attributes suitable for a DBMS_CLOUD_AI profile.

The result contains only values held by this provider instance. In particular, native provider endpoints remain available for network access configuration but are omitted from database profile payloads. Endpoint-only custom providers retain provider_endpoint.

provider_endpoint: str | None = 'api.openai.com'

provider_name: str = 'openai'

region: str | None = None

should_include_provider_endpoint() → bool

Whether to include provider_endpoint in a DBMS_CLOUD_AI profile.

4.1.8 OCIGenAIProvider

class select_ai.**OCIGenAIProvider**(*embedding_model: str | None = None*, *model: str | None = None*, *provider_name: str = 'oci'*, *provider_endpoint: str | None = None*, *region: str | None = None*, *oci_apiformat: str | None = None*, *oci_compartment_id: str | None = None*, *oci_endpoint_id: str | None = None*, *oci_runtimetype: str | None = None*)

OCI Gen AI specific attributes

Parameters

- **oci_apiformat** (*str*) – Specifies the format in which the API expects data to be sent and received. Supported values are 'COHERE' and 'GENERIC'
- **oci_compartment_id** (*str*) – Specifies the OCID of the compartment you are permitted to access when calling the OCI Generative AI service
- **oci_endpoint_id** (*str*) – This attributes indicates the endpoint OCID of the Oracle dedicated AI hosting cluster
- **oci_runtimetype** (*str*) – This attribute indicates the runtime type of the provided model. The supported values are 'COHERE' and 'LLAMA'

classmethod create(**, provider_name: str | None = None, **kwargs*)

dict(*exclude_null=True*)

embedding_model: str | None = None

json(*exclude_null=True*)

classmethod key_alias(*k*)

classmethod keys()

model: str | None = None

oci_apiformat: str | None = None

oci_compartment_id: str | None = None

oci_endpoint_id: str | None = None

oci_runtimetype: str | None = None

profile_dict(*exclude_null=True*)

Return provider attributes suitable for a DBMS_CLOUD_AI profile.

The result contains only values held by this provider instance. In particular, native provider endpoints remain available for network access configuration but are omitted from database profile payloads. Endpoint-only custom providers retain provider_endpoint.

provider_endpoint: str | None = None

provider_name: str = 'oci'

region: str | None = None

should_include_provider_endpoint() → bool

Whether to include provider_endpoint in a DBMS_CLOUD_AI profile.

4.1.9 GoogleProvider

class select_ai.**GoogleProvider**(*embedding_model: str | None = None*, *model: str | None = None*, *provider_name: str = 'google'*, *provider_endpoint: str | None = 'generativelanguage.googleapis.com'*, *region: str | None = None*)

Google AI specific attributes

classmethod create(*, *provider_name: str | None = None*, ***kwargs*)

dict(*exclude_null=True*)

embedding_model: str | None = None

json(*exclude_null=True*)

classmethod key_alias(*k*)

classmethod keys()

model: str | None = None

profile_dict(*exclude_null=True*)

Return provider attributes suitable for a DBMS_CLOUD_AI profile.

The result contains only values held by this provider instance. In particular, native provider endpoints remain available for network access configuration but are omitted from database profile payloads. Endpoint-only custom providers retain provider_endpoint.

provider_endpoint: str | None = 'generativelanguage.googleapis.com'

provider_name: str = 'google'

region: str | None = None

should_include_provider_endpoint() → bool

Whether to include provider_endpoint in a DBMS_CLOUD_AI profile.

4.1.10 HuggingFaceProvider

class select_ai.**HuggingFaceProvider**(*embedding_model: str | None = None*, *model: str | None = None*, *provider_name: str = 'huggingface'*, *provider_endpoint: str | None = 'api-inference.huggingface.co'*, *region: str | None = None*)

HuggingFace specific attributes

classmethod create(*, *provider_name: str | None = None*, ***kwargs*)

dict(*exclude_null=True*)

embedding_model: str | None = None

json(*exclude_null=True*)

classmethod key_alias(*k*)

classmethod keys()

model: str | None = None

profile_dict(*exclude_null=True*)

Return provider attributes suitable for a DBMS_CLOUD_AI profile.

The result contains only values held by this provider instance. In particular, native provider endpoints remain available for network access configuration but are omitted from database profile payloads. Endpoint-only custom providers retain provider_endpoint.

provider_endpoint: str | None = 'api-inference.huggingface.co'

provider_name: str = 'huggingface'

region: str | None = None

should_include_provider_endpoint() → bool

Whether to include provider_endpoint in a DBMS_CLOUD_AI profile.

4.1.11 Enable AI service provider

Enable using Sync API

This method adds ACL allowing database users to invoke AI provider's HTTP endpoint. For non-HTTP or port-specific network access, use the network ACL helpers described in *Privileges*.

```
import os

import select_ai

admin_user = os.getenv("SELECT_AI_ADMIN_USER")
password = os.getenv("SELECT_AI_ADMIN_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
select_ai_user = os.getenv("SELECT_AI_USER")

select_ai.connect(user=admin_user, password=password, dsn=dsn)
select_ai.grant_http_access(
    users=select_ai_user, provider_endpoint="api.OPENAI.com"
)
print("Enabled AI provider for user: ", select_ai_user)
```

output:

```
Enabled AI provider for user: <select_ai_db_user>
```

Enable using Async API

```
import asyncio
import os

import select_ai

admin_user = os.getenv("SELECT_AI_ADMIN_USER")
password = os.getenv("SELECT_AI_ADMIN_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
select_ai_user = os.getenv("SELECT_AI_USER")


async def main():
    await select_ai.async_connect(user=admin_user, password=password, dsn=dsn)
    await select_ai.async_grant_http_access(
        users=select_ai_user, provider_endpoint="*.openai.azure.com"
    )
    print("Enabled AI provider for user: ", select_ai_user)


asyncio.run(main())
```

output:

```
Enabled AI provider for user: <select_ai_db_user>
```

4.1.12 Cloud provider profile samples

The repository includes complete profile-creation samples for AWS Bedrock, Azure OpenAI, and Google Gemini. Each sample:

- grants the Select AI database user HTTP access to the provider endpoint;
- creates or replaces a provider credential; and
- creates or replaces a profile, then sends a test chat request.

Run the samples from the repository root after setting the common database variables:

- `SELECT_AI_ADMIN_USER` and `SELECT_AI_ADMIN_PASSWORD`;
- `SELECT_AI_USER` and `SELECT_AI_PASSWORD`;
- `SELECT_AI_DB_CONNECT_STRING`; and
- the provider-specific API key or access-key variables described below.

Do not store production secrets in source files or commit them to the repository. Adjust the sample region, model, resource, deployment, and object names for the provider account and database used by your application.

AWS Bedrock

The AWS Bedrock profile sample uses `AWSProvider` with the following provider settings:

- `region`: AWS Bedrock region;
- `model`: text-generation model identifier; and
- `embedding_model`: embedding model identifier.

Set `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` before running it:

```
python samples/profile_create_aws.py
```

First, load the configuration and grant network access as the administrator:

```
import os
from pprint import pformat

import select_ai

AWS_HOST = "bedrock-runtime.us-east-1.amazonaws.com"
CREDENTIAL_NAME = "AWS_CRED"
PROFILE_NAME = "aws_bedrock_meta_prf"

admin_user = os.environ["SELECT_AI_ADMIN_USER"]
admin_password = os.environ["SELECT_AI_ADMIN_PASSWORD"]
app_user = os.environ["SELECT_AI_USER"]
app_password = os.environ["SELECT_AI_PASSWORD"]
dsn = os.environ["SELECT_AI_DB_CONNECT_STRING"]

# Equivalent to DBMS_NETWORK_ACL_ADMIN.APPEND_HOST_ACE(..., 'http').
select_ai.connect(user=admin_user, password=admin_password, dsn=dsn)
try:
    select_ai.grant_network_access(
        users=app_user,
        host=AWS_HOST,
        privileges="http",
    )
finally:
    select_ai.disconnect()
```

Then connect as the application user to create the credential and profile. Append this code to the setup above:

```
select_ai.connect(user=app_user, password=app_password, dsn=dsn)
try:
    select_ai.create_credential(
        {
            "credential_name": CREDENTIAL_NAME,
            "username": os.environ["AWS_ACCESS_KEY_ID"],
            "password": os.environ["AWS_SECRET_ACCESS_KEY"],
        },
        replace=True,
    )

    profile = select_ai.Profile(
        profile_name=PROFILE_NAME,
        attributes=select_ai.ProfileAttributes(
            credential_name=CREDENTIAL_NAME,
            provider=select_ai.AWSProvider(
                region="us-east-1",
                model="meta.llama3-70b-instruct-v1:0",
                embedding_model="amazon.titan-embed-text-v1",
            ),
            object_list=[{"owner": app_user, "name": "CUSTOMERS"}],
            conversation=True,
            temperature=1,
            max_tokens=1500,
        ),
        replace=True,
    )

    print("Created profile:", profile.profile_name)
    print(pformat(profile.get_attributes().dict(exclude_null=False)))
    print("Chat response:", profile.chat("Reply with: AWS chat succeeded."))
finally:
    select_ai.disconnect()
```

The sample creates `aws_bedrock_meta_prf` and prints the saved attributes and a test response:

output:

```
Created profile: aws_bedrock_meta_prf
{'credential_name': 'AWS_CRED', 'provider': AWSProvider(...), ...}
Chat response: AWS chat succeeded.
```

Azure OpenAI

The Azure OpenAI profile sample uses `AzureProvider` with the following provider settings:

- `azure_resource_name`: Azure OpenAI resource name;
- `azure_deployment_name`: text-generation deployment name; and
- `azure_embedding_deployment_name`: embedding deployment name.

Set `AZURE_API_KEY` before running it:

```
python samples/profile_create_azure.py
```

First, load the configuration and grant network access as the administrator:

```
import os
from pprint import pformat

import select_ai

AZURE_HOST = "adbst-ai-resource-japan-east.openai.azure.com"
AZURE_RESOURCE = "ADBST-AI-RESOURCE-JAPAN-EAST"
AZURE_DEPLOYMENT = "ADBST-AI-RESOURCE-JAPAN-EAST-DEPLOYMENT"
AZURE_EMBEDDING_DEPLOYMENT = (
    "ADBST-AI-RESOURCE-JAPAN-EAST-text-embedding-3-large"
)
CREDENTIAL_NAME = "AZUREAI_CRED"
PROFILE_NAME = "azureai_prf"

admin_user = os.environ["SELECT_AI_ADMIN_USER"]
admin_password = os.environ["SELECT_AI_ADMIN_PASSWORD"]
app_user = os.environ["SELECT_AI_USER"]
app_password = os.environ["SELECT_AI_PASSWORD"]
dsn = os.environ["SELECT_AI_DB_CONNECT_STRING"]

# Equivalent to DBMS_NETWORK_ACL_ADMIN.APPEND_HOST_ACE(..., 'http').
select_ai.connect(user=admin_user, password=admin_password, dsn=dsn)
try:
    select_ai.grant_network_access(
        users=app_user,
        host=AZURE_HOST,
        privileges="http",
    )
finally:
    select_ai.disconnect()
```

Then connect as the application user to create the credential and profile. Append this code to the setup above:

```
select_ai.connect(user=app_user, password=app_password, dsn=dsn)
try:
    select_ai.create_credential(
        {
            "credential_name": CREDENTIAL_NAME,
            "username": "azure",
            "password": os.environ["AZURE_API_KEY"],
        },
        replace=True,
    )

    profile = select_ai.Profile(
        profile_name=PROFILE_NAME,
        attributes=select_ai.ProfileAttributes(
            credential_name=CREDENTIAL_NAME,
            provider=select_ai.AzureProvider(
                azure_resource_name=AZURE_RESOURCE,
                azure_deployment_name=AZURE_DEPLOYMENT,
                azure_embedding_deployment_name=AZURE_EMBEDDING_DEPLOYMENT,
            ),
            object_list=[{"owner": app_user, "name": "CUSTOMERS"}],
            conversation=True,
            temperature=1,
            max_tokens=1500,
            seed=20,
        ),
        replace=True,
    )
    p = select_ai.Profile.fetch(profile_name=PROFILE_NAME)
    print(p)
    print("Created profile:", profile.profile_name)
    print(pformat(profile.get_attributes().dict(exclude_null=False)))
    print("Chat response:", profile.chat("Reply with: Azure chat succeeded."))
finally:
    select_ai.disconnect()
```

The sample creates `azureai_prf`, fetches it to verify the saved profile, and prints a test response:

output:

```
Profile(profile_name=azureai_prf, ...)
Created profile: azureai_prf
{'credential_name': 'AZUREAI_CRED', 'provider': AzureProvider(...), ...}
Chat response: Azure chat succeeded.
```

Google Gemini (GCP)

The Google Gemini profile sample uses `GoogleProvider` with the following provider settings:

- `model`: Gemini text-generation model identifier; and
- `embedding_model`: Gemini embedding model identifier.

Set `GOOGLE_API_KEY` before running it:

```
python samples/profile_create_gcp.py
```

First, load the configuration and grant network access as the administrator:

```
import os
from pprint import pformat

import select_ai

GCP_HOST = "generativelanguage.googleapis.com"
CREDENTIAL_NAME = "GOOGLE_CRED"
PROFILE_NAME = "google_gemini_3_6_flash"

admin_user = os.environ["SELECT_AI_ADMIN_USER"]
admin_password = os.environ["SELECT_AI_ADMIN_PASSWORD"]
app_user = os.environ["SELECT_AI_USER"]
app_password = os.environ["SELECT_AI_PASSWORD"]
dsn = os.environ["SELECT_AI_DB_CONNECT_STRING"]

# Equivalent to DBMS_NETWORK_ACL_ADMIN.APPEND_HOST_ACE(..., 'http').
select_ai.connect(user=admin_user, password=admin_password, dsn=dsn)
try:
    select_ai.grant_network_access(
        users=app_user,
        host=GCP_HOST,
        privileges="http",
    )
finally:
    select_ai.disconnect()
```

Then connect as the application user to create the credential and profile. Append this code to the setup above:

```
select_ai.connect(user=app_user, password=app_password, dsn=dsn)
try:
    select_ai.create_credential(
        {
            "credential_name": CREDENTIAL_NAME,
            "username": "GOOGLE",
            "password": os.environ["GOOGLE_API_KEY"],
        },
        replace=True,
    )

    profile = select_ai.Profile(
        profile_name=PROFILE_NAME,
        attributes=select_ai.ProfileAttributes(
            credential_name=CREDENTIAL_NAME,
            provider=select_ai.GoogleProvider(
                embedding_model="gemini-embedding-001",
                model="gemini-3.6-flash",
            ),
            temperature=1,
            max_tokens=1500,
            seed=20,
        ),
        replace=True,
    )

    print("Created profile:", profile.profile_name)
    print(pformat(profile.get_attributes().dict(exclude_null=False)))
    print("Chat response:", profile.chat("Reply with: GCP chat succeeded."))
finally:
    select_ai.disconnect()
```

The sample creates `google_gemini_3_6_flash` and prints the saved attributes and a test response:

output:

```
Created profile: google_gemini_3_6_flash
{'credential_name': 'GOOGLE_CRED', 'provider': GoogleProvider(...), ...}
Chat response: GCP chat succeeded.
```

4.1.13 Disable AI service provider

This method removes the ACL entry that allows database users to invoke an AI provider's HTTP endpoint.

Disable using Sync API

```
import os

import select_ai

admin_user = os.getenv("SELECT_AI_ADMIN_USER")
password = os.getenv("SELECT_AI_ADMIN_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
select_ai_user = os.getenv("SELECT_AI_USER")

select_ai.connect(user=admin_user, password=password, dsn=dsn)
select_ai.revoke_http_access(
    users=select_ai_user, provider_endpoint="*.openai.azure.com"
)
print("Disabled AI provider for user: ", select_ai_user)
```

output:

```
Disabled AI provider for user:  <select_ai_db_user>
```

Disable using Async API

```
import asyncio
import os

import select_ai

admin_user = os.getenv("SELECT_AI_ADMIN_USER")
password = os.getenv("SELECT_AI_ADMIN_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
select_ai_user = os.getenv("SELECT_AI_USER")


async def main():
    await select_ai.async_connect(user=admin_user, password=password, dsn=dsn)
    await select_ai.async_revoke_http_access(
        users=select_ai_user, provider_endpoint="*.openai.azure.com"
    )
    print("Disabled AI provider for user: ", select_ai_user)


asyncio.run(main())
```

output:

```
Disabled AI provider for user:  <select_ai_db_user>
```

CHAPTER

FIVE

CREDENTIAL

A credential object securely stores authentication details from your AI provider for use by Oracle Database. Select AI profiles, vector indexes, and agent tools refer to the credential later by `credential_name`; the secret values are stored in Oracle Database and are not passed again when the profile or tool runs.

A credential is created in the connected user's schema by `DBMS_CLOUD.CREATE_CREDENTIAL`. Create credentials while connected as the database user that will own and use them. Before creating credentials, make sure the user has the required Select AI package privileges. If the credential will be used to call an external AI provider, the database user also needs network access to that provider endpoint.

Every credential object must include `credential_name` and the fields required by the target provider. The library accepts the following credential keys: `credential_name`, `username`, `password`, `user_ocid`, `tenancy_ocid`, `private_key`, `fingerprint`, and `comments`.

The following table shows AI providers and corresponding credential object formats.

Table 1: AI provider and expected credential format

AI provider	Credential format
Anthropic	`{ "credential_name": "ANTHROPIC_CRED", "username": "anthropic", "password": "sk-ant-xxx", }`
AWS Bedrock	`{ "credential_name": "AWS_BEDROCK_CRED", "username": "<aws_access_key_id>", "password": "<aws_secret_access_key>", }`
Azure OpenAI	`{ "credential_name": "AZURE_OPENAI_CRED", "username": "azure", "password": "<azure_openai_api_key>", }`

continues on next page

Table 1 – continued from previous page

AI provider	Credential format
Cohere	`{ "credential_name": "COHERE_CRED", "username": "cohere", "password": "<cohere_api_key>", }`
Google	`{ "credential_name": "GOOGLE_CRED", "username": "google", "password": "<google_api_key>", }`
HuggingFace	`{ "credential_name": "HUGGINGFACE_CRED", "username": "hf", "password": "hf_xxx", }`
OCI Gen AI	`{ "credential_name": "OCI_GENAI_CRED", "user_ocid": "<user_ocid>", "tenancy_ocid": "<tenancy_ocid>", "private_key": "<private_key_contents>", "fingerprint": "<fingerprint>", }`
OpenAI	`{ "credential_name": "OPENAI_CRED", "username": "openai", "password": "sk-xxx", }`

5.1 Create credential

In this example, we create a credential object to authenticate to OCI Gen AI service provider:

The synchronous and asynchronous samples below use the OCI Python SDK to read an existing API-signing configuration. Install this sample-specific dependency:

```
python -m pip install oci
```

Configure the `DEFAULT` profile in `~/.oci/config` with `user`, `tenancy`, `fingerprint`, `region`, and `key_file`. The `key_file` must point to a readable private key whose public key is registered with the OCI user. See OCI SDK configuration. The samples call `oci.config.from_file()` with its default path and profile; edit that call if you use another configuration. Keep the configuration and private key outside your source repository.

The `oci` package is needed only to load this configuration in the samples. The Select AI credential API itself accepts the credential dictionary shown above without requiring the OCI SDK.

Pass `replace=True` when you want to recreate an existing credential with the same name. Without `replace=True`, creating a credential that already exists raises a database error.

Pass `public_synonym=True` to also create a public synonym with the same name as the credential. This requires the `CREATE PUBLIC SYNONYM` system privilege. The credential samples create the synonym and remove it during the delete step.

5.1.1 Create Sync API

```
import os

import oci
import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)

# Default config file and profile
default_config = oci.config.from_file()
oci.config.validate_config(default_config)
with open(default_config["key_file"]) as fp:
    key_contents = fp.read()
credential = {
    "credential_name": "my_oci_ai_profile_key",
    "user_ocid": default_config["user"],
    "tenancy_ocid": default_config["tenancy"],
    "private_key": key_contents,
    "fingerprint": default_config["fingerprint"],
}
select_ai.create_credential(
    credential=credential,
    replace=True,
    public_synonym=True,
)
print("Created credential: ", credential["credential_name"])
```

output:

```
Created credential:  my_oci_ai_profile_key
```

5.1.2 Create Async API

```
import asyncio
import os

import oci
import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    default_config = oci.config.from_file()
    oci.config.validate_config(default_config)
    with open(default_config["key_file"]) as fp:
        key_contents = fp.read()
    credential = {
        "credential_name": "my_oci_ai_profile_key",
        "user_ocid": default_config["user"],
        "tenancy_ocid": default_config["tenancy"],
        "private_key": key_contents,
        "fingerprint": default_config["fingerprint"],
    }
    await select_ai.async_create_credential(
        credential=credential,
        replace=True,
        public_synonym=True,
    )
    print("Created credential: ", credential["credential_name"])


asyncio.run(main())
```

output:

```
Created credential:  my_oci_ai_profile_key
```

5.2 Delete credential

Use `select_ai.delete_credential(...)` to drop a credential that is no longer needed. Pass `force=True` when cleanup should succeed even if the credential does not exist. Pass `public_synonym=True` to drop the credential's public synonym as well; this requires the `DROP PUBLIC SYNONYM` system privilege.

5.2.1 Delete Sync API

```
import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
select_ai.delete_credential(
    credential_name="my_oci_ai_profile_key",
    force=True,
    public_synonym=True,
)
print("Deleted credential: my_oci_ai_profile_key")
```

output:

```
Deleted credential: my_oci_ai_profile_key
```

5.2.2 Delete Async API

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    await select_ai.async_delete_credential(
        credential_name="my_oci_ai_profile_key",
        force=True,
        public_synonym=True,
    )
    print("Deleted credential: my_oci_ai_profile_key")


asyncio.run(main())
```

output:

```
Deleted credential: my_oci_ai_profile_key
```

5.3 Credential access

Use `select_ai.grant_credential_access(...)` and `select_ai.revoke_credential_access(...)` to grant or revoke credential access for a database user or role. The asynchronous equivalents are `async_grant_credential_access(...)` and `async_revoke_credential_access(...)`.

```
select_ai.grant_credential_access(
    "MY_PROVIDER_CREDENTIAL",
    "APP_USER",
)
select_ai.revoke_credential_access(
    "MY_PROVIDER_CREDENTIAL",
    "APP_USER",
)
```

Use `await` with the asynchronous equivalents:

```
await select_ai.async_grant_credential_access(
    "MY_PROVIDER_CREDENTIAL",
    "APP_USER",
)
await select_ai.async_revoke_credential_access(
    "MY_PROVIDER_CREDENTIAL",
    "APP_USER",
)
```

CHAPTER

SIX

SHARING AND OWNERSHIP

6.1 Sharing and ownership

Share a saved profile, vector index, agent team, or credential with a database user or role. Run grants and revokes while connected as the object owner. The recipient also needs the database and package *Privileges* required to use that object.

The examples below use owner `APP_OWNER` and recipient `APP_USER`. Replace these names with your schemas. Each example assumes that its named object already exists. Sharing an object does not grant access to other database objects or services used by the workflow.

6.1.1 Connect before running an example

For synchronous examples, connect as the object owner:

```
import os
import select_ai

select_ai.connect(
    user=os.environ["SELECT_AI_USER"],        # APP_OWNER
    password=os.environ["SELECT_AI_PASSWORD"],
    dsn=os.environ["SELECT_AI_DB_CONNECT_STRING"],
)
```

Call `select_ai.disconnect()` when finished. For asynchronous examples, use `await select_ai.async_connect(...)` with the same arguments inside an async function, and call `await select_ai.async_disconnect()` when finished. The async snippets below belong inside that function.

6.1.2 Profiles

Grant access to an existing profile while connected as `APP_OWNER`:

```
profile = select_ai.Profile.fetch("OCI_AI_PROFILE")
profile.grant_access("APP_USER")
```

In a separate connection as `APP_USER`, fetch the shared profile by its name and owner. The `qualified_name` property identifies both:

```
profile = select_ai.Profile.fetch("OCI_AI_PROFILE", owner="APP_OWNER")
print(profile.qualified_name)
print(profile.chat(prompt="What is retrieval augmented generation?"))
```

The qualified name printed is `APP_OWNER.OCI_AI_PROFILE`. The chat answer depends on the configured provider. To withdraw access, reconnect as `APP_OWNER` and run:

```
profile = select_ai.Profile.fetch("OCI_AI_PROFILE")
profile.revoke_access("APP_USER")
```

The async sequence uses `AsyncProfile`. Run each step in the indicated user's connection:

```
# As APP_OWNER: share the profile.
profile = await select_ai.AsyncProfile.fetch("OCI_AI_PROFILE")
await profile.grant_access("APP_USER")

# In a separate connection as APP_USER: use the shared profile.
shared = await select_ai.AsyncProfile.fetch(
    "OCI_AI_PROFILE", owner="APP_OWNER"
)
print(shared.qualified_name)
print(await shared.chat(prompt="What is retrieval augmented generation?"))

# Reconnect as APP_OWNER when access should end.
await profile.revoke_access("APP_USER")
```

`Profile.list()` and `AsyncProfile.list()` also accept `owner`. If it is omitted, they list the current schema. See *List profiles* and *List profiles asynchronously* for connection setup, filtering, and reuse.

6.1.3 Vector indexes

Share an existing vector index while connected as `APP_OWNER`:

```
index = select_ai.VectorIndex.fetch("PRODUCT_DOCS")
index.grant_access("APP_USER")
```

In a separate connection as `APP_USER`, address the index by its owner:

```
index = select_ai.VectorIndex.fetch("PRODUCT_DOCS", owner="APP_OWNER")
print(index.qualified_name)  # APP_OWNER.PRODUCT_DOCS
```

Reconnect as `APP_OWNER` to withdraw access:

```
index = select_ai.VectorIndex.fetch("PRODUCT_DOCS")
index.revoke_access("APP_USER")
```

Use `AsyncVectorIndex` for the same sequence in an async application:

```
# As APP_OWNER.
index = await select_ai.AsyncVectorIndex.fetch("PRODUCT_DOCS")
await index.grant_access("APP_USER")

# In a separate connection as APP_USER.
shared = await select_ai.AsyncVectorIndex.fetch(
    "PRODUCT_DOCS", owner="APP_OWNER"
)
print(shared.qualified_name)

# Reconnect as APP_OWNER when access should end.
await index.revoke_access("APP_USER")
```

`VectorIndex.list(owner="APP_OWNER")` lists visible indexes in that schema; `AsyncVectorIndex.list(owner="APP_OWNER")` returns an async iterator. See *Vector Index* for creating and using an index for RAG.

6.1.4 Agent teams

While connected as the team owner, grant access to an existing team:

```
from select_ai.agent import Team

team = Team.fetch("MOVIE_AGENT_TEAM")
team.grant_access("APP_USER")
```

When the recipient no longer needs access, run this as the team owner:

```
team.revoke_access("APP_USER")
```

For async applications, use `AsyncTeam` and await each operation:

```
from select_ai.agent import AsyncTeam

team = await AsyncTeam.fetch("MOVIE_AGENT_TEAM")
await team.grant_access("APP_USER")

# Revoke later, while connected as the team owner.
await team.revoke_access("APP_USER")
```

Team fetch and list operations use the connected user's agent-team views and do not accept an `owner` argument. See *AI Agent* for team creation and execution.

6.1.5 Credentials

Credentials use module-level access functions. While connected as the credential owner, share an existing provider credential:

```
select_ai.grant_credential_access("MY_PROVIDER_CREDENTIAL", "APP_USER")
```

Revoke access when the recipient no longer needs it:

```
select_ai.revoke_credential_access("MY_PROVIDER_CREDENTIAL", "APP_USER")
```

The async functions take the same arguments:

```
await select_ai.async_grant_credential_access(
    "MY_PROVIDER_CREDENTIAL", "APP_USER"
)

# Revoke later, while connected as the credential owner.
await select_ai.async_revoke_credential_access(
    "MY_PROVIDER_CREDENTIAL", "APP_USER"
)
```

Access functions return `None` after the grant or revoke completes. They do not print a confirmation message.

Credential creation and deletion can also manage a public synonym with the same name. Pass `public_synonym=True` to `create_credential()` or `delete_credential()` (or their async equivalents). Creating or dropping the synonym requires `CREATE PUBLIC SYNONYM` or `DROP PUBLIC SYNONYM` respectively. See *Credential* for credential setup.

6.1.6 Runnable sharing scripts

The repository also provides a synchronous sharing script and an asynchronous sharing script that exercise all four object types. Use these when you want to check the APIs together after working through the individual examples above.

Set the common database connection variables and `SELECT_AI_SHARE_GRANTEE`. The connected user must own all four objects. The scripts use these defaults; override them if your objects have other names:

Table 1: Sharing script configuration

Environment variable	Default
`SELECT_AI_PROFILE_NAME`	`oci_ai_profile` (sync); `async_oci_ai_profile` (async)
`SELECT_AI_VECTOR_INDEX_NAME`	`test_vector_index`
`SELECT_AI_TEAM_NAME`	`MOVIE_AGENT_TEAM`
`SELECT_AI_CREDENTIAL_NAME`	`my_oci_ai_profile_key`
`SELECT_AI_OWNER`	Current schema; if set, use the connected owner's schema.

Run either script from the repository root:

```
export SELECT_AI_SHARE_GRANTEE=APP_USER
python samples/sharing.py
# Or run the async version:
python samples/async/sharing.py
```

Each script prints the owner-qualified profile and vector-index names, then confirms each grant and revoke. The scripts revoke every grant they make; use the individual examples above to leave an object shared for application use.

CHAPTER

SEVEN

PROFILE ATTRIBUTES

7.1 ProfileAttributes

This class defines attributes that manage and configure the behavior of an AI profile. `ProfileAttributes` objects are created with `select_ai.ProfileAttributes()` and passed to `select_ai.Profile` or `select_ai.AsyncProfile` when creating or updating a profile.

Profile attributes describe what the profile can access, which AI provider it uses, how much metadata is sent to the model, and how generation should behave. Provider-specific settings, such as OCI region or Azure deployment name, are configured on the provider object and assigned to the `provider` attribute.

7.1.1 Common required attributes

Most profiles need these attributes:

- `provider`: A `select_ai.Provider` object, such as `select_ai.OCIGenAIProvider` or `select_ai.OpenAIProvider`.
- `credential_name`: The database credential used to authenticate with the AI provider.
- `object_list`: The schemas, tables, or views that Select AI can use when generating SQL from natural language prompts.

For example:

```
attributes = select_ai.ProfileAttributes(
    provider=select_ai.OCIGenAIProvider(
        region="us-chicago-1",
        oci_apiformat="GENERIC",
    ),
    credential_name="my_oci_ai_profile_key",
    object_list=[
        {"owner": "SH", "name": "CUSTOMERS"},
        {"owner": "SH", "name": "SALES"},
    ],
)
```

7.1.2 Attribute groups

Table 1: Profile attribute groups

Attribute	Purpose
`provider`	Selects the AI provider, model, endpoint, and provider-specific options.
`credential_name`	Names the database credential used to authenticate with the AI provider.
`object_list`	Defines which schemas, tables, or views are eligible for natural language to SQL generation.
`object_list_mode`	Controls whether Select AI sends metadata for the most relevant objects or for all eligible objects.
`enforce_object_list`	Restricts generated SQL to objects in `object_list`.
`comments`, `constraints`, `annotations`	Controls whether additional database metadata is included in the prompt sent to the model.
`case_sensitive_values`	Helps prompts that depend on case-sensitive database values.
`max_tokens`, `temperature`, `stop_tokens`, `seed`	Tunes model generation behavior.
`conversation`	Enables conversation history for context-aware chat workflows.
`additional_instructions`	Provides persistent guidance, business rules, or response constraints for requests that use the profile.
`source_language`, `target_language`	Set default languages for `Profile.translate()` and `AsyncProfile.translate()`. Per-call values override these defaults. If no source language is configured or supplied per call, the provider detects it. A target language must be supplied either per call or in the profile.
`vector_index_name`, `enable_sources`, `enable_source_offsets`, `enable_custom_source_uri`	Configures retrieval-augmented generation and source reporting for vector index workflows.

7.1.3 Object list examples

Include supported objects from a schema in the profile's metadata scope:

```
object_list = [{"owner": "SH"}]
```

Include selected tables:

```
object_list = [
    {"owner": "SH", "name": "CUSTOMERS"},
    {"owner": "SH", "name": "SALES"},
    {"owner": "SH", "name": "PRODUCTS"},
]
```

The object list does not grant database privileges. The connected user must already have the privileges needed to access the underlying objects.

Restrict generated SQL to the selected objects:

```
attributes = select_ai.ProfileAttributes(
    provider=provider,
    credential_name="my_oci_ai_profile_key",
    object_list=[
        {"owner": "SH", "name": "CUSTOMERS"},
        {"owner": "SH", "name": "SALES"},
```

(continues on next page)

(continued from previous page)

```
    ],
    enforce_object_list=True,
)
```

7.1.4 Generation controls

Use generation controls when you need more predictable or constrained model responses:

```
attributes = select_ai.ProfileAttributes(
    provider=provider,
    credential_name="my_oci_ai_profile_key",
    object_list=[{"owner": "SH"}],
    max_tokens=1024,
    temperature=0.1,
    stop_tokens='[";"]',
)
```

7.1.5 Additional instructions

Use `additional_instructions` for guidance that should apply to every request made with a profile:

```
attributes = select_ai.ProfileAttributes(
    provider=provider,
    credential_name="my_oci_ai_profile_key",
    additional_instructions="Return concise, executable Python code.",
)
```

For guidance that applies to only one request, pass an `attributes` mapping to an action method instead of changing the saved profile.

class select_ai.**ProfileAttributes**(*additional_instructions: str | None = None*, *annotations: bool | None = None*, *case_sensitive_values: bool | None = None*, *comments: bool | None = None*, *constraints: bool | None = None*, *conversation: bool | None = None*, *credential_name: str | None = None*, *enable_custom_source_uri: bool | None = None*, *enable_sources: bool | None = None*, *enable_source_offsets: bool | None = None*, *enforce_object_list: bool | None = None*, *max_tokens: int | None = 1024*, *object_list: List[Mapping] | None = None*, *object_list_mode: str | None = None*, *provider: Provider | None = None*, *seed: int | None = None*, *source_language: str | None = None*, *stop_tokens: str | None = None*, *streaming: str | None = None*, *temperature: float | None = None*, *target_language: str | None = None*, *vector_index_name: str | None = None*)

Use this class to define attributes to manage and configure the behavior of an AI profile

Parameters

- **comments** (*bool*) – True to include column comments in the metadata used for generating SQL queries from natural language prompts.
- **constraints** (*bool*) – True to include referential integrity constraints such as primary and foreign keys in the metadata sent to the LLM.
- **conversation** (*bool*) – Indicates if conversation history is enabled for a profile.

- **credential_name** (*str*) – The name of the credential to access the AI provider APIs.
- **enforce_object_list** (*bool*) – Specifies whether to restrict the LLM to generate SQL that uses only tables covered by the object list.
- **additional_instructions** (*str*) – Persistent guidance that Select AI applies to requests that use the profile.
- **max_tokens** (*int*) – Denotes the number of tokens to return per generation. Default is 1024.
- **object_list** (*List[Mapping]*) – Array of JSON objects specifying the owner and object names that are eligible for natural language translation to SQL.
- **object_list_mode** (*str*) – Specifies whether to send metadata for the most relevant tables or all tables to the LLM. Supported values are - 'automated' and 'all'
- **provider** (select_ai.Provider) – AI Provider
- **seed** (*int*) – Signed 64-bit integer used to make model output more reproducible when the provider supports it.
- **source_language** (*str*) – Default language of text passed to the translate operation. If omitted, the translation provider can detect the source language.
- **stop_tokens** (*str*) – The generated text will be terminated at the beginning of the earliest stop sequence. Sequence will be incorporated into the text. The attribute value must be a valid array of string values in JSON format
- **temperature** (*float*) – Temperature is a non-negative float number used to tune the degree of randomness. Lower temperatures mean less random generations.
- **target_language** (*str*) – Default language into which text is translated. This is required by the database when no target language is supplied to the translate operation.
- **vector_index_name** (*str*) – Name of the vector index

additional_instructions: str | None = None

annotations: bool | None = None

async classmethod async_create(***kwargs*)

case_sensitive_values: bool | None = None

comments: bool | None = None

constraints: bool | None = None

conversation: bool | None = None

classmethod create(***kwargs*)

credential_name: str | None = None

dict(*exclude_null=True*)

enable_custom_source_uri: bool | None = None

enable_source_offsets: bool | None = None

enable_sources: bool | None = None

enforce_object_list: bool | None = None

json(*exclude_null=True*)

classmethod keys()

max_tokens: int | None = 1024

object_list: List[Mapping] | None = None

object_list_mode: str | None = None

provider: *Provider* | None = None

seed: int | None = None

set_attribute(*key*, *value*)

source_language: str | None = None

stop_tokens: str | None = None

streaming: str | None = None

target_language: str | None = None

temperature: float | None = None

vector_index_name: str | None = None

CHAPTER

EIGHT

PROFILE

An AI profile is a specification that includes the AI provider to use and other details regarding metadata and database objects required for generating responses to natural language prompts.

An AI profile object can be created using `select_ai.Profile()`. Creating a profile stores the profile in Oracle Database. Later, you can instantiate `select_ai.Profile(profile_name="...")` to reuse an existing database profile without passing all attributes again.

Before creating a profile, make sure the database user has the required privileges, a credential for the AI provider, network access to the provider endpoint, and access to the database objects included in the profile. See *Privileges*, *Credential*, *Provider*, and *ProfileAttributes*.

8.1 Profile lifecycle

The usual profile lifecycle is:

- Create a provider object.
- Create `ProfileAttributes` with the provider, credential name, and object list.
- Create the profile with `select_ai.Profile(...)`.
- Reuse the profile later by name.
- Enable or disable the profile without deleting it.
- Update profile attributes when provider settings or object scope changes.
- Delete profiles that are no longer needed.

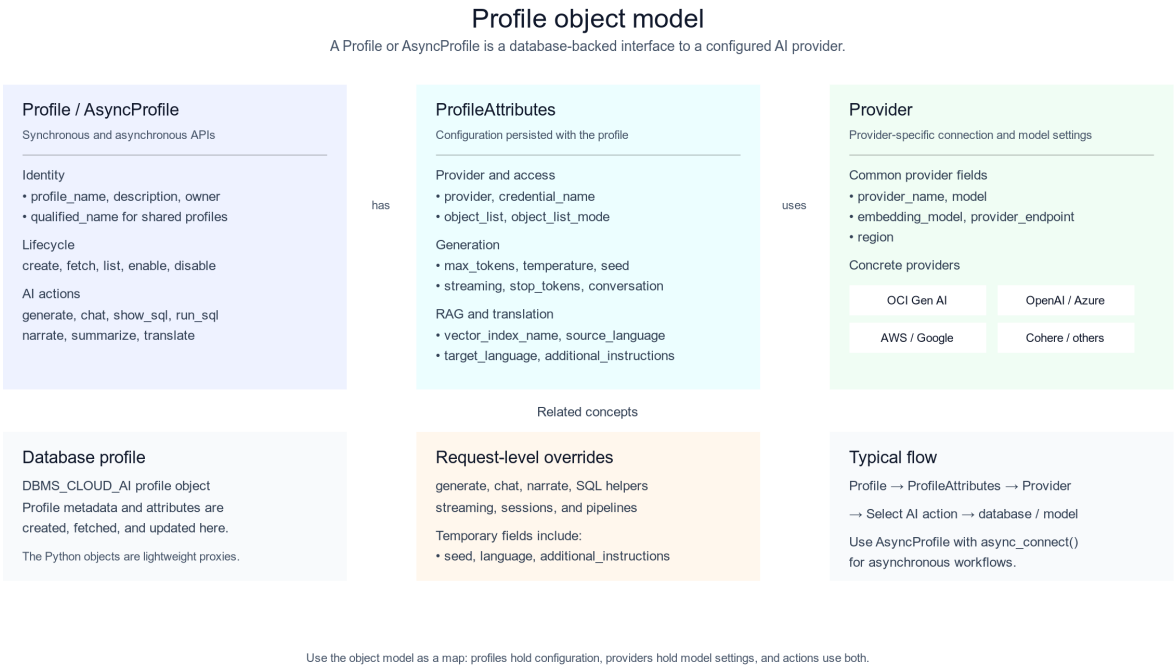
`replace=True` recreates a profile when a profile with the same name already exists. `merge=True` fetches the existing profile and updates it with the non-null attributes passed by the caller.

8.2 Profile actions

Table 1: Common profile actions

Method	Description
`show_sql()`	Generates SQL for a natural language prompt without executing it.
`run_sql()`	Generates SQL, executes it, and returns a `pandas.DataFrame`.
`narrate()`	Generates SQL, executes it, and returns a natural language answer.
`explain_sql()`	Explains the generated SQL for a prompt.
`show_prompt()`	Shows the prompt sent to the model.
`chat()`	Sends a general chat prompt to the model.
`summarize()`	Summarizes inline content or content referenced by a URI.
`translate()`	Translates text from one language to another.

8.2.1 Profile Object Model



8.2.2 Base `Profile` API

class select_ai.**BaseProfile**(*profile_name: str | None = None*, *attributes: ProfileAttributes | None = None*, *description: str | None = None*, *merge: bool | None = False*, *replace: bool | None = False*, *raise_error_if_exists: bool | None = True*, *raise_error_on_empty_attributes: bool | None = False*, *owner: str | None = None*)

BaseProfile is an abstract base class representing a Profile for Select AI's interactions with AI service providers (LLMs). Use either select_ai.Profile or select_ai.AsyncProfile to instantiate an AI profile object.

:param str profile_name : Name of the profile

Parameters

- **attributes** (*select_ai.ProfileAttributes*) – Object specifying AI profile attributes
- **description** (*str*) – Description of the profile
- **owner** (*str*) – Database user that owns the profile
- **merge** (*bool*) – Fetches the profile from database, merges the non-null attributes and saves it back in the database. Default value is False
- **replace** (*bool*) – Replaces the profile and attributes in the database. Default value is False
- **raise_error_if_exists** (*bool*) – Raise ProfileExistsError if profile exists in the database and replace = False and merge = False. Default value is True
- **raise_error_on_empty_attributes** (*bool*) – Raise ProfileEmptyAttributesError, if profile attributes are empty in database. Default value is False.

property qualified_name: `str | None`

Return the owner-qualified profile name when owner is known.

8.2.3 Profile API

class select_ai.**Profile**(**args*, ***kwargs*)

Profile class represents an AI Profile. It defines attributes and methods to interact with the underlying AI Provider. All methods in this class are synchronous or blocking

add_negative_feedback(*prompt_spec: Tuple[str, Action] | None = None*, *sql_id: str | None = None*, *response: str | None = None*, *feedback_content: str | None = None*)

Give negative feedback to the LLM

Parameters

- **prompt_spec** (*Tuple[str, select_ai.action.Action]*) – First element is the prompt; second is the corresponding action
- **sql_id** (*str*) – SQL identifier from V$MAPPED_SQL view
- **response** (*str*) – Expected SQL from LLM
- **feedback_content** (*str*) – Actual feedback in natural language

add_positive_feedback(*prompt_spec: Tuple[str, Action] | None = None*, *sql_id: str | None = None*)

Give positive feedback to the LLM

Parameters

- **prompt_spec** (*Tuple[str, select_ai.action.Action]*) – First element is the prompt; second is the corresponding action
- **sql_id** (*str*) – SQL identifier from V$MAPPED_SQL view

chat(*prompt: str*, *params: Mapping = None*, *stream: bool = False*, *chunk_size: int = 8192*, ***, *attributes: Mapping = None*) → str | Generator[str, None, None]

Chat with the LLM

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM request
- **attributes** (*Mapping*) – Profile attributes to override for this request
- **stream** (*bool*) – Return an iterator of response chunks
- **chunk_size** (*int*) – Number of characters to read per stream chunk

Returns

str

chat_session(*conversation: Conversation*, *delete: bool = False*)

Starts a new chat session for context-aware conversations

Parameters

- **conversation** (Conversation) – Conversation object to use for this chat session
- **delete** (*bool*) – Delete conversation after session ends

Returns

create(*replace: int | None = False*) → None

Create an AI Profile in the Database

Parameters
replace (*bool*) – Set True to replace else False

Returns
None

Raises
oracledb.DatabaseError

delete(*force=False*) → None

Deletes an AI profile from the database

Parameters
force (*bool*) – Ignores errors if AI profile does not exist.

Returns
None

Raises
oracledb.DatabaseError

delete_feedback(*prompt_spec: Tuple[str,* Action*] = None*, *sql_id: str | None = None*)

Delete feedback from the database

Parameters

- **prompt_spec** (*Tuple[str, select_ai.action.Action]*) – First element is the prompt; second is the corresponding action
- **sql_id** (*str*) – SQL identifier from V$MAPPED_SQL view

classmethod delete_profile(*profile_name: str*, *force: bool = False*)

Class method to delete an AI profile from the database

Parameters

- **profile_name** (*str*) – Name of the AI profile
- **force** (*bool*) – Ignores errors if AI profile does not exist.

Returns
None

Raises
oracledb.DatabaseError

disable() → None

Disable this AI profile in the database.

Returns
None

Raises
oracledb.DatabaseError

enable() → None

Enable this AI profile in the database.

Returns
None

Raises
oracledb.DatabaseError

explain_sql(*prompt: str*, *params: Mapping = None*, *stream: bool = False*, *chunk_size: int = 8192*, ***, *attributes: Mapping = None*) → str | Generator[str, None, None]

Explain the generated SQL

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM request
- **attributes** (*Mapping*) – Profile attributes to override for this request
- **stream** (*bool*) – Return an iterator of response chunks
- **chunk_size** (*int*) – Number of characters to read per stream chunk

Returns

str

classmethod fetch(*profile_name: str*, *owner: str | None = None*) → *Profile*

Create a proxy Profile object from fetched attributes saved in the database

Parameters

- **profile_name** (*str*) – The name of the AI profile
- **owner** (*str*) – Owner of a shared profile. Defaults to current schema.

Returns

select_ai.Profile

Raises

ProfileNotFoundError

generate(*prompt: str*, *action: Action | None = Action.RUNSQL*, *params: Mapping = None*, *stream: bool = False*, *chunk_size: int = 8192*, ***, *attributes: Mapping = None*) → DataFrame | str | Generator[str, None, None] | None

Perform AI translation using this profile

Parameters

- **prompt** (*str*) – Natural language prompt to translate
- **action** (select_ai.action.Action)
- **params** – Parameters to include in the LLM request. For e.g. conversation_id for context-aware chats
- **attributes** (*Mapping*) – Profile attributes to override for this request
- **stream** (*bool*) – Return an iterator of response chunks
- **chunk_size** (*int*) – Number of characters to read per stream chunk

Returns

Union[pandas.DataFrame, str]

generate_synthetic_data(*synthetic_data_attributes: SyntheticDataAttributes*) → None

Generate synthetic data for a single table, multiple tables or a full schema.

Parameters

synthetic_data_attributes (select_ai.SyntheticDataAttributes)

Returns

None

Raises
oracledb.DatabaseError

`get_attributes`() → *ProfileAttributes*

Get AI profile attributes from the Database

Returns
select_ai.ProfileAttributes

`grant_access`(*user_or_role_name: str*) → None

Grant a database user or role access to this AI profile.

Parameters
user_or_role_name (*str*) – Database user or role receiving access.

Returns
None

Raises
oracledb.DatabaseError

`classmethod list`(*profile_name_pattern: str = '.*'*, *owner: str | None = None*) → Generator[*Profile*, None, None]

List AI Profiles saved in the database.

Parameters

- **profile_name_pattern** (*str*) – Regular expressions can be used to specify a pattern. Function REGEXP_LIKE is used to perform the match. Default value is ".*" i.e. match all AI profiles.
- **owner** (*str*) – Owner of shared profiles. Defaults to current schema.

Returns
Iterator[Profile]

`narrate`(*prompt: str*, *params: Mapping = None*, *stream: bool = False*, *chunk_size: int = 8192*, ***, *attributes: Mapping = None*) → str | Generator[str, None, None]

Narrate the result of the SQL

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM request
- **attributes** (*Mapping*) – Profile attributes to override for this request
- **stream** (*bool*) – Return an iterator of response chunks
- **chunk_size** (*int*) – Number of characters to read per stream chunk

Returns
str

`property qualified_name: str | None`

Return the owner-qualified profile name when owner is known.

`revoke_access`(*user_or_role_name: str*) → None

Revoke a database user or role's access to this AI profile.

Parameters
user_or_role_name (*str*) – Database user or role losing access.

Returns
None

Raises
oracledb.DatabaseError

run_sql(*prompt: str*, *params: Mapping = None*, ***, *attributes: Mapping = None*) → DataFrame

Run the generate SQL statement and return a pandas Dataframe built using the result set

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM request
- **attributes** (*Mapping*) – Profile attributes to override for this request

Returns
pandas.DataFrame

set_attribute(*attribute_name: str*, *attribute_value: bool | str | int | float | Provider*)

Updates AI profile attribute on the Python object and also saves it in the database

Parameters

- **attribute_name** (*str*) – Name of the AI profile attribute
- **attribute_value** (*Union[bool, str, int, float, Provider]*) – Value of the profile attribute

Returns
None

set_attributes(*attributes: ProfileAttributes*)

Updates AI profile attributes on the Python object and also saves it in the database

Parameters
attributes (*select_ai.ProfileAttributes*) – Object specifying AI profile attributes

Returns
None

show_prompt(*prompt: str*, *params: Mapping = None*, *stream: bool = False*, *chunk_size: int = 8192*, ***, *attributes: Mapping = None*) → str | Generator[str, None, None]

Show the prompt sent to LLM

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM request
- **attributes** (*Mapping*) – Profile attributes to override for this request
- **stream** (*bool*) – Return an iterator of response chunks
- **chunk_size** (*int*) – Number of characters to read per stream chunk

Returns
str

show_sql(*prompt: str*, *params: Mapping = None*, *stream: bool = False*, *chunk_size: int = 8192*, ***, *attributes: Mapping = None*) → str | Generator[str, None, None]

Show the generated SQL

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM request
- **attributes** (*Mapping*) – Profile attributes to override for this request
- **stream** (*bool*) – Return an iterator of response chunks
- **chunk_size** (*int*) – Number of characters to read per stream chunk

Returns

str

summarize(*content: str = None*, *prompt: str = None*, *location_uri: str = None*, *credential_name: str = None*, *params:* SummaryParams *= None*) → str

Generate summary

Parameters

- **prompt** (*str*) – Natural language prompt to guide the summary generation
- **content** (*str*) – Specifies the text you want to summarize
- **location_uri** (*str*) – Provides the URI where the text is stored or the path to a local file stored
- **credential_name** (*str*) – Identifies the credential object used to authenticate with the object store
- **params** (select_ai.summary.SummaryParams) – Parameters to include in the LLM request

translate(*text: str*, *source_language: str | None = None*, *target_language: str | None = None*) → str | None

Translate text using the supplied languages or the profile defaults.

Parameters

- **text** (*str*) – Text to translate
- **source_language** (*str*) – Source language. When omitted, the profile value is used; if the profile does not define one, the provider detects the source language.
- **target_language** (*str*) – Target language. When omitted, the profile value is used.

Returns

str

8.2.4 Create Profile

The following example creates an OCI Gen AI profile that can generate SQL over objects in the SH schema.

```
import os
from pprint import pformat

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
provider = select_ai.OCIGenAIProvider(
    region="us-chicago-1", oci_apiformat="GENERIC", model="openai.gpt-4.1"
)
profile_attributes = select_ai.ProfileAttributes(
    credential_name="my_oci_ai_profile_key",
    object_list=[{"owner": "SH"}],
    provider=provider,
)
profile = select_ai.Profile(
    profile_name="oci_ai_profile",
    attributes=profile_attributes,
    description="MY OCI AI Profile",
    replace=True,
)
print("Created profile ", profile.profile_name)
profile_attributes = profile.get_attributes()
print(
    "Profile attributes are: ",
    pformat(profile_attributes.dict(exclude_null=False)),
)
```

output:

```
Created profile  oci_ai_profile
Profile attributes are:  {'annotations': None,
 'case_sensitive_values': None,
 'comments': None,
 'constraints': None,
 'conversation': None,
 'credential_name': 'my_oci_ai_profile_key',
 'enable_source_offsets': None,
 'enable_sources': None,
 'enforce_object_list': None,
 'max_tokens': '1024',
 'object_list': '[{"owner":"SH"}]',
 'object_list_mode': None,
 'provider': OCIGenAIProvider(embedding_model=None,
                              model=None,
                              provider_name='oci',
                              provider_endpoint=None,
```

(continues on next page)

(continued from previous page)

```
                              region='us-chicago-1',
                              oci_apiformat='GENERIC',
                              oci_compartment_id=None,
                              oci_endpoint_id=None,
                              oci_runtimetype=None),
 'seed': None,
 'stop_tokens': None,
 'streaming': None,
 'temperature': None,
 'vector_index_name': None}
```

8.2.5 Reuse Profile

After a profile has been created, instantiate `Profile` with only the profile name to reuse the database profile:

```
profile = select_ai.Profile(profile_name="oci_ai_profile")
sql = profile.show_sql(prompt="How many promotions?")
```

Use `Profile.fetch(...)` when you want to create a proxy object from a saved database profile and raise an error if the profile does not exist:

```
profile = select_ai.Profile.fetch("oci_ai_profile")
```

8.2.6 Enable and disable Profile

Use `Profile.disable()` to make an existing profile unavailable without deleting it. Call `Profile.enable()` to make it available again. These methods change the database profile status and return `None`.

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
profile_name = os.getenv("SELECT_AI_PROFILE_NAME", "oci_ai_profile")

select_ai.connect(user=user, password=password, dsn=dsn)
profile = select_ai.Profile(profile_name=profile_name)

profile.disable()
try:
    print("Disabled profile:", profile.profile_name)
finally:
    profile.enable()
    print("Enabled profile:", profile.profile_name)
```

The sample uses `SELECT_AI_PROFILE_NAME` when set, otherwise it operates on `oci_ai_profile`. It always re-enables the profile during cleanup:

output:

```
Disabled profile: oci_ai_profile
Enabled profile: oci_ai_profile
```

8.2.7 Update Profile

Use `set_attribute(...)` to update one profile attribute or `set_attributes(...)` to update several attributes. Updates are saved to the database profile.

```
profile = select_ai.Profile(profile_name="oci_ai_profile")
profile.set_attribute("temperature", 0.1)

profile.set_attributes(
    select_ai.ProfileAttributes(
        max_tokens=2048,
        enforce_object_list=True,
    )
)
```

8.2.8 Delete Profile

Use `delete(...)` or `Profile.delete_profile(...)` to remove a profile from the database. Pass `force=True` when cleanup should succeed even if the profile does not exist.

```
profile = select_ai.Profile(profile_name="oci_ai_profile")
profile.delete(force=True)

select_ai.Profile.delete_profile("oci_ai_profile", force=True)
```

8.2.9 Narrate

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
profile = select_ai.Profile(
    profile_name="oci_ai_profile",
)
narration = profile.narrate(prompt="How many promotions?")
print(narration)
```

output:

```
There are 503 promotions in the database.
```

8.2.10 Show SQL

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
profile = select_ai.Profile(profile_name="oci_ai_profile")
sql = profile.show_sql(prompt="How many promotions ?")
print(sql)
```

output:

```
SELECT
COUNT("p"."PROMO_ID") AS "Number of Promotions"
FROM "SH"."PROMOTIONS" "p"
```

8.2.11 Explain SQL

Use `explain_sql(...)` to generate SQL and return a natural language explanation without executing the SQL.

```
profile = select_ai.Profile(profile_name="oci_ai_profile")
explanation = profile.explain_sql(prompt="How many promotions?")
print(explanation)
```

8.2.12 Show Prompt

Use `show_prompt(...)` to inspect the prompt that Select AI sends to the model. This is useful when tuning profile attributes, object lists, comments, constraints, and provider settings.

```
profile = select_ai.Profile(profile_name="oci_ai_profile")
prompt = profile.show_prompt(prompt="How many promotions?")
print(prompt)
```

8.2.13 Run SQL

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
profile = select_ai.Profile(profile_name="oci_ai_profile")
df = profile.run_sql(prompt="How many promotions ?")
print(df.columns)
print(df)
```

output:

```
Index(['Number of Promotions'], dtype='object')
    Number of Promotions
0                    503
```

8.2.14 Chat

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
profile = select_ai.Profile(profile_name="oci_ai_profile")
response = profile.chat(prompt="What is OCI ?")
print(response)
```

output:

```
OCI stands for Oracle Cloud Infrastructure. It is a comprehensive cloud computing␣
↪platform provided by Oracle Corporation that offers a wide range of services for␣
↪computing, storage, networking, database, and more.
...
...
OCI competes with other major cloud providers, including Amazon Web Services (AWS),␣
↪Microsoft Azure, Google Cloud Platform (GCP), and IBM Cloud.
```

8.2.15 Streaming chat

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
profile = select_ai.Profile(profile_name="oci_ai_profile")

for chunk in profile.chat(
    prompt="What is OCI ?", stream=True, chunk_size=4096
):
    print(chunk, end="")
print()
```

`stream=True` lets callers consume generated CLOB responses chunk by chunk, reducing memory pressure and making it easier to progressively forward output to files, services, or user interfaces. Streaming text APIs return an iterator of `str` chunks. The `chunk_size` parameter controls the number of CLOB characters read per chunk; it is not a byte count.

Streaming is supported by `generate()`, `chat()`, `narrate()`, `explain_sql()`, `show_sql()`, and `show_prompt()`. It is not supported for `run_sql()`, which returns a `pandas.DataFrame`.

The SDK calls the database function before reading the returned CLOB in chunks. This is CLOB consumption, rather than a guarantee that model tokens arrive while generation is in progress. The first chunk is available only after the database function returns. A partially consumed iterator keeps its cursor and connection open. Close it if you stop early, and consume session streams before leaving `chat_session()`:

```
from contextlib import closing

with closing(profile.chat("Explain connection pooling.", stream=True)) as chunks:
    for chunk in chunks:
        print(chunk, end="")
        # Breaking here also closes the iterator on context exit.
```

8.2.16 Request-level profile attribute overrides

The `attributes` keyword accepts a mapping of profile attributes to apply to one request. These values override the corresponding saved profile attributes for that request only; they do not update the profile in the database. The mapping must contain JSON-serializable values.

Request-level attributes are available on the following synchronous APIs:

- `Profile.generate()`, `chat()`, and `narrate()`.
- SQL helpers: `explain_sql()`, `run_sql()`, `show_sql()`, and `show_prompt()`.
- `Session.chat()`, `narrate()`, `explain_sql()`, `run_sql()`, `show_sql()`, and `show_prompt()`.
- Streaming calls to the text-producing methods, by combining `stream=True` with `attributes=...`. `run_sql()` does not support streaming.

The mapping can include attributes such as `additional_instructions`, the integer `seed`, and `source_language` or `target_language`. Use `ProfileAttributes` for profile-wide defaults; use this mapping when an override should

apply only to the current request.

For `translate()`, pass language values through its named `source_language` and `target_language` arguments. The `attributes` mapping described here applies to the generate/action APIs.

```
request_attributes = {
    "additional_instructions": "Answer in one sentence.",
    "seed": 42,
    "source_language": "en",
    "target_language": "de",
}

response = profile.chat(
    prompt="What is Oracle Cloud Infrastructure?",
    attributes=request_attributes,
)
```

The same mapping can be used with streaming and session requests:

```
for chunk in profile.chat(
    prompt="Summarize Oracle Cloud Infrastructure.",
    stream=True,
    attributes=request_attributes,
):
    print(chunk, end="")

with profile.chat_session(conversation) as session:
    response = session.chat(
        prompt="Give one more detail.",
        attributes=request_attributes,
    )
```

The following sample uses `show_prompt()` to verify that the request-only `additional_instructions` value reached the generated prompt:

```

import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
profile_name = os.getenv("SELECT_AI_PROFILE_NAME", "oci_ai_profile")

select_ai.connect(user=user, password=password, dsn=dsn)
profile = select_ai.Profile(profile_name=profile_name)

instruction = "Include the exact marker PYSAI_REQUEST_ATTRIBUTES_SAMPLE."
request_attributes = {
    "additional_instructions": instruction,
    "seed": 42,
    "source_language": "en",
    "target_language": "de",
```

(continues on next page)

(continued from previous page)

```
}

prompt = profile.show_prompt(
    prompt="Tell me about Oracle Cloud Infrastructure.",
    attributes=request_attributes,
)
print("Request attributes applied:", instruction in prompt)
```

output:

```
Request attributes applied: True
```

8.2.17 Summarize

Summarize inline content

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

content = """
A gas cloud in our galaxy, Sagittarius B2, contains enough alcohol to brew 400
trillion pints of beer, and some stars are so cool that you could touch them
without being burned. Meanwhile, on the exoplanet 55 Cancri e, a form of
"hot ice" exists where high pressure prevents water from becoming gas even at
high temperatures. Additionally, some ancient stars found in the Milky Way's
halo are much older than the Sun, providing clues about the early universe and
its composition
"""

select_ai.connect(user=user, password=password, dsn=dsn)
profile = select_ai.Profile(profile_name="oci_ai_profile")
summary = profile.summarize(content=content)
print(summary)
```

output:

```
A gas cloud in the Sagittarius B2 galaxy contains a large amount of alcohol,
while some stars are cool enough to touch without being burned. The exoplanet
55 Cancri e has a unique form of "hot ice" where water remains solid despite
high temperatures due to high pressure. Ancient stars in the Milky Way's halo
are older than the Sun, providing insights into the early universe and its composition,
offering clues about the universe's formation and evolution.
```

Summarize content accessible via a URL

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
profile = select_ai.Profile(profile_name="oci_ai_profile")
summary = profile.summarize(
    location_uri="https://en.wikipedia.org/wiki/Astronomy"
)
print(summary)
```

output:

```
Astronomy is a natural science that studies celestial objects and phenomena,
using mathematics, physics, and chemistry to explain their origin and evolution.
The field has a long history, with early civilizations making methodical
observations of the night sky, and has since split into observational and
theoretical branches. Observational astronomy focuses on acquiring data
from observations, while theoretical astronomy develops computer or
analytical models to describe astronomical objects and phenomena. The study
of astronomy has led to numerous discoveries, including the existence of
galaxies, the expansion of the universe, and the detection of gravitational
waves. Astronomers use various methods, such as radio, infrared, optical,
ultraviolet, X-ray, and gamma-ray astronomy, to study objects and events in
the universe. The field has also led to the development of new technologies and
has inspired new areas of research, such as astrobiology and the search for
extraterrestrial life. Overall, astronomy is a dynamic and constantly evolving
field that seeks to understand the universe and its many mysteries.
```

8.2.18 Translate

`Profile.translate()` accepts optional `source_language` and `target_language` arguments. Per-call values take precedence over the profile's `source_language` and `target_language` defaults.

If `source_language` is omitted for both the call and the profile, source language detection is delegated to the translation provider. If `target_language` is omitted, the profile's default is used; a target language must be supplied in one of those two places.

Configure profile-level defaults with `ProfileAttributes` when creating or updating a profile:

```
language_defaults = select_ai.ProfileAttributes(
    source_language="en",
    target_language="de",
)

profile.set_attributes(language_defaults)
print(profile.translate(text="Thank you"))
```

When only the target is supplied at the call site, the provider can detect the source language:

```
print(profile.translate(text="Thank you", target_language="de"))
```

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
profile_name = os.getenv("SELECT_AI_PROFILE_NAME", "oci_ai_profile")

select_ai.connect(user=user, password=password, dsn=dsn)
profile = select_ai.Profile(profile_name=profile_name)
response = profile.translate(text="Thank you", target_language="de")
print(response)
```

output:

```
Danke
```

8.2.19 Share a profile

While connected as the profile owner, grant access to an existing profile:

```
profile = select_ai.Profile.fetch("OCI_AI_PROFILE")
profile.grant_access("APP_USER")
```

The recipient can fetch the profile with `owner="APP_OWNER"` in a separate connection. When access should end, run `profile.revoke_access("APP_USER")` as the owner. See *Sharing and ownership* for the complete owner and recipient sequence.

8.2.20 Give feedback on generated SQL

Use feedback after reviewing the SQL produced for a prompt. Identify the request with a `(prompt, Action)` tuple or a `sql_id` from `V$MAPPED_SQL`. The example below assumes an open connection and an existing profile over the `SH` schema:

```
profile = select_ai.Profile.fetch("OCI_AI_PROFILE")
prompt = "How many promotions?"
sql = profile.show_sql(prompt=prompt)
print(sql)
```

After inspecting the SQL, record positive feedback if it answers the question:

```
profile.add_positive_feedback(
    prompt_spec=(prompt, select_ai.Action.SHOWSQL)
)
```

For an incorrect answer, use `add_negative_feedback()` with the same request identifier, `response` containing the expected SQL, and `feedback_content` explaining the correction. Remove obsolete feedback with `delete_feedback()` and the same request identifier.

8.2.21 List profiles

Use `Profile.list()` to discover saved profiles in the current schema. Call it on the class; you do not need to create an empty profile first. Each returned object includes `profile_name`, `owner`, and the owner-qualified `qualified_name`.

Connect and list your profiles

Set `SELECT_AI_USER`, `SELECT_AI_PASSWORD`, and `SELECT_AI_DB_CONNECT_STRING` before running this complete script. It prints one qualified name per profile and closes the connection afterward:

```
import os

import select_ai

select_ai.connect(
    user=os.environ["SELECT_AI_USER"],
    password=os.environ["SELECT_AI_PASSWORD"],
    dsn=os.environ["SELECT_AI_DB_CONNECT_STRING"],
)
try:
    for profile in select_ai.Profile.list():
        print(profile.qualified_name)
finally:
    select_ai.disconnect()
```

For a schema named `APP_OWNER` containing two profiles, the output could be:

```
APP_OWNER.OCI_AI_PROFILE
APP_OWNER.OCI_VECTOR_AI_PROFILE
```

Your results depend on the profiles visible in your schema; result order is not guaranteed. An empty result means no profiles match the requested scope.

Filter by name

With a connection open, use `profile_name_pattern` to match names with a regular expression. This example lists names starting with `OCI_`:

```
for profile in select_ai.Profile.list(profile_name_pattern="^OCI_"):
    print(profile.qualified_name)
```

Find and reuse a shared profile

Pass `owner` to list profiles in another schema. The connected user must have access to those profiles; specifying an owner does not grant access. While connected as the recipient, list the profiles shared by `APP_OWNER`:

```
for profile in select_ai.Profile.list(owner="APP_OWNER"):
    print(profile.qualified_name)
```

Once you know the profile name, fetch that profile with the same owner and use the returned object directly:

```
profile = select_ai.Profile.fetch(
    "OCI_AI_PROFILE", owner="APP_OWNER"
)
print(profile.qualified_name)  # APP_OWNER.OCI_AI_PROFILE
response = profile.chat(prompt="What is retrieval augmented generation?")
print(response)
```

See *Sharing and ownership* for the owner's grant and revoke steps.

CHAPTER

NINE

ASYNC PROFILE

An `AsyncProfile` object can be created with `select_ai.AsyncProfile()`. `AsyncProfile` supports concurrent programming with asyncio. Unless explicitly noted as synchronous, the `AsyncProfile` methods should be used with `await`.

Use `AsyncProfile` in applications that already use `select_ai.async_connect()` or `select_ai.create_pool_async()`. The async profile object uses the same database profile objects as `Profile`; only the Python API style changes.

Because `AsyncProfile` initializes itself from the database, create or fetch instances with `await`:

```
async_profile = await select_ai.AsyncProfile(
    profile_name="async_oci_ai_profile",
)
```

Before creating an async profile, make sure the database user has the required privileges, a credential for the AI provider, network access to the provider endpoint, and access to the database objects included in the profile. See *Privileges*, *Credential*, *Provider*, and *ProfileAttributes*.

9.1 Async profile lifecycle

The usual async profile lifecycle is:

- Create or reuse an async database connection or async pool.
- Create a provider object.
- Create `ProfileAttributes` with the provider, credential name, and object list.
- Create the profile with `await select_ai.AsyncProfile(...)`.
- Reuse the profile later by name.
- Enable or disable the profile without deleting it.
- Update profile attributes when provider settings or object scope changes.
- Delete profiles that are no longer needed.

`replace=True` recreates a profile when a profile with the same name already exists. `merge=True` fetches the existing profile and updates it with the non-null attributes passed by the caller.

9.2 Async profile actions

Table 1: Common async profile actions

Method	Description
`show_sql()`	Generates SQL for a natural language prompt without executing it.
`run_sql()`	Generates SQL, executes it, and returns a `pandas.DataFrame`.
`narrate()`	Generates SQL, executes it, and returns a natural language answer.
`explain_sql()`	Explains the generated SQL for a prompt.
`show_prompt()`	Shows the prompt sent to the model.
`chat()`	Sends a general chat prompt to the model.
`summarize()`	Summarizes inline content or content referenced by a URI.
`translate()`	Translates text from one language to another.

9.2.1 `AsyncProfile` API

class select_ai.**AsyncProfile**(**args*, ***kwargs*)

AsyncProfile defines methods to interact with the underlying AI Provider asynchronously.

async add_negative_feedback(*prompt_spec: Tuple[str,* Action*]* | *None* = *None*, *sql_id: str* | *None* = *None*, *response: str* | *None* = *None*, *feedback_content: str* | *None* = *None*)

Give negative feedback to the LLM

Parameters

- **prompt_spec** (*Tuple[str, select_ai.action.Action]*) – First element is the prompt; second is the corresponding action
- **sql_id** (*str*) – SQL identifier from V$MAPPED_SQL view
- **response** (*str*) – Expected SQL from LLM
- **feedback_content** (*str*) – Actual feedback in natural language

async add_positive_feedback(*prompt_spec: Tuple[str,* Action*]* | *None* = *None*, *sql_id: str* | *None* = *None*)

Give positive feedback to the LLM

Parameters

- **prompt_spec** (*Tuple[str, select_ai.action.Action]*) – First element is the prompt; second is the corresponding action
- **sql_id** (*str*) – SQL identifier from V$MAPPED_SQL view

async chat(*prompt*, *params: Mapping* = *None*, *stream: bool* = *False*, *chunk_size: int* = *8192*, ***, *attributes: Mapping* = *None*) → str | AsyncGenerator[str, None]

Asynchronously chat with the LLM

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM request
- **attributes** (*Mapping*) – Profile attributes to override for this request
- **stream** (*bool*) – Return an async iterator of response chunks
- **chunk_size** (*int*) – Number of characters to read per stream chunk

Returns
str

chat_session(*conversation:* AsyncConversation, *delete: bool = False*)

Starts a new chat session for context-aware conversations

Parameters

- **conversation** (AsyncConversation) – Conversation object to use for this chat session
- **delete** (*bool*) – Delete conversation after session ends

async create(*replace: int | None = False*) → None

Asynchronously create an AI Profile in the Database

Parameters
replace (*bool*) – Set True to replace else False

Returns
None

Raises
oracledb.DatabaseError

async delete(*force=False*) → None

Asynchronously deletes an AI profile from the database

Parameters
force (*bool*) – Ignores errors if AI profile does not exist.

Returns
None

Raises
oracledb.DatabaseError

async delete_feedback(*prompt_spec: Tuple[str,* Action*] = None*, *sql_id: str | None = None*)

Delete feedback from the database

Parameters

- **prompt_spec** (*Tuple[str, select_ai.action.Action]*) – First element is the prompt; second is the corresponding action
- **sql_id** (*str*) – SQL identifier from V$MAPPED_SQL view

async classmethod delete_profile(*profile_name: str*, *force: bool = False*)

Asynchronously deletes an AI profile from the database

Parameters

- **profile_name** (*str*) – Name of the AI profile
- **force** (*bool*) – Ignores errors if AI profile does not exist.

Returns
None

Raises
oracledb.DatabaseError

async disable() → None

Asynchronously disable this AI profile in the database.

Returns
None

Raises
oracledb.DatabaseError

async enable() → None

Asynchronously enable this AI profile in the database.

Returns
None

Raises
oracledb.DatabaseError

async explain_sql(*prompt: str*, *params: Mapping = None*, *stream: bool = False*, *chunk_size: int = 8192*, *, *attributes: Mapping = None*)

Explain the generated SQL

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM request
- **attributes** (*Mapping*) – Profile attributes to override for this request
- **stream** (*bool*) – Return an async iterator of response chunks
- **chunk_size** (*int*) – Number of characters to read per stream chunk

Returns
str

async classmethod fetch(*profile_name: str*, *owner: str | None = None*) → *AsyncProfile*

Asynchronously create an AI Profile object from attributes saved in the database

Parameters

- **profile_name** (*str*) – Name of the AI profile.
- **owner** (*str*) – Owner of a shared profile. Defaults to current schema.

Returns
select_ai.Profile

Raises
ProfileNotFoundError

async generate(*prompt: str*, *action=Action.SHOWSQL*, *params: Mapping = None*, *stream: bool = False*, *chunk_size: int = 8192*, *, *attributes: Mapping = None*) → DataFrame | str | AsyncGenerator[str, None] | None

Asynchronously perform AI translation using this profile

Parameters

- **prompt** (*str*) – Natural language prompt to translate
- **action** (select_ai.action.Action)
- **params** – Parameters to include in the LLM request. For e.g. conversation_id for context-aware chats
- **attributes** (*Mapping*) – Profile attributes to override for this request

- **stream** (*bool*) – Return an async iterator of response chunks
- **chunk_size** (*int*) – Number of characters to read per stream chunk

Returns
Union[pandas.DataFrame, str]

async generate_synthetic_data(*synthetic_data_attributes:* SyntheticDataAttributes) → None

Generate synthetic data for a single table, multiple tables or a full schema.

Parameters
synthetic_data_attributes (select_ai.SyntheticDataAttributes)

Returns
None

Raises
oracledb.DatabaseError

async get_attributes() → *ProfileAttributes*

Asynchronously gets AI profile attributes from the Database

Returns
select_ai.ProfileAttributes

Raises
ProfileNotFoundError

async grant_access(*user_or_role_name: str*) → None

Asynchronously grant a user or role access to this AI profile.

classmethod list(*profile_name_pattern: str = '.*'*, *owner: str | None = None*) → AsyncGenerator[*AsyncProfile*, None]

Asynchronously list AI Profiles saved in the database.

Parameters

- **profile_name_pattern** (*str*) – Regular expressions can be used to specify a pattern. Function REGEXP_LIKE is used to perform the match. Default value is ".*" i.e. match all AI profiles.
- **owner** (*str*) – Owner of shared profiles. Defaults to current schema.

Returns
Iterator[Profile]

async narrate(*prompt*, *params: Mapping = None*, *stream: bool = False*, *chunk_size: int = 8192*, ***, *attributes: Mapping = None*) → str | AsyncGenerator[str, None]

Narrate the result of the SQL

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM request
- **attributes** (*Mapping*) – Profile attributes to override for this request
- **stream** (*bool*) – Return an async iterator of response chunks
- **chunk_size** (*int*) – Number of characters to read per stream chunk

Returns
str

property qualified_name: str | None

Return the owner-qualified profile name when owner is known.

async revoke_access(*user_or_role_name: str*) → None

Asynchronously revoke a user or role's profile access.

async run_pipeline(*prompt_specifications: List[Tuple[str,* Action*]]*, *continue_on_error: bool = False*, ***, *attributes: Mapping = None*) → List[object]

Submit prompt/action pairs as a database pipeline

Parameters

- **prompt_specifications** (*List[Tuple[str, select_ai.action.Action]]*) – List of 2-element tuples. First element is the prompt and second is the corresponding action
- **continue_on_error** (*bool*) – True to continue on error else False
- **attributes** (*Mapping*) – Profile attributes to override for every request in the pipeline

Returns

Ordered list of strings for successful operations and driver error objects for failures when continue_on_error=True. RUNSQL results remain JSON strings; they are not converted to DataFrames.

Operations execute sequentially. Round-trip reduction requires a supporting database version; reading returned CLOBs adds calls. With continue_on_error=False, an operation failure raises an exception.

async run_sql(*prompt*, *params: Mapping = None*, ***, *attributes: Mapping = None*) → DataFrame

Explain the generated SQL

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM request
- **attributes** (*Mapping*) – Profile attributes to override for this request

Returns

pandas.DataFrame

async set_attribute(*attribute_name: str*, *attribute_value: bool | str | int | float |* Provider)

Updates AI profile attribute on the Python object and also saves it in the database

Parameters

- **attribute_name** (*str*) – Name of the AI profile attribute
- **attribute_value** (*Union[bool, str, int, float]*) – Value of the profile attribute

Returns

None

async set_attributes(*attributes:* ProfileAttributes)

Updates AI profile attributes on the Python object and also saves it in the database

Parameters

attributes (ProfileAttributes) – Object specifying AI profile attributes

Returns

None

async **show_prompt**(*prompt: str*, *params: Mapping = None*, *stream: bool = False*, *chunk_size: int = 8192*, ***, *attributes: Mapping = None*)

Show the prompt sent to LLM

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM request
- **attributes** (*Mapping*) – Profile attributes to override for this request
- **stream** (*bool*) – Return an async iterator of response chunks
- **chunk_size** (*int*) – Number of characters to read per stream chunk

Returns

str

async **show_sql**(*prompt*, *params: Mapping = None*, *stream: bool = False*, *chunk_size: int = 8192*, ***, *attributes: Mapping = None*)

Show the generated SQL

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM request
- **attributes** (*Mapping*) – Profile attributes to override for this request
- **stream** (*bool*) – Return an async iterator of response chunks
- **chunk_size** (*int*) – Number of characters to read per stream chunk

Returns

str

async **summarize**(*content: str = None*, *prompt: str = None*, *location_uri: str = None*, *credential_name: str = None*, *params:* SummaryParams *= None*) → str

Generate summary

Parameters

- **prompt** (*str*) – Natural language prompt to guide the summary generation
- **content** (*str*) – Specifies the text you want to summarize
- **location_uri** (*str*) – Provides the URI where the text is stored or the path to a local file stored
- **credential_name** (*str*) – Identifies the credential object used to authenticate with the object store
- **params** (select_ai.summary.SummaryParams) – Parameters to include in the LLM request

async **translate**(*text: str*, *source_language: str | None = None*, *target_language: str | None = None*) → str | None

Translate text using the supplied languages or the profile defaults.

Parameters

- **text** (*str*) – Text to translate

- **source_language** (*str*) – Source language. When omitted, the profile value is used; if the profile does not define one, the provider detects the source language.
- **target_language** (*str*) – Target language. When omitted, the profile value is used.

Returns

str

9.2.2 Async Profile creation

The following example creates an OCI Gen AI profile that can generate SQL over objects in the SH schema.

```
import asyncio
import os
from pprint import pformat

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


# This example shows how to asynchronously generate SQLs nad run SQLs
async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    provider = select_ai.OCIGenAIProvider(
        region="us-chicago-1", oci_apiformat="GENERIC"
    )
    profile_attributes = select_ai.ProfileAttributes(
        credential_name="my_oci_ai_profile_key",
        object_list=[{"owner": "SH"}],
        provider=provider,
    )
    async_profile = await select_ai.AsyncProfile(
        profile_name="async_oci_ai_profile",
        attributes=profile_attributes,
        description="MY OCI AI Profile",
        replace=True,
    )
    print("Created async profile ", async_profile.profile_name)
    profile_attributes = await async_profile.get_attributes()
    print(
        "Profile attributes: ",
        pformat(profile_attributes.dict(exclude_null=False)),
    )


asyncio.run(main())
```

output:

```
Created async profile  async_oci_ai_profile
Profile attributes:  {'annotations': None,
 'case_sensitive_values': None,
 'comments': None,
 'constraints': None,
 'conversation': None,
 'credential_name': 'my_oci_ai_profile_key',
 'enable_source_offsets': None,
 'enable_sources': None,
 'enforce_object_list': None,
```

(continues on next page)

(continued from previous page)

```
'max_tokens': '1024',
'object_list': '[{"owner":"SH"}]',
'object_list_mode': None,
'provider': OCIGenAIProvider(embedding_model=None,
                             model=None,
                             provider_name='oci',
                             provider_endpoint=None,
                             region='us-chicago-1',
                             oci_apiformat='GENERIC',
                             oci_compartment_id=None,
                             oci_endpoint_id=None,
                             oci_runtimetype=None),
'seed': None,
'stop_tokens': None,
'streaming': None,
'temperature': None,
'vector_index_name': None}
```

9.2.3 Reuse Async Profile

After a profile has been created, instantiate `AsyncProfile` with only the profile name to reuse the database profile:

```
async_profile = await select_ai.AsyncProfile(
    profile_name="async_oci_ai_profile",
)
sql = await async_profile.show_sql(prompt="How many promotions?")
```

Use `AsyncProfile.fetch(...)` when you want to create a proxy object from a saved database profile and raise an error if the profile does not exist:

```
async_profile = await select_ai.AsyncProfile.fetch(
    "async_oci_ai_profile"
)
```

9.2.4 Enable and disable AsyncProfile

Use `await AsyncProfile.disable()` to make an existing profile unavailable without deleting it. Call `await AsyncProfile.enable()` to make it available again. These methods change the database profile status and return `None`.

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
profile_name = os.getenv("SELECT_AI_PROFILE_NAME", "oci_ai_profile")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    profile = await select_ai.AsyncProfile(profile_name=profile_name)

    await profile.disable()
    try:
        print("Disabled profile:", profile.profile_name)
    finally:
        await profile.enable()
        print("Enabled profile:", profile.profile_name)


asyncio.run(main())
```

The sample uses `SELECT_AI_PROFILE_NAME` when set, otherwise it operates on `oci_ai_profile`. It always re-enables the profile during cleanup:

output:

```
Disabled profile: oci_ai_profile
Enabled profile: oci_ai_profile
```

9.2.5 Update Async Profile

Use `set_attribute(...)` to update one profile attribute or `set_attributes(...)` to update several attributes. Updates are saved to the database profile.

```
async_profile = await select_ai.AsyncProfile(
    profile_name="async_oci_ai_profile",
)
await async_profile.set_attribute("temperature", 0.1)

await async_profile.set_attributes(
    select_ai.ProfileAttributes(
        max_tokens=2048,
        enforce_object_list=True,
    )
)
```

9.2.6 Delete Async Profile

Use `delete(...)` or `AsyncProfile.delete_profile(...)` to remove a profile from the database. Pass `force=True` when cleanup should succeed even if the profile does not exist.

```
async_profile = await select_ai.AsyncProfile(
    profile_name="async_oci_ai_profile",
)
await async_profile.delete(force=True)

await select_ai.AsyncProfile.delete_profile(
    "async_oci_ai_profile",
    force=True,
)
```

9.2.7 Async explain SQL

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_profile = await select_ai.AsyncProfile(
        profile_name="async_oci_ai_profile",
    )
    response = await async_profile.explain_sql("How many promotions ?")
    print(response)


asyncio.run(main())
```

output:

```
To answer the question "How many promotions", we need to write a SQL query that counts␣
↪the number of rows in the "PROMOTIONS" table. Here is the query:

```sql
SELECT
  COUNT("p"."PROMO_ID") AS "Number of Promotions"
FROM
  "SH"."PROMOTIONS" "p";
```

Explanation:

* We use the `COUNT` function to count the number of rows in the table.
* We use the table alias `"p"` to refer to the `"PROMOTIONS"` table.
* We enclose the table name and column name in double quotes to make them case-sensitive.
* We use the `AS` keyword to give an alias to the count column, making it easier to read.

This query will return the total number of promotions in the `"PROMOTIONS"` table.
```

9.2.8 Async run SQL

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


# This example shows how to asynchronously run sql
async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_profile = await select_ai.AsyncProfile(
        profile_name="async_oci_ai_profile",
    )
    # run_sql returns a pandas df
    df = await async_profile.run_sql("How many promotions?")
    print(df)


asyncio.run(main())
```

output:

```
   PROMOTION_COUNT
0              503
```

9.2.9 Async show SQL

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_profile = await select_ai.AsyncProfile(
        profile_name="async_oci_ai_profile",
    )
    response = await async_profile.show_sql("How many promotions?")
    print(response)


asyncio.run(main())
```

output:

```
SELECT COUNT("p"."PROMO_ID") AS "PROMOTION_COUNT" FROM "SH"."PROMOTIONS" "p"
```

9.2.10 Async show prompt

Use `show_prompt(...)` to inspect the prompt that Select AI sends to the model. This is useful when tuning profile attributes, object lists, comments, constraints, and provider settings.

```
async_profile = await select_ai.AsyncProfile(
    profile_name="async_oci_ai_profile",
)
prompt = await async_profile.show_prompt(prompt="How many promotions?")
print(prompt)
```

9.2.11 Async concurrent SQL

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    select_ai.create_pool_async(
        user=user, password=password, dsn=dsn,
        min_size=1, max_size=3, increment=1,
    )
    tasks = []
    try:
        async_profile = await select_ai.AsyncProfile(
            profile_name="async_oci_ai_profile",
        )
        # Each operation acquires its own connection from the pool.
        tasks = [
            asyncio.create_task(async_profile.show_sql("How many customers?")),
            asyncio.create_task(async_profile.run_sql("How many promotions?")),
            asyncio.create_task(async_profile.explain_sql("How many promotions?")),
        ]
        for task in asyncio.as_completed(tasks):
            print(await task)
    finally:
        # Finish cancellation before closing connections, including on errors.
        for task in tasks:
            if not task.done():
                task.cancel()
        await asyncio.gather(*tasks, return_exceptions=True)
        await select_ai.async_disconnect()


asyncio.run(main())
```

output:

```
SELECT COUNT("c"."CUST_ID") AS "customer_count" FROM "SH"."CUSTOMERS" "c"

To answer the question "How many promotions", we need to write a SQL query that counts␣
↪the number of rows in the "PROMOTIONS" table. Here is the query:

```sql
SELECT
  COUNT("p"."PROMO_ID") AS "number_of_promotions"
FROM
  "SH"."PROMOTIONS" "p";
```
```

(continues on next page)

(continued from previous page)

```
Explanation:

* We use the `COUNT` function to count the number of rows in the table.
* We use the table alias `"p"` to refer to the `"PROMOTIONS"` table.
* We specify the schema name `"SH"` to ensure that we are accessing the correct table.
* We enclose the table name, schema name, and column name in double quotes to make them␣
↪case-sensitive.
* The `AS` keyword is used to give an alias to the calculated column, in this case, `
↪"number_of_promotions"`.

This query will return the total number of promotions in the `"PROMOTIONS"` table.

   PROMOTION_COUNT
0              503
```

9.2.12 Async chat

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_profile = await select_ai.AsyncProfile(
        profile_name="async_oci_ai_profile"
    )

    # Asynchronously send multiple chat prompts
    chat_tasks = [
        async_profile.chat(prompt="What is OCI ?"),
        async_profile.chat(prompt="What is OML4PY?"),
        async_profile.chat(prompt="What is Autonomous Database ?"),
    ]
    for chat_task in asyncio.as_completed(chat_tasks):
        result = await chat_task
        print(result)


asyncio.run(main())
```

output:

```
OCI stands for several things depending on the context:

1. **Oracle Cloud Infrastructure (OCI)**: This is a cloud computing service offered by␣
↪Oracle Corporation. It provides a range of services including computing, storage,␣
↪networking, database, and more, allowing businesses to build, deploy, and manage␣
↪applications and services in a secure and scalable manner.

...
..
OML4PY provides a Python interface to OML, allowing users to create, manipulate, and␣
↪analyze models using Python scripts. It enables users to leverage the power of OML and␣
↪OMF from within Python, making it easier to integrate modeling and simulation into␣
↪larger workflows and applications.
...
...

An Autonomous Database is a type of database that uses artificial intelligence (AI) and␣
↪machine learning (ML) to automate many of the tasks typically performed by a database␣
↪administrator (DBA)
...
...
```

9.2.13 Async streaming chat

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_profile = await select_ai.AsyncProfile(
        profile_name="async_oci_ai_profile"
    )

    chunks = await async_profile.chat(
        prompt="What is OCI ?", stream=True, chunk_size=4096
    )
    async for chunk in chunks:
        print(chunk, end="")
    print()


asyncio.run(main())
```

`stream=True` lets callers consume generated CLOB responses chunk by chunk, reducing memory pressure and making it easier to progressively forward output to files, services, or user interfaces. Async streaming text APIs return an async iterator of `str` chunks after the awaited method call. The `chunk_size` parameter controls the number of CLOB characters read per chunk; it is not a byte count.

Streaming is supported by `generate()`, `chat()`, `narrate()`, `explain_sql()`, `show_sql()`, and `show_prompt()`. It is not supported for `run_sql()`, which returns a `pandas.DataFrame`.

The SDK awaits the database function before reading its returned CLOB in chunks. The first chunk is available after that function returns; this does not guarantee token delivery while the model is generating. A partially consumed iterator retains its cursor and connection. Use `aclosing()` when iteration can stop early, and consume session streams before leaving the `chat_session()` context:

```
from contextlib import aclosing

chunks = await async_profile.chat("Explain connection pooling.", stream=True)
async with aclosing(chunks):
    async for chunk in chunks:
        print(chunk, end="")
```

9.2.14 Async request-level profile attribute overrides

The `attributes` keyword accepts a mapping of profile attributes to apply to one async request. These values override the corresponding saved profile attributes for that request only; they do not update the profile in the database. The mapping must contain JSON-serializable values.

Request-level attributes are available on the following asynchronous APIs:

- `AsyncProfile.generate()`, `chat()`, and `narrate()`.
- SQL helpers: `explain_sql()`, `run_sql()`, `show_sql()`, and `show_prompt()`.
- `AsyncSession.chat()`, `narrate()`, `explain_sql()`, `run_sql()`, `show_sql()`, and `show_prompt()`.
- `AsyncProfile.run_pipeline()`, which applies the same mapping to every prompt/action request in the pipeline.
- Streaming calls to the text-producing methods, by combining `stream=True` with `attributes=...`. `run_sql()` does not support streaming.

The mapping can include attributes such as `additional_instructions`, the integer `seed`, and `source_language` or `target_language`. Use `ProfileAttributes` for profile-wide defaults; use this mapping when an override should apply only to the current request.

For `translate()`, pass language values through its named `source_language` and `target_language` arguments. The `attributes` mapping described here applies to the generate/action APIs.

```
request_attributes = {
    "additional_instructions": "Answer in one sentence.",
    "seed": 42,
    "source_language": "en",
    "target_language": "de",
}

response = await async_profile.chat(
    prompt="What is Oracle Cloud Infrastructure?",
    attributes=request_attributes,
)
```

For async streaming, await the method call first and then iterate over the returned async iterator. The same mapping can be used with an async session:

```
chunks = await async_profile.chat(
    prompt="Summarize Oracle Cloud Infrastructure.",
    stream=True,
    attributes=request_attributes,
)
async for chunk in chunks:
    print(chunk, end="")

async with async_profile.chat_session(conversation) as session:
    response = await session.chat(
        prompt="Give one more detail.",
        attributes=request_attributes,
    )
```

The following sample uses `show_prompt()` to verify that the request-only `additional_instructions` value reached the generated prompt:

```
import asyncio
import os

import select_ai
```

(continues on next page)

(continued from previous page)

```
user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
profile_name = os.getenv("SELECT_AI_PROFILE_NAME", "async_oci_ai_profile")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_profile = await select_ai.AsyncProfile(profile_name=profile_name)

    instruction = "Include the exact marker PYSAI_REQUEST_ATTRIBUTES_SAMPLE."
    request_attributes = {
        "additional_instructions": instruction,
        "seed": 42,
        "source_language": "en",
        "target_language": "de",
    }

    prompt = await async_profile.show_prompt(
        prompt="Tell me about Oracle Cloud Infrastructure.",
        attributes=request_attributes,
    )
    print("Request attributes applied:", instruction in prompt)


asyncio.run(main())
```

output:

```
Request attributes applied: True
```

9.2.15 Summarize

Summarize inline content

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

content = """
A gas cloud in our galaxy, Sagittarius B2, contains enough alcohol to brew 400
trillion pints of beer, and some stars are so cool that you could touch them
without being burned. Meanwhile, on the exoplanet 55 Cancri e, a form of
"hot ice" exists where high pressure prevents water from becoming gas even at
high temperatures. Additionally, some ancient stars found in the Milky Way's
halo are much older than the Sun, providing clues about the early universe and
its composition
"""


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_profile = await select_ai.AsyncProfile(
        profile_name="async_oci_ai_profile",
    )
    summary = await async_profile.summarize(content=content)
    print(summary)


asyncio.run(main())
```

output:

```
A gas cloud in the Sagittarius B2 galaxy contains a large amount of alcohol,
while some stars are cool enough to touch without being burned. The exoplanet
55 Cancri e has a unique form of "hot ice" where water remains solid despite
high temperatures due to high pressure. Ancient stars in the Milky Way's halo
are older than the Sun, providing insights into the early universe and its composition,
offering clues about the universe's formation and evolution.
```

Summarize content accessible via a URL

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_profile = await select_ai.AsyncProfile(
        profile_name="async_oci_ai_profile",
    )
    summary = await async_profile.summarize(
        location_uri="https://en.wikipedia.org/wiki/Astronomy"
    )
    print(summary)


asyncio.run(main())
```

output:

```
Astronomy is a natural science that studies celestial objects and phenomena,
using mathematics, physics, and chemistry to explain their origin and evolution.
The field has a long history, with early civilizations making methodical
observations of the night sky, and has since split into observational and
theoretical branches. Observational astronomy focuses on acquiring data
from observations, while theoretical astronomy develops computer or
analytical models to describe astronomical objects and phenomena. The study
of astronomy has led to numerous discoveries, including the existence of
galaxies, the expansion of the universe, and the detection of gravitational
waves. Astronomers use various methods, such as radio, infrared, optical,
ultraviolet, X-ray, and gamma-ray astronomy, to study objects and events in
the universe. The field has also led to the development of new technologies and
has inspired new areas of research, such as astrobiology and the search for
extraterrestrial life. Overall, astronomy is a dynamic and constantly evolving
field that seeks to understand the universe and its many mysteries.
```

9.2.16 Translate

`AsyncProfile.translate()` accepts optional `source_language` and `target_language` arguments. Per-call values take precedence over the profile's `source_language` and `target_language` defaults.

If `source_language` is omitted for both the call and the profile, source language detection is delegated to the translation provider. If `target_language` is omitted, the profile's default is used; a target language must be supplied in one of those two places.

Configure profile-level defaults with `ProfileAttributes` when creating or updating a profile:

```
language_defaults = select_ai.ProfileAttributes(
    source_language="en",
    target_language="de",
)

await async_profile.set_attributes(language_defaults)
print(await async_profile.translate(text="Thank you"))
```

When only the target is supplied at the call site, the provider can detect the source language:

```
print(await async_profile.translate(text="Thank you", target_language="de"))
```

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
profile_name = os.getenv("SELECT_AI_PROFILE_NAME", "async_oci_ai_profile")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_profile = await select_ai.AsyncProfile(
        profile_name=profile_name,
    )
    response = await async_profile.translate(
        text="Thank you",
        target_language="de",
    )
    print(response)


asyncio.run(main())
```

output:

```
Danke
```

9.2.17 Async pipeline

Pipeline compatibility and results

`run_pipeline()` requires `python-oracledb>=2.4.0`, which provides `create_pipeline()` and `AsyncConnection.run_pipeline()`. If an older driver lacks these APIs, upgrade it with `python -m pip install --upgrade oracledb`. Async operations use Thin mode.

The driver documents true pipelining for Oracle AI Database 26ai and later. On older databases it executes operations individually, without reducing round trips. Pipeline operations execute sequentially, even when true pipelining is available. Select AI returns CLOBs, and reading these adds database calls; do not assume the complete SDK operation uses one round trip. See python-oracledb pipelining.

Results preserve input order. Successful results are strings read from CLOBs. For `Action.RUNSQL`, this is the database's JSON response, rather than the `pandas.DataFrame` returned by `AsyncProfile.run_sql()`. With `continue_on_error=False` (the default), an operation error raises an exception. With `continue_on_error=True`, failed entries contain driver error objects; check their type before using them as text:

```
results = await async_profile.run_pipeline(
    [("How many promotions?", select_ai.Action.SHOWSQL)],
    continue_on_error=True,
)
for result in results:
    if isinstance(result, str):
        print(result)
    else:
        print(f"Pipeline operation failed: {result}")
```

`AsyncProfile.run_pipeline()` also accepts `attributes=...`. The mapping is applied to every prompt/action request in the pipeline, while the saved profile remains unchanged.

```
results = await async_profile.run_pipeline(
    prompt_specifications,
    attributes={
        "additional_instructions": "Keep each answer concise.",
        "seed": 42,
    },
)
```

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_profile = await select_ai.AsyncProfile(
        profile_name="async_oci_ai_profile"
    )
    prompt_specifications = [
```

(continues on next page)

(continued from previous page)

```
        ("What is Oracle Autonomous Database?", select_ai.Action.CHAT),
        ("Generate SQL to list all customers?", select_ai.Action.SHOWSQL),
        (
            "Explain the query: SELECT * FROM sh.products",
            select_ai.Action.EXPLAINSQL,
        ),
        ("Explain the query: SELECT * FROM sh.products", "INVALID ACTION"),
    ]

    # Operations execute sequentially; round-trip savings depend on the
    # database version. Successful results are strings, including RUNSQL JSON.
    results = await async_profile.run_pipeline(
        prompt_specifications, continue_on_error=True
    )
    for i, result in enumerate(results):
        print(
            f"Result {i} for prompt '{prompt_specifications[i][0]}' is: {result}"
        )


asyncio.run(main())
```

output:

```
Result 0 for prompt 'What is Oracle Autonomous Database?' is: Oracle Autonomous Database␣
↪is a cloud-based database service that uses artificial intelligence (AI) and machine␣
↪learning (ML) to automate many of the tasks associated with managing a database. It is␣
↪a self-driving, self-securing, and self-repairing database that eliminates the need␣
↪for manual database administration, allowing users to focus on higher-level tasks.


Result 1 for prompt 'Generate SQL to list all customers?' is: SELECT "c"."CUST_ID" AS
↪"Customer ID", "c"."CUST_FIRST_NAME" AS "First Name", "c"."CUST_LAST_NAME" AS "Last␣
↪Name", "c"."CUST_EMAIL" AS "Email" FROM "SH"."CUSTOMERS" "c"

Result 2 for prompt 'Explain the query: SELECT * FROM sh.products' is: ```sql
SELECT
  p.*
FROM
  "SH"."PRODUCTS" p;
```

Explanation:

This query is designed to retrieve all columns (`*`) from the `"SH"."PRODUCTS"` table.

Here's a breakdown of the query components:

Result 3 for prompt 'Explain the query: SELECT * FROM sh.products' is: ORA-20000:␣
↪Invalid action - INVALID ACTION
```
```

9.2.18 Share a profile asynchronously

While connected as the profile owner, grant access to an existing profile:

```
profile = await select_ai.AsyncProfile.fetch("OCI_AI_PROFILE")
await profile.grant_access("APP_USER")
```

The recipient can fetch the profile with `owner="APP_OWNER"` in a separate connection. When access should end, run `await profile.revoke_access("APP_USER")` as the owner. See *Sharing and ownership* for the complete owner and recipient sequence.

9.2.19 Give async feedback on generated SQL

Use feedback after reviewing the SQL produced for a prompt. Identify the request with a `(prompt, Action)` tuple or a `sql_id` from `V$MAPPED_SQL`. The example below assumes an open async connection and an existing profile over the `SH` schema:

```
profile = await select_ai.AsyncProfile.fetch("OCI_AI_PROFILE")
prompt = "How many promotions?"
sql = await profile.show_sql(prompt=prompt)
print(sql)
```

After inspecting the SQL, record positive feedback if it answers the question:

```
await profile.add_positive_feedback(
    prompt_spec=(prompt, select_ai.Action.SHOWSQL)
)
```

For an incorrect answer, use `add_negative_feedback()` with the same request identifier, `response` containing the expected SQL, and `feedback_content` explaining the correction. Remove obsolete feedback with `delete_feedback()` and the same request identifier. Await these methods when using `AsyncProfile`.

9.2.20 List profiles asynchronously

Use `AsyncProfile.list()` to discover saved profiles in the current schema. Call it on the class; you do not need to create an empty profile first. Each returned object includes `profile_name`, `owner`, and the owner-qualified `qualified_name`.

Connect and list your profiles

Set `SELECT_AI_USER`, `SELECT_AI_PASSWORD`, and `SELECT_AI_DB_CONNECT_STRING` before running this complete script. It prints one qualified name per profile and closes the connection afterward:

```
import asyncio
import os

import select_ai


async def main():
    await select_ai.async_connect(
        user=os.environ["SELECT_AI_USER"],
        password=os.environ["SELECT_AI_PASSWORD"],
        dsn=os.environ["SELECT_AI_DB_CONNECT_STRING"],
    )
    try:
        async for profile in select_ai.AsyncProfile.list():
            print(profile.qualified_name)
    finally:
        await select_ai.async_disconnect()


asyncio.run(main())
```

For a schema named `APP_OWNER` containing two profiles, the output could be:

```
APP_OWNER.OCI_AI_PROFILE
APP_OWNER.OCI_VECTOR_AI_PROFILE
```

Your results depend on the profiles visible in your schema; result order is not guaranteed. An empty result means no profiles match the requested scope.

Filter by name

With a connection open, use `profile_name_pattern` to match names with a regular expression. This example lists names starting with `OCI_`:

```
async for profile in select_ai.AsyncProfile.list(profile_name_pattern="^OCI_"):
    print(profile.qualified_name)
```

`AsyncProfile.list()` returns an async iterator. Use `async for` directly; do not await the iterator. Put these snippets inside your async function before disconnecting.

Find and reuse a shared profile

Pass `owner` to list profiles in another schema. The connected user must have access to those profiles; specifying an owner does not grant access. While connected as the recipient, list the profiles shared by `APP_OWNER`:

```
async for profile in select_ai.AsyncProfile.list(owner="APP_OWNER"):
    print(profile.qualified_name)
```

Once you know the profile name, fetch that profile with the same owner and use the returned object directly:

```
profile = await select_ai.AsyncProfile.fetch(
    "OCI_AI_PROFILE", owner="APP_OWNER"
)
print(profile.qualified_name)  # APP_OWNER.OCI_AI_PROFILE
response = await profile.chat(prompt="What is retrieval augmented generation?")
print(response)
```

See *Sharing and ownership* for the owner's grant and revoke steps.

CHAPTER

TEN

CONVERSATION

Conversations in Select AI represent an interactive exchange between the user and the system, enabling users to query or interact with the database through a series of natural language prompts.

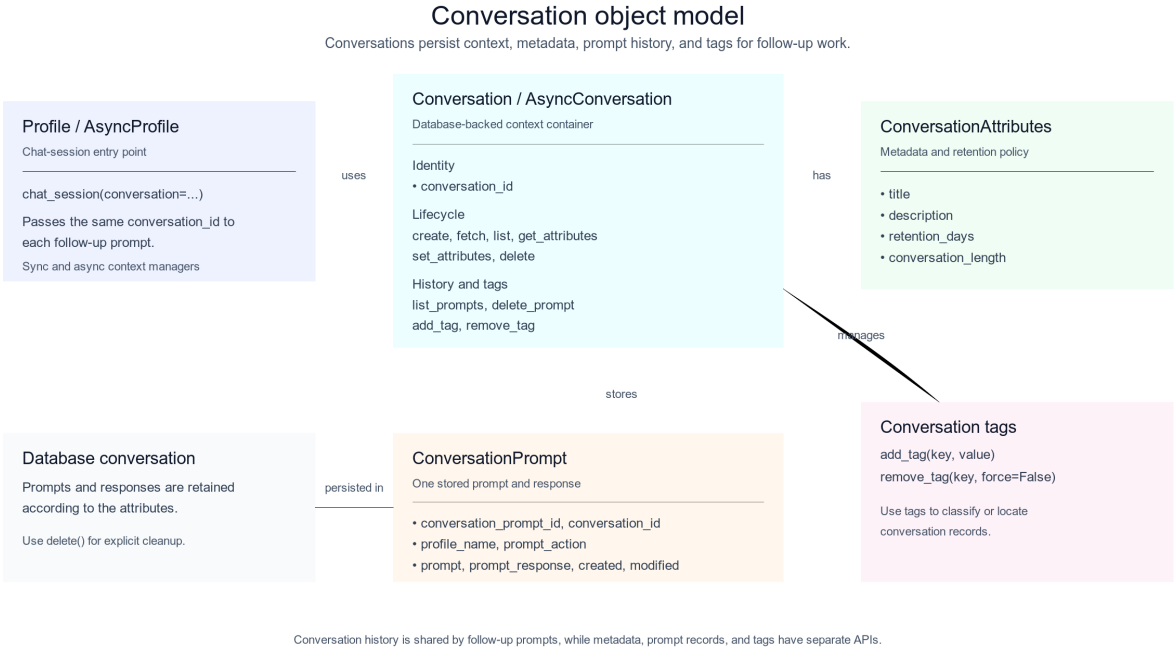
A conversation is stored in the database and identified by a `conversation_id`. Pass that conversation to `Profile.chat_session()` or `AsyncProfile.chat_session()` when follow-up prompts should use prior prompts as context. This is useful for chat workflows where the user asks a question, then asks follow-up questions such as "explain that further" or "show another example".

Use conversations when you need context across multiple prompts. For one-off prompts, call profile methods such as `chat()`, `show_sql()`, or `narrate()` directly without creating a conversation.

The usual lifecycle is:

1. Create `ConversationAttributes` with a title, optional description, retention period, and conversation length.
2. Create a `Conversation` or `AsyncConversation` object.
3. Use the conversation in `profile.chat_session(...)`.
4. List, fetch, or update the conversation metadata when needed.
5. Delete the conversation when the stored history is no longer needed.

10.1 Conversation Object model


Conversation object model
Conversations persist context, metadata, prompt history, and tags for follow-up work.
Profile / AsyncProfile
Chat-session entry point
chat_session(conversation=...)
Passes the same conversation_id to each follow-up prompt.
Sync and async context managers
uses
Conversation / AsyncConversation
Database-backed context container
Identity
• conversation_id
Lifecycle
create, fetch, list, get_attributes
set_attributes, delete
History and tags
list_prompts, delete_prompt
add_tag, remove_tag
has
ConversationAttributes
Metadata and retention policy
• title
• description
• retention_days
• conversation_length
manages
stores
Conversation tags
add_tag(key, value)
remove_tag(key, force=False)
Use tags to classify or locate conversation records.
Database conversation
Prompts and responses are retained according to the attributes.
Use delete() for explicit cleanup.
persisted in
ConversationPrompt
One stored prompt and response
• conversation_prompt_id, conversation_id
• profile_name, prompt_action
• prompt, prompt_response, created, modified
Conversation history is shared by follow-up prompts, while metadata, prompt records, and tags have separate APIs.

10.2 ConversationAttributes

`ConversationAttributes` controls the metadata and retention behavior for a conversation:

Attribute	Use
`title`	Human-readable conversation title. If omitted, the default is `"New Conversation"`.
`descriptio`	Optional description of the conversation topic.
`retention_`	Number of days to keep the conversation in the database from its creation date. Use `datetime.timedelta(days=...)`. A value of `datetime.timedelta(days=0)` keeps the conversation until it is manually deleted.
`conversati`	Number of prompts retained in the conversation context. The default is `10`.

Example:

```
import datetime

attributes = select_ai.ConversationAttributes(
    title="Sales analysis",
    description="Follow-up questions about quarterly sales",
    retention_days=datetime.timedelta(days=14),
    conversation_length=20,
)
```

class select_ai.**ConversationAttributes**(*title: str | None = 'New Conversation'*, *description: str | None = None*, *retention_days: timedelta | None = datetime.timedelta(days=7)*, *conversation_length: int | None = 10*)

Conversation Attributes

Parameters

- **title** (*str*) – Conversation Title
- **description** (*str*) – Description of the conversation topic
- **retention_days** (*datetime.timedelta*) – The number of days the conversation will be stored in the database from its creation date. If value is 0, the conversation will not be removed unless it is manually deleted by delete
- **conversation_length** (*int*) – Number of prompts to store for this conversation

conversation_length: int | None = 10

description: str | None = None

dict(*exclude_null=True*)

json(*exclude_null=True*)

classmethod keys()

retention_days: timedelta | None = datetime.timedelta(days=7)

title: str | None = 'New Conversation'

10.3 Conversation API

class select_ai.**Conversation**(*conversation_id: str | None = None*, *attributes:* ConversationAttributes | *None = None*)

Conversation class can be used to create, update and delete conversations in the database

Typical usage is to combine this conversation object with an AI Profile.chat_session() to have context-aware conversations with the LLM provider

Parameters

- **conversation_id** (*str*) – Conversation ID
- **attributes** (ConversationAttributes) – Conversation attributes

add_tag(*tag_key: str*, *tag_value: str*) → None

Add or update a tag on this conversation.

Parameters

- **tag_key** (*str*) – Tag name.
- **tag_value** (*str*) – Value associated with the tag name.

create() → str

Creates a new conversation and returns the conversation_id to be used in context-aware conversations with LLMs

Returns
conversation_id

delete(*force: bool = False*)

Drops the conversation

delete_prompt(*conversation_prompt_id: str*, *force: bool = False*) → None

Delete a stored prompt from this conversation.

Parameters

- **conversation_prompt_id** (*str*) – Identifier of the prompt to delete.
- **force** (*bool*) – Ignore a missing prompt when True.

classmethod fetch(*conversation_id: str*) → *Conversation*

Fetch conversation attributes from the database and build a proxy object

Parameters
conversation_id (*str*) – Conversation ID

get_attributes() → *ConversationAttributes*

Get attributes of the conversation from the database

classmethod list() → Iterator[*Conversation*]

List all conversations

Returns
Iterator[Conversation]

list_prompts() → Iterator[*ConversationPrompt*]

List the prompts and responses stored for this conversation.

Returns
Prompts in creation order.

remove_tag(*tag_key: str*, *force: bool = False*) → None

Remove a tag from this conversation.

Parameters

- **tag_key** (*str*) – Tag name to remove.
- **force** (*bool*) – Ignore a missing tag when True.

set_attributes(*attributes:* ConversationAttributes)

Updates the attributes of the conversation in the database

The synchronous API is used with `select_ai.connect()` or `select_ai.create_pool()`. Important methods:

Method	Use
`create()`	Create a database conversation and return its `conversation_id`.
`fetch(conversation_id)`	Build a `Conversation` object from an existing database conversation.
`get_attributes()`	Read conversation metadata from the database.
`set_attributes(attributes)`	Update the title, description, retention period, or conversation length.
`list()`	Iterate over conversations visible to the current user.
`list_prompts()`	Iterate over prompts and responses stored in this conversation.
`delete_prompt(conversation_prompt_id, force=False)`	Delete one stored prompt. Use `force=True` to ignore a missing prompt.
`add_tag(tag_key, tag_value)`	Add a tag or update the value for an existing tag key.
`remove_tag(tag_key, force=False)`	Remove a tag. Use `force=True` to ignore a missing tag.
`delete(force=False)`	Drop the conversation. Use `force=True` to ignore missing-conversation errors.

`Profile.chat_session(conversation=..., delete=False)` is a context manager. If the conversation has attributes but no `conversation_id`, the session creates it automatically. While the context manager is active, every `session.chat(...)` call passes the same `conversation_id` to Select AI, so follow-up prompts can use the conversation history. If `delete=True`, the conversation is deleted when the session exits.

10.3.1 Create conversation

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
conversation_attributes = select_ai.ConversationAttributes(
    title="History of Science",
    description="LLM's understanding of history of science",
)
conversation = select_ai.Conversation(attributes=conversation_attributes)
conversation_id = conversation.create()

print("Created conversation with conversation id: ", conversation_id)
```

output:

```
Created conversation with conversation id:  3AB2ED3E-7E52-8000-E063-BE1A000A15B6
```

10.3.2 Chat session

Use `chat_session()` to keep context across multiple chat prompts. The second prompt in this example can refer to the previous answer because both prompts use the same database conversation.

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
profile = select_ai.Profile(profile_name="oci_ai_profile")
conversation_attributes = select_ai.ConversationAttributes(
    title="History of Science",
    description="LLM's understanding of history of science",
)
conversation = select_ai.Conversation(attributes=conversation_attributes)
with profile.chat_session(conversation=conversation, delete=True) as session:
    print(
        "Conversation ID for this session is:",
        conversation.conversation_id,
    )
    response = session.chat(
        prompt="What is importance of history of science ?"
    )
    print(response)
    response = session.chat(
        prompt="Elaborate more on 'Learning from past mistakes'"
    )
    print(response)
```

output:

```
Conversation ID for this session is: 380A1910-5BF2-F7A1-E063-D81A000A3FDA

The importance of the history of science lies in its ability to provide a comprehensive␣
↪understanding of the development of scientific knowledge and its impact on society.␣
↪Here are some key reasons why the history of science is important:

1. **Contextualizing Scientific Discoveries**: The history of science helps us␣
↪understand the context in which scientific discoveries were made, including the social,
↪ cultural, and intellectual climate of the time. This context is essential for␣
↪appreciating the significance and relevance of scientific findings.

..
..

The history of science is replete with examples of mistakes, errors, and misconceptions␣
↪that have occurred over time. By studying these mistakes, scientists and researchers␣
↪can gain valuable insights into the pitfalls and challenges that have shaped the␣
↪development of scientific knowledge. Learning from past mistakes is essential for␣
```

(continues on next page)

(continued from previous page)

```
↪several reasons:
...
...
```

10.3.3 Prompt history and tags

`Conversation.list_prompts()` returns an iterator of `ConversationPrompt` objects in creation order. Each object includes the prompt identifier, prompt text, response text, conversation metadata, and timestamps. Use the identifier with `Conversation.delete_prompt()` to remove a stored prompt from the conversation.

Conversation tags are key-value pairs. `Conversation.add_tag()` creates a tag or updates the value for an existing key. `Conversation.remove_tag()` removes a tag; pass `force=True` when a missing tag should not raise an error.

class select_ai.**ConversationPrompt**(*conversation_prompt_id: str*, *conversation_id: str*, *conversation_title: str*, *profile_name: str | None*, *prompt_action: str | None*, *prompt: str | None*, *prompt_response: str | None*, *created: datetime | None*, *modified: datetime | None*, *client_identifier: str | None*, *client_ip: str | None*, *sid: int | None*, *serial: int | None*)

A prompt and response stored in a conversation's history.

client_identifier: str | None

client_ip: str | None

conversation_id: str

conversation_prompt_id: str

conversation_title: str

created: datetime | None

modified: datetime | None

profile_name: str | None

prompt: str | None

prompt_action: str | None

prompt_response: str | None

serial: int | None

sid: int | None

```
conversation.add_tag("PROJECT", "SELECT_AI")
prompts = list(conversation.list_prompts())
for prompt in prompts:
    print(prompt.conversation_prompt_id, prompt.prompt)
if prompts:
    conversation.delete_prompt(prompts[-1].conversation_prompt_id)
conversation.remove_tag("PROJECT")
```

The complete sample creates a conversation, stores one prompt through a chat session, lists the resulting `ConversationPrompt`, deletes it, and exercises tag creation, update, and removal:

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
```

(continues on next page)

(continued from previous page)

```
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
profile_name = os.getenv("SELECT_AI_PROFILE_NAME", "oci_ai_profile")

select_ai.connect(user=user, password=password, dsn=dsn)

conversation = select_ai.Conversation(
    attributes=select_ai.ConversationAttributes(title="Prompt and tag sample")
)
conversation.create()
profile = select_ai.Profile(profile_name=profile_name)

try:
    conversation.add_tag("PROJECT", "SELECT_AI")
    conversation.add_tag("PROJECT", "SELECT_AI_SAMPLES")  # Updates the tag.
    with profile.chat_session(conversation=conversation) as session:
        print(session.chat("Reply with exactly: conversation history works."))

    prompts = list(conversation.list_prompts())
    for prompt in prompts:
        print(f"Prompt {prompt.conversation_prompt_id}: {prompt.prompt}")

    if prompts:
        conversation.delete_prompt(prompts[-1].conversation_prompt_id)
    print("Prompts after deletion:", list(conversation.list_prompts()))
    conversation.remove_tag("PROJECT")
finally:
    conversation.delete(force=True)
```

The conversation ID, prompt ID, and model response vary between runs. Example output (generated IDs and model responses vary):

output:

```
conversation history works.
Prompt <generated prompt id>: Reply with exactly: conversation history works.
Prompts after deletion: []
```

10.3.4 List conversations

Listing returns `Conversation` objects with their `conversation_id` and metadata. It does not replay the conversation transcript.

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
for conversation in select_ai.Conversation().list():
    print(conversation.conversation_id)
    print(conversation.attributes)
```

output:

```
5275A80-A290-DA17-E063-151B000AD3B4
ConversationAttributes(title='History of Science', description="LLM's understanding of␣
↪history of science", retention_days=7)

37DF777F-F3DA-F084-E063-D81A000A53BE
ConversationAttributes(title='History of Science', description="LLM's understanding of␣
↪history of science", retention_days=7)
```

10.3.5 Delete conversation

Delete conversations that are no longer needed, especially when `retention_days` is set to `0` or when the content should not remain in the database after a session ends. For temporary sessions, prefer `profile.chat_session(conversation=conversation, delete=True)`.

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
conversation = select_ai.Conversation(
    conversation_id="37DDC22E-11C8-3D49-E063-D81A000A85FE"
)
conversation.delete(force=True)
print(
    "Deleted conversation with conversation id: ",
    conversation.conversation_id,
)
```

output:

```
Deleted conversation with conversation id:  37DDC22E-11C8-3D49-E063-D81A000A85FE
```

10.4 AsyncConversation API

class select_ai.**AsyncConversation**(*conversation_id: str | None = None*, *attributes:* ConversationAttributes | *None = None*)

AsyncConversation class can be used to create, update and delete conversations in the database in an async manner

Typical usage is to combine this conversation object with an AsyncProfile.chat_session() to have context-aware conversations

Parameters

- **conversation_id** (*str*) – Conversation ID
- **attributes** (ConversationAttributes) – Conversation attributes

async add_tag(*tag_key: str*, *tag_value: str*) → None

Add or update a tag on this conversation.

Parameters

- **tag_key** (*str*) – Tag name.
- **tag_value** (*str*) – Value associated with the tag name.

async create() → str

Creates a new conversation and returns the conversation_id to be used in context-aware conversations with LLMs

Returns
conversation_id

async delete(*force: bool = False*)

Delete the conversation

async delete_prompt(*conversation_prompt_id: str*, *force: bool = False*) → None

Delete a stored prompt from this conversation.

Parameters

- **conversation_prompt_id** (*str*) – Identifier of the prompt to delete.
- **force** (*bool*) – Ignore a missing prompt when True.

async classmethod fetch(*conversation_id: str*) → *AsyncConversation*

Fetch conversation attributes from the database

async get_attributes() → *ConversationAttributes*

Get attributes of the conversation from the database

classmethod list() → AsyncGenerator[*AsyncConversation*, None]

List all conversations

Returns
AsyncGenerator[AsyncConversation, None]

async list_prompts() → AsyncGenerator[*ConversationPrompt*, None]

List the prompts and responses stored for this conversation.

Returns
Prompts in creation order.

async remove_tag(*tag_key: str*, *force: bool = False*) → None

Remove a tag from this conversation.

Parameters

- **tag_key** (*str*) – Tag name to remove.
- **force** (*bool*) – Ignore a missing tag when True.

async set_attributes(*attributes:* ConversationAttributes)

Updates the attributes of the conversation

The async API mirrors the synchronous API and is used with `select_ai.async_connect()` or `select_ai.create_pool_async()`.

Synchronous API	Async API
`Conversation.create()`	`await AsyncConversation.create()`
`Conversation.fetch(...)`	`await AsyncConversation.fetch(...)`
`Conversation.get_attributes()`	`await AsyncConversation.get_attributes()`
`Conversation.set_attributes(...)`	`await AsyncConversation.set_attributes(...)`
`for conversation in Conversation.list()`	`async for conversation in AsyncConversation.list()`
`Conversation.delete(...)`	`await AsyncConversation.delete(...)`
`Conversation.list_prompts()`	`async for prompt in AsyncConversation.list_prompts()`
`Conversation.delete_prompt(...)`	`await AsyncConversation.delete_prompt(...)`
`Conversation.add_tag(...)`	`await AsyncConversation.add_tag(...)`
`Conversation.remove_tag(...)`	`await AsyncConversation.remove_tag(...)`
`with profile.chat_session(...)`	`async with async_profile.chat_session(...)`

10.4.1 Async chat session

Use `AsyncProfile.chat_session()` in async applications. The conversation is created automatically when the object has attributes and no `conversation_id`.

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_profile = await select_ai.AsyncProfile(
        profile_name="async_oci_ai_profile"
    )
    conversation_attributes = select_ai.ConversationAttributes(
        title="History of Science",
        description="LLM's understanding of history of science",
    )
    async_conversation = select_ai.AsyncConversation(
        attributes=conversation_attributes
    )

    async with async_profile.chat_session(
        conversation=async_conversation, delete=True
    ) as async_session:
        response = await async_session.chat(
            prompt="What is importance of history of science ?"
        )
        print(response)
        response = await async_session.chat(
            prompt="Elaborate more on 'Learning from past mistakes'"
        )
        print(response)


asyncio.run(main())
```

output:

```
Conversation ID for this session is: 380A1910-5BF2-F7A1-E063-D81A000A3FDA

The importance of the history of science lies in its ability to provide a comprehensive␣
↪understanding of the development of scientific knowledge and its impact on society.␣
↪Here are some key reasons why the history of science is important:

1. **Contextualizing Scientific Discoveries**: The history of science helps us␣
↪understand the context in which scientific discoveries were made, including the social,
↪ cultural, and intellectual climate of the time. This context is essential for␣
```

(continues on next page)

(continued from previous page)

```
↪appreciating the significance and relevance of scientific findings.

..
..

The history of science is replete with examples of mistakes, errors, and misconceptions␣
↪that have occurred over time. By studying these mistakes, scientists and researchers␣
↪can gain valuable insights into the pitfalls and challenges that have shaped the␣
↪development of scientific knowledge. Learning from past mistakes is essential for␣
↪several reasons:
...
...
```

10.4.2 Async prompt history and tags

Use prompt history to inspect or remove stored prompts, and tags to attach application metadata to a conversation. This example first creates a conversation and records a chat prompt so there is history to inspect.

Create a conversation and save a prompt

With an async connection open, run the following inside an async function. The saved profile `oci_ai_profile` must already exist:

```
conversation = select_ai.AsyncConversation(
    attributes=select_ai.ConversationAttributes(title="Prompt history demo")
)
await conversation.create()
profile = await select_ai.AsyncProfile(profile_name="oci_ai_profile")
async with profile.chat_session(conversation=conversation) as session:
    response = await session.chat("Explain retrieval augmented generation.")
    print(response)
```

Tag and inspect the conversation

A second `add_tag()` call with the same key updates its value. `list_prompts()` returns an async iterator of `ConversationPrompt` objects:

```
await conversation.add_tag("PROJECT", "SELECT_AI")
await conversation.add_tag("PROJECT", "SELECT_AI_SAMPLES")
prompts = [prompt async for prompt in conversation.list_prompts()]
for prompt in prompts:
    print(prompt.conversation_prompt_id, prompt.prompt)
```

Remove a stored prompt and its tag

Delete by prompt ID, then remove the tag by key. Check for an empty history before choosing a prompt to delete:

```
if prompts:
    await conversation.delete_prompt(prompts[-1].conversation_prompt_id)
await conversation.remove_tag("PROJECT")
```

Use `force=True` on `delete_prompt()` or `remove_tag()` when a missing prompt or tag should be ignored. When finished with this temporary conversation, call `await conversation.delete(force=True)`.

Run the complete example

The complete async prompt-history script adds connection setup and a `finally` block that deletes the temporary conversation. Set the common database environment variables and optionally set `SELECT_AI_PROFILE_NAME` (default `oci_ai_profile`), then run from the repository root:

```
python samples/async/conversation_prompts_tags.py
```

Example output (prompt IDs and model responses vary):

```
conversation history works.
Prompt <generated prompt id>: Reply with exactly: conversation history works.
Prompts after deletion: []
```

10.4.3 Async list conversations

`AsyncConversation.list()` is an async iterator.

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async for conversation in select_ai.AsyncConversation().list():
        print(conversation.conversation_id)
        print(conversation.attributes)


asyncio.run(main())
```

output:

```
5275A80-A290-DA17-E063-151B000AD3B4
ConversationAttributes(title='History of Science', description="LLM's understanding of␣
↪history of science", retention_days=7)

37DF777F-F3DA-F084-E063-D81A000A53BE
ConversationAttributes(title='History of Science', description="LLM's understanding of␣
↪history of science", retention_days=7)
```

CHAPTER

ELEVEN

VECTOR INDEX

11.1 Vector Index

`VectorIndex` supports Retrieval Augmented Generation (RAG). It converts source documents into vector embeddings, stores the embeddings in a vector store, and links the vector index to a Select AI profile. When that profile is used for natural language generation, Select AI can retrieve semantically similar content from the vector index and use that content as grounding context for the response.

A vector index is useful when the answer should come from files or documents that are not represented as relational tables. Typical sources include documents in Object Storage, product manuals, generated reports, logs, JSON files, and other text-heavy content that should be searched by meaning rather than exact keywords.

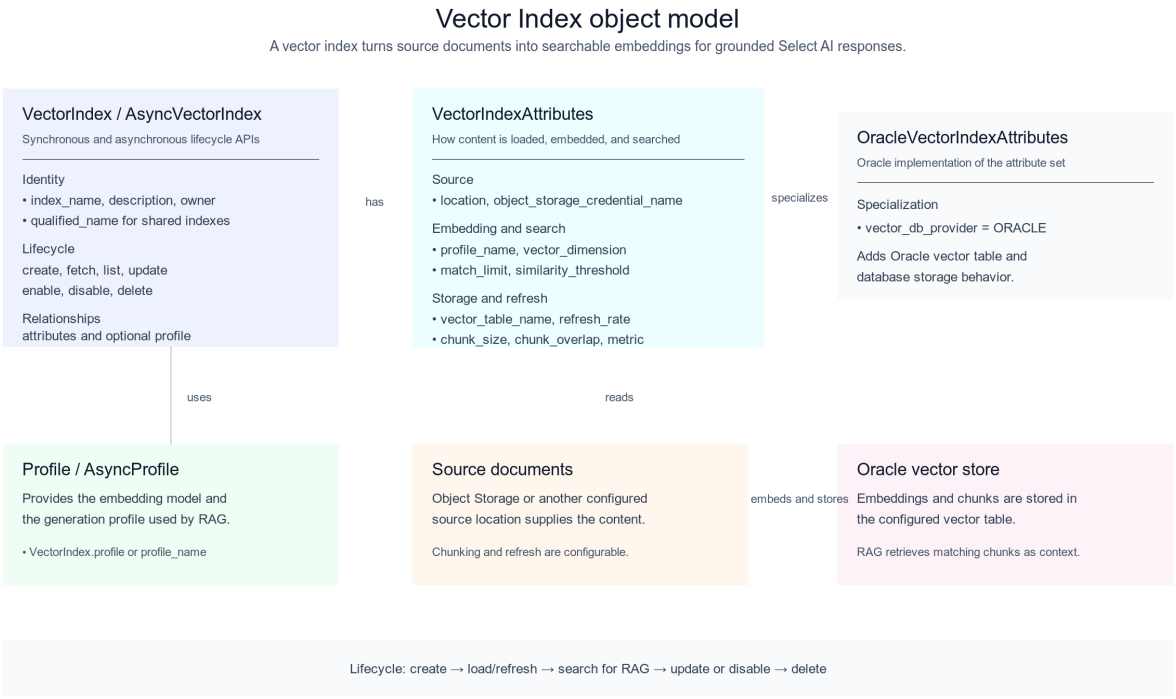
Before creating a vector index, make sure the database user has:

- A Select AI profile with an AI provider that supports embeddings.
- A credential for the AI provider used by the profile.
- A credential for the object storage location if the source objects are not public.
- Network access to the AI provider endpoint and the source location. See *Privileges* for network ACL helpers.

The usual lifecycle is:

1. Create a profile with a provider and embedding model.
2. Create `OracleVectorIndexAttributes` with the source location and storage credential.
3. Create `VectorIndex` and call `create()`.
4. Use the linked profile for RAG actions such as `narrate()`.
5. Fetch, list, update, disable, enable, or delete the index as needed.

11.2 `VectorIndex` Object Model


Vector Index object model
A vector index turns source documents into searchable embeddings for grounded Select AI responses.
VectorIndex / AsyncVectorIndex
Synchronous and asynchronous lifecycle APIs
Identity
• index_name, description, owner
• qualified_name for shared indexes
Lifecycle
create, fetch, list, update
enable, disable, delete
Relationships
attributes and optional profile
has
VectorIndexAttributes
How content is loaded, embedded, and searched
Source
• location, object_storage_credential_name
Embedding and search
• profile_name, vector_dimension
• match_limit, similarity_threshold
Storage and refresh
• vector_table_name, refresh_rate
• chunk_size, chunk_overlap, metric
specializes
OracleVectorIndexAttributes
Oracle implementation of the attribute set
Specialization
• vector_db_provider = ORACLE
Adds Oracle vector table and
database storage behavior.
uses
reads
Profile / AsyncProfile
Provides the embedding model and
the generation profile used by RAG.
• VectorIndex.profile or profile_name
Source documents
Object Storage or another configured
source location supplies the content.
Chunking and refresh are configurable.
embeds and stores
Oracle vector store
Embeddings and chunks are stored in
the configured vector table.
RAG retrieves matching chunks as context.
Lifecycle: create → load/refresh → search for RAG → update or disable → delete

11.3 VectorIndexAttributes

A `VectorIndexAttributes` object can be created with `select_ai.VectorIndexAttributes()`. Also check vector index attributes

For Oracle vector indexes, use `OracleVectorIndexAttributes`. It sets `vector_db_provider` to `VectorDBProvider.ORACLE` and is the preferred attribute class for the examples in this guide.

Common attributes:

Attribute	Use
`location`	Object Storage URI or source location containing the documents to embed.
`object_storage_cre`	Credential used to read the source location.
`profile_name`	Select AI profile used to create embeddings and answer RAG prompts. If omitted during `create()`, it is taken from the `profile` object passed to `VectorIndex`.
`chunk_size` and `chunk_overlap`	Control how source text is split before embedding. Larger chunks keep more context together; overlap helps preserve context across chunk boundaries.
`match_limit`	Maximum number of matching chunks returned during semantic search.
`similarity_thresho`	Minimum similarity score required for retrieved chunks to be considered relevant.
`vector_distance_me`	Distance metric used to compare embeddings. Supported values include `COSINE`, `EUCLIDEAN`, `L2_SQUARED`, `DOT`, `MANHATTAN`, and `HAMMING`.
`refresh_rate`	Refresh interval, in minutes, for loading new or changed source data.
`vector_table_name`	Name of the table used to store vector embeddings and chunked data. Leave unset unless you need to control the storage table name.
`enable_sources`	Include filenames and source links in RAG output when supported by the profile and model response.

Example attributes:

```
attributes = select_ai.OracleVectorIndexAttributes(
    location="https://objectstorage.us-ashburn-1.oraclecloud.com/n/example/b/docs/o/
↪product-guides",
    object_storage_credential_name="object_store_credential",
    chunk_size=1024,
    chunk_overlap=128,
    match_limit=5,
    similarity_threshold=0.5,
    vector_distance_metric=select_ai.VectorDistanceMetric.COSINE,
    refresh_rate=1440,
)
```

The embedding model is configured on the provider inside the linked `ProfileAttributes`. Keep the profile provider and vector index attributes together conceptually: the profile decides how embeddings are generated, while the vector index attributes decide where content is read from, how it is chunked, and how the vector store is searched.

class select_ai.**VectorIndexAttributes**(*chunk_size: int | None = None*, *chunk_overlap: int | None = None*, *enable_sources: bool | None = None*, *location: str | None = None*, *match_limit: int | None = None*, *object_storage_credential_name: str | None = None*, *profile_name: str | None = None*, *refresh_rate: int | None = None*, *similarity_threshold: float | None = None*, *vector_distance_metric:* VectorDistanceMetric *| None = None*, *vector_db_endpoint: str | None = None*, *vector_db_credential_name: str | None = None*, *vector_db_provider:* VectorDBProvider *| None = None*, *vector_dimension: int | None = None*, *vector_table_name: str | None = None*, *pipeline_name: str | None = None*)

Attributes of a vector index help to manage and configure the behavior of the vector index.

Parameters

- **chunk_size** (*int*) – Text size of chunking the input data.
- **chunk_overlap** (*int*) – Specifies the amount of overlapping characters between adjacent chunks of text.
- **enable_sources** – Provides document source links and filenames in RAG output
- **location** (*str*) – Location of the object store.
- **match_limit** (*int*) – Specifies the maximum number of results to return in a vector search query
- **object_storage_credential_name** (*str*) – Name of the credentials for accessing object storage.
- **profile_name** (*str*) – Name of the AI profile which is used for embedding source data and user prompts.
- **refresh_rate** (*int*) – Interval of updating data in the vector store. The unit is minutes.
- **similarity_threshold** (*float*) – Defines the minimum level of similarity required for two items to be considered a match
- **vector_distance_metric** (select_ai.vector_index.VectorDistanceMetric) – Specifies the type of distance calculation used to compare vectors in a database
- **vector_db_provider** (select_ai.vector_index.VectorDBProvider) – Name of the Vector database provider. Default value is "oracle"
- **vector_db_endpoint** (*str*) – Endpoint to access the Vector database
- **vector_db_credential_name** (*str*) – Name of the credentials for accessing Vector database
- **vector_dimension** (*int*) – Specifies the number of elements in each vector within the vector store
- **vector_table_name** (*str*) – Specifies the name of the table or collection to store vector embeddings and chunked data

chunk_overlap: int | None = None

chunk_size: int | None = None

classmethod **create**(*, *vector_db_provider: str | None = None*, ***kwargs*)

dict(*exclude_null=True*)

enable_sources: *bool* | *None* = None

json(*exclude_null=True*)

classmethod keys()

location: *str* | *None* = None

match_limit: *int* | *None* = None

object_storage_credential_name: *str* | *None* = None

pipeline_name: *str* | *None* = None

profile_name: *str* | *None* = None

refresh_rate: *int* | *None* = None

similarity_threshold: *float* | *None* = None

vector_db_credential_name: *str* | *None* = None

vector_db_endpoint: *str* | *None* = None

vector_db_provider: *VectorDBProvider* | *None* = None

vector_dimension: *int* | *None* = None

vector_distance_metric: *VectorDistanceMetric* | *None* = None

vector_table_name: *str* | *None* = None

11.3.1 OracleVectorIndexAttributes

class select_ai.**OracleVectorIndexAttributes**(*chunk_size: int | None = None*, *chunk_overlap: int | None = None*, *enable_sources: bool | None = None*, *location: str | None = None*, *match_limit: int | None = None*, *object_storage_credential_name: str | None = None*, *profile_name: str | None = None*, *refresh_rate: int | None = None*, *similarity_threshold: float | None = None*, *vector_distance_metric: VectorDistanceMetric | None = None*, *vector_db_endpoint: str | None = None*, *vector_db_credential_name: str | None = None*, *vector_db_provider: VectorDBProvider | None = VectorDBProvider.ORACLE*, *vector_dimension: int | None = None*, *vector_table_name: str | None = None*, *pipeline_name: str | None = None*)

Oracle specific vector index attributes

chunk_overlap: *int* | *None* = None

chunk_size: *int* | *None* = None

classmethod create(**, vector_db_provider: str | None = None, **kwargs*)

dict(*exclude_null=True*)

enable_sources: *bool* | *None* = None

json(*exclude_null=True*)

classmethod keys()

location: str | None = None

match_limit: int | None = None

object_storage_credential_name: str | None = None

pipeline_name: str | None = None

profile_name: str | None = None

refresh_rate: int | None = None

similarity_threshold: float | None = None

vector_db_credential_name: str | None = None

vector_db_endpoint: str | None = None

vector_db_provider: *VectorDBProvider* | None = 'oracle'

vector_dimension: int | None = None

vector_distance_metric: *VectorDistanceMetric* | None = None

vector_table_name: str | None = None

11.4 VectorIndex API

A `VectorIndex` object can be created with `select_ai.VectorIndex()`

class select_ai.**VectorIndex**(*profile:* BaseProfile | *None* = *None*, *index_name:* *str* | *None* = *None*, *description:* *str* | *None* = *None*, *attributes:* VectorIndexAttributes | *None* = *None*, *owner:* *str* | *None* = *None*)

VectorIndex objects let you manage vector indexes

Parameters

- **index_name** (*str*) – The name of the vector index
- **description** (*str*) – The description of the vector index
- **attributes** (select_ai.VectorIndexAttributes) – The attributes of the vector index

create(*replace:* *bool* | *None* = *False*, *wait_for_completion:* *bool* = *False*)

Create a vector index in the database and populates the index
with data from an object store bucket using an async scheduler job

Parameters

- **replace** (*bool*) – Replace vector index if it exists
- **wait_for_completion** (*bool*) – True to wait for index creation

Returns
None

delete(*include_data:* *bool* | *None* = *True*, *force:* *bool* | *None* = *False*)

This procedure removes a vector store index

Parameters

- **include_data** (*bool*) – Indicates whether to delete both the customer's vector store and vector index along with the vector index object
- **force** (*bool*) – Indicates whether to ignore errors that occur if the vector index does not exist

Returns
None

Raises
oracledb.DatabaseError

classmethod delete_index(*index_name:* *str*, *include_data:* *bool* = *True*, *force:* *bool* = *False*)

Class method to remove a vector store index

Parameters

- **index_name** (*str*) – The name of the vector index
- **include_data** (*bool*) – Indicates whether to delete both the customer's vector store and vector index along with the vector index object
- **force** (*bool*) – Indicates whether to ignore errors that occur if the vector index does not exist

Returns
None

Raises
oracledb.DatabaseError

`disable()`

This procedure disables a vector index object in the current database. When disabled, an AI profile cannot use the vector index, and the system does not load data into the vector store as new data is added to the object store and does not perform indexing, searching or querying based on the index.

Returns
None

Raises
oracledb.DatabaseError

`enable()`

This procedure enables or activates a previously disabled vector index object. Generally, when you create a vector index, by default it is enabled such that the AI profile can use it to perform indexing and searching.

Returns
None

Raises
oracledb.DatabaseError

`classmethod fetch(index_name: str, owner: str | None = None) → VectorIndex`

Fetches vector index attributes from the database and builds a proxy object for the passed index_name

Parameters

- **index_name** (*str*) – The name of the vector index
- **owner** (*str*) – Owner of a shared index. Defaults to current schema.

`get_attributes() → VectorIndexAttributes`

Get attributes of this vector index

Returns
select_ai.VectorIndexAttributes

Raises
VectorIndexNotFoundError

`get_next_refresh_timestamp() → datetime | None`

Returns the UTC timestamp of the next scheduled refresh

`get_profile() → Profile`

Get Profile object linked to this vector index

Returns
select_ai.Profile

Raises
ProfileNotFoundError

`grant_access(user_or_role_name: str) → None`

Grant a database user or role access to this vector index.

`classmethod list(index_name_pattern: str = '.*', owner: str | None = None) → Iterator[VectorIndex]`

List Vector Indexes

Parameters

- **index_name_pattern** (*str*) – Regular expressions can be used to specify a pattern. Function REGEXP_LIKE is used to perform the match. Default value is ".*" i.e. match all vector indexes.
- **owner** (*str*) – Owner of shared indexes. Defaults to current schema.

Returns
Iterator[VectorIndex]

property qualified_name: str | None

Return the owner-qualified index name when owner is known.

revoke_access(*user_or_role_name: str*) → None

Revoke a database user or role's vector index access.

set_attribute(*attribute_name: str*, *attribute_value: str | int | float*)

This procedure updates an existing vector store index with a specified value of the vector index attribute.

Parameters

- **attribute_name** (*str*) – Custom attribute name
- **attribute_value** (*Union[str, int, float]*) – Attribute Value

set_attributes(*attributes:* VectorIndexAttributes = *None*)

This procedure updates an existing vector store index with a specified value of the vector index attributes. Specify multiple attributes by passing an object of type :class *VectorIndexAttributes*

Parameters
attributes (select_ai.VectorIndexAttributes) – Use this to update multiple attribute values

Returns
None

Raises
oracledb.DatabaseError

Use the synchronous API in scripts, notebooks, and command-line tools that use `select_ai.connect()`. Use `AsyncVectorIndex` in applications already using `asyncio` and `select_ai.async_connect()` or an async connection pool.

Important lifecycle methods:

Method	Use
`create(replace=Fal` `wait_for_completio`	Create the database vector index and start the load pipeline. If `replace=True` and the index already exists, the existing index is dropped and recreated. Use `wait_for_completion=True` when the next step depends on the initial load being complete.
`fetch(index_name, owner=None)`	Build a `VectorIndex` proxy from database metadata, including attributes and the linked profile when it still exists.
`list(index_name_pa *", owner=None)`	Iterate over vector indexes visible to the current user or owned by the specified schema. The pattern is evaluated with Oracle `REGEXP_LIKE`.
`set_attribute()` and `set_attributes()`	Update one or more index attributes.
`get_next_refresh_t`	Return the next scheduled refresh timestamp in UTC when the index has a refresh rate and a recorded pipeline execution.
`disable()` and `enable()`	Pause or resume use of the vector index for loading, indexing, searching, and querying.
`delete(include_dat` `force=False)`	Drop the vector index. `include_data=True` also removes associated vector store data. `force=True` ignores missing-index errors.

Check the examples below to understand how to create vector indexes.

11.4.1 Create vector index

In the following example, vector database provider is Oracle and objects used to create embeddings reside in OCI Object Storage. The profile uses an OCI Generative AI provider with an embedding model, and the vector index is linked to that profile during `create()`.

Upload `samples/data/refund-policy.txt` from the repository to a bucket you can read. The document says customers have 30 calendar days to request a refund. Configure the sample with the URI of that uploaded file and the name of an existing database credential authorized to read it:

```
export SELECT_AI_DOCUMENT_URI='https://objectstorage.REGION.oraclecloud.com/n/NAMESPACE/
↪b/BUCKET/o/refund-policy.txt'
export SELECT_AI_OBJECT_STORAGE_CREDENTIAL='OBJECT_STORE_CRED'
export SELECT_AI_PROVIDER_CREDENTIAL='my_oci_ai_profile_key'
```

Replace `REGION`, `NAMESPACE`, and `BUCKET` with your storage values. Use the common database connection variables from *Connection*. The provider credential and storage credential can be the same credential when it has both permissions. Optionally set `SELECT_AI_OCI_REGION` and `SELECT_AI_EMBEDDING_MODEL` for your OCI configuration. The default region is `us-chicago-1` and embedding model is `cohere.embed-v4.0`; choose a model available in your account and region.

The creation samples replace `test_vector_index` and their named RAG profile. Use a development schema, or edit these names before running if you need to preserve existing definitions. The samples close their database connections but leave the profile and index available for the next step.

From the repository root, run `python samples/vector_index_create.py`. For async, use `python samples/async/vector_index_create.py`. Both examples wait for the initial load to complete. Run either workflow in its own schema, or delete the existing sample index before switching between them.

```
import os

import select_ai

user = os.environ["SELECT_AI_USER"]
password = os.environ["SELECT_AI_PASSWORD"]
dsn = os.environ["SELECT_AI_DB_CONNECT_STRING"]
# Upload samples/data/refund-policy.txt to your bucket, then set its URI.
source_uri = os.environ["SELECT_AI_DOCUMENT_URI"]
provider_credential = os.getenv(
    "SELECT_AI_PROVIDER_CREDENTIAL", "my_oci_ai_profile_key"
)
storage_credential = os.environ["SELECT_AI_OBJECT_STORAGE_CREDENTIAL"]

select_ai.connect(user=user, password=password, dsn=dsn)
try:
    provider = select_ai.OCIGenAIProvider(
        region=os.getenv("SELECT_AI_OCI_REGION", "us-chicago-1"),
        oci_apiformat="GENERIC",
        embedding_model=os.getenv(
            "SELECT_AI_EMBEDDING_MODEL", "cohere.embed-v4.0"
        ),
    )
    profile = select_ai.Profile(
        profile_name="oci_vector_ai_profile",
        attributes=select_ai.ProfileAttributes(
            credential_name=provider_credential,
```

(continues on next page)

(continued from previous page)

```
            provider=provider,
        ),
        description="Profile for the sample refund policy",
        replace=True,
    )
    vector_index = select_ai.VectorIndex(
        index_name="test_vector_index",
        attributes=select_ai.OracleVectorIndexAttributes(
            location=source_uri,
            object_storage_credential_name=storage_credential,
        ),
        description="Sample refund policy",
        profile=profile,
    )
    vector_index.create(replace=True, wait_for_completion=True)
    print("Created vector index: test_vector_index")
finally:
    select_ai.disconnect()
```

output:

```
Created vector index: test_vector_index
```

11.4.2 List vector index

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
vector_index = select_ai.VectorIndex()
for index in vector_index.list(index_name_pattern="^test"):
    print("Vector index", index.index_name)
    print("Vector index profile", index.profile)
```

The script prints each matching index name and its linked profile. Results depend on the indexes visible to the connected user.

11.4.3 Fetch vector index

You can fetch the vector index attributes and associated AI profile using the class method `VectorIndex.fetch(index_name, owner=None)`. Pass `owner` for an index shared from another schema. The returned object exposes the resolved `owner` and the owner-qualified `qualified_name`. Fetch is useful when the index was created earlier or by another process and you want to inspect or update it without recreating the original Python object.

```
vector_index = select_ai.VectorIndex.fetch(
    "PRODUCT_DOCS",
    owner="APP_OWNER",
)
print(vector_index.qualified_name)
```

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)

vector_index = select_ai.VectorIndex.fetch(index_name="test_vector_index")
print(vector_index.attributes)
print(vector_index.profile)
print(vector_index.get_next_refresh_timestamp())
```

The script prints the saved index attributes, its linked profile, and the next refresh timestamp. The URI and credentials reflect your configuration; the next refresh can be `None` when no refresh execution has been recorded.

11.4.4 Update vector index attributes

To update attributes, use either `vector_index.set_attribute()` or `vector_index.set_attributes()`. Use `set_attribute()` for a single value and `set_attributes()` when updating several values together.

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
select_ai.connect(user=user, password=password, dsn=dsn)
vector_index = select_ai.VectorIndex(
    index_name="test_vector_index",
)

# Use vector_index.set_attributes to update a multiple attributes
updated_attributes = select_ai.OracleVectorIndexAttributes(refresh_rate=1450)
vector_index.set_attributes(attributes=updated_attributes)

# Use vector_index.set_attribute to update a single attribute
vector_index.set_attribute(
    attribute_name="similarity_threshold", attribute_value=0.5
)
print(vector_index.attributes)
```

The script sets `refresh_rate` to 1450 minutes and `similarity_threshold` to 0.5, then prints the attributes held by the Python object.

11.4.5 RAG using vector index

After `create()` succeeds, the profile has its `vector_index_name` set to the new index. Use that profile with text-returning actions such as `narrate()` to retrieve relevant chunks from the vector index and ground the answer in the indexed content.

```
import os

import select_ai

user = os.environ["SELECT_AI_USER"]
password = os.environ["SELECT_AI_PASSWORD"]
dsn = os.environ["SELECT_AI_DB_CONNECT_STRING"]

select_ai.connect(user=user, password=password, dsn=dsn)
try:
    profile = select_ai.Profile(
        profile_name="oci_vector_ai_profile"
    )
    response = profile.narrate(
        "How many days do customers have to request a refund?"
    )
    print(response)
finally:
    select_ai.disconnect()
```

The answer should identify the **30 calendar day** refund window. Exact wording and source formatting depend on the provider and profile settings. For example:

output:

```
Customers can request a refund within 30 calendar days of purchase.
```

If no answer is grounded in the policy, verify that the uploaded URI and storage credential are correct, that the initial load completed, and that this profile is linked to `test_vector_index`. Run the corresponding RAG script after creating the index. See *Delete vector index* to remove the sample index and its data; delete the sample profile separately when finished.

11.4.6 Delete vector index

Use `delete()` when the index is no longer needed. By default, `include_data=True` removes the vector index metadata and the associated vector store data. Set `include_data=False` only when you intentionally want to keep the underlying vector store data.

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
vector_index = select_ai.VectorIndex(index_name="test_vector_index")
vector_index.delete(force=True)
print("Deleted vector index: test_vector_index")
```

output:

```
Deleted vector index: test_vector_index
```

11.5 AsyncVectorIndex API

An AsyncVectorIndex object can be created with select_ai.AsyncVectorIndex()

class select_ai.**AsyncVectorIndex**(*profile:* BaseProfile | *None* = *None*, *index_name:* *str* | *None* = *None*, *description:* *str* | *None* = *None*, *attributes:* VectorIndexAttributes | *None* = *None*, *owner:* *str* | *None* = *None*)

AsyncVectorIndex objects let you manage vector indexes using async APIs. Use this for non-blocking concurrent requests

Parameters

- **index_name** (*str*) – The name of the vector index
- **description** (*str*) – The description of the vector index
- **attributes** (VectorIndexAttributes) – The attributes of the vector index

async create(*replace:* *bool* | *None* = *False*, *wait_for_completion:* *bool* | *None* = *False*) → None

Create a vector index in the database and populates it with data from an object store bucket using an async scheduler job

Parameters

- **replace** (*bool*) – True to replace existing vector index
- **wait_for_completion** (*bool*) – True to wait for index creation

async delete(*include_data:* *bool* | *None* = *True*, *force:* *bool* | *None* = *False*) → None

This procedure removes a vector store index.

Parameters

- **include_data** (*bool*) – Indicates whether to delete both the customer's vector store and vector index along with the vector index object.
- **force** (*bool*) – Indicates whether to ignore errors that occur if the vector index does not exist.

Returns
None

Raises
oracledb.DatabaseError

async classmethod delete_index(*index_name:* *str*, *include_data:* *bool* = *True*, *force:* *bool* = *False*)

Class method to remove a vector store index

Parameters

- **index_name** (*str*) – The name of the vector index
- **include_data** (*bool*) – Indicates whether to delete both the customer's vector store and vector index along with the vector index object
- **force** (*bool*) – Indicates whether to ignore errors that occur if the vector index does not exist

Returns
None

Raises
oracledb.DatabaseError

async disable() → None

This procedure disables a vector index object in the current database. When disabled, an AI profile cannot use the vector index, and the system does not load data into the vector store as new data is added to the object store and does not perform indexing, searching or querying based on the index.

Returns
None

Raises
oracledb.DatabaseError

async enable() → None

This procedure enables or activates a previously disabled vector index object. Generally, when you create a vector index, by default it is enabled such that the AI profile can use it to perform indexing and searching.

Returns
None

Raises
oracledb.DatabaseError

async classmethod fetch(*index_name: str*, *owner: str | None = None*) → *AsyncVectorIndex*

Fetches vector index attributes from the database and builds a proxy object for the passed index_name

Parameters

- **index_name** (*str*) – The name of the vector index
- **owner** (*str*) – Owner of a shared index. Defaults to current schema.

async get_attributes() → *VectorIndexAttributes*

Get attributes of a vector index

Returns
select_ai.VectorIndexAttributes

Raises
VectorIndexNotFoundError

async get_next_refresh_timestamp() → datetime | None

Return the UTC timestamp for the next scheduled refresh.

async get_profile() → *AsyncProfile*

Get AsyncProfile object linked to this vector index

Returns
select_ai.AsyncProfile

Raises
ProfileNotFoundError

async grant_access(*user_or_role_name: str*) → None

Asynchronously grant a user or role access to this vector index.

classmethod list(*index_name_pattern: str = '.*'*, *owner: str | None = None*) → AsyncGenerator[*AsyncVectorIndex*, None]

List Vector Indexes.

Parameters

- **index_name_pattern** (*str*) – Regular expressions can be used to specify a pattern. Function REGEXP_LIKE is used to perform the match. Default value is ".*" i.e. match all vector indexes.
- **owner** (*str*) – Owner of shared indexes. Defaults to current schema.

Returns
AsyncGenerator[VectorIndex]

property qualified_name: str | None

Return the owner-qualified index name when owner is known.

async revoke_access(*user_or_role_name: str*) → None

Asynchronously revoke a user or role's vector index access.

async set_attribute(*attribute_name: str*, *attribute_value: str | int | float*) → None

This procedure updates an existing vector store index with a specified value of the vector index attribute.

Parameters

- **attribute_name** (*str*) – Custom attribute name
- **attribute_value** (*Union[str, int, float]*) – Attribute Value

async set_attributes(*attributes:* VectorIndexAttributes) → None

This procedure updates an existing vector store index with a specified value of the vector index attribute. multiple attributes by passing an object of type :class *VectorIndexAttributes*

Parameters
attributes (select_ai.VectorIndexAttributes) – Use this to update multiple attribute values

Returns
None

Raises
oracledb.DatabaseError

The async API mirrors the synchronous API. Async profile construction and vector index methods that access the database must be awaited, and `AsyncVectorIndex.list()` is an async iterator.

Synchronous API	Async API
`select_ai.connect(...)`	`await select_ai.async_connect(...)`
`Profile(...)`	`await AsyncProfile(...)`
`VectorIndex.create(...)`	`await AsyncVectorIndex.create(...)`
`VectorIndex.fetch(...)`	`await AsyncVectorIndex.fetch(...)`
`for index in VectorIndex.list(...)`	`async for index in AsyncVectorIndex.list(...)`
`profile.narrate(...)`	`await async_profile.narrate(...)`

11.5.1 Async create vector index

```
import asyncio
import os

import select_ai

user = os.environ["SELECT_AI_USER"]
password = os.environ["SELECT_AI_PASSWORD"]
dsn = os.environ["SELECT_AI_DB_CONNECT_STRING"]
# Upload samples/data/refund-policy.txt to your bucket, then set its URI.
source_uri = os.environ["SELECT_AI_DOCUMENT_URI"]
provider_credential = os.getenv(
    "SELECT_AI_PROVIDER_CREDENTIAL", "my_oci_ai_profile_key"
)
storage_credential = os.environ["SELECT_AI_OBJECT_STORAGE_CREDENTIAL"]


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    try:
        provider = select_ai.OCIGenAIProvider(
            region=os.getenv("SELECT_AI_OCI_REGION", "us-chicago-1"),
            oci_apiformat="GENERIC",
            embedding_model=os.getenv(
                "SELECT_AI_EMBEDDING_MODEL", "cohere.embed-v4.0"
            ),
        )
        profile = await select_ai.AsyncProfile(
            profile_name="async_oci_vector_ai_profile",
            attributes=select_ai.ProfileAttributes(
                credential_name=provider_credential,
                provider=provider,
            ),
            description="Profile for the sample refund policy",
            replace=True,
        )
        vector_index = select_ai.AsyncVectorIndex(
            index_name="test_vector_index",
            attributes=select_ai.OracleVectorIndexAttributes(
                location=source_uri,
                object_storage_credential_name=storage_credential,
            ),
            description="Sample refund policy",
            profile=profile,
        )
        await vector_index.create(replace=True, wait_for_completion=True)
        print("Created vector index: test_vector_index")
    finally:
        await select_ai.async_disconnect()


asyncio.run(main())
```

output:

```
Created vector index: test_vector_index
```

11.5.2 Async list vector index

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    vector_index = select_ai.AsyncVectorIndex()
    async for index in vector_index.list(index_name_pattern="^test"):
        print("Vector index", index.index_name)
        print("Vector index profile", index.profile)


asyncio.run(main())
```

The script prints matching index names and their linked profiles. Results depend on your schema and configuration.

11.5.3 Async fetch vector index

You can fetch the vector index attributes and associated AI profile using the class method `AsyncVectorIndex.fetch(index_name)`

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_vector_index = await select_ai.AsyncVectorIndex.fetch(
        index_name="test_vector_index"
    )
    print(async_vector_index.attributes)
    print(async_vector_index.profile)
    print(await async_vector_index.get_next_refresh_timestamp())


asyncio.run(main())
```

The script prints the saved attributes and linked profile. Source URIs, credentials, and other values depend on the index you created.

11.5.4 Async update vector index attributes

To update attributes, use either `async_vector_index.set_attribute()` or `async_vector_index.set_attributes()`

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_vector_index = select_ai.AsyncVectorIndex(
        index_name="test_vector_index",
    )

    # Use vector_index.set_attributes to update a multiple attributes
    updated_attributes = select_ai.OracleVectorIndexAttributes(
        refresh_rate=1450
    )
    await async_vector_index.set_attributes(attributes=updated_attributes)

    # Use vector_index.set_attribute to update a single attribute
    await async_vector_index.set_attribute(
        attribute_name="similarity_threshold", attribute_value=0.5
    )
    print(async_vector_index.attributes)


asyncio.run(main())
```

The script updates the refresh rate and similarity threshold, then prints the attributes held by the Python object.

11.5.5 Async RAG using vector index

```
import os

import select_ai

user = os.environ["SELECT_AI_USER"]
password = os.environ["SELECT_AI_PASSWORD"]
dsn = os.environ["SELECT_AI_DB_CONNECT_STRING"]


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    try:
        profile = await select_ai.AsyncProfile(
            profile_name="async_oci_vector_ai_profile"
        )
        response = await profile.narrate(
            "How many days do customers have to request a refund?"
        )
        print(response)
    finally:
        await select_ai.async_disconnect()


asyncio.run(main())
```

The answer should identify the **30 calendar day** refund window. Exact wording and source formatting depend on the provider and profile settings. For example:

output:

```
Customers can request a refund within 30 calendar days of purchase.
```

If no answer is grounded in the policy, verify that the uploaded URI and storage credential are correct, that the initial load completed, and that this profile is linked to `test_vector_index`. Run the corresponding RAG script after creating the index. See *Delete vector index* to remove the sample index and its data; delete the sample profile separately when finished.

Share a vector index

While connected as the index owner, grant access to an existing index:

```
index = select_ai.VectorIndex.fetch("PRODUCT_DOCS")
index.grant_access("APP_USER")
```

In a separate connection as the recipient, use `VectorIndex.fetch("PRODUCT_DOCS", owner="APP_OWNER")`. It returns an object whose `qualified_name` is `APP_OWNER.PRODUCT_DOCS`. When access should end, run `index.revoke_access("APP_USER")` as the owner.

For an async application, use `AsyncVectorIndex` and await each operation:

```
# Run as the index owner.
index = await select_ai.AsyncVectorIndex.fetch("PRODUCT_DOCS")
await index.grant_access("APP_USER")

# Run later as the index owner, when access should end.
await index.revoke_access("APP_USER")
```

See *Sharing and ownership* for the recipient's async fetch example and connection setup.

CHAPTER

TWELVE

SYNTHETIC DATA

Synthetic data generation uses a Select AI profile to populate database tables with generated rows. It is useful for demos, development, testing, and prototyping when representative data is needed but production data should not be copied.

Synthetic data is inserted into the target tables in the connected database schema or in the schema identified by `owner_name` or `object_list`. Before running generation, make sure the connected user has privileges on the target tables and that the Select AI profile is configured with a provider and credential.

Use synthetic data generation with care in shared schemas. The API writes rows to the target tables; use dedicated test tables or schemas when experimenting.

12.1 Generation modes

Use `object_name` for a single target table:

```
attributes = select_ai.SyntheticDataAttributes(
    object_name="MOVIE",
    record_count=100,
    user_prompt="the release date for the movies should be in 2019",
)
```

Use `object_list` for multiple target tables in one request:

```
attributes = select_ai.SyntheticDataAttributes(
    object_list=[
        {
            "owner": "SH",
            "name": "MOVIE",
            "record_count": 100,
            "user_prompt": (
                "the release date for the movies should be in 2019"
            ),
        },
        {"owner": "SH", "name": "ACTOR", "record_count": 10},
        {"owner": "SH", "name": "DIRECTOR", "record_count": 5},
    ]
)
```

Exactly one of `object_name` or `object_list` must be set.

12.2 Generation parameters

Use `SyntheticDataParams` to control how generation is performed:

```
params = select_ai.SyntheticDataParams(
    sample_rows=100,
    table_statistics=True,
    priority="HIGH",
    comments=True,
)

attributes = select_ai.SyntheticDataAttributes(
    object_name="MOVIE",
    record_count=100,
    user_prompt="Generate movie data for releases in 2019.",
    params=params,
)
```

`sample_rows` controls how many existing rows are used as examples for the model. `table_statistics` and `comments` include additional table metadata. `priority` controls resource priority for generation work; supported values are `HIGH`, `MEDIUM`, and `LOW`. All parameters are optional. Parameters that are not supplied are omitted from the request, allowing the database to apply its defaults.

12.3 Sync and async APIs

Use `Profile.generate_synthetic_data(...)` for synchronous applications and `await AsyncProfile.generate_synthetic_data(...)` for asynchronous applications:

```
profile = select_ai.Profile(profile_name="oci_ai_profile")
profile.generate_synthetic_data(
    synthetic_data_attributes=attributes,
)
```

```
async_profile = await select_ai.AsyncProfile(
    profile_name="async_oci_ai_profile",
)
await async_profile.generate_synthetic_data(
    synthetic_data_attributes=attributes,
)
```

For additional database-side attribute details, see the generate_synthetic_data PL/SQL API.

12.3.1 SyntheticDataAttributes

class select_ai.**SyntheticDataAttributes**(*object_name: str | None = None*, *object_list: List[Mapping] | None = None*, *owner_name: str | None = None*, *params: SyntheticDataParams | None = None*, *record_count: int | None = None*, *user_prompt: str | None = None*)

Attributes to control generation of synthetic data

Parameters

- **object_name** (*str*) – Table name to populate synthetic data
- **object_list** (*List[Mapping]*) – Use this to generate synthetic data on multiple tables
- **owner_name** (*str*) – Database user who owns the referenced object. Default value is connected user's schema
- **record_count** (*int*) – Number of records to generate
- **user_prompt** (*str*) – User prompt to guide generation of synthetic data For e.g. "the release date for the movies should be in 2019"

dict(*exclude_null=True*)

json(*exclude_null=True*)

classmethod keys()

object_list: List[Mapping] | None = None

object_name: str | None = None

owner_name: str | None = None

params: *SyntheticDataParams* | None = None

prepare()

record_count: int | None = None

user_prompt: str | None = None

12.3.2 SyntheticDataParams

class select_ai.**SyntheticDataParams**(*sample_rows: int | None = None*, *table_statistics: bool | None = None*, *priority: str | None = None*, *comments: bool | None = None*)

Optional parameters to control generation of synthetic data

Parameters

- **sample_rows** (*int*) – number of rows from the table to use as a sample to guide the LLM in data generation
- **table_statistics** (*bool*) – Enable or disable the use of table statistics information. When omitted, the database default is used.
- **priority** (*str*) – Assign a priority value that defines the number of parallel requests sent to the LLM for generating synthetic data. Tasks with a higher priority will consume more database resources and complete faster. Possible values are: HIGH, MEDIUM, LOW. When omitted, the database default is used.
- **comments** (*bool*) – Enable or disable sending comments to the LLM to guide data generation. When omitted, the database default is used.

comments: bool | None = None

dict(*exclude_null=True*)

json(*exclude_null=True*)

classmethod keys()

priority: str | None = None

sample_rows: int | None = None

table_statistics: bool | None = None

12.3.3 Single table synthetic data

The below example shows single table synthetic data generation

Single Table Sync API

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
profile = select_ai.Profile(profile_name="oci_ai_profile")
synthetic_data_params = select_ai.SyntheticDataParams(
    sample_rows=100, table_statistics=True, priority="HIGH"
)
synthetic_data_attributes = select_ai.SyntheticDataAttributes(
    object_name="MOVIE",
    user_prompt="the release date for the movies should be in 2019",
    params=synthetic_data_params,
    record_count=100,
)
profile.generate_synthetic_data(
    synthetic_data_attributes=synthetic_data_attributes
)
```

output:

```
SQL> select count(*) from movie;

  COUNT(*)
----------
       100
```

Single Table Async API

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_profile = await select_ai.AsyncProfile(
        profile_name="async_oci_ai_profile",
    )
    synthetic_data_params = select_ai.SyntheticDataParams(
        sample_rows=100, table_statistics=True, priority="HIGH"
    )
    synthetic_data_attributes = select_ai.SyntheticDataAttributes(
        object_name="MOVIE",
        user_prompt="the release date for the movies should be in 2019",
        params=synthetic_data_params,
        record_count=100,
    )
    await async_profile.generate_synthetic_data(
        synthetic_data_attributes=synthetic_data_attributes
    )


asyncio.run(main())
```

output:

```
SQL> select count(*) from movie;

  COUNT(*)
----------
       100
```

12.3.4 Multi table synthetic data

The below example shows multi-table synthetic data generation

Multi table Sync API

```
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)
profile = select_ai.Profile(profile_name="oci_ai_profile")
synthetic_data_params = select_ai.SyntheticDataParams(
    sample_rows=100, table_statistics=True, priority="HIGH"
)
object_list = [
    {
        "owner": user,
        "name": "MOVIE",
        "record_count": 100,
        "user_prompt": "the release date for the movies should be in 2019",
    },
    {"owner": user, "name": "ACTOR", "record_count": 10},
    {"owner": user, "name": "DIRECTOR", "record_count": 5},
]
synthetic_data_attributes = select_ai.SyntheticDataAttributes(
    object_list=object_list, params=synthetic_data_params
)
profile.generate_synthetic_data(
    synthetic_data_attributes=synthetic_data_attributes
)
```

output:

```
SQL> select count(*) from actor;

  COUNT(*)
----------
    40

SQL> select count(*) from director;

  COUNT(*)
----------
    13

SQL> select count(*) from movie;

  COUNT(*)
----------
```

(continues on next page)

(continued from previous page)

```
        300
```

Multi table Async API

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_profile = await select_ai.AsyncProfile(
        profile_name="async_oci_ai_profile",
    )
    synthetic_data_params = select_ai.SyntheticDataParams(
        sample_rows=100, table_statistics=True, priority="HIGH"
    )
    object_list = [
        {
            "owner": user,
            "name": "MOVIE",
            "record_count": 100,
            "user_prompt": "the release date for the movies should be in 2019",
        },
        {"owner": user, "name": "ACTOR", "record_count": 10},
        {"owner": user, "name": "DIRECTOR", "record_count": 5},
    ]
    synthetic_data_attributes = select_ai.SyntheticDataAttributes(
        object_list=object_list, params=synthetic_data_params
    )
    await async_profile.generate_synthetic_data(
        synthetic_data_attributes=synthetic_data_attributes
    )


asyncio.run(main())
```

output:

```
SQL> select count(*) from actor;

  COUNT(*)
----------
    40

SQL> select count(*) from director;
```

(continues on next page)

(continued from previous page)

```
  COUNT(*)
----------
    13

SQL> select count(*) from movie;

  COUNT(*)
----------
       300
```

CHAPTER

THIRTEEN

SUMMARY

Summarization uses a Select AI profile to summarize inline text or content available from a URI. The profile supplies the AI provider, model, credential, and generation settings. The `summarize` APIs are available on both `Profile` and `AsyncProfile`.

Use one content source per call:

- `content` for inline text.
- `location_uri` for content available from a URL, object storage URI, or supported file location.

Use `credential_name` when the `location_uri` requires a database credential, such as object storage access. Use `prompt` to guide what the summary should focus on.

13.1 Inline content

```
profile = select_ai.Profile(profile_name="oci_ai_profile")

summary = profile.summarize(
    content="Long text to summarize...",
    prompt="Summarize the key business implications.",
)
print(summary)
```

13.2 Content from a URI

```
profile = select_ai.Profile(profile_name="oci_ai_profile")

summary = profile.summarize(
    location_uri="https://en.wikipedia.org/wiki/Astronomy",
)
print(summary)
```

13.3 Content from object storage

Pass `credential_name` when the target location requires authentication:

```
profile = select_ai.Profile(profile_name="oci_ai_profile")

summary = profile.summarize(
```

(continues on next page)

(continued from previous page)

```
    location_uri=(
        "https://objectstorage.us-ashburn-1.oraclecloud.com/"
        "n/namespace/b/bucket/o/document.txt"
    ),
    credential_name="OBJECT_STORE_CRED",
)
print(summary)
```

13.4 Summary parameters

Use `SummaryParams` to control output length, output style, chunk processing, and extractiveness:

```
params = select_ai.summary.SummaryParams(
    min_words=50,
    max_words=150,
    summary_style=select_ai.summary.Style.LIST,
    chunk_processing_method=(
        select_ai.summary.ChunkProcessingMethod.MAP_REDUCE
    ),
    extractiveness_level=select_ai.summary.ExtractivenessLevel.MEDIUM,
)

summary = profile.summarize(
    content="Long text to summarize...",
    params=params,
)
```

13.5 Async summary

```
async_profile = await select_ai.AsyncProfile(
    profile_name="async_oci_ai_profile",
)

summary = await async_profile.summarize(
    content="Long text to summarize...",
    prompt="Summarize the main points.",
)
print(summary)
```

13.6 Validation

`summarize` requires exactly one of `content` or `location_uri`. Passing both, or passing neither, raises an error.

13.6.1 SummaryParams

class select_ai.summary.**SummaryParams**(*min_words: int | None = None*, *max_words: int | None = None*, *summary_style: Style | None = None*, *chunk_processing_method: ChunkProcessingMethod | None = None*, *extractiveness_level: ExtractivenessLevel | None = None*)

Customize summary generation using these parameters

Parameters

- **min_words** (*int*) – approximate minimum number of words the generated summary is expected to contain.
- **max_words** (*int*) – approximate maximum number of words the generated summary is expected to contain.
- **summary_style** (select_ai.summary.Style) – Specifies the format style for the summary
- **chunk_processing_method** (select_ai.summary.ChunkProcessingMethod) – When the text exceeds the token limit that the LLM can process, it must be split into manageable chunks
- **extractiveness_level** (select_ai.summary.ExtractivenessLevel) – Determines how closely the summary follows the original wording of the input

chunk_processing_method: *ChunkProcessingMethod* | None = None

dict(*exclude_null=True*)

extractiveness_level: *ExtractivenessLevel* | None = None

json(*exclude_null=True*)

classmethod keys()

max_words: int | None = None

min_words: int | None = None

summary_style: *Style* | None = None

13.6.2 ChunkProcessingMethod

class select_ai.summary.**ChunkProcessingMethod**(**values*)

When the text exceeds the token limit that the LLM can process, it must be split into manageable chunks. This parameter enables you to choose the method for processing these chunks - ChunkProcessingMethod.ITERATIVE_REFINEMENT - ChunkProcessingMethod.MAP_REDUCE

ITERATIVE_REFINEMENT = 'iterative_refinement'

MAP_REDUCE = 'map_reduce'

capitalize()

Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold()

Return a version of the string suitable for caseless comparisons.

center(*width*, *fillchar=' '*, */ (Positional-only parameter separator (PEP 570))*)

Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

count()

Return the number of non-overlapping occurrences of substring sub in string S[start:end].

Optional arguments start and end are interpreted as in slice notation.

encode(*encoding='utf-8'*, *errors='strict'*)

Encode the string using the codec registered for encoding.

encoding
The encoding in which to encode the string.

errors
The error handling scheme to use for encoding errors. The default is 'strict' meaning that encoding errors raise a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith()

Return True if the string ends with the specified suffix, False otherwise.

suffix
A string or a tuple of strings to try.

start
Optional start position. Default: start of the string.

end
Optional stop position. Default: end of the string.

expandtabs(*tabsize=8*)

Return a copy where all tab characters are expanded using spaces.

If tabsize is not given, a tab size of 8 characters is assumed.

find()

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.

format(**args*, ***kwargs*)

Return a formatted version of the string, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map(*mapping*, /)

Return a formatted version of the string, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

index()

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Raises ValueError when the substring is not found.

isalnum()

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha()

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii()

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal()

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit()

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier()

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as "def" or "class".

islower()

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

isnumeric()

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

isprintable()

Return True if all characters in the string are printable, False otherwise.

A character is printable if repr() may use it in its output.

isspace()

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

istitle()

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

isupper()

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

join(*iterable*, /)

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'

ljust(*width*, *fillchar=' '*, /)

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

lower()

Return a copy of the string converted to lowercase.

lstrip(*chars=None*, /)

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

static maketrans()

Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

partition(*sep*, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original string and two empty strings.

removeprefix(*prefix*, /)

Return a str with the given prefix string removed if present.

If the string starts with the prefix string, return string[len(prefix):]. Otherwise, return a copy of the original string.

removesuffix(*suffix*, /)

Return a str with the given suffix string removed if present.

If the string ends with the suffix string and that suffix is not empty, return string[:-len(suffix)]. Otherwise, return a copy of the original string.

replace(*old*, *new*, /, *count=-1*)

Return a copy with all occurrences of substring old replaced by new.

count
Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

rfind()

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.

rindex()

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Raises ValueError when the substring is not found.

rjust(*width*, *fillchar=' '*, /)

Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

rpartition(*sep*, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty strings and the original string.

rsplit(*sep=None*, *maxsplit=-1*)

Return a list of the substrings in the string, using sep as the separator string.

sep
The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including n r t f and spaces) and will discard empty strings from the result.

maxsplit
Maximum number of splits. -1 (the default value) means no limit.

Splitting starts at the end of the string and works to the front.

rstrip(*chars=None*, /)

Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

`split`(*sep=None*, *maxsplit=-1*)

Return a list of the substrings in the string, using sep as the separator string.

sep
The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including n r t f and spaces) and will discard empty strings from the result.

maxsplit
Maximum number of splits. -1 (the default value) means no limit.

Splitting starts at the front of the string and works to the end.

Note, str.split() is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

`splitlines`(*keepends=False*)

Return a list of the lines in the string, breaking at line boundaries.

Line breaks are not included in the resulting list unless keepends is given and true.

`startswith`()

Return True if the string starts with the specified prefix, False otherwise.

prefix
A string or a tuple of strings to try.

start
Optional start position. Default: start of the string.

end
Optional stop position. Default: end of the string.

`strip`(*chars=None*, */*)

Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

`swapcase`()

Convert uppercase characters to lowercase and lowercase characters to uppercase.

`title`()

Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

`translate`(*table*, */*)

Replace each character in the string using the given translation table.

table
Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via __getitem__, for instance a dictionary or list. If this operation raises LookupError, the character is left untouched. Characters mapped to None are deleted.

`upper`()

Return a copy of the string converted to uppercase.

zfill(*width*, */*)

Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

13.6.3 ExtractivenessLevel

class select_ai.summary.**ExtractivenessLevel**(**values*)

Determines how closely the summary follows the original wording of the input. It controls the degree to which the model extracts versus rephrases it. The following are the options: - ExtractivenessLevel.LOW - ExtractivenessLevel.MEDIUM - ExtractivenessLevel.HIGH

HIGH = 'high'

LOW = 'low'

MEDIUM = 'medium'

capitalize()

Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold()

Return a version of the string suitable for caseless comparisons.

center(*width*, *fillchar=' '*, */*)

Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

count()

Return the number of non-overlapping occurrences of substring sub in string S[start:end].

Optional arguments start and end are interpreted as in slice notation.

encode(*encoding='utf-8'*, *errors='strict'*)

Encode the string using the codec registered for encoding.

encoding
The encoding in which to encode the string.

errors
The error handling scheme to use for encoding errors. The default is 'strict' meaning that encoding errors raise a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith()

Return True if the string ends with the specified suffix, False otherwise.

suffix
A string or a tuple of strings to try.

start
Optional start position. Default: start of the string.

end
Optional stop position. Default: end of the string.

expandtabs(*tabsize=8*)

Return a copy where all tab characters are expanded using spaces.

If tabsize is not given, a tab size of 8 characters is assumed.

find()

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.

format(**args*, ***kwargs*)

Return a formatted version of the string, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map(*mapping*, /)

Return a formatted version of the string, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

index()

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Raises ValueError when the substring is not found.

isalnum()

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha()

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii()

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal()

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit()

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier()

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as "def" or "class".

islower()

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

isnumeric()

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

isprintable()

Return True if all characters in the string are printable, False otherwise.

A character is printable if repr() may use it in its output.

isspace()

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

istitle()

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

isupper()

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

join(*iterable*, /)

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'

ljust(*width*, *fillchar=' '*, /)

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

lower()

Return a copy of the string converted to lowercase.

lstrip(*chars=None*, /)

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

static maketrans()

Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

partition(*sep*, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original string and two empty strings.

removeprefix(*prefix*, /)

Return a str with the given prefix string removed if present.

If the string starts with the prefix string, return string[len(prefix):]. Otherwise, return a copy of the original string.

removesuffix(*suffix*, /)

Return a str with the given suffix string removed if present.

If the string ends with the suffix string and that suffix is not empty, return string[:-len(suffix)]. Otherwise, return a copy of the original string.

replace(*old*, *new*, /, *count=-1*)

Return a copy with all occurrences of substring old replaced by new.

count
Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

rfind()

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.

rindex()

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Raises ValueError when the substring is not found.

rjust(*width*, *fillchar=' '*, /)

Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

rpartition(*sep*, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty strings and the original string.

rsplit(*sep=None*, *maxsplit=-1*)

Return a list of the substrings in the string, using sep as the separator string.

sep
The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including n r t f and spaces) and will discard empty strings from the result.

maxsplit
Maximum number of splits. -1 (the default value) means no limit.

Splitting starts at the end of the string and works to the front.

`rstrip`(*chars=None*, */*)

Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

`split`(*sep=None*, *maxsplit=-1*)

Return a list of the substrings in the string, using sep as the separator string.

sep
The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including n r t f and spaces) and will discard empty strings from the result.

maxsplit
Maximum number of splits. -1 (the default value) means no limit.

Splitting starts at the front of the string and works to the end.

Note, str.split() is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

`splitlines`(*keepends=False*)

Return a list of the lines in the string, breaking at line boundaries.

Line breaks are not included in the resulting list unless keepends is given and true.

`startswith`()

Return True if the string starts with the specified prefix, False otherwise.

prefix
A string or a tuple of strings to try.

start
Optional start position. Default: start of the string.

end
Optional stop position. Default: end of the string.

`strip`(*chars=None*, */*)

Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

`swapcase`()

Convert uppercase characters to lowercase and lowercase characters to uppercase.

`title`()

Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

`translate`(*table*, */*)

Replace each character in the string using the given translation table.

table
Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via __getitem__, for instance a dictionary or list. If this operation raises LookupError, the character is left untouched. Characters mapped to None are deleted.

`upper()`

Return a copy of the string converted to uppercase.

`zfill(`*width*`, /)`

Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

13.6.4 SummaryStyle

class select_ai.summary.Style(*values)

Specifies the format style for the summary. The following are the available summary format options: - Style.PARAGRAPH - the summary is presented in one or more paragraphs. - Style.LIST - the summary is a list of key points from the text.

LIST = 'list'

PARAGRAPH = 'paragraph'

capitalize()

Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold()

Return a version of the string suitable for caseless comparisons.

center(*width*, *fillchar=' '*, */*)

Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

count()

Return the number of non-overlapping occurrences of substring sub in string S[start:end].

Optional arguments start and end are interpreted as in slice notation.

encode(*encoding='utf-8'*, *errors='strict'*)

Encode the string using the codec registered for encoding.

encoding
The encoding in which to encode the string.

errors
The error handling scheme to use for encoding errors. The default is 'strict' meaning that encoding errors raise a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith()

Return True if the string ends with the specified suffix, False otherwise.

suffix
A string or a tuple of strings to try.

start
Optional start position. Default: start of the string.

end
Optional stop position. Default: end of the string.

expandtabs(*tabsize=8*)

Return a copy where all tab characters are expanded using spaces.

If tabsize is not given, a tab size of 8 characters is assumed.

find()

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.

format(**args*, ***kwargs*)

Return a formatted version of the string, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map(*mapping*, /)

Return a formatted version of the string, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

index()

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Raises ValueError when the substring is not found.

isalnum()

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha()

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii()

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal()

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit()

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier()

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as "def" or "class".

islower()

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

isnumeric()

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

isprintable()

Return True if all characters in the string are printable, False otherwise.

A character is printable if repr() may use it in its output.

isspace()

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

istitle()

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

isupper()

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

join(*iterable*, /)

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'

ljust(*width*, *fillchar=' '*, /)

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

lower()

Return a copy of the string converted to lowercase.

lstrip(*chars=None*, /)

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

static maketrans()

Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

partition(*sep*, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original string and two empty strings.

removeprefix(*prefix*, /)

Return a str with the given prefix string removed if present.

If the string starts with the prefix string, return string[len(prefix):]. Otherwise, return a copy of the original string.

removesuffix(*suffix*, /)

Return a str with the given suffix string removed if present.

If the string ends with the suffix string and that suffix is not empty, return string[:-len(suffix)]. Otherwise, return a copy of the original string.

replace(*old*, *new*, /, *count=-1*)

Return a copy with all occurrences of substring old replaced by new.

count
Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

rfind()

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.

rindex()

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Raises ValueError when the substring is not found.

rjust(*width*, *fillchar=' '*, /)

Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

rpartition(*sep*, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty strings and the original string.

rsplit(*sep=None*, *maxsplit=-1*)

Return a list of the substrings in the string, using sep as the separator string.

sep
The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including n r t f and spaces) and will discard empty strings from the result.

maxsplit
Maximum number of splits. -1 (the default value) means no limit.

Splitting starts at the end of the string and works to the front.

rstrip(*chars=None*, /)

Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

`split`(*sep=None*, *maxsplit=-1*)

Return a list of the substrings in the string, using sep as the separator string.

sep
The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including n r t f and spaces) and will discard empty strings from the result.

maxsplit
Maximum number of splits. -1 (the default value) means no limit.

Splitting starts at the front of the string and works to the end.

Note, str.split() is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

`splitlines`(*keepends=False*)

Return a list of the lines in the string, breaking at line boundaries.

Line breaks are not included in the resulting list unless keepends is given and true.

`startswith`()

Return True if the string starts with the specified prefix, False otherwise.

prefix
A string or a tuple of strings to try.

start
Optional start position. Default: start of the string.

end
Optional stop position. Default: end of the string.

`strip`(*chars=None*, */*)

Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

`swapcase`()

Convert uppercase characters to lowercase and lowercase characters to uppercase.

`title`()

Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

`translate`(*table*, */*)

Replace each character in the string using the given translation table.

table
Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via __getitem__, for instance a dictionary or list. If this operation raises LookupError, the character is left untouched. Characters mapped to None are deleted.

`upper`()

Return a copy of the string converted to uppercase.

zfill(*width*, /)

Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

CHAPTER

FOURTEEN

AI AGENT

`select_ai.agent` adds a thin Python layer over Oracle Autonomous Database's `DBMS_CLOUD_AI_AGENT` package so you can define tools, compose tasks, wire up agents and run teams from Python using the existing Select AI connection objects.

- Keep agent state and orchestration in the database
- Register callable tools (PL/SQL procedure or functions, SQL, external HTTP endpoints, Slack or Email notifications) and attach them to tasks
- Group agents into teams and invoke them with a single API call

Agent workflows build on the same setup used by profiles: connect to Oracle Database, create or reuse a Select AI profile, create credentials for any external service used by tools, and grant network access for external endpoints. See *Connection*, *Profile*, *Credential*, and *Privileges*.

The usual agent workflow is:

- Create tools that an agent can use.
- Create tasks that describe the work and list the tools available for that task.
- Create an agent with a role and an LLM profile.
- Create a team that pairs agents with tasks.
- Run the team with a user prompt.

Tools, tasks, agents, and teams are database objects. Use `replace=True` when you want to recreate an existing object with the same name, and `force=True` when cleanup should succeed even if the object does not exist.

14.1 Tool

A callable which Select AI agent can invoke to accomplish a certain task. Users can either register built-in tools or create a custom tool using a PL/SQL stored procedure.

Use focused tools with clear instructions. The task and agent prompts decide when tools are used, so tool names, descriptions, and instructions should be specific enough for the model to choose the right tool.

14.1.1 Supported Tools

Following class methods of `select_ai.agent.Tool` class can be used to create tools. Invoking them will create a proxy object in the Python layer and persist the tool in the Database using `DBMS_CLOUD_AI_AGENT.CREATE_TOOL`

Table 1: Select AI Agent Tools

Tool Type	Class Method	Arguments
`EMAIL`	`select_ai.agent.Tool.create_email_notification_tool`	• `tool_name` • `credential_name` • `recipient` • `sender` • `smtp_host`
`SQL`	`select_ai.agent.Tool.create_sql_tool`	• `tool_name` • `profile_name`
`SLACK`	`select_ai.agent.Tool.create_slack_notification_tool`	• `tool_name` • `credential_name` • `channel`
`WEBSEARCH`	`select_ai.agent.Tool.create_websearch_tool`	• `tool_name` • `credential_name`
`PL/SQL custom tool`	`select_ai.agent.Tool.create_pl_sql_tool`	• `tool_name` • `function`
`RAG`	`select_ai.agent.Tool.create_rag_tool`	• `tool_name` • `profile_name`

14.1.2 Tool selection

Table 2: When to use each tool

Tool type	Use case
`SQL`	Ask questions over database objects using a Select AI profile.
`RAG`	Answer questions using content indexed by a vector index profile.
`WEBSEARCH`	Search public web content using a web search credential.
`SLACK`	Send a Slack notification from an agent workflow.
`EMAIL`	Send an email notification from an agent workflow.
`PL/SQL custom tool`	Call a database procedure or function for application-specific work.

Notification and web search tools require credentials and network access for the external service. SQL and RAG tools require existing Select AI profiles.

class select_ai.agent.**ToolAttributes**(*instruction: str | None = None*, *function: str | None = None*, *tool_params:* ToolParams | *None = None*, *tool_inputs: List[Mapping] | None = None*, *tool_type:* ToolType | *None = None*)

AI Tool attributes

Parameters

- **instruction** (*str*) – Statement that describes what the tool should accomplish and how to do it. This text is included in the prompt sent to the LLM.
- **function** – Specifies the PL/SQL procedure or function to call when the tool is used
- **tool_params** (select_ai.agent.ToolParams) – Tool parameters for built-in tools
- **tool_inputs** (*List[Mapping]*) – Describes input arguments. Similar to column comments in a table. Each mapping can include keys such as `name` and `description`.

classmethod create(***kwargs*)

dict(*exclude_null=True*)

function: str | None = None

instruction: str | None = None

json(*exclude_null=True*)

classmethod keys()

tool_inputs: List[Mapping] | None = None

tool_params: *ToolParams* | None = None

tool_type: *ToolType* | None = None

class select_ai.agent.**ToolParams**(*_REQUIRED_FIELDS: List | None = None*, *credential_name: str | None = None*, *endpoint: str | None = None*, *notification_type:* NotificationType | *None = None*, *profile_name: str | None = None*, *recipient: str | None = None*, *sender: str | None = None*, *channel: str | None = None*, *smtp_host: str | None = None*, *subject: str | None = None*)

Parameters to register a built-in Tool

Parameters

- **credential_name** (*str*) – Used by SLACK, EMAIL and WEBSEARCH tools
- **endpoint** (*str*) – Send HTTP requests to this endpoint
- **select_ai.agent.NotificationType** – Either SLACK or EMAIL
- **profile_name** (*str*) – Name of AI profile to use
- **recipient** (*str*) – Recipient used for EMAIL notification
- **sender** (*str*) – Sender used for EMAIL notification
- **channel** (*str*) – Slack channel to use
- **smtp_host** (*str*) – SMTP host to use for EMAIL notification
- **subject** (*str*) – Email subject to use

channel: str | None = None

classmethod create(*, *tool_type:* ToolType | *None* = *None*, ***kwargs*)

credential_name: str | None = None

dict(*exclude_null=True*)

endpoint: str | None = None

json(*exclude_null=True*)

classmethod keys()

notification_type: *NotificationType* | None = None

profile_name: str | None = None

recipient: str | None = None

sender: str | None = None

smtp_host: str | None = None

subject: str | None = None

class select_ai.agent.**Tool**(*tool_name: str | None = None*, *description: str | None = None*, *attributes: ToolAttributes | None = None*)

create(*enabled: bool | None = True*, *replace: bool | None = False*)

classmethod **create_built_in_tool**(*tool_name: str*, *tool_params: ToolParams*, *tool_type: ToolType*, *description: str | None = None*, *replace: bool | None = False*, *instruction: str | None = None*) → *Tool*

Register a built-in tool

Parameters

- **tool_name** (*str*) – The name of the tool
- **tool_params** (select_ai.agent.ToolParams) – Parameters required by built-in tool
- **tool_type** (select_ai.agent.tool.ToolType) – The built-in tool type
- **description** (*str*) – Description of the tool
- **replace** (*bool*) – Whether to replace the existing tool. Default value is False
- **instruction** (*str*) – A clear, concise statement that describes what the tool should accomplish and how to do it. This text is included in the prompt sent to the LLM.

Returns

select_ai.agent.Tool

classmethod **create_email_notification_tool**(*tool_name: str*, *credential_name: str*, *recipient: str*, *sender: str*, *smtp_host: str*, *description: str | None*, *subject: str | None = None*, *replace: bool = False*, *instruction: str | None = None*) → *Tool*

Register an email notification tool

Parameters

- **tool_name** (*str*) – The name of the tool
- **credential_name** (*str*) – The name of the credential
- **recipient** (*str*) – The recipient of the email
- **sender** (*str*) – The sender of the email
- **smtp_host** (*str*) – The SMTP host of the email server
- **description** (*str*) – The description of the tool
- **subject** (*str*) – Subject of the email.
- **replace** (*bool*) – Whether to replace the existing tool. Default value is False
- **instruction** (*str*) – A clear, concise statement that describes what the tool should accomplish and how to do it. This text is included in the prompt sent to the LLM.

Returns

select_ai.agent.Tool

classmethod **create_pl_sql_tool**(*tool_name: str*, *function: str*, *description: str | None = None*, *replace: bool = False*, *instruction: str | None = None*) → *Tool*

Create a custom tool to invoke PL/SQL procedure or function

Parameters

- **tool_name** (*str*) – The name of the tool

- **function** (*str*) – The name of the PL/SQL procedure or function
- **description** (*str*) – The description of the tool
- **replace** (*bool*) – Whether to replace existing tool. Default value is False
- **instruction** (*str*) – A clear, concise statement that describes what the tool should accomplish and how to do it. This text is included in the prompt sent to the LLM.

classmethod create_rag_tool(*tool_name: str*, *profile_name: str*, *description: str | None = None*, *replace: bool = False*, *instruction: str | None = None*) → *Tool*

Register a RAG tool, which will use a VectorIndex linked AI Profile

Parameters

- **tool_name** (*str*) – The name of the tool
- **profile_name** (*str*) – The name of the profile to use for Vector Index based RAG
- **description** (*str*) – The description of the tool
- **replace** (*bool*) – Whether to replace existing tool. Default value is False
- **instruction** (*str*) – A clear, concise statement that describes what the tool should accomplish and how to do it. This text is included in the prompt sent to the LLM.

classmethod create_slack_notification_tool(*tool_name: str*, *credential_name: str*, *channel: str*, *description: str | None = None*, *replace: bool = False*, *instruction: str | None = None*) → *Tool*

Register a Slack notification tool

Parameters

- **tool_name** (*str*) – The name of the Slack notification tool
- **credential_name** (*str*) – The name of the Slack credential
- **channel** (*str*) – The name of the Slack channel
- **description** (*str*) – The description of the Slack notification tool
- **replace** (*bool*) – Whether to replace existing tool. Default value is False
- **instruction** (*str*) – A clear, concise statement that describes what the tool should accomplish and how to do it. This text is included in the prompt sent to the LLM.

classmethod create_sql_tool(*tool_name: str*, *profile_name: str*, *description: str | None = None*, *replace: bool = False*, *instruction: str | None = None*) → *Tool*

Register a SQL tool to perform natural language to SQL translation

Parameters

- **tool_name** (*str*) – The name of the tool
- **profile_name** (*str*) – The name of the profile to use for SQL translation
- **description** (*str*) – The description of the tool
- **replace** (*bool*) – Whether to replace existing tool. Default value is False
- **instruction** (*str*) – A clear, concise statement that describes what the tool should accomplish and how to do it. This text is included in the prompt sent to the LLM.

classmethod create_websearch_tool(*tool_name: str*, *credential_name: str*, *description: str | None*, *replace: bool = False*, *instruction: str | None = None*) → *Tool*

Register a built-in websearch tool to search information on the web

Parameters

- **tool_name** (*str*) – The name of the tool
- **credential_name** (*str*) – The name of the credential object storing OpenAI credentials
- **description** (*str*) – The description of the tool
- **replace** (*bool*) – Whether to replace the existing tool
- **instruction** (*str*) – A clear, concise statement that describes what the tool should accomplish and how to do it. This text is included in the prompt sent to the LLM.

delete(*force: bool = False*)

Delete AI Tool from the database

Parameters

force (*bool*) – Force the deletion. Default value is False.

classmethod delete_tool(*tool_name: str*, *force: bool = False*)

Class method to delete AI Tool from the database

Parameters

- **tool_name** (*str*) – The name of the tool
- **force** (*bool*) – Force the deletion. Default value is False.

describe_tool() → str | None

Return this tool's JSON metadata and function arguments.

disable()

Disable AI Tool

enable()

Enable AI Tool

classmethod fetch(*tool_name: str*) → *Tool*

Fetch AI Tool attributes from the Database and build a proxy object in the Python layer

Parameters

tool_name (*str*) – The name of the AI Task

Returns

select_ai.agent.Tool

Raises

select_ai.errors.AgentToolNotFoundError – If the AI Tool is not found

classmethod list(*tool_name_pattern: str = '.*'*) → Iterator[*Tool*]

List AI Tools

Parameters

tool_name_pattern (*str*) – Regular expressions can be used to specify a pattern. Function REGEXP_LIKE is used to perform the match. Default value is ".*" i.e. match all tool name.

Returns

Iterator[Tool]

run_tool(*input: str*) → str | None

Run this tool directly and return its result.

Parameters
input (*str*) – Tool input payload.

Returns
The tool result.

Return type
str or None

set_attribute(*attribute_name: str*, *attribute_value: Any*) → None

Set the attribute of the AI Agent tool specified by *attribute_name* and *attribute_value*.

set_attributes(*attributes:* ToolAttributes) → None

Set the attributes of the AI Agent tool

14.1.3 Create Tool

The following example shows creation of an AI agent tool to perform natural language translation to SQL using an OCI AI profile

```
import os
from pprint import pformat

import select_ai
import select_ai.agent

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)

profile_attributes = select_ai.ProfileAttributes(
    credential_name="my_oci_ai_profile_key",
    object_list=[
        {"owner": user, "name": "MOVIE"},
        {"owner": user, "name": "ACTOR"},
        {"owner": user, "name": "DIRECTOR"},
    ],
    provider=select_ai.OCIGenAIProvider(
        region="us-chicago-1",
        oci_apiformat="GENERIC",
        model="meta.llama-4-maverick-17b-128e-instruct-fp8",
    ),
)
profile = select_ai.Profile(
    profile_name="LLAMA_4_MAVERICK",
    attributes=profile_attributes,
    description="MY OCI AI Profile",
    replace=True,
)

# Use the OCI AI Profile to perform natural
# language SQL translation
sql_tool = select_ai.agent.Tool.create_sql_tool(
    tool_name="MOVIE_SQL_TOOL",
    description="My Select AI MOVIE SQL agent tool",
    profile_name="LLAMA_4_MAVERICK",
    replace=True,
)
print(sql_tool.tool_name)
print(pformat(sql_tool.attributes))
```

output:

```
MOVIE_SQL_TOOL

ToolAttributes(instruction=None,
               function=None,
               tool_params=SQLToolParams(_REQUIRED_FIELDS=None,
                                         credential_name=None,
                                         endpoint=None,
                                         notification_type=None,
                                         profile_name='oci_ai_profile',
                                         recipient=None,
                                         sender=None,
                                         channel=None,
                                         smtp_host=None),
               tool_inputs=None,
               tool_type=<ToolType.SQL: 'SQL'>)
```

14.1.4 List Tools

```
import os

import select_ai
import select_ai.agent

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)

for tool in select_ai.agent.Tool.list():
    print(tool.tool_name)
```

output:

```
WEB_SEARCH_TOOL
MOVIE_SQL_TOOL
LLM_CHAT_TOOL
```

14.1.5 Inspect and run a tool

Use `Tool.describe_tool()` to return JSON metadata for a tool, including its function arguments. Use `Tool.run_tool(input)` to invoke a tool directly with an input payload. Direct execution is useful for testing a tool independently of a team workflow.

```
tool = Tool.fetch("MOVIE_SQL_TOOL")
print(tool.describe_tool())
print(tool.run_tool('{"query": "How many movies are there?"}'))
```

The complete example creates a temporary PL/SQL tool, describes it, invokes it, and removes the temporary database objects:

```

import os
import uuid

import select_ai
from select_ai.agent import Tool

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)

suffix = uuid.uuid4().hex.upper()
function_name = f"SAMPLE_CALCULATE_AGE_{suffix}"
tool_name = f"SAMPLE_AGE_TOOL_{suffix}"

with select_ai.cursor() as cursor:
    cursor.execute(
        f"""
        CREATE OR REPLACE FUNCTION {function_name}(p_birth_date DATE)
        RETURN NUMBER IS
        BEGIN
            RETURN TRUNC(MONTHS_BETWEEN(SYSDATE, p_birth_date) / 12);
        END;
        """
    )

tool = Tool.create_pl_sql_tool(
    tool_name=tool_name,
    function=function_name,
    description="Calculate age from a birth date",
)

try:
    print("Tool description:")
    print(tool.describe_tool())
    print("Tool result:")
    print(tool.run_tool('{"p_birth_date":"2000-01-01"}'))
finally:
    tool.delete(force=True)
```

(continues on next page)

(continued from previous page)

```
    with select_ai.cursor() as cursor:
        cursor.execute(f"DROP FUNCTION {function_name}")
```

The generated tool name and calculated age vary between runs. Representative output is:

output:

```
Tool description:
{"tool_name": "SAMPLE_AGE_TOOL_<generated suffix>", ...}
Tool result:
<calculated age in years>
```

14.2 Task

Each task is identified by a `task_name` and includes a set of attributes that guide the agent's behavior during execution. Key attributes include the `instruction`, which describes the task's purpose and provides context for the agent to reason about when and how to use it, and the `tools` list, which specifies which tools the agent can choose from to accomplish the task. An optional `input` field allows a task to depend on the output of prior tasks, enabling task chaining and multi-step workflows.

class select_ai.agent.**TaskAttributes**(*instruction: str*, *tools: List[str] | None = None*, *input: str | None = None*, *enable_human_tool: bool | None = True*)

AI Task attributes

Parameters

- **instruction** (*str*) – Statement describing what the task is meant to accomplish
- **tools** (*List[str]*) – List of tools the agent can use to execute the task
- **input** (*str*) – Task name whose output will be automatically provided by select ai to LLM
- **enable_human_tool** (*bool*) – Enable agent to ask question to user when it requires information or clarification during a task. Default value is True.

dict(*exclude_null=True*)

enable_human_tool: bool | None = True

input: str | None = None

instruction: str

json(*exclude_null=True*)

classmethod keys()

tools: List[str] | None = None

class select_ai.agent.**Task**(*task_name: str | None = None*, *description: str | None = None*, *attributes: TaskAttributes | None = None*)

select_ai.agent.Task class lets you create, delete, enable, disable and list AI Tasks

Parameters

- **task_name** (*str*) – The name of the AI task
- **description** (*str*) – Optional description of the AI task
- **attributes** (*select_ai.agent.TaskAttributes*) – AI task attributes

create(*enabled: bool | None = True*, *replace: bool | None = False*)

Create a task that a Select AI agent can include in its reasoning process

Parameters

- **enabled** (*bool*) – Whether the AI Task should be enabled. Default value is True.
- **replace** (*bool*) – Whether the AI Task should be replaced. Default value is False.

delete(*force: bool = False*)

Delete AI Task from the database

Parameters

force (*bool*) – Force the deletion. Default value is False.

classmethod delete_task(*task_name: str*, *force: bool = False*)

Class method to delete AI Task from the database

Parameters

- **task_name** (*str*) – The name of the AI Task
- **force** (*bool*) – Force the deletion. Default value is False.

disable()

Disable AI Task

enable()

Enable AI Task

classmethod fetch(*task_name: str*) → *Task*

Fetch AI Task attributes from the Database and build a proxy object in the Python layer

Parameters

task_name (*str*) – The name of the AI Task

Returns

select_ai.agent.Task

Raises

select_ai.errors.AgentTaskNotFoundError – If the AI Task is not found

classmethod list(*task_name_pattern: str | None = '.*'*) → Iterator[*Task*]

List AI Tasks

Parameters

task_name_pattern (*str*) – Regular expressions can be used to specify a pattern. Function REGEXP_LIKE is used to perform the match. Default value is ".*" i.e. match all tasks.

Returns

Iterator[Task]

set_attribute(*attribute_name: str*, *attribute_value: Any*)

Set a single AI Task attribute specified using name and value

Parameters

- **attribute_name** (*str*) – The name of the AI Task attribute
- **attribute_value** (*str*) – The value of the AI Task attribute

set_attributes(*attributes:* TaskAttributes)

Set AI Task attributes

Parameters

attributes (select_ai.agent.TaskAttributes) – Multiple attributes can be specified by passing a TaskAttributes object

14.2.1 Create Task

In the following task, we use the `MOVIE_SQL_TOOL` created in the previous step

The `instruction` is the main task prompt. Use placeholders such as `{query}` when the user prompt should be inserted into the task. The `tools` list limits which tools the agent can use for the task.

```
import os
from pprint import pformat

import select_ai
import select_ai.agent
from select_ai.agent import Task, TaskAttributes

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
select_ai.connect(user=user, password=password, dsn=dsn)

task = Task(
    task_name="ANALYZE_MOVIE_TASK",
    description="Search for movies in the database",
    attributes=TaskAttributes(
        instruction="Help the user with their request about movies. "
        "User question: {query}. "
        "You can use SQL tool to search the data from database",
        tools=["MOVIE_SQL_TOOL"],
        enable_human_tool=False,
    ),
)
task.create(replace=True)
print(task.task_name)
print(pformat(task.attributes))
```

output:

```
ANALYZE_MOVIE_TASK

TaskAttributes(instruction='Help the user with their request about movies. '
                           'User question: {query}. You can use SQL tool to '
                           'search the data from database',
               tools=['MOVIE_SQL_TOOL'],
               input=None,
               enable_human_tool=False)
```

14.2.2 List Tasks

```
import os

import select_ai
import select_ai.agent

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)

for task in select_ai.agent.Task.list():
    print(task.task_name)
```

output:

```
WEB_SEARCH_TASK
ANALYZE_MOVIE_TASK
```

14.3 Agent

A Select AI Agent is defined using `agent_name`, its `attributes` and an optional description. The attributes must include key agent properties such as `profile_name` which specifies the LLM profile used for prompt generation and `role`, which outlines the agent's intended role and behavioral context.

The agent profile supplies the model used for planning and reasoning. The role should describe the agent's domain, boundaries, and expected behavior. Keep the role specific to the tasks assigned to the agent.

class select_ai.agent.**AgentAttributes**(*profile_name: str*, *role: str*, *enable_human_tool: bool | None = True*, *supervisor: bool | None = None*)

AI Agent Attributes

Parameters

- **profile_name** (*str*) – Name of the AI Profile which agent will use to send request to LLM
- **role** (*str*) – Agent's role also sent to LLM
- **enable_human_tool** (*bool*) – Enable human tool support. Agent will ask question to the user for any clarification
- **supervisor** (*bool*) – Whether the agent supervises other agents in a supervised team.

dict(*exclude_null=True*)

enable_human_tool: bool | None = True

json(*exclude_null=True*)

classmethod keys()

profile_name: str

role: str

supervisor: bool | None = None

class select_ai.agent.**Agent**(*agent_name: str | None = None*, *description: str | None = None*, *attributes: AgentAttributes | None = None*)

select_ai.agent.Agent class lets you create, delete, enable, disable and list AI agents

Parameters

- **agent_name** (*str*) – The name of the AI Agent
- **description** (*str*) – Optional description of the AI agent
- **attributes** (select_ai.agent.AgentAttributes) – AI agent attributes

create(*enabled: bool | None = True*, *replace: bool | None = False*)

Register a new AI Agent within the Select AI framework

Parameters

- **enabled** (*bool*) – Whether the AI Agent should be enabled. Default value is True.
- **replace** (*bool*) – Whether the AI Agent should be replaced. Default value is False.

delete(*force: bool | None = False*)

Delete AI Agent from the database

Parameters

force (*bool*) – Force the deletion. Default value is False.

classmethod **delete_agent**(*agent_name: str*, *force: bool | None = False*)

Class method to delete AI Agent from the database

Parameters

- **agent_name** (*str*) – The name of the AI Agent
- **force** (*bool*) – Force the deletion. Default value is False.

disable()

Disable AI Agent

enable()

Enable AI Agent

classmethod **fetch**(*agent_name: str*) → *Agent*

Fetch AI Agent attributes from the Database and build a proxy object in the Python layer

Parameters

agent_name (*str*) – The name of the AI Agent

Returns

select_ai.agent.Agent

Raises

select_ai.errors.AgentNotFoundError – If the AI Agent is not found

classmethod **list**(*agent_name_pattern: str | None = '.*'*) → Iterator[*Agent*]

List AI agents matching a pattern

Parameters

agent_name_pattern (*str*) – Regular expressions can be used to specify a pattern. Function REGEXP_LIKE is used to perform the match. Default value is ".*" i.e. match all agent names.

Returns

Iterator[Agent]

set_attribute(*attribute_name: str*, *attribute_value: Any*) → None

Set a single AI Agent attribute specified using name and value

set_attributes(*attributes:* AgentAttributes) → None

Set AI Agent attributes

Parameters

attributes (select_ai.agent.AgentAttributes) – Multiple attributes can be specified by passing an AgentAttributes object

14.3.1 Create Agent

```
import os

import select_ai
from select_ai.agent import (
    Agent,
    AgentAttributes,
)

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
select_ai.connect(user=user, password=password, dsn=dsn)
agent_attributes = AgentAttributes(
    profile_name="LLAMA_4_MAVERICK",
    role="You are an AI Movie Analyst. "
    "Your can help answer a variety of questions related to movies. ",
    enable_human_tool=False,
)
agent = Agent(
    agent_name="MOVIE_ANALYST",
    attributes=agent_attributes,
)
agent.create(enabled=True, replace=True)
print("Created Agent:", agent)
```

output:

```
Created Agent: Agent(agent_name=MOVIE_ANALYST,
attributes=AgentAttributes(profile_name='LLAMA_4_MAVERICK',
role='You are an AI Movie Analyst.
Your can help answer a variety of questions related to movies. ',
enable_human_tool=False), description=None)
```

14.4 Team

AI Agent Team coordinates the execution of multiple agents working together to fulfill a user request. Each team is uniquely identified by a `team_name` and configured through a set of `attributes` that define its composition and execution strategy. The `agents` attribute specifies an array of agent-task pairings, allowing users to assign specific tasks to designated agents. User can perform multiple tasks by assigning the same agent to different tasks. The `process` attribute defines how tasks should be executed.

Currently, `process="sequential"` is used to execute task assignments in order. Reuse the same agent in multiple team entries when the agent should perform multiple tasks.

For example:

```
attributes = TeamAttributes(
    agents=[
        {"name": "MOVIE_ANALYST", "task": "ANALYZE_MOVIE_TASK"},
        {"name": "MOVIE_ANALYST", "task": "SUMMARIZE_MOVIE_TASK"},
    ],
    process="sequential",
)
```

class select_ai.agent.**TeamAttributes**(*agents: List[Mapping]*, *process: str = 'sequential'*, *supervisor_agent: str | None = None*, *supervisor_task: str | None = None*)

AI agent team attributes

Parameters

- **agents** (*List[Mapping]*) – A List of Python dictionaries, each defining the agent and the task name. [{"name": "<agent_name>", "task": "<task_name>"}]
- **process** (*str*) – Execution order of tasks. Currently only "sequential" is supported.
- **supervisor_agent** (*str*) – Name of the agent that supervises the team when using a supervised execution process.
- **supervisor_task** (*str*) – Database-generated task associated with the supervisor agent.

agents: List[Mapping]

dict(*exclude_null=True*)

json(*exclude_null=True*)

classmethod keys()

process: str = 'sequential'

supervisor_agent: str | None = None

supervisor_task: str | None = None

14.4.1 Supervised teams

A supervised team uses a dedicated agent to coordinate the other agents in the team. Set `supervisor=True` on that agent's `AgentAttributes` and pass the agent name in `TeamAttributes.supervisor_agent`. The worker agent and task assignments remain in the team's `agents` list.

supervisor_task is created by the database when the team is created. Do not set it when constructing TeamAttributes. Fetch the team after creation to read the generated value from fetched.attributes.supervisor_task.

```
from select_ai.agent import Agent, AgentAttributes, Team, TeamAttributes

# MOVIE_ANALYST and ANALYZE_MOVIE_TASK must already exist.
supervisor = Agent(
    agent_name="MOVIE_SUPERVISOR",
    attributes=AgentAttributes(
        profile_name="oci_ai_profile",
        role="You supervise and coordinate the team.",
        supervisor=True,
    ),
)

supervisor.create()

team = Team(
    team_name="MOVIE_AGENT_TEAM",
    attributes=TeamAttributes(
        agents=[
            {"name": "MOVIE_ANALYST", "task": "ANALYZE_MOVIE_TASK"}
        ],
        process="sequential",
        supervisor_agent=supervisor.agent_name,
    ),
)
team.create()
```

Agent and team constructors build Python objects; create() persists them in Oracle Database. The example uses new names and raises an error if those objects already exist. Use replace=True on create() only when you intend to replace existing definitions.

The complete example creates a supervised team, runs a prompt through the supervisor workflow, and inspects the generated supervisor task and team metadata. Team.run() starts the team workflow; the configured supervisor agent coordinates the worker agent as part of that call.

Set the common database connection variables and SELECT_AI_PROFILE_NAME (default oci_ai_profile) to an existing profile. Run python samples/agent/team_supervisor_inspect.py from the repository root. The script removes the objects and conversation it creates during cleanup.

First, configure the connection and define the task and worker agent:

```
import os
import uuid

import select_ai
from select_ai.agent import (
    Agent,
    AgentAttributes,
    Task,
    TaskAttributes,
    Team,
    TeamAttributes,
)
from select_ai.conversation import Conversation, ConversationAttributes

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
profile_name = os.getenv("SELECT_AI_PROFILE_NAME", "oci_ai_profile")
suffix = uuid.uuid4().hex.upper()

select_ai.connect(user=user, password=password, dsn=dsn)

task = Task(
    task_name=f"SAMPLE_TASK_{suffix}",
    attributes=TaskAttributes(
        instruction="Answer the user's question: {query}",
        tools=[],
        enable_human_tool=False,
    ),
)
worker = Agent(
    agent_name=f"SAMPLE_WORKER_{suffix}",
    attributes=AgentAttributes(
        profile_name=profile_name,
        role="You answer user questions.",
        enable_human_tool=False,
    ),
)
```

Append the supervisor, team, and conversation definitions:

```
supervisor = Agent(
    agent_name=f"SAMPLE_SUPERVISOR_{suffix}",
    attributes=AgentAttributes(
        profile_name=profile_name,
        role="You supervise and coordinate the team.",
        enable_human_tool=False,
        supervisor=True,
    ),
)
team = Team(
    team_name=f"SAMPLE_TEAM_{suffix}",
    attributes=TeamAttributes(
        agents=[{"name": worker.agent_name, "task": task.task_name}],
        process="sequential",
        supervisor_agent=supervisor.agent_name,
    ),
)

conversation = Conversation(
    attributes=ConversationAttributes(
        title="Supervised team sample",
        description="Conversation for the supervised team sample",
    )
)
```

Finally, create the objects, run the team, inspect its metadata, and clean up:

```
created_objects = []
conversation_created = False
try:
    task.create(replace=True)
    created_objects.append(task)
    worker.create(replace=True)
    created_objects.append(worker)
    supervisor.create(replace=True)
    created_objects.append(supervisor)
    team.create(replace=True)
    created_objects.append(team)

    conversation.create()
    conversation_created = True
    response = team.run(
        prompt="What is the capital of France? Answer in one sentence.",
        params={"conversation_id": conversation.conversation_id},
    )
    if response is None or (
        isinstance(response, str) and response.startswith("Task failed:")
    ):
        raise RuntimeError(f"Supervised team run failed: {response}")
    print("Team response:", response)

    fetched = Team.fetch(team.team_name)
    print("Supervisor agent:", fetched.attributes.supervisor_agent)
    print("Supervisor task:", fetched.attributes.supervisor_task)
    print("Team description:", team.describe_team())
    print("Team tools:", team.list_tools())
finally:
    if conversation_created:
        conversation.delete(force=True)
    for obj in reversed(created_objects):
        obj.delete(force=True)
```

Example output (generated names, model responses, and JSON metadata vary):

output:

```
Team response: <answer from the supervised team>
Supervisor agent: SAMPLE_SUPERVISOR_<generated suffix>
Supervisor task: <generated supervisor task>
Team description: <JSON team metadata>
Team tools: <JSON tool metadata>
```

class select_ai.agent.**Team**(*team_name: str*, *attributes:* TeamAttributes | *None* = *None*, *description: str* | *None* = *None*)

A Team of AI agents work together to accomplish tasks select_ai.agent.Team class lets you create, delete, enable, disable and list AI Tasks.

Parameters

- **team_name** (*str*) – The name of the AI team
- **description** (*str*) – Optional description of the AI team
- **attributes** (select_ai.agent.TeamAttributes) – AI team attributes

create(*enabled: bool* | *None* = *True*, *replace: bool* | *None* = *False*)

Create a team of AI agents that work together to accomplish tasks.

Parameters

- **enabled** (*bool*) – Whether the AI agent team should be enabled. Default value is True.
- **replace** (*bool*) – Whether the AI agent team should be replaced. Default value is False.

delete(*force: bool* | *None* = *False*)

Delete an AI agent team from the database

Parameters

force (*bool*) – Force the deletion. Default value is False.

classmethod delete_team(*team_name: str*, *force: bool* | *None* = *False*)

Class method to delete an AI agent team from the database

Parameters

- **team_name** (*str*) – The name of the AI team
- **force** (*bool*) – Force the deletion. Default value is False.

describe_team() → str | None

Return this team's JSON metadata and aggregated skills.

disable()

Disable the AI agent team

enable()

Enable the AI agent team

export(*object_storage_credential_name: str* | *None* = *None*, *location: str* | *None* = *None*, *params: str* | *Mapping* | *None* = *None*) → str | None

Export this AI agent team specification.

If object storage details are provided, the specification is written to the given location and None is returned. Otherwise, the specification is returned as a string.

Parameters

- **object_storage_credential_name** (*str*) – Optional credential name used to write the exported specification to object storage. Must be specified together with `location`.
- **location** (*str*) – Optional object storage URI where the exported specification should be written. Must be specified together with `object_storage_credential_name`.
- **params** (*str or Mapping*) – Optional export parameters. May be a JSON string or a Python mapping.

Returns
Exported team specification as a JSON string when exporting inline, or None when exporting to object storage.

Return type
str or None

classmethod export_team(*team_name: str*, *object_storage_credential_name: str | None = None*, *location: str | None = None*, *params: str | Mapping | None = None*) → str | None

Export an AI agent team specification.

If object storage details are provided, the specification is written to the given location and None is returned. Otherwise, the specification is returned as a string.

Parameters

- **team_name** (*str*) – Name of the AI agent team to export.
- **object_storage_credential_name** (*str*) – Optional credential name used to write the exported specification to object storage. Must be specified together with `location`.
- **location** (*str*) – Optional object storage URI where the exported specification should be written. Must be specified together with `object_storage_credential_name`.
- **params** (*str or Mapping*) – Optional export parameters. May be a JSON string or a Python mapping.

Returns
Exported team specification as a JSON string when exporting inline, or None when exporting to object storage.

Return type
str or None

classmethod fetch(*team_name: str*) → *Team*

Fetch AI Team attributes from the Database and build a proxy object in the Python layer

Parameters
team_name (*str*) – The name of the AI Team

Returns
select_ai.agent.Team

Raises
select_ai.errors.AgentTeamNotFoundError – If the AI Team is not found

grant_access(*user_or_role_name: str*) → None

Grant a database user or role access to this agent team.

classmethod import_team(*profile_name: str*, *team_name: str | None = None*, *specification: str | Mapping | None = None*, *object_storage_credential_name: str | None = None*, *location: str | None = None*, *force: bool | None = False*, *params: str | Mapping | None = None*) → None

Import an AI agent team specification and create the associated team, agents, tasks, and tools in the database.

Parameters

- **profile_name** (*str*) – Name of the Select AI profile to use for the imported team and agents in the target database.
- **team_name** (*str*) – Optional name for the imported team. If omitted, the team name from the specification is used.

- **specification** (*str or Mapping*) – Team specification to import. May be a JSON string or a Python mapping. Omit this when importing from object storage.
- **object_storage_credential_name** (*str*) – Optional credential name used to read the specification from object storage. Must be specified together with `location`.
- **location** (*str*) – Optional object storage URI of the specification to import. Must be specified together with `object_storage_credential_name`.
- **force** (*bool*) – Whether to replace conflicting database objects during import. Default value is False.
- **params** (*str or Mapping*) – Optional import parameters. May be a JSON string or a Python mapping.

classmethod list(*team_name_pattern: str | None = '.*'*) → Iterator[*Team*]

List AI Agent Teams

Parameters

team_name_pattern (*str*) – Regular expressions can be used to specify a pattern. Function REGEXP_LIKE is used to perform the match. Default value is ".*" i.e. match all teams.

Returns

Iterator[Team]

list_tools() → str | None

Return JSON metadata for the tools available to this team.

revoke_access(*user_or_role_name: str*) → None

Revoke a database user or role's access to this agent team.

run(*prompt: str = None*, *params: Mapping = None*)

Start a new AI agent team or resume a paused one that is waiting for human input. If you provide an existing process ID and the associated team process is in the WAITING_FOR_HUMAN state, the function resumes the workflow using the input you provide as the human response

Parameters

- **prompt** (*str*) – Optional prompt for the user. If the task is in the RUNNING state, the input acts as a placeholder for the {query} in the task instruction. If the task is in the WAITING_FOR_HUMAN state, the input serves as the human response.
- **params** (*Mapping[str, str]*) – Optional parameters for the task. Supported keys include `conversation_id`, which identifies the conversation session associated with the agent team, and `variables`, which provides additional key-value input to the agent team.

set_attribute(*attribute_name: str*, *attribute_value: Any*) → None

Set the attribute of the AI Agent team specified by *attribute_name* and *attribute_value*.

set_attributes(*attributes:* TeamAttributes) → None

Set the attributes of the AI Agent team

14.4.2 Share a team

Grant or revoke access for a database user or role with `Team` methods. Run these methods as the team owner:

```
team = Team.fetch("MOVIE_AGENT_TEAM")
team.grant_access("APP_USER")
team.revoke_access("APP_USER")
```

Team fetch and list operations use the connected user's agent-team views and do not currently accept an `owner` argument.

14.4.3 Run Team

`Team.run(...)` starts the team workflow. The `prompt` argument is passed to the task and can be referenced by task instructions using `{query}`. `params` can include `conversation_id` to associate multiple runs with the same conversation and `variables` to pass additional key-value inputs.

```
result = team.run(
    prompt="Could you list the movies in the database?",
    params={
        "conversation_id": conversation_id,
        "variables": {"audience": "analyst"},
    },
)
```

```
import os
import uuid

import select_ai
from select_ai.agent import (
    Team,
    TeamAttributes,
)

conversation_id = str(uuid.uuid4())

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)

# Team
team = Team(
    team_name="MOVIE_AGENT_TEAM",
    attributes=TeamAttributes(
        agents=[{"name": "MOVIE_ANALYST", "task": "ANALYZE_MOVIE_TASK"}],
        process="sequential",
    ),
)
team.create(enabled=True, replace=True)

print(
    team.run(
        prompt="Could you list the movies in the database?",
        params={"conversation_id": conversation_id},
    )
)
```

output:

```
The database contains 100 movies with various titles, genres, and release
dates. The list includes a wide range of genres such as Action, Comedy, Drama,
Thriller, Romance, Adventure, Mystery, Sci-Fi, Historical, Biography, War,
Sports, Music, Documentary, Animated, Fantasy, Horror, Western, Family,
```

(continues on next page)

(continued from previous page)

```
and more. The release dates are primarily in January and February of 2019.
Here is a summary of the movies:

1. Action Movie (Action, 2019-01-01)
2. Comedy Film (Comedy, 2019-01-02)
3. Drama Series (Drama, 2019-01-03)
4. Thriller Night (Thriller, 2019-01-04)
5. Romance Story (Romance, 2019-01-05)
6. Adventure Time (Adventure, 2019-01-06)
7. Mystery Solver (Mystery, 2019-01-07)
8. Sci-Fi World (Sci-Fi, 2019-01-08)
9. Historical Epic (Historical, 2019-01-09)
10. Biographical (Biography, 2019-01-10)
... (list continues up to 100 movies)
```

14.4.4 Export and Import Team

Select AI agent teams can be exported into a portable specification and imported into the same database, a different database, or another Select AI service. The specification describes the team composition and the associated agent, task, and tool definitions that are needed to recreate the team.

`Team.export_team()` returns the specification as a JSON string by default. `Team.import_team()` accepts either that JSON string or a Python mapping containing the same team definition structure. In most cases, pass a `dict`, for example the result of `json.loads(exported_spec)`. Other JSON-serializable collections.abc.Mapping objects, such as `OrderedDict`, can also be used. On import, `profile_name` identifies the Select AI profile to use in the target database. `team_name` can be provided to create the imported team under a new name; this is useful when importing into the same database as the source team.

If imported object names conflict with existing agents, tasks, tools, or teams, set `force=True` to let the database replace the conflicting objects. Use this carefully when importing into a shared schema because conflicting components can be dropped and recreated.

```
import json
import os

import select_ai
from select_ai.agent import (
    Agent,
    AgentAttributes,
    Task,
    TaskAttributes,
    Team,
    TeamAttributes,
)

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
profile_name = os.getenv("SELECT_AI_PROFILE_NAME", "LLAMA_4_MAVERICK")

select_ai.connect(user=user, password=password, dsn=dsn)

task = Task(
    task_name="EXPORT_IMPORT_MOVIE_TASK",
    description="Task used by the team export/import sample",
    attributes=TaskAttributes(
        instruction="Help the user with movie questions. Question: {query}",
        tools=[],
        enable_human_tool=False,
    ),
)
task.create(replace=True)

agent = Agent(
    agent_name="EXPORT_IMPORT_MOVIE_ANALYST",
    description="Agent used by the team export/import sample",
    attributes=AgentAttributes(
        profile_name=profile_name,
        role="You are an AI Movie Analyst.",
        enable_human_tool=False,
```

(continues on next page)

(continued from previous page)

```
    ),
)
agent.create(enabled=True, replace=True)

source_team = Team(
    team_name="EXPORT_IMPORT_MOVIE_TEAM",
    attributes=TeamAttributes(
        agents=[
            {
                "name": agent.agent_name,
                "task": task.task_name,
            }
        ],
        process="sequential",
    ),
)
source_team.create(enabled=True, replace=True)

specification = json.loads(source_team.export())
print("Exported specification:")
print(json.dumps(specification, indent=2))

specification["name"] = "IMPORTED_MOVIE_ANALYST"
specification["task"]["task_name"] = "IMPORTED_ANALYZE_MOVIE_TASK"

Team.import_team(
    profile_name=profile_name,
    team_name="IMPORTED_MOVIE_AGENT_TEAM",
    specification=specification,
    force=True,
)

team = Team.fetch("IMPORTED_MOVIE_AGENT_TEAM")
print("Imported team:", team)
```

output:

```
Exported specification:
{
  "name": "EXPORT_IMPORT_MOVIE_ANALYST",
  "component_type": "Agent",
  "task": {
    "task_name": "EXPORT_IMPORT_MOVIE_TASK",
    "instruction": "Help the user with movie questions. Question: {query}",
    "task_attributes": {
      "enable_human_tool": "false",
      "tools": []
    }
  },
  "llm_config": {
    "name": "LLAMA_4_MAVERICK",
    "component_type": "oci"
```

(continues on next page)

(continued from previous page)

```
  }
}
Imported team: Team(team_name=IMPORTED_MOVIE_AGENT_TEAM, ...)
```

The same APIs can also read from or write to object storage by passing both `object_storage_credential_name` and `location`. When exporting to object storage, `Team.export_team()` writes the specification to the location and returns `None`. When importing from object storage, pass the same credential and location instead of `specification`.

14.4.5 Inspect a team

`Team.describe_team()` returns JSON metadata and the aggregated skills for a team. `Team.list_tools()` returns JSON metadata for the tools available to that team.

```
team = Team.fetch("MOVIE_AGENT_TEAM")
print(team.describe_team())
print(team.list_tools())
```

The supervised-team sample also demonstrates both inspection methods.

14.4.6 Lifecycle helpers

All agent object types support list, fetch, enable, disable, and delete operations.

```
for tool in select_ai.agent.Tool.list():
    print(tool.tool_name)

task = select_ai.agent.Task.fetch("ANALYZE_MOVIE_TASK")
agent = select_ai.agent.Agent.fetch("MOVIE_ANALYST")
team = select_ai.agent.Team.fetch("MOVIE_AGENT_TEAM")

team.disable()
team.enable()
team.delete(force=True)
```

Object definitions

Use `get_definition(object_type, object_name)` to retrieve the canonical PL/SQL block for recreating an AI `AGENT`, `TASK`, `TOOL`, or `TEAM`. The function returns `None` when the database does not return a definition.

```
from select_ai.agent import get_definition

definition = get_definition("TASK", "ANALYZE_MOVIE_TASK")
print(definition)
```

The asynchronous equivalent is `async_get_definition`. See the sync and async definition samples for a complete create, inspect, and cleanup flow.

```
import os
import uuid

import select_ai
from select_ai.agent import Task, TaskAttributes, get_definition

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

select_ai.connect(user=user, password=password, dsn=dsn)

task = Task(
    task_name=f"SAMPLE_DEFINITION_TASK_{uuid.uuid4().hex.upper()}",
    attributes=TaskAttributes(
        instruction="Answer the user's question: {query}", tools=[]
    ),
)
task.create(replace=True)

try:
    print(get_definition("TASK", task.task_name))
finally:
    task.delete(force=True)
```

The task name and exact PL/SQL formatting vary between runs and database versions. Example output (generated IDs and model responses vary):

output:

```
BEGIN
  DBMS_CLOUD_AI_AGENT.CREATE_TASK(...);
END;
/
```

14.5 Agent history

`TeamHistory`, `TaskHistory`, and `ToolHistory` provide typed, read-only access to the current user's Select AI Agent history views. They query only `USER_AI_AGENT_TEAM_HISTORY`, `USER_AI_AGENT_TASK_HISTORY`, and `USER_AI_AGENT_TOOL_HISTORY` respectively. Results are yielded newest first. Tool `input` and `output` values are decoded to Python objects when they contain valid JSON; other CLOB payloads are returned as strings.

```
from select_ai.agent import TaskHistory, TeamHistory, ToolHistory

for run in TeamHistory.list(team_name="MOVIE_AGENT_TEAM", limit=10):
    print(run.team_exec_id, run.state)

for run in TaskHistory.list(team_exec_id="<team execution id>"):
    print(run.task_name, run.result)

for call in ToolHistory.list(tool_name="MOVIE_SQL_TOOL", limit=20):
    print(call.input, call.output)
```

The sample retrieves a team's latest execution and uses its `team_exec_id` to retrieve the associated task and tool history.

class select_ai.agent.**TeamHistory**

Read runs from `C##CLOUD$SERVICE.USER_AI_AGENT_TEAM_HISTORY`.

classmethod list(*team_name: str | None = None*, *team_exec_id: str | None = None*, *limit: int | None = None*) → Iterator[*TeamHistoryEvent*]

Yield team runs ordered from newest to oldest.

class select_ai.agent.**TaskHistory**

Read task runs from `C##CLOUD$SERVICE.USER_AI_AGENT_TASK_HISTORY`.

classmethod list(*team_name: str | None = None*, *task_name: str | None = None*, *agent_name: str | None = None*, *team_exec_id: str | None = None*, *limit: int | None = None*) → Iterator[*TaskHistoryEvent*]

Yield task runs ordered from newest to oldest.

class select_ai.agent.**ToolHistory**

Read tool calls from `C##CLOUD$SERVICE.USER_AI_AGENT_TOOL_HISTORY`.

classmethod list(*tool_name: str | None = None*, *task_name: str | None = None*, *agent_name: str | None = None*, *team_exec_id: str | None = None*, *limit: int | None = None*) → Iterator[*ToolHistoryEvent*]

Yield tool calls ordered from newest to oldest.

The complete sample prints the latest team execution and the task and tool events associated with that execution:

```
from select_ai.agent import TaskHistory, TeamHistory, ToolHistory

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
team_name = "ORACLE_AI_DATABASE_AGENT"

select_ai.connect(user=user, password=password, dsn=dsn)
```

(continues on next page)

(continued from previous page)

```
# Replace team_name with team_exec_id when the application has recorded it.
for team_run in TeamHistory.list(team_name=team_name, limit=1):
    pprint(team_run)

    # team_exec_id scopes the remaining history to the same execution.
    for task_run in TaskHistory.list(team_exec_id=team_run.team_exec_id):
        pprint(task_run)

    for tool_run in ToolHistory.list(team_exec_id=team_run.team_exec_id):
        pprint(tool_run)
```

History depends on an existing execution for the configured team. Representative output is:

output:

```
TeamHistoryEvent(team_exec_id='<team execution id>', team_name='ORACLE_AI_DATABASE_AGENT
↪', state='<state>', ...)
TaskHistoryEvent(team_exec_id='<team execution id>', task_name='<task name>', state='
↪<state>', ...)
ToolHistoryEvent(invocation_id=<invocation id>, team_exec_id='<team execution id>', tool_
↪name='<tool name>', ...)
```

14.6 AI agent examples

14.6.1 Web Search Agent using OpenAI's GPT model

```
import os

import select_ai
from select_ai.agent import (
    Agent,
    AgentAttributes,
    Task,
    TaskAttributes,
    Team,
    TeamAttributes,
    Tool,
)

OPEN_AI_CREDENTIAL_NAME = "OPENAI_CRED"
OPEN_AI_PROFILE_NAME = "OPENAI_PROFILE"
SELECT_AI_AGENT_NAME = "WEB_SEARCH_AGENT"
SELECT_AI_TASK_NAME = "WEB_SEARCH_TASK"
SELECT_AI_TOOL_NAME = "WEB_SEARCH_TOOL"
SELECT_AI_TEAM_NAME = "WEB_SEARCH_TEAM"

USER_QUERIES = {
    "d917b055-e8a1-463a-a489-d4328a7b2210": "What are the key features for the product␣
↪highlighted at "
    "this URL https://www.oracle.com/artificial-intelligence/database-machine-learning",
    "c2e3ff20-f56d-40e7-987c-cc72740c75a5": "What is the main topic at this URL https://
↪www.oracle.com/artificial-intelligence/database-machine-learning",
    "25e23a25-07b9-4ed7-be11-f7e5e445d286": "What is the main topic at this URL https://
↪openai.com",
}

# connect
user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
select_ai.connect(user=user, password=password, dsn=dsn)

# Create Open AI credential
select_ai.create_credential(
    credential={
        "credential_name": OPEN_AI_CREDENTIAL_NAME,
        "username": "OPENAI",
        "password": os.getenv("OPEN_AI_API_KEY"),
    },
    replace=True,
)
print("Created credential: ", OPEN_AI_CREDENTIAL_NAME)

# # Create Open AI Profile
profile = select_ai.Profile(
```

(continues on next page)

(continued from previous page)

```
    profile_name=OPEN_AI_PROFILE_NAME,
    attributes=select_ai.ProfileAttributes(
        credential_name=OPEN_AI_CREDENTIAL_NAME,
        provider=select_ai.OpenAIProvider(model="gpt-4.1"),
    ),
    description="My Open AI Profile",
    replace=True,
)
print("Created profile: ", OPEN_AI_PROFILE_NAME)

# Create an AI Agent team
team = Team(
    team_name=SELECT_AI_TEAM_NAME,
    attributes=TeamAttributes(
        agents=[{"name": SELECT_AI_AGENT_NAME, "task": SELECT_AI_TASK_NAME}]
    ),
)
team.create(replace=True)

# Agent
agent = Agent(
    agent_name=SELECT_AI_AGENT_NAME,
    attributes=AgentAttributes(
        profile_name=OPEN_AI_PROFILE_NAME,
        enable_human_tool=False,
        role="You are a specialized web search agent that can access web page "
        "contents and respond to questions based on its content.",
    ),
)
agent.create(replace=True)

# Task
task = Task(
    task_name=SELECT_AI_TASK_NAME,
    attributes=TaskAttributes(
        instruction="Answer the user question about the provided URL:{query}",
        enable_human_tool=False,
        tools=[SELECT_AI_TOOL_NAME],
    ),
)
task.create(replace=True)

# Tool
web_search_tool = Tool.create_websearch_tool(
    tool_name=SELECT_AI_TOOL_NAME,
    credential_name=OPEN_AI_CREDENTIAL_NAME,
    description="Web Search Tool using OpenAI",
    replace=True,
)
print("Created tool: ", SELECT_AI_TOOL_NAME)

# Run the Agent Team
```

(continues on next page)

(continued from previous page)

```
for conversation_id, prompt in USER_QUERIES.items():
    response = team.run(prompt=prompt)
    print(response)
```

output:

```
Created credential:  OPENAI_CRED
Created profile:  OPENAI_PROFILE
Created tool:  WEB_SEARCH_TOOL
The key features of Oracle Database Machine Learning, as highlighted on the
 Oracle website, include:

- In-database machine learning: Build, train, and deploy machine learning
  models directly inside the Oracle Database, eliminating the need to move
  data.
- Support for multiple languages: Use SQL, Python, and R for machine
  learning tasks, allowing flexibility for data scientists and developers.
- Automated machine learning (AutoML): Automates feature selection, model
  selection, and hyperparameter tuning to speed up model development.
- Scalability and performance: Utilizes Oracle Database's scalability,
  security, and high performance for machine learning workloads.
- Integration with Oracle Cloud: Seamlessly integrates with Oracle
  Cloud Infrastructure for scalable and secure deployment.
- Security and governance: Inherits Oracle Database's robust security,
  data privacy, and governance features.
- Prebuilt algorithms: Offers a wide range of in-database algorithms for
  classification, regression, clustering, anomaly detection, and more.
- No data movement: Keeps data secure and compliant by performing
  analytics and machine learning where the data resides.

These features enable organizations to operationalize machine learning at
scale, improve productivity, and maintain data security and compliance.

The main topic at the URL https://www.oracle.com/artificial-intelligence/database-
↪machine-learning
is Oracle's database machine learning capabilities, specifically how Oracle
integrates artificial intelligence and machine learning features directly
into its database products. The page highlights how users can leverage these
built-in AI and ML tools to analyze data, build predictive models, and enhance
business applications without moving data outside the Oracle Database
environment.

The main topic of the website https://openai.com is artificial
intelligence research and development. OpenAI focuses on creating and
 promoting advanced AI technologies, including products like ChatGPT, and
 provides information about their research, products, and mission to ensure
 that artificial general intelligence benefits all of humanity.
```

CHAPTER

FIFTEEN

ASYNC AI AGENT

`select_ai.agent` also provides async interfaces to be used with `async` / `await` keywords. Use these classes in applications that already use `asyncio` and `select_ai.async_connect()` or `select_ai.create_pool_async()`.

The history API follows the same pattern. `AsyncTeamHistory`, `AsyncTaskHistory`, and `AsyncToolHistory` query only the current user's history views and yield typed events newest first. See *Async agent history* for the complete API reference and sample.

```
from select_ai.agent import AsyncToolHistory

async for call in AsyncToolHistory.list(limit=10):
    print(call.tool_name, call.output)
```

The async sample retrieves a team's latest execution and uses its `team_exec_id` to retrieve the associated task and tool history.

The async agent object model mirrors the synchronous agent object model:

Table 1: Sync and async agent APIs

Sync class	Async class
`select_ai.agent.Tool`	`select_ai.agent.AsyncTool`
`select_ai.agent.Task`	`select_ai.agent.AsyncTask`
`select_ai.agent.Agent`	`select_ai.agent.AsyncAgent`
`select_ai.agent.Team`	`select_ai.agent.AsyncTeam`

Create or reuse the same database objects as the synchronous APIs. Async methods must be awaited, and async list methods return async iterators.

```
await select_ai.async_connect(user=user, password=password, dsn=dsn)

async for tool in select_ai.agent.AsyncTool.list():
    print(tool.tool_name)
```

Tools, tasks, agents, and teams are database objects. Use `replace=True` when you want to recreate an existing object with the same name, and `force=True` when cleanup should succeed even if the object does not exist.

Table 2: Select AI Async Agent Tools

Tool Type	AsyncTool Class Method	Arguments
EMAIL	select_ai.agent.AsyncTool. create_email_notification_tool	• tool_name • credential_name • recipient • sender • smtp_host
SQL	select_ai.agent.AsyncTool.create_sql_tool	• tool_name • profile_name
SLACK	select_ai.agent.AsyncTool. create_slack_notification_tool	• tool_name • credential_name • channel
WEBSEARCH	select_ai.agent.AsyncTool. create_websearch_tool	• tool_name • credential_name
PL/SQL custom tool	select_ai.agent.AsyncTool. create_pl_sql_tool	• tool_name • function
RAG	select_ai.agent.AsyncTool.create_rag_tool	• tool_name • profile_name

Notification and web search tools require credentials and network access for the external service. SQL and RAG tools require existing Select AI profiles.

Tool selection follows the same guidance as *Agent*: use SQL tools for database questions, RAG tools for vector-index-backed content, notification tools for Slack or email, web search tools for public web content, and PL/SQL tools for application-specific database logic.

15.1 AsyncTool

class select_ai.agent.**AsyncTool**(*tool_name: str | None = None*, *description: str | None = None*, *attributes:* ToolAttributes *| None = None*)

async create(*enabled: bool | None = True*, *replace: bool | None = False*)

Create an AI Tool in the database

Parameters

- **enabled** (*Optional[bool]*) – Whether the tool should be enabled. Default: True
- **replace** (*Optional[bool]*) – Whether the tool should be replaced. Default: False

async classmethod create_built_in_tool(*tool_name: str*, *tool_params:* ToolParams, *tool_type:* ToolType, *description: str | None = None*, *replace: bool | None = False*, *instruction: str | None = None*) → *AsyncTool*

Register a built-in tool

Parameters

- **tool_name** (*str*) – The name of the tool
- **tool_params** (select_ai.agent.ToolParams) – Parameters required by built-in tool
- **tool_type** (select_ai.agent.tool.ToolType) – The built-in tool type
- **description** (*str*) – Description of the tool
- **replace** (*bool*) – Whether to replace the existing tool. Default value is False

- **instruction** (*str*) – A clear, concise statement that describes what the tool should accomplish and how to do it. This text is included in the prompt sent to the LLM.

Returns
select_ai.agent.Tool

async classmethod create_email_notification_tool(*tool_name: str*, *credential_name: str*, *recipient: str*, *sender: str*, *smtp_host: str*, *description: str | None*, *subject: str | None = None*, *replace: bool = False*, *instruction: str | None = None*) → *AsyncTool*

Register an email notification tool

Parameters

- **tool_name** (*str*) – The name of the tool
- **credential_name** (*str*) – The name of the credential
- **recipient** (*str*) – The recipient of the email
- **sender** (*str*) – The sender of the email
- **smtp_host** (*str*) – The SMTP host of the email server
- **description** (*str*) – The description of the tool
- **subject** (*str*) – Subject of the email.
- **replace** (*bool*) – Whether to replace the existing tool. Default value is False
- **instruction** (*str*) – A clear, concise statement that describes what the tool should accomplish and how to do it. This text is included in the prompt sent to the LLM.

Returns
select_ai.agent.Tool

async classmethod create_pl_sql_tool(*tool_name: str*, *function: str*, *description: str | None = None*, *replace: bool = False*, *instruction: str | None = None*) → *AsyncTool*

Create a custom tool to invoke PL/SQL procedure or function

Parameters

- **tool_name** (*str*) – The name of the tool
- **function** (*str*) – The name of the PL/SQL procedure or function
- **description** (*str*) – The description of the tool
- **replace** (*bool*) – Whether to replace existing tool. Default value is False
- **instruction** (*str*) – A clear, concise statement that describes what the tool should accomplish and how to do it. This text is included in the prompt sent to the LLM.

async classmethod create_rag_tool(*tool_name: str*, *profile_name: str*, *description: str | None = None*, *replace: bool = False*, *instruction: str | None = None*) → *AsyncTool*

Register a RAG tool, which will use a VectorIndex linked AI Profile

Parameters

- **tool_name** (*str*) – The name of the tool
- **profile_name** (*str*) – The name of the profile to use for Vector Index based RAG

- **description** (*str*) – The description of the tool
- **replace** (*bool*) – Whether to replace existing tool. Default value is False
- **instruction** (*str*) – A clear, concise statement that describes what the tool should accomplish and how to do it. This text is included in the prompt sent to the LLM.

async classmethod create_slack_notification_tool(*tool_name: str*, *credential_name: str*, *channel: str*, *description: str | None = None*, *replace: bool = False*, *instruction: str | None = None*) → *AsyncTool*

Register a Slack notification tool

Parameters

- **tool_name** (*str*) – The name of the Slack notification tool
- **credential_name** (*str*) – The name of the Slack credential
- **channel** (*str*) – The name of the Slack channel
- **description** (*str*) – The description of the Slack notification tool
- **replace** (*bool*) – Whether to replace existing tool. Default value is False
- **instruction** (*str*) – A clear, concise statement that describes what the tool should accomplish and how to do it. This text is included in the prompt sent to the LLM.

async classmethod create_sql_tool(*tool_name: str*, *profile_name: str*, *description: str | None = None*, *replace: bool = False*, *instruction: str | None = None*) → *AsyncTool*

Register a SQL tool to perform natural language to SQL translation

Parameters

- **tool_name** (*str*) – The name of the tool
- **profile_name** (*str*) – The name of the profile to use for SQL translation
- **description** (*str*) – The description of the tool
- **replace** (*bool*) – Whether to replace existing tool. Default value is False
- **instruction** (*str*) – A clear, concise statement that describes what the tool should accomplish and how to do it. This text is included in the prompt sent to the LLM.

async classmethod create_websearch_tool(*tool_name: str*, *credential_name: str*, *description: str | None*, *replace: bool = False*, *instruction: str | None = None*) → *AsyncTool*

Register a built-in websearch tool to search information on the web

Parameters

- **tool_name** (*str*) – The name of the tool
- **credential_name** (*str*) – The name of the credential object storing OpenAI credentials
- **description** (*str*) – The description of the tool
- **replace** (*bool*) – Whether to replace the existing tool
- **instruction** (*str*) – A clear, concise statement that describes what the tool should accomplish and how to do it. This text is included in the prompt sent to the LLM.

async delete(*force: bool = False*)

Delete AI Tool from the database

Parameters
force (*bool*) – Force the deletion. Default value is False.

async classmethod delete_tool(*tool_name: str*, *force: bool = False*)

Class method ot delete AI Tool from the database

Parameters

- **tool_name** (*str*) – The name of the tool
- **force** (*bool*) – Force the deletion. Default value is False.

async describe_tool() → str | None

Asynchronously return this tool's JSON metadata.

async disable()

Disable AI Tool

async enable()

Enable AI Tool

async classmethod fetch(*tool_name: str*) → *AsyncTool*

Fetch AI Tool attributes from the Database and build a proxy object in the Python layer

Parameters
tool_name (*str*) – The name of the AI Task

Returns
select_ai.agent.Tool

Raises
select_ai.errors.AgentToolNotFoundError – If the AI Tool is not found

classmethod list(*tool_name_pattern: str = '.*'*) → AsyncGenerator[*AsyncTool*, None]

List AI Tools

Parameters
tool_name_pattern (*str*) – Regular expressions can be used to specify a pattern. Function REGEXP_LIKE is used to perform the match. Default value is ".*" i.e. match all tool name.

Returns
Iterator[Tool]

async run_tool(*input: str*) → str | None

Asynchronously run this tool directly and return its result.

async set_attribute(*attribute_name: str*, *attribute_value: Any*) → None

Set the attribute of the AI Agent tool specified by *attribute_name* and *attribute_value*.

async set_attributes(*attributes:* ToolAttributes) → None

Set the attributes of the AI Agent tool

15.1.1 Create Tool

The following example shows async creation of an AI agent tool to perform natural language translation to SQL using an OCI AI profile

```
import asyncio
import os
from pprint import pformat

import select_ai
import select_ai.agent
from select_ai.agent import AsyncTool, ToolAttributes

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():

    await select_ai.async_connect(user=user, password=password, dsn=dsn)

    profile_attributes = select_ai.ProfileAttributes(
        credential_name="my_oci_ai_profile_key",
        object_list=[
            {"owner": user, "name": "MOVIE"},
            {"owner": user, "name": "ACTOR"},
            {"owner": user, "name": "DIRECTOR"},
        ],
        provider=select_ai.OCIGenAIProvider(
            region="us-chicago-1",
            oci_apiformat="GENERIC",
            model="meta.llama-4-maverick-17b-128e-instruct-fp8",
        ),
    )
    profile = await select_ai.AsyncProfile(
        profile_name="LLAMA_4_MAVERICK",
        attributes=profile_attributes,
        description="MY OCI AI Profile",
        replace=True,
    )

    # Create a tool which uses the OCI AI Profile to
    # perform natural language SQL translation
    sql_tool = await AsyncTool.create_sql_tool(
        tool_name="MOVIE_SQL_TOOL",
        description="My Select AI MOVIE SQL agent tool",
        profile_name="LLAMA_4_MAVERICK",
        replace=True,
    )
    print(sql_tool.tool_name)
    print(pformat(sql_tool.attributes))
```

(continues on next page)

(continued from previous page)

```
asyncio.run(main())
```

output:

```
MOVIE_SQL_TOOL

ToolAttributes(instruction=None,
               function=None,
               tool_params=SQLToolParams(_REQUIRED_FIELDS=None,
                                         credential_name=None,
                                         endpoint=None,
                                         notification_type=None,
                                         profile_name='oci_ai_profile',
                                         recipient=None,
                                         sender=None,
                                         channel=None,
                                         smtp_host=None),
               tool_inputs=None,
               tool_type=<ToolType.SQL: 'SQL'>)
```

15.1.2 List Tools

```
import os

import select_ai
from select_ai.agent import AsyncTool

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async for tool in AsyncTool.list():
        print(tool.tool_name)


asyncio.run(main())
```

output:

```
WEB_SEARCH_TOOL
MOVIE_SQL_TOOL
LLM_CHAT_TOOL
```

15.1.3 Inspect and run a tool

Use `AsyncTool.describe_tool()` to return JSON metadata for a tool, including its function arguments. Use `AsyncTool.run_tool(input)` to invoke a tool directly with an input payload.

```
tool = await AsyncTool.fetch("MOVIE_SQL_TOOL")
print(await tool.describe_tool())
print(await tool.run_tool('{"query": "How many movies are there?"}'))
```

The complete example creates a temporary PL/SQL tool, describes it, invokes it, and removes the temporary database objects:

```
import asyncio
import os
import uuid

import select_ai
from select_ai.agent import AsyncTool

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    suffix = uuid.uuid4().hex.upper()
    function_name = f"SAMPLE_CALCULATE_AGE_{suffix}"
    tool_name = f"SAMPLE_AGE_TOOL_{suffix}"

    async with select_ai.async_cursor() as cursor:
        await cursor.execute(
            f"""
            CREATE OR REPLACE FUNCTION {function_name}(p_birth_date DATE)
            RETURN NUMBER IS
            BEGIN
                RETURN TRUNC(MONTHS_BETWEEN(SYSDATE, p_birth_date) / 12);
            END;
            """
        )

    tool = await AsyncTool.create_pl_sql_tool(
        tool_name=tool_name,
        function=function_name,
        description="Calculate age from a birth date",
    )

    try:
        print("Tool description:")
        print(await tool.describe_tool())
        print("Tool result:")
        print(await tool.run_tool('{"p_birth_date":"2000-01-01"}'))
    finally:
```

(continues on next page)

(continued from previous page)

```
        await tool.delete(force=True)
        async with select_ai.async_cursor() as cursor:
            await cursor.execute(f"DROP FUNCTION {function_name}")


asyncio.run(main())
```

The generated tool name and calculated age vary between runs. Representative output is:

output:

```
Tool description:
{"tool_name": "SAMPLE_AGE_TOOL_<generated suffix>", ...}
Tool result:
<calculated age in years>
```

15.2 AsyncTask

class select_ai.agent.**AsyncTask**(*task_name: str | None = None*, *description: str | None = None*, *attributes: TaskAttributes | None = None*)

select_ai.agent.AsyncTask class lets you create, delete, enable, disable and list AI Tasks asynchronously

Parameters

- **task_name** (*str*) – The name of the AI task
- **description** (*str*) – Optional description of the AI task
- **attributes** (*select_ai.agent.TaskAttributes*) – AI task attributes

async create(*enabled: bool | None = True*, *replace: bool | None = False*)

Create a task that a Select AI agent can include in its reasoning process

Parameters

- **enabled** (*bool*) – Whether the AI Task should be enabled. Default value is True.
- **replace** (*bool*) – Whether the AI Task should be replaced. Default value is False.

async delete(*force: bool = False*)

Delete AI Task from the database

Parameters

force (*bool*) – Force the deletion. Default value is False.

async classmethod delete_task(*task_name: str*, *force: bool = False*)

Class method to delete AI Task from the database

Parameters

- **task_name** (*str*) – The name of the AI Task
- **force** (*bool*) – Force the deletion. Default value is False.

async disable()

Disable AI Task

async enable()

Enable AI Task

async classmethod fetch(*task_name: str*) → *AsyncTask*

Fetch AI Task attributes from the Database and build a proxy object in the Python layer

Parameters

task_name (*str*) – The name of the AI Task

Returns

select_ai.agent.Task

Raises

select_ai.errors.AgentTaskNotFoundError – If the AI Task is not found

classmethod list(*task_name_pattern: str | None = '.*'*) → AsyncGenerator[*AsyncTask*, None]

List AI Tasks

Parameters

task_name_pattern (*str*) – Regular expressions can be used to specify a pattern. Function REGEXP_LIKE is used to perform the match. Default value is ".*" i.e. match all tasks.

Returns
AsyncGenerator[Task]

async set_attribute(*attribute_name: str*, *attribute_value: Any*)

Set a single AI Task attribute specified using name and value

Parameters

- **attribute_name** (*str*) – The name of the AI Task attribute
- **attribute_value** (*str*) – The value of the AI Task attribute

async set_attributes(*attributes:* TaskAttributes)

Set AI Task attributes

Parameters
attributes (select_ai.agent.TaskAttributes) – Multiple attributes can be specified by passing a TaskAttributes object

15.2.1 Create Task

In the following task, we use the `MOVIE_SQL_TOOL` created in the previous step

The `instruction` is the main task prompt. Use placeholders such as `{query}` when the user prompt should be inserted into the task. The `tools` list limits which tools the agent can use for the task.

```
import asyncio
import os
from pprint import pformat

import select_ai
import select_ai.agent
from select_ai.agent import AsyncTask, TaskAttributes

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    task = AsyncTask(
        task_name="ANALYZE_MOVIE_TASK",
        description="Search for movies in the database",
        attributes=TaskAttributes(
            instruction="Help the user with their request about movies. "
            "User question: {query}. "
            "You can use SQL tool to search the data from database",
            tools=["MOVIE_SQL_TOOL"],
            enable_human_tool=False,
        ),
    )
    await task.create(replace=True)
    print(task.task_name)
    print(pformat(task.attributes))


asyncio.run(main())
```

output:

```
ANALYZE_MOVIE_TASK
TaskAttributes(instruction='Help the user with their request about movies. '
                           'User question: {query}. You can use SQL tool to '
                           'search the data from database',
               tools=['MOVIE_SQL_TOOL'],
               input=None,
               enable_human_tool=False)
```

15.2.2 List Tasks

```
import asyncio
import os

import select_ai
from select_ai.agent import AsyncTask

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async for task in AsyncTask.list():
        print(task.task_name)


asyncio.run(main())
```

output:

```
WEB_SEARCH_TASK
ANALYZE_MOVIE_TASK
```

15.3 AsyncAgent

class select_ai.agent.**AsyncAgent**(*agent_name: str | None = None*, *description: str | None = None*, *attributes: AgentAttributes | None = None*)

select_ai.agent.AsyncAgent class lets you create, delete, enable, disable and list AI agents asynchronously

Parameters

- **agent_name** (*str*) – The name of the AI Agent
- **description** (*str*) – Optional description of the AI agent
- **attributes** (*select_ai.agent.AgentAttributes*) – AI agent attributes

async create(*enabled: bool | None = True*, *replace: bool | None = False*)

Register a new AI Agent within the Select AI framework

Parameters

- **enabled** (*bool*) – Whether the AI Agent should be enabled. Default value is True.
- **replace** (*bool*) – Whether the AI Agent should be replaced. Default value is False.

async delete(*force: bool | None = False*)

Delete AI Agent from the database

Parameters

force (*bool*) – Force the deletion. Default value is False.

async classmethod delete_agent(*agent_name: str*, *force: bool | None = False*)

Class method to delete AI Agent from the database

Parameters

- **agent_name** (*str*) – The name of the AI Agent
- **force** (*bool*) – Force the deletion. Default value is False.

async disable()

Disable AI Agent

async enable()

Enable AI Agent

async classmethod fetch(*agent_name: str*) → *AsyncAgent*

Fetch AI Agent attributes from the Database and build a proxy object in the Python layer

Parameters

agent_name (*str*) – The name of the AI Agent

Returns

select_ai.agent.Agent

Raises

select_ai.errors.AgentNotFoundError – If the AI Agent is not found

classmethod list(*agent_name_pattern: str | None = '.*'*) → AsyncGenerator[*AsyncAgent*, None]

List AI agents matching a pattern

Parameters

agent_name_pattern (*str*) – Regular expressions can be used to specify a pattern. Function REGEXP_LIKE is used to perform the match. Default value is ".*" i.e. match all agent names.

Returns
AsyncGenerator[AsyncAgent]

async set_attribute(*attribute_name: str*, *attribute_value: Any*) → None

Set a single AI Agent attribute specified using name and value

async set_attributes(*attributes:* AgentAttributes) → None

Set AI Agent attributes

Parameters
attributes (select_ai.agent.AgentAttributes) – Multiple attributes can be specified by passing an AgentAttributes object

15.3.1 Create Agent

```
import asyncio
import os

import select_ai
from select_ai.agent import (
    AgentAttributes,
    AsyncAgent,
)


async def main():
    user = os.getenv("SELECT_AI_USER")
    password = os.getenv("SELECT_AI_PASSWORD")
    dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    agent_attributes = AgentAttributes(
        profile_name="LLAMA_4_MAVERICK",
        role="You are an AI Movie Analyst. "
        "Your can help answer a variety of questions related to movies. ",
        enable_human_tool=False,
    )
    agent = AsyncAgent(
        agent_name="MOVIE_ANALYST",
        attributes=agent_attributes,
    )
    await agent.create(enabled=True, replace=True)
    print("Created Agent:", agent)


asyncio.run(main())
```

output:

```
Created Agent: Agent(agent_name=MOVIE_ANALYST,
attributes=AgentAttributes(profile_name='LLAMA_4_MAVERICK',
role='You are an AI Movie Analyst.
Your can help answer a variety of questions related to movies. ',
enable_human_tool=False), description=None)
```

15.3.2 List Agents

```
import asyncio
import os

import select_ai
from select_ai.agent import AsyncAgent


async def main():
    user = os.getenv("SELECT_AI_USER")
    password = os.getenv("SELECT_AI_PASSWORD")
```

(continues on next page)

(continued from previous page)

```
    dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async for agent in AsyncAgent.list():
        print(agent.agent_name)


asyncio.run(main())
```

output:

```
WEB_SEARCH_AGENT
MOVIE_ANALYST
```

15.4 AsyncTeam

class select_ai.agent.**AsyncTeam**(*team_name: str*, *attributes:* TeamAttributes | *None* = *None*, *description: str* | *None* = *None*)

A Team of AI agents work together to accomplish tasks select_ai.agent.Team class lets you create, delete, enable, disable and list AI Tasks.

Parameters

- **team_name** (*str*) – The name of the AI team
- **description** (*str*) – Optional description of the AI team
- **attributes** (select_ai.agent.TeamAttributes) – AI team attributes

async create(*enabled: bool* | *None* = *True*, *replace: bool* | *None* = *False*)

Create a team of AI agents that work together to accomplish tasks.

Parameters

- **enabled** (*bool*) – Whether the AI agent team should be enabled. Default value is True.
- **replace** (*bool*) – Whether the AI agent team should be replaced. Default value is False.

async delete(*force: bool* | *None* = *False*)

Delete an AI agent team from the database

Parameters
force (*bool*) – Force the deletion. Default value is False.

async classmethod delete_team(*team_name: str*, *force: bool* | *None* = *False*)

Class method to delete an AI agent team from the database

Parameters

- **team_name** (*str*) – The name of the AI team
- **force** (*bool*) – Force the deletion. Default value is False.

async describe_team() → str | None

Asynchronously return this team's JSON metadata and skills.

async disable()

Disable the AI agent team

async enable()

Enable the AI agent team

async export(*object_storage_credential_name: str* | *None* = *None*, *location: str* | *None* = *None*, *params: str* | *Mapping* | *None* = *None*) → str | None

Export this AI agent team specification.

If object storage details are provided, the specification is written to the given location and None is returned. Otherwise, the specification is returned as a string.

Parameters

- **object_storage_credential_name** (*str*) – Optional credential name used to write the exported specification to object storage. Must be specified together with `location`.
- **location** (*str*) – Optional object storage URI where the exported specification should be written. Must be specified together with `object_storage_credential_name`.

- **params** (*str or Mapping*) – Optional export parameters. May be a JSON string or a Python mapping.

Returns
Exported team specification as a JSON string when exporting inline, or None when exporting to object storage.

Return type
str or None

async classmethod export_team(*team_name: str*, *object_storage_credential_name: str | None = None*, *location: str | None = None*, *params: str | Mapping | None = None*) → str | None

Export an AI agent team specification.

If object storage details are provided, the specification is written to the given location and None is returned. Otherwise, the specification is returned as a string.

Parameters

- **team_name** (*str*) – Name of the AI agent team to export.
- **object_storage_credential_name** (*str*) – Optional credential name used to write the exported specification to object storage. Must be specified together with `location`.
- **location** (*str*) – Optional object storage URI where the exported specification should be written. Must be specified together with `object_storage_credential_name`.
- **params** (*str or Mapping*) – Optional export parameters. May be a JSON string or a Python mapping.

Returns
Exported team specification as a JSON string when exporting inline, or None when exporting to object storage.

Return type
str or None

async classmethod fetch(*team_name: str*) → *AsyncTeam*

Fetch AI Team attributes from the Database and build a proxy object in the Python layer

Parameters
team_name (*str*) – The name of the AI Team

Returns
select_ai.agent.Team

Raises
select_ai.errors.AgentTeamNotFoundError – If the AI Team is not found

async grant_access(*user_or_role_name: str*) → None

Asynchronously grant a user or role access to this agent team.

async classmethod import_team(*profile_name: str*, *team_name: str | None = None*, *specification: str | Mapping | None = None*, *object_storage_credential_name: str | None = None*, *location: str | None = None*, *force: bool | None = False*, *params: str | Mapping | None = None*) → None

Import an AI agent team specification and create the associated team, agents, tasks, and tools in the database.

Parameters

- **profile_name** (*str*) – Name of the Select AI profile to use for the imported team and agents in the target database.
- **team_name** (*str*) – Optional name for the imported team. If omitted, the team name from the specification is used.
- **specification** (*str or Mapping*) – Team specification to import. May be a JSON string or a Python mapping. Omit this when importing from object storage.
- **object_storage_credential_name** (*str*) – Optional credential name used to read the specification from object storage. Must be specified together with `location`.
- **location** (*str*) – Optional object storage URI of the specification to import. Must be specified together with `object_storage_credential_name`.
- **force** (*bool*) – Whether to replace conflicting database objects during import. Default value is False.
- **params** (*str or Mapping*) – Optional import parameters. May be a JSON string or a Python mapping.

classmethod list(*team_name_pattern: str | None = '.*'*) → AsyncGenerator[*AsyncTeam*, None]

List AI Agent Teams

Parameters

team_name_pattern (*str*) – Regular expressions can be used to specify a pattern. Function REGEXP_LIKE is used to perform the match. Default value is ".*" i.e. match all teams.

Returns

Iterator[Team]

async list_tools() → str | None

Asynchronously return JSON metadata for this team's tools.

async revoke_access(*user_or_role_name: str*) → None

Asynchronously revoke a user or role's agent team access.

async run(*prompt: str = None*, *params: Mapping = None*)

Start a new AI agent team or resume a paused one that is waiting for human input. If you provide an existing process ID and the associated team process is in the WAITING_FOR_HUMAN state, the function resumes the workflow using the input you provide as the human response

Parameters

- **prompt** (*str*) – Optional prompt for the user. If the task is in the RUNNING state, the input acts as a placeholder for the {query} in the task instruction. If the task is in the WAITING_FOR_HUMAN state, the input serves as the human response.
- **params** (*Mapping[str, str]*) – Optional parameters for the task. Supported keys include `conversation_id`, which identifies the conversation session associated with the agent team, and `variables`, which provides additional key-value input to the agent team.

async set_attribute(*attribute_name: str*, *attribute_value: Any*) → None

Set the attribute of the AI Agent team specified by *attribute_name* and *attribute_value*.

async set_attributes(*attributes:* TeamAttributes) → None

Set the attributes of the AI Agent team

15.4.1 Share an async team

Grant or revoke access for a database user or role with `AsyncTeam` methods. Run these methods as the team owner:

```
team = await AsyncTeam.fetch("MOVIE_AGENT_TEAM")
await team.grant_access("APP_USER")
await team.revoke_access("APP_USER")
```

Team fetch and list operations use the connected user's agent-team views and do not currently accept an `owner` argument.

15.4.2 Supervised teams

Configure an asynchronous supervised team in the same way as a synchronous team. Set `supervisor=True` on the coordinating agent and pass its name as `TeamAttributes.supervisor_agent`. Keep the worker agent and task assignments in the team's `agents` list.

The database creates the supervisor task when the team is created, so `supervisor_task` should not be supplied by the application. Fetch the team after creation to read the generated value from `fetched.attributes.supervisor_task`.

```
from select_ai.agent import (
    AgentAttributes, AsyncAgent, AsyncTeam, TeamAttributes,
)

# Inside an async function, with an open async connection.
# MOVIE_ANALYST and ANALYZE_MOVIE_TASK must already exist.
supervisor = AsyncAgent(
    agent_name="MOVIE_SUPERVISOR",
    attributes=AgentAttributes(
        profile_name="oci_ai_profile",
        role="You supervise and coordinate the team.",
        supervisor=True,
    ),
)

await supervisor.create()

team = AsyncTeam(
    team_name="MOVIE_AGENT_TEAM",
    attributes=TeamAttributes(
        agents=[
            {"name": "MOVIE_ANALYST", "task": "ANALYZE_MOVIE_TASK"}
        ],
        process="sequential",
        supervisor_agent=supervisor.agent_name,
    ),
)
await team.create()
```

Constructors build Python objects. Await `create()` to persist each object in Oracle Database. The example expects new supervisor and team names; pass `replace=True` to `create()` only when replacement is intended.

The complete asynchronous example creates the team, runs a prompt through the supervisor workflow, and inspects its generated supervisor task and metadata. `AsyncTeam.run()` starts the team workflow; the configured supervisor agent coordinates the worker agent as part of that call.

Set the common database connection variables and `SELECT_AI_PROFILE_NAME` (default `oci_ai_profile`) to an existing profile. Run `python samples/agent/async/team_supervisor_inspect.py` from the repository root. The script removes its generated objects and conversation during cleanup.

Configure the imports and connection values:

```
import asyncio
import os
import uuid

import select_ai
from select_ai.agent import (
    AgentAttributes,
    AsyncAgent,
    AsyncTask,
    AsyncTeam,
    TaskAttributes,
    TeamAttributes,
)
from select_ai.conversation import (
    AsyncConversation,
    ConversationAttributes,
)

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
profile_name = os.getenv("SELECT_AI_PROFILE_NAME", "oci_ai_profile")
```

Append the async function and its task, agent, and team definitions:

```
async def main():
    suffix = uuid.uuid4().hex.upper()
    await select_ai.async_connect(user=user, password=password, dsn=dsn)

    task = AsyncTask(
        task_name=f"SAMPLE_TASK_{suffix}",
        attributes=TaskAttributes(
            instruction="Answer the user's question: {query}",
            tools=[],
            enable_human_tool=False,
        ),
    )
    worker = AsyncAgent(
        agent_name=f"SAMPLE_WORKER_{suffix}",
        attributes=AgentAttributes(
            profile_name=profile_name,
            role="You answer user questions.",
            enable_human_tool=False,
        ),
    )
    supervisor = AsyncAgent(
        agent_name=f"SAMPLE_SUPERVISOR_{suffix}",
        attributes=AgentAttributes(
            profile_name=profile_name,
            role="You supervise and coordinate the team.",
            enable_human_tool=False,
            supervisor=True,
        ),
    )
    team = AsyncTeam(
        team_name=f"SAMPLE_TEAM_{suffix}",
        attributes=TeamAttributes(
            agents=[{"name": worker.agent_name, "task": task.task_name}],
            process="sequential",
            supervisor_agent=supervisor.agent_name,
        ),
    )

    conversation = AsyncConversation(
        attributes=ConversationAttributes(
            title="Supervised team sample",
            description="Conversation for the supervised team sample",
        )
    )
```

Continue inside `main()` with creation, execution, inspection, and cleanup. Keep this block indented as part of the function, then add the entry point:

```
    created_objects = []
    conversation_created = False
    try:
        await task.create(replace=True)
        created_objects.append(task)
        await worker.create(replace=True)
        created_objects.append(worker)
        await supervisor.create(replace=True)
        created_objects.append(supervisor)
        await team.create(replace=True)
        created_objects.append(team)

        await conversation.create()
        conversation_created = True
        response = await team.run(
            prompt="What is the capital of France? Answer in one sentence.",
            params={"conversation_id": conversation.conversation_id},
        )
        if response is None or (
            isinstance(response, str) and response.startswith("Task failed:")
        ):
            raise RuntimeError(f"Supervised team run failed: {response}")
        print("Team response:", response)

        fetched = await AsyncTeam.fetch(team.team_name)
        print("Supervisor agent:", fetched.attributes.supervisor_agent)
        print("Supervisor task:", fetched.attributes.supervisor_task)
        print("Team description:", await team.describe_team())
        print("Team tools:", await team.list_tools())
    finally:
        if conversation_created:
            await conversation.delete(force=True)
        for obj in reversed(created_objects):
            await obj.delete(force=True)


asyncio.run(main())
```

Example output (generated names, model responses, and JSON metadata vary):

output:

```
Team response: <answer from the supervised team>
Supervisor agent: SAMPLE_SUPERVISOR_<generated suffix>
Supervisor task: <generated supervisor task>
Team description: <JSON team metadata>
Team tools: <JSON tool metadata>
```

15.4.3 Run Team

`AsyncTeam.run(...)` starts the team workflow. The `prompt` argument is passed to the task and can be referenced by task instructions using `{query}`. `params` can include `conversation_id` to associate multiple runs with the same conversation and `variables` to pass additional key-value inputs.

```
result = await team.run(
    prompt="Could you list the movies in the database?",
    params={
        "conversation_id": conversation_id,
        "variables": {"audience": "analyst"},
    },
)
```

```
import asyncio
import os
import uuid

import select_ai
from select_ai.agent import (
    AsyncTeam,
    TeamAttributes,
)

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    team = AsyncTeam(
        team_name="MOVIE_AGENT_TEAM",
        attributes=TeamAttributes(
            agents=[{"name": "MOVIE_ANALYST", "task": "ANALYZE_MOVIE_TASK"}],
            process="sequential",
        ),
    )
    await team.create(enabled=True, replace=True)
    print(
        await team.run(
            prompt="Could you list the movies in the database?",
            params={"conversation_id": str(uuid.uuid4())},
        )
    )


asyncio.run(main())
```

output:

```
The database contains 100 movies with various titles, genres, and release
dates. The list includes a wide range of genres such as Action, Comedy, Drama,
Thriller, Romance, Adventure, Mystery, Sci-Fi, Historical, Biography, War,
```

(continues on next page)

(continued from previous page)

```
Sports, Music, Documentary, Animated, Fantasy, Horror, Western, Family,
and more. The release dates are primarily in January and February of 2019.
Here is a summary of the movies:

1. Action Movie (Action, 2019-01-01)
2. Comedy Film (Comedy, 2019-01-02)
3. Drama Series (Drama, 2019-01-03)
4. Thriller Night (Thriller, 2019-01-04)
5. Romance Story (Romance, 2019-01-05)
6. Adventure Time (Adventure, 2019-01-06)
7. Mystery Solver (Mystery, 2019-01-07)
8. Sci-Fi World (Sci-Fi, 2019-01-08)
9. Historical Epic (Historical, 2019-01-09)
10. Biographical (Biography, 2019-01-10)
... (list continues up to 100 movies)
```

15.4.4 Export and Import Team

Select AI agent teams can be exported into a portable specification and imported into the same database, a different database, or another Select AI service. The specification describes the team composition and the associated agent, task, and tool definitions that are needed to recreate the team.

`AsyncTeam.export_team()` returns the specification as a JSON string by default. `AsyncTeam.import_team()` accepts either that JSON string or a Python mapping containing the same team definition structure. In most cases, pass a `dict`, for example the result of `json.loads(exported_spec)`. Other JSON-serializable collections.abc.Mapping objects, such as `OrderedDict`, can also be used. On import, `profile_name` identifies the Select AI profile to use in the target database. `team_name` can be provided to create the imported team under a new name; this is useful when importing into the same database as the source team.

If imported object names conflict with existing agents, tasks, tools, or teams, set `force=True` to let the database replace the conflicting objects. Use this carefully when importing into a shared schema because conflicting components can be dropped and recreated.

```
import asyncio
import json
import os

import select_ai
from select_ai.agent import (
    AgentAttributes,
    AsyncAgent,
    AsyncTask,
    AsyncTeam,
    TaskAttributes,
    TeamAttributes,
)

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
profile_name = os.getenv("SELECT_AI_PROFILE_NAME", "LLAMA_4_MAVERICK")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)

    task = AsyncTask(
        task_name="EXPORT_IMPORT_MOVIE_TASK",
        description="Task used by the team export/import sample",
        attributes=TaskAttributes(
            instruction="Help the user with movie questions. Question: {query}",
            tools=[],
            enable_human_tool=False,
        ),
    )
    await task.create(replace=True)

    agent = AsyncAgent(
        agent_name="EXPORT_IMPORT_MOVIE_ANALYST",
        description="Agent used by the team export/import sample",
        attributes=AgentAttributes(
```

(continues on next page)

(continued from previous page)

```
            profile_name=profile_name,
            role="You are an AI Movie Analyst.",
            enable_human_tool=False,
        ),
    )
    await agent.create(enabled=True, replace=True)

    source_team = AsyncTeam(
        team_name="EXPORT_IMPORT_MOVIE_TEAM",
        attributes=TeamAttributes(
            agents=[
                {
                    "name": agent.agent_name,
                    "task": task.task_name,
                }
            ],
            process="sequential",
        ),
    )
    await source_team.create(enabled=True, replace=True)

    specification = json.loads(await source_team.export())
    print("Exported specification:")
    print(json.dumps(specification, indent=2))

    specification["name"] = "IMPORTED_MOVIE_ANALYST"
    specification["task"]["task_name"] = "IMPORTED_ANALYZE_MOVIE_TASK"

    await AsyncTeam.import_team(
        profile_name=profile_name,
        team_name="IMPORTED_MOVIE_AGENT_TEAM",
        specification=specification,
        force=True,
    )

    team = await AsyncTeam.fetch("IMPORTED_MOVIE_AGENT_TEAM")
    print("Imported team:", team)


asyncio.run(main())
```

output:

```
Exported specification:
{
  "name": "EXPORT_IMPORT_MOVIE_ANALYST",
  "component_type": "Agent",
  "task": {
    "task_name": "EXPORT_IMPORT_MOVIE_TASK",
    "instruction": "Help the user with movie questions. Question: {query}",
    "task_attributes": {
      "enable_human_tool": "false",
```

(continues on next page)

(continued from previous page)

```
      "tools": []
    }
  },
  "llm_config": {
    "name": "LLAMA_4_MAVERICK",
    "component_type": "oci"
  }
}
Imported team: AsyncTeam(team_name=IMPORTED_MOVIE_AGENT_TEAM, ...)
```

The same APIs can also read from or write to object storage by passing both `object_storage_credential_name` and `location`. When exporting to object storage, `AsyncTeam.export_team()` writes the specification to the location and returns `None`. When importing from object storage, pass the same credential and location instead of `specification`.

15.4.5 Inspect a team

`AsyncTeam.describe_team()` returns JSON metadata and the aggregated skills for a team. `AsyncTeam.list_tools()` returns JSON metadata for the tools available to that team.

```
team = await AsyncTeam.fetch("MOVIE_AGENT_TEAM")
print(await team.describe_team())
print(await team.list_tools())
```

The asynchronous supervised-team sample also demonstrates both inspection methods.

15.4.6 Lifecycle helpers

All async agent object types support list, fetch, enable, disable, and delete operations.

```
async for tool in select_ai.agent.AsyncTool.list():
    print(tool.tool_name)

task = await select_ai.agent.AsyncTask.fetch("ANALYZE_MOVIE_TASK")
agent = await select_ai.agent.AsyncAgent.fetch("MOVIE_ANALYST")
team = await select_ai.agent.AsyncTeam.fetch("MOVIE_AGENT_TEAM")

await team.disable()
await team.enable()
await team.delete(force=True)
```

Object definitions

Use `async_get_definition(object_type, object_name)` to asynchronously retrieve the canonical PL/SQL block for recreating an AI `AGENT`, `TASK`, `TOOL`, or `TEAM`. The function returns `None` when the database does not return a definition.

```
from select_ai.agent import async_get_definition

definition = await async_get_definition("TASK", "ANALYZE_MOVIE_TASK")
print(definition)
```

See the asynchronous definition sample for a complete create, inspect, and cleanup flow.

```
import asyncio
import os
import uuid

import select_ai
from select_ai.agent import AsyncTask, TaskAttributes, async_get_definition

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    task = AsyncTask(
        task_name=f"SAMPLE_DEFINITION_TASK_{uuid.uuid4().hex.upper()}",
        attributes=TaskAttributes(
            instruction="Answer the user's question: {query}", tools=[]
        ),
    )
    await task.create(replace=True)

    try:
        print(await async_get_definition("TASK", task.task_name))
    finally:
        await task.delete(force=True)


asyncio.run(main())
```

The task name and exact PL/SQL formatting vary between runs and database versions. Representative output is:

output:

```
BEGIN
  DBMS_CLOUD_AI_AGENT.CREATE_TASK(...);
END;
/
```

15.5 Async agent history

`AsyncTeamHistory`, `AsyncTaskHistory`, and `AsyncToolHistory` provide asynchronous, read-only access to the current user's Select AI Agent history views. Their `list()` methods return async iterators ordered from newest to oldest.

Use `team_exec_id` from a team event to scope task and tool history to the same execution. Filters such as `team_name`, `task_name`, `agent_name`, and `tool_name` can be used when an execution identifier is not available. Tool `input` and `output` values are decoded to Python objects when they contain valid JSON.

```
from select_ai.agent import (
    AsyncTaskHistory,
    AsyncTeamHistory,
    AsyncToolHistory,
)

async for team_run in AsyncTeamHistory.list(
    team_name="ORACLE_AI_DATABASE_AGENT",
    limit=1,
):
    print(team_run)
    async for task_run in AsyncTaskHistory.list(
        team_exec_id=team_run.team_exec_id
    ):
        print(task_run)
    async for tool_run in AsyncToolHistory.list(
        team_exec_id=team_run.team_exec_id
    ):
        print(tool_run)
```

The complete sample retrieves a team's latest execution and uses its `team_exec_id` to retrieve the associated task and tool history:

class select_ai.agent.**AsyncTeamHistory**

Asynchronously read runs from C##CLOUD$SERVICE.USER_AI_AGENT_TEAM_HISTORY.

classmethod list(*team_name: str | None = None*, *team_exec_id: str | None = None*, *limit: int | None = None*) → AsyncGenerator[*TeamHistoryEvent*, None]

Yield team runs ordered from newest to oldest.

class select_ai.agent.**AsyncTaskHistory**

Asynchronously read task runs from C##CLOUD$SERVICE.USER_AI_AGENT_TASK_HISTORY.

classmethod list(*team_name: str | None = None*, *task_name: str | None = None*, *agent_name: str | None = None*, *team_exec_id: str | None = None*, *limit: int | None = None*) → AsyncGenerator[*TaskHistoryEvent*, None]

Yield task runs ordered from newest to oldest.

class select_ai.agent.**AsyncToolHistory**

Asynchronously read tool calls from C##CLOUD$SERVICE.USER_AI_AGENT_TOOL_HISTORY.

classmethod list(*tool_name: str | None = None*, *task_name: str | None = None*, *agent_name: str | None = None*, *team_exec_id: str | None = None*, *limit: int | None = None*) → AsyncGenerator[*ToolHistoryEvent*, None]

Yield tool calls ordered from newest to oldest.

```
import select_ai
from select_ai.agent import (
    AsyncTaskHistory,
    AsyncTeamHistory,
    AsyncToolHistory,
)

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")
team_name = "ORACLE_AI_DATABASE_AGENT"


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)

    # Replace team_name with team_exec_id when the application has recorded it.
    async for team_run in AsyncTeamHistory.list(team_name=team_name, limit=1):
        pprint(team_run)

        # team_exec_id scopes the remaining history to the same execution.
        async for task_run in AsyncTaskHistory.list(
            team_exec_id=team_run.team_exec_id
        ):
            pprint(task_run)

        async for tool_run in AsyncToolHistory.list(
            team_exec_id=team_run.team_exec_id
        ):
            pprint(tool_run)


asyncio.run(main())
```

History depends on an existing execution for the configured team. Representative output is:

output:

```
TeamHistoryEvent(team_exec_id='<team execution id>', team_name='ORACLE_AI_DATABASE_AGENT
↪', state='<state>', ...)
TaskHistoryEvent(team_exec_id='<team execution id>', task_name='<task name>', state='
↪<state>', ...)
ToolHistoryEvent(invocation_id=<invocation id>, team_exec_id='<team execution id>', tool_
↪name='<tool name>', ...)
```

15.5.1 List Teams

```
import asyncio
import os

import select_ai
from select_ai.agent import AsyncTeam

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async for team in AsyncTeam.list():
        print(team.team_name)


asyncio.run(main())
```

output:

```
WEB_SEARCH_TEAM
MOVIE_AGENT_TEAM
```

15.6 Async AI agent examples

15.6.1 Web Search Agent using OpenAI's GPT model

```
import asyncio
import os

import select_ai
from select_ai.agent import (
    AgentAttributes,
    AsyncAgent,
    AsyncTask,
    AsyncTeam,
    AsyncTool,
    TaskAttributes,
    TeamAttributes,
)

OPEN_AI_CREDENTIAL_NAME = "OPENAI_CRED"
OPEN_AI_PROFILE_NAME = "OPENAI_PROFILE"
SELECT_AI_AGENT_NAME = "WEB_SEARCH_AGENT"
SELECT_AI_TASK_NAME = "WEB_SEARCH_TASK"
SELECT_AI_TOOL_NAME = "WEB_SEARCH_TOOL"
SELECT_AI_TEAM_NAME = "WEB_SEARCH_TEAM"

USER_QUERIES = {
    "d917b055-e8a1-463a-a489-d4328a7b2210": "What are the key features for the product␣
↪highlighted at "
    "this URL https://www.oracle.com/artificial-intelligence/database-machine-learning",
    "c2e3ff20-f56d-40e7-987c-cc72740c75a5": "What is the main topic at this URL https://
↪www.oracle.com/artificial-intelligence/database-machine-learning",
    "25e23a25-07b9-4ed7-be11-f7e5e445d286": "What is the main topic at this URL https://
↪openai.com",
}

# connect
user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)

    # Create Open AI credential
    await select_ai.async_create_credential(
        credential={
            "credential_name": OPEN_AI_CREDENTIAL_NAME,
            "username": "OPENAI",
            "password": os.getenv("OPEN_AI_API_KEY"),
        },
        replace=True,
    )
```

(continues on next page)

(continued from previous page)

```
    print("Created credential: ", OPEN_AI_CREDENTIAL_NAME)

    # # Create Open AI Profile
    profile = await select_ai.AsyncProfile(
        profile_name=OPEN_AI_PROFILE_NAME,
        attributes=select_ai.ProfileAttributes(
            credential_name=OPEN_AI_CREDENTIAL_NAME,
            provider=select_ai.OpenAIProvider(model="gpt-4.1"),
        ),
        description="My Open AI Profile",
        replace=True,
    )
    print("Created profile: ", OPEN_AI_PROFILE_NAME)

    # Create an AI Agent team
    team = AsyncTeam(
        team_name=SELECT_AI_TEAM_NAME,
        attributes=TeamAttributes(
            agents=[
                {"name": SELECT_AI_AGENT_NAME, "task": SELECT_AI_TASK_NAME}
            ]
        ),
    )
    await team.create(replace=True)

    # Agent
    agent = AsyncAgent(
        agent_name=SELECT_AI_AGENT_NAME,
        attributes=AgentAttributes(
            profile_name=OPEN_AI_PROFILE_NAME,
            enable_human_tool=False,
            role="You are a specialized web search agent that can access web page "
            "contents and respond to questions based on its content.",
        ),
    )
    await agent.create(replace=True)

    # Task
    task = AsyncTask(
        task_name=SELECT_AI_TASK_NAME,
        attributes=TaskAttributes(
            instruction="Answer the user question about the provided URL:{query}",
            enable_human_tool=False,
            tools=[SELECT_AI_TOOL_NAME],
        ),
    )
    await task.create(replace=True)

    # Tool
    web_search_tool = await AsyncTool.create_websearch_tool(
        tool_name=SELECT_AI_TOOL_NAME,
        credential_name=OPEN_AI_CREDENTIAL_NAME,
```

(continues on next page)

(continued from previous page)

```
        description="Web Search Tool using OpenAI",
        replace=True,
    )
    print("Created tool: ", SELECT_AI_TOOL_NAME)

    # Run the Agent Team
    for conversation_id, prompt in USER_QUERIES.items():
        response = await team.run(
            prompt=prompt, params={"conversation_id": conversation_id}
        )
        print(response)


asyncio.run(main())
```

output:

```
Created credential:  OPENAI_CRED
Created profile:  OPENAI_PROFILE
Created tool:  WEB_SEARCH_TOOL
The key features of Oracle Database Machine Learning, as highlighted on the
 Oracle website, include:

- In-database machine learning: Build, train, and deploy machine learning
  models directly inside the Oracle Database, eliminating the need to move
  data.
- Support for multiple languages: Use SQL, Python, and R for machine
  learning tasks, allowing flexibility for data scientists and developers.
- Automated machine learning (AutoML): Automates feature selection, model
  selection, and hyperparameter tuning to speed up model development.
- Scalability and performance: Utilizes Oracle Database's scalability,
  security, and high performance for machine learning workloads.
- Integration with Oracle Cloud: Seamlessly integrates with Oracle
  Cloud Infrastructure for scalable and secure deployment.
- Security and governance: Inherits Oracle Database's robust security,
  data privacy, and governance features.
- Prebuilt algorithms: Offers a wide range of in-database algorithms for
  classification, regression, clustering, anomaly detection, and more.
- No data movement: Keeps data secure and compliant by performing
  analytics and machine learning where the data resides.

These features enable organizations to operationalize machine learning at
scale, improve productivity, and maintain data security and compliance.

The main topic at the URL https://www.oracle.com/artificial-intelligence/database-
↪machine-learning
is Oracle's database machine learning capabilities, specifically how Oracle
integrates artificial intelligence and machine learning features directly
into its database products. The page highlights how users can leverage these
built-in AI and ML tools to analyze data, build predictive models, and enhance
business applications without moving data outside the Oracle Database
environment.
```

(continues on next page)

(continued from previous page)

```
The main topic of the website https://openai.com is artificial
intelligence research and development. OpenAI focuses on creating and
 promoting advanced AI technologies, including products like ChatGPT, and
 provides information about their research, products, and mission to ensure
 that artificial general intelligence benefits all of humanity.
```

CHAPTER

SIXTEEN

A2A INTEGRATION

16.1 Agent2Agent (A2A)

Select AI for Python can expose an Oracle Database AI Agent Team through the Agent2Agent (A2A) protocol. A2A clients can discover the team from an Agent Card, submit database prompts, and retrieve task results through the same JSON-RPC endpoint.

Make two independent choices: **where the server runs** (topology) and **when database connection values are supplied** (fixed or dynamic). For example, a clustered deployment can use fixed credentials, and a standalone deployment can ask each client for credentials.

16.1.1 Choose a topology

- **Standalone:** run one public `select-ai a2a serve` process. It handles A2A requests and database work locally. Start here for a single server; Consul and a separate worker are not required.
- **Clustered:** run a public gateway with `select-ai a2a serve --deployment clustered` and one or more `select-ai a2a worker` processes. Consul lets the gateway discover healthy workers and route each session back to its worker. Use this when database sessions need to run across multiple workers. See *Why clustered deployments use Consul* for the role of this additional service.

16.1.2 Why clustered deployments use Consul

Consul is a separate service that keeps a directory of running workers and records which worker owns each A2A session. The gateway consults this directory to answer two questions: "Where can I start a new session?" and "Where should I send the next request for an existing session?"

This matters because each database session runs in a child process on one worker. That process holds the session's database connection and credentials. Later prompts, task-status requests, and cancellation requests must reach that same worker. A different worker cannot continue using its live connection.

Consul provides three things for the gateway:

- **Worker discovery:** each worker registers its address when it starts. The gateway can find workers without a manually maintained list of addresses.
- **Worker health:** workers send regular heartbeats to Consul to report that they are running. The gateway selects a worker whose health check is passing when opening a new session.
- **Session and task routing:** the gateway records the worker assigned to a session and the session associated with each task. Subsequent requests can reach the worker that owns them, even when other workers are available.

For example, suppose two workers, A and B, are running. The gateway opens your session on worker A and records that assignment in Consul. Your next prompt and a request to check your task's status both go to A. Another client's new session can run on B. If A stops, the gateway cannot move its live database connection to B; the client must reconnect to open a new session. See *Session isolation, routing, and cleanup* for expiry and reconnection behavior.

Consul stores worker addresses and temporary routing records. Database passwords are not stored in Consul; task results and conversation history are stored in Oracle Database. See *Persistent Oracle-backed state* for the database storage details.

When do you need it? The current clustered implementation requires Consul, even with only one worker or fixed database credentials. Standalone deployment handles sessions locally and does not require Consul. For a first deployment, use standalone unless you need separate worker processes across machines.

16.1.3 Choose how connections are configured

Each session needs four values: a database DSN, username, password, and an existing AI Agent Team name. Configure these on `a2a serve` with `--dsn`, `--user`, `--password`, and `--team`, or their environment fallbacks.

- **Fixed:** configure all four values before starting the server. Clients send prompts without completing a connection form. In standalone topology, the server uses a shared connection pool. In clustered topology, the gateway automatically opens a session on a worker using the configured values.
- **Dynamic:** leave one or more values unset. The client's first message receives an A2UI connection form asking for those missing values. After the client submits the form, the server opens an isolated database session and the client can send prompts using that A2A context ID. The client must support the A2UI handshake; see the runnable samples below.

For example, configure `--dsn` and `--team` for an application that always uses one database and one team, but lets each user connect with their own username and password. The form then asks only for username and password. Leave all four values unset to let the client choose the database and team as well. Values configured on the server cannot be changed by the client.

"Dynamic" describes collecting connection values when a session starts. Changing a database or user requires a new session. It does not switch the connection underneath an existing session.

Table 1: Supported combinations

Topology	Fixed connection	Dynamic connection
Standalone	Shared async pool; streaming supported.	Local child process per session; task polling supported.
Clustered	Worker session opened automatically; task polling supported.	Worker session opened after form submission; task polling supported.

Dynamic standalone and all clustered sessions do not support streaming. Standalone dynamic mode requires one Uvicorn process. Clustered mode requires Consul and at least one worker even when the connection is fixed. See *A2A CLI examples* for commands and *Command Line Interface* for options.

16.1.4 CLI-based deployment on any cloud

The A2A runtimes are provided as ordinary Select AI CLI commands, so they are not tied to a particular cloud provider. Run the commands on a local machine, a virtual machine, a container platform, or a Kubernetes service in any cloud that can reach the required Oracle Database. Clustered deployments also need Consul; standalone deployments do not.

The `--deployment` option defaults to `standalone`. Set it to `clustered` to use a gateway and workers. There is no separate connection-mode CLI flag: the connection is fixed when all four connection values are supplied and dynamic when any are missing. Environment values count as supplied values. See *A2A CLI examples* for complete commands.

The commands can be packaged into the platform's preferred container or process deployment. The Google Cloud scripts documented below are convenience automation for a Cloud Run/GKE deployment; they are not required to use the A2A CLI commands on another cloud or on self-managed infrastructure.

The public routes are the same in all supported combinations:

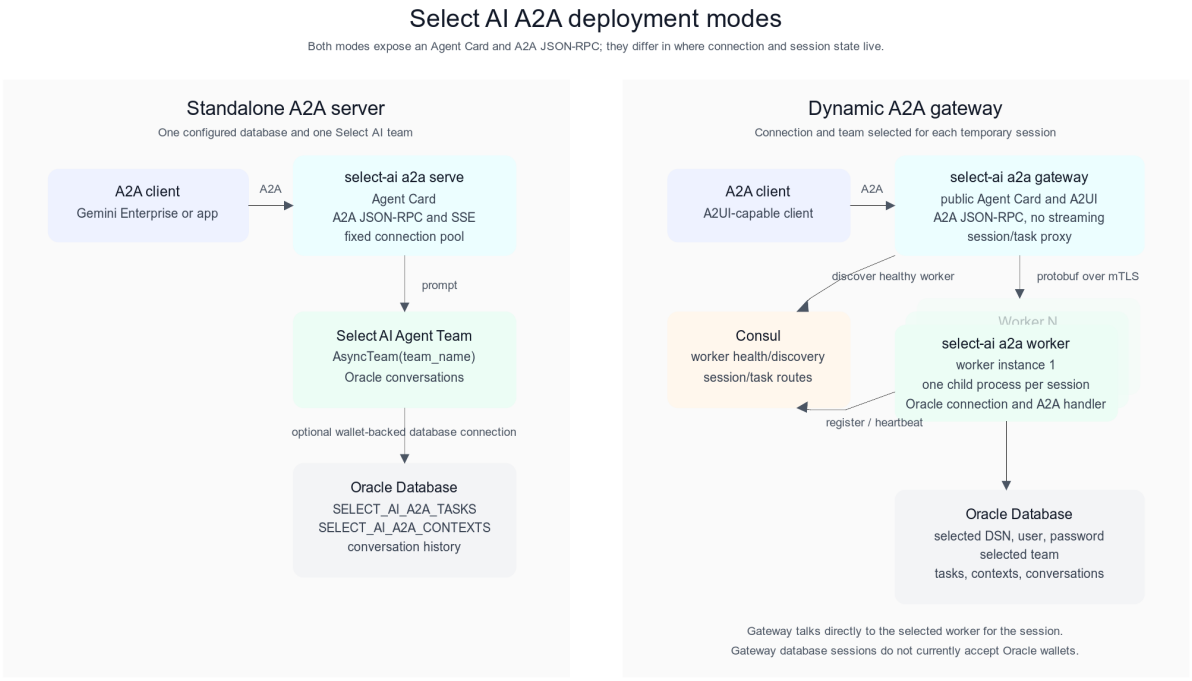


Fig. 1: Select AI A2A topologies. Dynamic sessions can run inside the standalone server or through the clustered gateway and worker processes.

`/.well-known/agent-card.json`
: Agent Card discovery endpoint.

`/a2a/jsonrpc/`
: A2A JSON-RPC endpoint. The endpoint accepts the A2A 1.0 method names and the A2A v0.3 compatibility method names.

16.1.5 Installation and prerequisites

Install the A2A dependencies before running the server, gateway, or worker:

```
python -m pip install 'select_ai[a2a]'
```

All modes require:

- an Oracle Database where the Select AI Agent Team is installed;
- a database user with the privileges required by Select AI and the team; and
- network access from the process to Oracle Database.

The database user also needs `CREATE TABLE`, `CREATE VIEW`, and sufficient quota in its default tablespace for A2A task and context storage. Ask a database administrator to grant these privileges to each database user that will open an A2A session. The SDK's `grant_privileges()` helper grants package execute privileges; it does not grant these storage privileges or tablespace quota.

At initialization, the runtime creates missing internal tables and executes `CREATE OR REPLACE VIEW` for its read-only views. `CREATE VIEW` is therefore required even when the tables already exist. See *Persistent Oracle-backed state*.

The fixed standalone server needs all database values at startup. Dynamic standalone and clustered sessions can collect

missing values through A2UI. Clustered deployment additionally requires Consul and at least one worker; workers must be able to reach both Consul and Oracle Database.

16.1.6 Protocol compatibility

The server and gateway use the same JSON-RPC URL for A2A 1.0 and v0.3. The protocol adapter translates the v0.3 method names and Agent Card shape for clients that have not migrated to A2A 1.0.

Table 2: Common A2A operations

Operation	A2A 1.0	A2A v0.3 compatibility name
Send a message	`SendMessage`	`message/send`
Stream a message	`SendStreamingMessage`	`message/stream`
Get a task	`GetTask`	`tasks/get`
List tasks	`ListTasks`	`tasks/list`
Cancel a task	`CancelTask`	`tasks/cancel`

For A2A 1.0 clients, use `A2A-Version: 1.0` when the client library requires an explicit version header. A v0.3 client uses the compatibility names. The `select-ai a2a agent-card` command prints a v0.3 Agent Card that can be registered with Gemini Enterprise.

16.1.7 Standalone A2A server

Fixed database connection

When all four connection values are configured at startup, the standalone server uses one shared asynchronous connection pool for the selected team. This mode supports streaming and does not need Consul or a separate worker. Replace the example values with your database username, password, host, port, service name, team, and CA-file path. Supply all four connection values to use fixed standalone mode; omitting any value selects dynamic standalone mode.

```
select-ai a2a serve \
    --deployment standalone \
    --team ORACLE_AI_DATABASE_AGENT \
    --user DB_USERNAME \
    --password 'DB_PASSWORD' \
    --dsn 'dbhost.example.com:1521/db_service' \
    --database-tls-ca-file /run/secrets/database-ca.pem \
    --host 0.0.0.0 \
    --port 8000 \
    --public-url https://a2a.example.com
```

The `--database-tls-ca-file` option is only needed when the database certificate is signed by a CA not already trusted by the system. The `--public-url` value is the externally reachable base URL placed into the Agent Card; set it when a proxy or load balancer sits in front of the process.

The server prints the discovery URL when it starts:

```
A2A Agent Card: https://a2a.example.com/.well-known/agent-card.json
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
```

The exact Uvicorn startup lines vary by version and configuration.

Dynamic standalone sessions

If one or more of `--dsn`, `--user`, `--password`, and `--team` are omitted, the standalone server asks the A2A client for those values through an A2UI form. The server creates an isolated child process for each session. This is still standalone: it needs no Consul service and no `a2a worker` process. Run one Uvicorn process for this mode. Dynamic standalone sessions support task polling but do not support streaming.

```
select-ai a2a serve \
    --deployment standalone \
    --host 0.0.0.0 \
    --port 8000 \
    --public-url https://a2a.example.com
```

If you set some connection values at startup, the form asks only for the remaining values. See *A2A CLI examples* for the clustered gateway and worker commands.

End-user OAuth and session ownership

By default, `select-ai a2a serve` does not require an application OAuth token. Each A2A context maps to its own Select AI conversation, while records use a shared owner scope (`anonymous` in fixed standalone mode and `a2a-conversation` in session-based modes). A hosting platform such as private Cloud Run can authenticate the calling service, but the A2A server does not verify which human user owns the conversation.

Add `--require-oauth` to require a bearer token and derive the owner scope from it. Configure a trusted upstream service to validate the token first:

```
select-ai a2a serve \
    --deployment standalone \
    --require-oauth \
    --team ORACLE_AI_DATABASE_AGENT \
    --user DB_USERNAME \
    --password 'DB_PASSWORD' \
    --dsn 'dbhost.example.com:1521/db_service' \
    --database-tls-ca-file /run/secrets/database-ca.pem \
    --host 0.0.0.0 \
    --port 8000 \
    --public-url https://a2a.example.com
```

Every request to the A2A JSON-RPC endpoint must then contain `Authorization: Bearer ...`. Agent Card discovery remains public. The SDK requires bearer syntax and derives an owner scope, but does not validate token signatures, expiration, issuer, or audience. It accepts arbitrary nonempty bearer values. This flag alone does not establish a verified end-user identity.

The deployment must validate tokens in a trusted authenticating proxy or service before forwarding them, and prevent callers from reaching the A2A listener through a path that bypasses that validation. Cloud Run IAM validation of a calling service is separate from validation of the end-user token.

The server uses the derived owner and A2A context to scope task and context records and the Select AI conversation. For an upstream-validated JWT with `iss` and `sub` claims, ownership remains stable across token refreshes. For an opaque access token, ownership is derived from a hash of the token; refreshing it starts a new owner scope. Tokens are not persisted by this authentication backend.

The Agent Card advertises bearer security only with `--require-oauth`.

Agent Card discovery

Fetch the card after starting the server:

```
curl -sS https://a2a.example.com/.well-known/agent-card.json | jq
```

The discovery response is the A2A v0.3-compatible representation used by clients such as Gemini Enterprise. It identifies the JSON-RPC endpoint and advertises streaming. The server's internal A2A 1.0 card declares both a JSON-RPC 1.0 interface and a JSON-RPC v0.3 interface at that same endpoint.

When a service has a public URL, the portable v0.3 card can also be printed without starting the server:

```
select-ai a2a agent-card \
    --team ORACLE_AI_DATABASE_AGENT \
    --public-url https://a2a.example.com
```

This command prints JSON similar to:

```
{
  "protocolVersion": "0.3",
  "name": "ORACLE_AI_DATABASE_AGENT",
  "url": "https://a2a.example.com/a2a/jsonrpc/",
  "capabilities": {"streaming": true},
  "defaultInputModes": ["text/plain"],
  "defaultOutputModes": ["text/plain"]
}
```

The package version, description, and skill list are also included in the actual output.

Tasks, polling, streaming, and cancellation

The fixed standalone server supports these execution patterns:

- **Blocking**: send a message without a non-blocking option. The request waits until the task reaches a terminal state and returns the completed task.
- **Non-blocking and polling**: send a v0.3 request with `configuration.blocking: false` or an A2A 1.0 request with `configuration.returnImmediately: true`. Poll `tasks/get` or `GetTask` until the task reaches `completed`, `failed`, `canceled`, or `rejected`.
- **Streaming**: use `message/stream` or `SendStreamingMessage`. The Agent Card advertises `streaming: true` and the endpoint returns the protocol's streaming events over the HTTP streaming response.
- **Cancellation**: call `tasks/cancel` or `CancelTask` with the task ID. The server updates the task through the A2A task handler and returns its canceled state when cancellation succeeds.

The repository includes the blocking_task.py sample and the task_poll.py sample:

```
import json
import uuid
from urllib.request import Request, urlopen

ENDPOINT = "http://127.0.0.1:8000/a2a/jsonrpc/"
PROMPT = "What were last month's sales by product category?"


# There is intentionally no "configuration": {"blocking": false} here.
# Omitting it is blocking by default, so this call waits for the database work
# to finish before the server returns the Task.
request = Request(
    ENDPOINT,
    data=json.dumps(
        {
            "jsonrpc": "2.0",
            "id": str(uuid.uuid4()),
            "method": "message/send",
            "params": {
                "message": {
                    "messageId": str(uuid.uuid4()),
                    "role": "user",
                    "parts": [{"kind": "text", "text": PROMPT}],
                }
            },
        }
    ).encode(),
    headers={"Content-Type": "application/json"},
    method="POST",
)

with urlopen(request) as response:  # noqa: S310
    body = json.load(response)
if "error" in body:
    raise RuntimeError(body["error"])

task = body["result"]
print(f"Task {task['id']}: {task['status']['state']}")
print(json.dumps(task, indent=2))
```

The blocking sample omits `configuration`. Example output (task IDs and model responses vary):

```
Task 7e1...: completed
{
  "id": "7e1...",
  "status": {"state": "completed", "timestamp": "..."},
  "artifacts": [{"name": "database-agent-result", "parts": ["..."]}]
}
```

The task ID, timestamp, and database answer vary for each run.

First, define the endpoint, prompt, terminal states, and JSON-RPC helper:

```
import json
import time
import uuid
from urllib.request import Request, urlopen

ENDPOINT = "http://127.0.0.1:8000/a2a/jsonrpc/"
PROMPT = "What were last month's sales by product category?"
TERMINAL_STATES = {"completed", "failed", "canceled", "rejected"}


def call(method, params):
    """Make one A2A v0.3 JSON-RPC call."""
    request = Request(
        ENDPOINT,
        data=json.dumps(
            {
                "jsonrpc": "2.0",
                "id": str(uuid.uuid4()),
                "method": method,
                "params": params,
            }
        ).encode(),
        headers={"Content-Type": "application/json"},
        method="POST",
    )
    with urlopen(request) as response:  # noqa: S310
        body = json.load(response)
    if "error" in body:
        raise RuntimeError(body["error"])
    return body["result"]
```

Append the send-and-poll loop to that setup:

```
# blocking=False returns immediately with a Task. Database work continues on
# the server while this client polls tasks/get.
task = call(
    "message/send",
    {
        "message": {
            "messageId": str(uuid.uuid4()),
            "role": "user",
            "parts": [{"kind": "text", "text": PROMPT}],
        },
        "configuration": {"blocking": False},
    },
)

task_id = task["id"]
print(f"Task {task_id}: {task['status']['state']}")

while task["status"]["state"] not in TERMINAL_STATES:
    time.sleep(1)
    task = call("tasks/get", {"id": task_id})
    print(f"Task {task_id}: {task['status']['state']}")

print(json.dumps(task, indent=2))
```

The polling sample sends a non-blocking request and then calls `tasks/get`:

```
Task 42b...: submitted
Task 42b...: working
Task 42b...: completed
{
  "id": "42b...",
  "status": {"state": "completed", "timestamp": "..."},
  "artifacts": [{"name": "database-agent-result", "parts": ["..."]}]
}
```

16.1.8 Persistent Oracle-backed state

Both standalone and clustered deployments use Oracle implementations of the A2A task and context stores. In fixed standalone mode, the server initializes these stores when the application starts. In dynamic standalone mode, each session child initializes the stores after connecting. In clustered mode, `select-ai a2a serve --deployment clustered` does not connect to Oracle itself; the `select-ai a2a worker` command starts the internal worker, and each connected worker session initializes the stores after it opens its database connection. On first initialization, the stores create these tables if they do not already exist:

DBMS_AI_A2A_TASKS$ and DBMS_AI_A2A_TASKS
: Stores the task ID, context ID, serialized task JSON, owner, and update timestamp. It supports task retrieval, filtering, listing, pagination, and deletion. The object with the $ suffix is the writable internal table; the object without it is a read-only view of the same columns.

DBMS_AI_A2A_CONTEXTS$ and DBMS_AI_A2A_CONTEXTS
: Maps an A2A context and owner to an Oracle conversation ID. The object with the $ suffix is the writable internal table; the object without it is a read-only view of the same columns.

When a request starts a new context, Select AI creates an `AsyncConversation` and passes its ID to `AsyncTeam.run`. Later messages in the same A2A context reuse that conversation, so the team can use the conversation history. In standalone modes, task and context ownership is resolved from the A2A request context in fixed standalone mode; an absent owner is stored as `anonymous`. Dynamic standalone and clustered runtimes pass the gateway's derived owner scope to the child runtime, using `a2a-conversation` when OAuth is not required. Routing binds the owner and A2A context ID to an internal session ID. That internal ID identifies the child process; it is separate from the client-visible A2A context ID.

The tables hold task and context metadata, while the conversation prompts and responses remain in the regular Select AI conversation storage. This means a server or worker process restart does not discard task records, context mappings, or conversation history, provided the same database and owner scope are used. Gateway routing records and worker child processes are temporary: after a worker session expires or is lost, submit the connection form again so the gateway can recreate the session and route requests to the durable Oracle state. The gateway's temporary connection-form task itself is not stored in Oracle; durable A2A tasks are created by the worker after the database session is ready.

16.1.9 Wallet-based database connections

Wallet-based connections apply to the fixed standalone `select-ai a2a serve` path. Dynamic standalone sessions and clustered worker sessions do not currently accept wallet settings. Pass the unzipped wallet directory and password with CLI options:

```
select-ai a2a serve \
    --deployment standalone \
    --team ORACLE_AI_DATABASE_AGENT \
    --user DB_USERNAME \
    --password 'DB_PASSWORD' \
    --dsn 'dbhost.example.com:1521/db_service' \
    --wallet-location /secure/wallet/Wallet_database \
    --wallet-password 'WALLET_PASSWORD' \
    --host 0.0.0.0 \
    --port 8000 \
    --public-url https://a2a.example.com
```

`--wallet-location` is used as both the wallet location and the Oracle configuration directory. Keep wallet files and passwords outside the source tree. The standalone deployment script can upload a wallet archive to Secret Manager and expand it into ephemeral Cloud Run storage; see *Standalone Google Cloud deployment* below.

16.1.10 Private database CA certificates

For a database whose server certificate is signed by a private CA, add `--database-tls-ca-file /path/to/database-ca.pem` to the relevant `select-ai a2a serve` command. The CLI reads PEM from that file and uses it to validate Oracle Database TLS in fixed standalone, dynamic standalone, and clustered sessions. In clustered mode, the gateway sends the certificate to the worker when it opens a session. The separate `--worker-tls-ca-file` option validates the worker certificate on the gateway-to-worker connection.

16.1.11 Database call timeout

Use `--database-call-timeout-seconds` to configure the maximum duration of an Oracle Database call. It defaults to 120 seconds and sets the timeout for fixed standalone connections and dynamic standalone or clustered worker sessions. In clustered mode, the gateway uses this value to derive its request deadline, so set the same value on the gateway and worker.

16.1.12 Clustered A2A gateway and worker

The clustered topology separates the public A2A protocol endpoint from the database-bearing runtime. The gateway can start without database credentials; an A2UI form collects them from the client. Values configured on the gateway are prefilled and the form asks only for the missing values. A worker registers with Consul and waits for the gateway to assign sessions. Each worker session uses the persistent Oracle-backed task, context, and conversation state described in *Persistent Oracle-backed state*; Consul stores only temporary routing metadata.

Consul provides the gateway's service-discovery and lightweight routing control plane. Workers register their address and health check with Consul; the gateway asks Consul for a passing worker when a session is opened. The gateway then stores the session-to-worker and task-to-session routes in Consul so later requests go directly to the worker that owns the session. Consul does not store database credentials, task payloads, or conversation history.

For local development, install the Consul command-line binary before starting the local agent. HashiCorp provides package-manager and precompiled-binary instructions in the Consul installation guide. After installation, verify that the binary is on `PATH`:

```
consul version
```

The `-dev` agent below is intentionally for local development only. It uses an in-memory, single-node Consul server bound to loopback. For the Google Cloud deployment, do not install or run a local agent: the deployment creates Consul in GKE from the Consul manifest.

Start the three local components in separate terminals:

```
# Terminal 1: Consul
consul agent -dev -bind=127.0.0.1 -client=127.0.0.1

# Terminal 2: worker
select-ai a2a worker \
    --host 127.0.0.1 \
    --port 8081 \
    --consul-url http://127.0.0.1:8500 \
    --worker-id local-worker \
    --worker-endpoint http://127.0.0.1:8081

# Terminal 3: public gateway
select-ai a2a serve \
    --deployment clustered \
    --host 127.0.0.1 \
    --port 8000 \
    --public-url http://127.0.0.1:8000 \
    --consul-url http://127.0.0.1:8500
```

The worker uses `--consul-url`, `--worker-id`, and the optional `--worker-endpoint` when registering with Consul. Their environment fallbacks are `CONSUL_HTTP_URL`, `WORKER_ID`, and `WORKER_ENDPOINT`; explicit command-line values take precedence. `--port` is both the listening port and the registered port. Cluster manifests can set `WORKER_ADDRESS` to the pod IP when no advertised endpoint is supplied. Clustered serve uses `PUBLIC_URL`, `CONSUL_HTTP_URL`, `WORKER_SERVICE` (default `select-ai-a2a-worker`), and `SESSION_TTL_SECONDS`.

Pass any fixed connection values with `--dsn`, `--user`, `--password`, and `--team`. The generated form contains only the remaining values. Use `--a2ui-form` to supply a validated custom form for those missing values.

The worker must be able to resolve the submitted DSN. For a TNS alias, set `TNS_ADMIN` in the worker terminal before starting it. The gateway session path accepts a DSN, username, and password; it does not currently accept an Oracle wallet.

The gateway's Agent Card advertises `streaming: false`. It supports blocking tasks, non-blocking tasks, task retrieval/listing, and cancellation by forwarding the request to the worker session. Streaming and push-notification operations are rejected because the gateway does not proxy a live stream.

Clustered deployment with a fixed connection

Yes, a clustered deployment can use fixed connection information. Start Consul and the worker as shown above, then run this gateway command in place of the dynamic gateway command. Supply the connection values on the gateway; the gateway passes them to the selected worker when opening each session:

```
select-ai a2a serve \
    --deployment clustered \
    --host 127.0.0.1 \
    --port 8000 \
    --public-url http://127.0.0.1:8000 \
    --consul-url http://127.0.0.1:8500 \
    --dsn 'dbhost.example.com:1521/db_service' \
    --user APP_USER \
    --password "$SELECT_AI_PASSWORD" \
    --team ORACLE_AI_DATABASE_AGENT
```

Set `SELECT_AI_PASSWORD` to the database password before running the command. The client sends an ordinary A2A prompt; the gateway skips the form and creates the worker session automatically. Workers still isolate sessions in child processes. All clients use the configured database identity and team. Fixed clustered connections retain the clustered restrictions: no streaming and no wallet parameters for the database session.

To collect each user's credentials instead, remove `--user` and `--password` and unset their environment fallbacks `SELECT_AI_USER` and `SELECT_AI_PASSWORD` in the gateway environment. Keep `--dsn` and `--team`. The same gateway and workers now serve dynamic sessions whose forms ask only for username and password.

A2UI database-connection form

When connection properties are missing, the first `message/send` for a context returns a temporary task containing a database connection form. The form is an A2UI v0.9 data artifact with the `application/json+a2ui` MIME type. It contains only the values not configured on the server:

Table 3: Dynamic connection form

Field	Purpose
`connection_url`	Oracle connect descriptor, simplified connect string, or TNS alias available to the worker.
`username`	Database user for this session.
`password`	Database password. The form renders this field as obscured input.
`ai_agent`	Select AI Agent Team to run in this session.
`submit_database_conne`	A2UI action that submits exactly the displayed values.

For example, a clustered server started with `--dsn` and `--team` renders only username and password. Server-configured values are immutable and cannot be overridden by a form submission. When all four values are configured, the server skips the form and opens the worker session automatically.

Use `--a2ui-form PATH` to replace the generated form. The file must be a top-level JSON array of A2UI v0.9 operations using the advertised catalog and one consistent template `surfaceId`. It must contain exactly one `submit_database_connection` action whose context submits exactly the missing canonical properties: `connection_url`, `username`, `password`, and/or `ai_agent`. A collected password must use an obscured `TextField`

and the template must not contain a password default. The server validates the file at startup and replaces its template surface ID with a fresh ID every time the form is displayed.

After the action is submitted in clustered mode, the gateway selects a healthy worker through Consul and opens a session. In dynamic standalone mode, the server opens the session locally. The worker starts a child process, calls `select_ai.async_connect` with the supplied DSN/user/password, validates the team, initializes the Oracle task and context stores, and returns a `database-session` artifact. The client can then send ordinary database prompts with the same A2A context ID.

Credentials are used to open the session and are not written to Consul. Consul stores only the worker endpoint and expiration for the session, plus task-to- session routing metadata. The credentials remain in the worker child process for the lifetime of that session, so use TLS for client-to-gateway traffic and follow the security policies for any client that renders the form.

The dynamic-session samples perform this handshake automatically. They use the repository-local helper module samples/a2a/dynamic/_common.py; `_common` is not a package that users install with `pip`. Running the samples from the repository root as shown below makes that helper available automatically.

The helper functions are:

`call(method, params)`
: Sends one A2A v0.3 JSON-RPC request to the configured server endpoint.

`connect(prompt)`
: Sends the initial prompt, reads the A2UI connection form, submits the `SELECT_AI_DB_CONNECT_STRING`, `SELECT_AI_USER`, `SELECT_AI_PASSWORD`, and `SELECT_AI_A2A_TEAM` values, and returns the A2A context ID for the connected session.

`send_prompt(prompt, context_id, blocking=None)`
: Sends a database prompt in an existing dynamic session. Passing `blocking=False` adds the non-blocking request option.

`print_task_summary(task)`
: Prints the task state, result artifact name, and text parts without dumping the connection-form details.

If you copy a dynamic-session sample into another directory, copy the _common.py helper with it, or replace these helpers with an A2A client implementation of your own.

The complete dynamic-session setup is also documented in the dynamic sample README. The examples below are the blocking_task.py sample and the task_poll.py sample.

```
from _common import connect, print_task_summary, send_prompt

PROMPT = "What were last month's sales by product category?"


context_id = connect(PROMPT)
task = send_prompt(PROMPT, context_id)
if task["artifacts"][0]["name"] != "database-agent-result":
    raise RuntimeError(
        "Expected a database result, got " f"{task['artifacts'][0]['name']}"
    )
print_task_summary(task)
```

Set the database values before running the sample:

```
export SELECT_AI_DB_CONNECT_STRING='database-dsn'
export SELECT_AI_USER='database-user'
export SELECT_AI_PASSWORD='database-password'
export SELECT_AI_A2A_TEAM='ORACLE_AI_DATABASE_AGENT'
```

Run it with:

```
python samples/a2a/dynamic/blocking_task.py
```

Example output (task IDs and model responses vary):

```
Task 15c...: completed
Artifact: database-agent-result
The database contains ...
```

The dynamic polling sample uses the same connection-form handshake, then passes `configuration.blocking: false` and polls `tasks/get`:

```
import time

from _common import call, connect, print_task_summary, send_prompt

PROMPT = "What were last month's sales by product category?"
TERMINAL_STATES = {"completed", "failed", "canceled", "rejected"}


context_id = connect(PROMPT)
task = send_prompt(PROMPT, context_id, blocking=False)
print(f"Task {task['id']}: {task['status']['state']}")

while task["status"]["state"] not in TERMINAL_STATES:
    time.sleep(1)
    task = call("tasks/get", {"id": task["id"]})
    print(f"Task {task['id']}: {task['status']['state']}")

if task["artifacts"][0]["name"] != "database-agent-result":
    raise RuntimeError(
        "Expected a database result, got " f"{task['artifacts'][0]['name']}"
    )
print_task_summary(task, include_task=False)
```

```
Task 91a...: submitted
Task 91a...: working
Task 91a...: completed
Artifact: database-agent-result
The database contains ...
```

Session isolation, routing, and cleanup

Each successful connection action binds the owner and A2A context ID to one worker session with its own internal session ID. The worker owns an operating-system child process for that session. The child process owns its Select AI asynchronous connection, A2A handler, task store, context store, and database conversation access. Requests for different sessions therefore do not share database credentials or a connection pool.

The default session lifetime is 900 seconds (15 minutes). Change it with `--session-ttl-seconds` on the worker and gateway, or with the gateway's `SESSION_TTL_SECONDS` environment variable. The worker reaper checks for expired or dead child processes and terminates them. The gateway also checks the expiry stored in Consul. When a route or child session is no longer available, the gateway returns an A2UI connection form so the client can reconnect rather than silently sending a request to a different database.

Consul is used for two kinds of routing metadata:

- the worker service registration and health TTL, which let the gateway select a passing worker; and
- `select-ai-a2a/sessions/` and `select-ai-a2a/tasks/` key-value records, which route a context or task back to the worker that owns it.

The task and conversation contents are not stored in Consul. They are stored in Oracle by the selected worker. A worker registers a health check and sends heartbeats; it deregisters on shutdown. The gateway selects healthy workers round-robin when opening new sessions.

Public gateway HTTPS

The `--tls-*` options on `select-ai a2a serve` configure the public A2A listener. They do not configure gateway-to-worker traffic.

- With no TLS options, Uvicorn listens over HTTP.
- Supply `--tls-cert-file` and `--tls-key-file` together to serve HTTPS. Clients verify the gateway certificate using their configured trust store.
- Add `--tls-ca-file` to require clients to present a certificate signed by that CA. This enables mTLS on the public connection. The CA file requires the certificate and key.

For example, this command serves HTTPS without requiring client certificates:

```
select-ai a2a serve \
    --team ORACLE_AI_DATABASE_AGENT \
    --user DB_USERNAME \
    --password DB_PASSWORD \
    --dsn 'dbhost.example.com:1521/db_service' \
    --host 0.0.0.0 \
    --port 8000 \
    --public-url https://a2a.example.com \
    --tls-cert-file /run/secrets/gateway.crt \
    --tls-key-file /run/secrets/gateway.key
```

Add `--tls-ca-file /run/secrets/client-ca.pem` when public clients must authenticate with client certificates. `--public-url` controls the URL advertised in the Agent Card; the TLS options control Uvicorn's listener.

Gateway-to-worker HTTP and TLS

The gateway-to-worker connection is independent of the public A2A listener. The public listener can use HTTPS while workers use HTTP, or the gateway can use HTTPS to workers while its public listener uses HTTP.

Table 4: Gateway-to-worker TLS options

Option	Process	Purpose
`--tls-cert-file` and `--tls-key-file`	Worker	Enable HTTPS on the worker. Supply both together.
`--tls-ca-file`	Worker	Optional CA for requiring and verifying gateway client certificates. Supplying it enables the worker side of mTLS.
`--worker-tls-ca-file`	Gateway	Optional CA bundle for verifying the worker's server certificate. Supply it alone for HTTPS with a private CA and no mTLS. If omitted, Requests uses its default CA bundle (which can be overridden with `REQUESTS_CA_BUNDLE`).
`--worker-tls-cert-file` and `--worker-tls-key-file`	Gateway	Optional gateway client certificate and key. Supply both together; the worker must require and trust this client certificate for mTLS.

Whenever a gateway `--worker-tls-*` option is supplied, register the worker with an HTTPS URL using `--worker-endpoint`. If no gateway worker TLS option is supplied, Requests still verifies an HTTPS worker certificate against its default CA bundle. When the worker has no advertised endpoint, the gateway builds an HTTP URL from the worker address and port in the Consul record.

For HTTPS without mTLS, configure the worker's certificate and key, then advertise its HTTPS URL. If the certificate uses a private CA, give that CA to the gateway with `--worker-tls-ca-file`. Do not set the gateway client certificate or the worker's `--tls-ca-file` in this mode:

```
select-ai a2a worker \
    --worker-endpoint https://worker.example.com:8443 \
    --tls-cert-file /run/secrets/worker.crt \
    --tls-key-file /run/secrets/worker.key

select-ai a2a serve \
    --deployment clustered \
    --worker-tls-ca-file /run/secrets/private-worker-ca.pem
```

For mTLS, the worker requires a client certificate with `--tls-ca-file`. The gateway presents its certificate and key with `--worker-tls-cert-file` and `--worker-tls-key-file`. The gateway also needs `--worker-tls-ca-file` when the worker certificate is signed by a private CA; omit it when the Requests default CA bundle trusts the worker certificate. The two directions can use the same CA or different CAs:

```
select-ai a2a worker \
    --worker-endpoint https://worker.example.com:8443 \
    --tls-cert-file /run/secrets/worker.crt \
    --tls-key-file /run/secrets/worker.key \
    --tls-ca-file /run/secrets/gateway-client-ca.pem

select-ai a2a serve \
    --deployment clustered \
    --worker-tls-ca-file /run/secrets/worker-server-ca.pem \
    --worker-tls-cert-file /run/secrets/gateway-client.crt \
    --worker-tls-key-file /run/secrets/gateway-client.key
```

The worker CA validates the gateway's client certificate. The gateway CA validates the worker's server certificate. This protects only the gateway-to-worker connection; it does not configure TLS for the public A2A listener or Oracle Database connections.

16.1.13 Google Cloud deployment

The repository contains complete deployment workflows under the gcloud directory. The two scripts deploy different topologies, so choose the directory that matches the desired runtime model.

Standalone Google Cloud deployment

gcloud/standalone/deploy.sh deploys one private Cloud Run service. Select its connection provisioning explicitly:

```
gcloud/standalone/deploy.sh \
    --project PROJECT_ID \
    --connection-mode dynamic \
    --build
```

The script creates or reuses an Artifact Registry repository, runtime service account, any requested Secret Manager secrets, and Cloud Run service. With no database options, dynamic standalone mode fixes neither the Connection URL nor AI Agent; its A2UI form requests Connection URL, username, password, and AI Agent. Pass `--db-dsn-secret` or `--a2a-team` to fix either value independently, or `--rotate-db-config` to configure the deployment's Connection URL. Fixed mode prompts for Connection URL, username, and password, requires `--a2a-team`, and creates the shared startup pool. The service is private; the script grants `run.routes.invoke` to the active deployment identity and the Gemini Enterprise Discovery Engine service agent.

Use `--build` when the current checkout should become a new container image. The script submits gcloud/standalone/cloudbuild.yaml to Cloud Build, which builds `docker/Dockerfile` from the repository root and pushes the image to Artifact Registry. Without `--build`, a later invocation reuses the image already deployed and only updates Cloud Run configuration or secrets.

Important standalone options include:

- `--connection-mode`: required `fixed` or `dynamic` provisioning.
- `--service`: Cloud Run service name; use a different service for each deployment profile.
- `--a2a-team`: fix the AI Agent in the deployment; required in fixed mode.
- `--require-oauth`: require an upstream-validated end-user bearer token and derive an owner scope as well as the A2A conversation scope. The SDK does not validate the token itself; see *End-user OAuth and session ownership*.
- `--pool-max-size`: maximum Oracle connections per Cloud Run instance.
- `--max-instances`: Cloud Run instance limit; dynamic standalone mode requires one.
- `--wallet-archive`: upload or replace an Autonomous Database wallet ZIP in fixed mode.
- `--db-dsn-secret`: fix the Connection URL using an existing Secret Manager secret.
- `--rotate-db-config`: prompt for and create or rotate the deployment's database values.

For a wallet deployment, `--wallet-archive` stores the ZIP and wallet password in service-specific Secret Manager secrets. Cloud Run mounts the ZIP read-only; the launcher expands it into ephemeral `/tmp` storage, verifies the wallet, and sets `SELECT_AI_WALLET_LOCATION` before starting the A2A server. Do not commit wallet files or put them in the container image.

The standalone script prints the deployed Agent Card JSON at the end. The service is ready for a client when the printed card points to the final Cloud Run URL and the client can invoke the private service.

The Google Cloud scripts do not require end-user OAuth by default. Cloud Run IAM authenticates Gemini Enterprise, and `a2a serve` separates database sessions by A2A conversation without asserting a verified human owner. Add `--require-oauth` to the deployment command when user ownership is required, and ensure a trusted upstream service validates the end-user token. The flag alone does not verify identity. Gemini Enterprise must then be configured with end-user OAuth rather than **Skip & Finish**. It uses `X-Serverless-Authorization` for Cloud Run IAM and sends the user token separately in `Authorization`.

Clustered Google Cloud deployment

gcloud/cluster/deploy.sh deploys the clustered topology: Cloud Run for the public server, GKE Autopilot for worker replicas, and Consul for discovery and routing.

```
gcloud/cluster/deploy.sh \
    --project PROJECT_ID \
    --region us-central1 \
    --worker-replicas 3
```

The script and gcloud/cluster/cloudbuild.yaml perform these steps:

1. Enable the required Google Cloud APIs and create or reuse Artifact Registry and a GKE Autopilot cluster.
2. Configure additive VPC DNS so StatefulSet worker names resolve to current Pod IPs after a worker is recreated.
3. Deploy the A2A namespace, internal Consul service, and worker service.
4. Build and publish the image with Cloud Build.
5. Deploy the requested worker replicas with `select-ai a2a worker` and the selected session TTL.
6. Deploy Cloud Run with `select-ai a2a serve --deployment clustered` and set `PUBLIC_URL` to the final Cloud Run URL.

The Cloud Run server uses direct VPC egress to reach the internal Consul load-balancer address and worker endpoints. The GKE worker service is headless service discovery, not a public load balancer. Consul selects a healthy worker when a session opens; the route is then pinned to that worker.

Important clustered options include:

- `--cluster` and `--gke-dns-domain`: GKE Autopilot cluster and immutable additive DNS domain.
- `--network` and `--subnet`: VPC path used by Cloud Run to reach the internal worker infrastructure.
- `--worker-replicas`: number of worker runtimes available for new sessions.
- `--session-ttl-seconds`: lifetime of temporary database sessions.
- `--conversation-retention-days`: number of days to retain Oracle conversations (default 7; use 0 to disable automatic removal).
- `--enable-worker-mtls`: create and use server-to-worker certificates.
- `--rotate-worker-mtls`: replace the existing test certificates and restart the worker workload.

The default clustered deployment uses private-VPC HTTP between Cloud Run and GKE. With `--enable-worker-mtls`, the script creates short-lived test PKI material in Secret Manager, creates the Kubernetes TLS secrets, deploys the worker StatefulSet variant, and mounts the server client certificate into Cloud Run. The worker certificate uses the GKE StatefulSet DNS name, so the `--gke-dns-domain` value must remain consistent with the cluster.

The clustered deployment does not upload an Oracle wallet because the dynamic worker session path currently accepts only DSN/user/password. If database mTLS is required, use the standalone deployment or provide a separate database connection mechanism to the worker implementation.

The deployment files are intended to be read together:

Table 5: Google Cloud A2A deployment files

File	Role
gcloud/standalone/deploy.sh	Creates and updates one private Cloud Run standalone server.
gcloud/standalone/cloudbuild.yaml	Builds and publishes the standalone container image.
gcloud/standalone/README.md	Documents IAM, secrets, wallet archives, and update behavior.
gcloud/cluster/deploy.sh	Creates or reuses the clustered GKE/Cloud Run topology and supplies build substitutions.
gcloud/cluster/cloudbuild.yaml	Builds the image, deploys Consul and workers, and deploys Cloud Run.
gcloud/cluster/gke manifests	Namespace, Consul, headless worker service, and HTTP or mTLS worker workloads.
gcloud/cluster/README.md	Explains the topology, DNS, mTLS test mode, and operational details.

16.1.14 Troubleshooting and security notes

- If the Agent Card advertises the wrong URL, set `--public-url` or `PUBLIC_URL`. The URL must be the client-visible base URL, not an internal container address.
- If the gateway returns the connection form repeatedly, check that the worker is passing in Consul, that the worker can resolve and connect to Oracle, and that the session TTL has not expired.
- If a task can no longer be retrieved after a worker restart, submit the A2UI form again. Consul routes are intentionally tied to the worker session, even though task data is durable in Oracle.
- Do not put database passwords in source files, container images, Consul KV, or deployment logs. Use Secret Manager for fixed standalone credentials and protect the A2UI/gateway path with HTTPS.
- Gateway mTLS and Oracle wallet mTLS are different controls. Gateway mTLS authenticates the internal HTTP peer; an Oracle wallet authenticates the database connection and is currently available only to the standalone server path.

See *Command Line Interface* for the common CLI connection options and *Connection* for Select AI database connection setup.

CHAPTER

SEVENTEEN

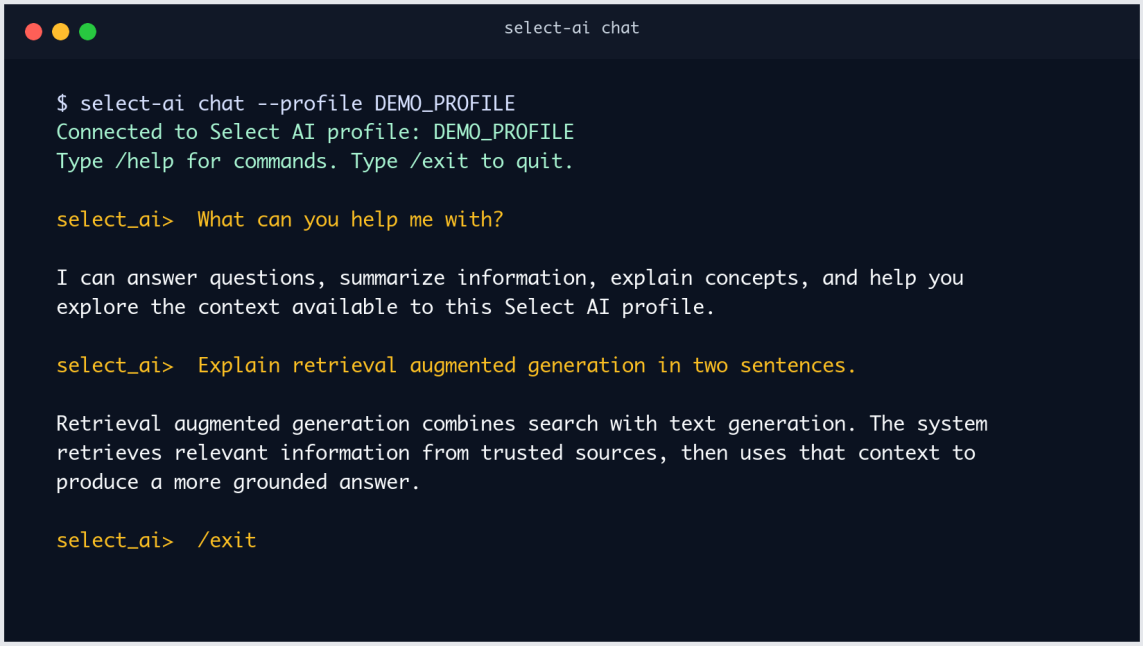
COMMAND LINE INTERFACE

17.1 Command line interface

The `select-ai` command line interface (CLI) provides a terminal workflow for using Select AI profiles without writing Python code. It is intended for quick exploration, profile validation, prompt testing, SQL generation, summarization, translation, and interactive chat against an existing profile.

The CLI is useful for developers who want a fast shell workflow and for non-developers who are comfortable running terminal commands but do not want to write Python code. It can help users check whether a profile is configured correctly, inspect generated SQL, try prompts during development, or interact with a curated profile through a simple command.

The CLI works with Select AI profiles and can also run the Select AI A2A server, clustered gateway and worker, and Agent Card generator. RAG is supported when the selected profile is already configured with a vector index. The *A2A guide* explains the difference between fixed and dynamic connections and standalone and clustered topologies; this page gives the corresponding CLI commands.



The package provides an optional `select-ai` command line tool. Install the CLI extra to use profile, SQL, and chat commands. Install the separate `a2a` extra for the A2A server and worker commands. If A2A dependencies are missing,

invoking the A2A command group or a server or worker command reports how to install them; the other CLI commands remain available:

```
pip install 'select_ai[cli]'
# Install A2A support:
pip install 'select_ai[a2a]'
```

Use `select-ai --help` to view the available command groups and `select-ai <command> --help` to view options for a specific command.

The A2A examples below pass database credentials with CLI options so the commands are easy to copy and adapt. Environment-variable equivalents are listed in the connection-options table.

17.1.1 Connection options

Database-backed commands accept the following connection options. The `select-ai a2a serve` command accepts these options directly. In standalone mode, providing all four connection values at startup selects the fixed connection path; omitting values opens a dynamic A2UI form for them. In clustered mode, the gateway can preconfigure any values and the form collects the rest. The worker receives the resolved values when it opens a session. For a private database CA, use `--database-tls-ca-file`; the CLI reads PEM from the specified file.

Table 1: Connection options

Option	Environment variable
`--user`	`SELECT_AI_USER`
`--password`	`SELECT_AI_PASSWORD`
`--dsn`	`SELECT_AI_DB_CONNECT_STRING`
`--wallet-location`	`SELECT_AI_WALLET_LOCATION`
`--wallet-password`	`SELECT_AI_WALLET_PASSWORD`
`--database-tls-ca-file`	`SELECT_AI_DATABASE_TLS_CA_FILE`

Profile, SQL, and chat commands prompt for the database password when neither `--password` nor `SELECT_AI_PASSWORD` is set. For `a2a serve`, omitting any connection value selects dynamic mode, and the A2A client supplies the missing values through an A2UI form. Supply `--dsn`, `--user`, `--password`, and `--team` for fixed standalone mode. Wallet options apply to fixed standalone A2A connections; dynamic sessions do not currently support Oracle wallet authentication. The A2A CLI examples below include a host, port, service name, credentials, and optional database CA file.

17.1.2 Interactive chat

The `chat` subcommand starts an interactive profile chat Read-Eval-Print Loop (REPL). A REPL is a terminal session that reads each prompt you type, evaluates it, prints the response, and then waits for the next prompt. Pass an existing Select AI profile with `--profile`:

```
select-ai chat --profile OCI_AI_PROFILE
```

The REPL uses `Profile.chat_session()` so prompts in the same terminal session share conversation context. Responses stream by default. Use `--no-stream` to print each response after it is fully generated.

```
Connected to Select AI profile: OCI_AI_PROFILE
Type /help for commands. Type /exit to quit.
select_ai> What tables can I ask about?
...
select_ai> /exit
```

Useful options:

- `--user`, `--password`, and `--dsn` override the environment values.
- `--wallet-location` and `--wallet-password` configure wallet connections.
- `--chunk-size` controls the number of CLOB characters read per stream chunk.
- `--conversation-length` controls how many prompts are retained in context.
- `--keep-conversation` keeps the database conversation after the REPL exits.

Inside the REPL, use these commands:

Table 2: Chat REPL commands

Command	Description
`/help`	Show available REPL commands.
`/clear`	Start a fresh database conversation.
`/exit`	Exit the chat session.
`/quit`	Exit the chat session.

17.1.3 SQL commands

SQL operations are one-shot subcommands instead of a REPL:

```
select-ai sql show --profile OCI_AI_PROFILE "count movies by genre"
select-ai sql run --profile OCI_AI_PROFILE "count movies by genre"
select-ai sql explain --profile OCI_AI_PROFILE "count movies by genre"
select-ai sql narrate --profile OCI_AI_PROFILE "count movies by genre"
```

`show`, `explain`, and `narrate` stream text output by default. Use `--no-stream` to print the response after it is fully generated, and `--chunk-size` to control the number of CLOB characters read per stream chunk.

`run` executes the generated SQL and prints the returned result table. It does not support streaming.

17.1.4 Profile commands

Summarize and translate are available under the `profile` command group:

```
select-ai profile list
select-ai profile list --pattern "OCI.*"

select-ai profile summarize --profile OCI_AI_PROFILE "Text to summarize"
select-ai profile summarize --profile OCI_AI_PROFILE --file notes.txt
select-ai profile summarize \
    --profile OCI_AI_PROFILE \
    --location-uri https://example.com/article.txt
select-ai profile summarize \
    --profile OCI_AI_PROFILE \
    --location-uri https://objectstorage.example.com/n/namespace/b/bucket/o/file.txt \
    --credential-name OBJECT_STORE_CRED

select-ai profile translate \
    --profile OCI_AI_PROFILE \
    --source-language English \
```

(continues on next page)

(continued from previous page)

```
    --target-language German \
    "Thank you"
```

`profile list` prints profile names visible to the connected database user. Use `--pattern` to filter names with a regular expression.

`profile summarize` accepts one content source at a time: inline text, `--file`, or `--location-uri`. Use `--prompt` to guide the summary and `--credential-name` when the location URI requires an object storage credential.

17.1.5 A2A commands

The `a2a` command group exposes the cloud-neutral A2A runtimes. Use `select-ai a2a --help` or a command-specific `--help` option to see the current defaults:

```
select-ai a2a --help
select-ai a2a serve --help
select-ai a2a worker --help
select-ai a2a agent-card --help
```

Table 3: A2A CLI commands

Command	Purpose
`select-ai a2a serve`	Start the public A2A HTTP server. The default standalone topology needs no worker or Consul. Add `--deployment clustered` to use a gateway, workers, and Consul.
`select-ai a2a worker`	Start an internal database worker for clustered topology. Standalone topology does not use this command.
`select-ai a2a agent-card`	Print a Gemini Enterprise-compatible A2A v0.3 Agent Card without starting a server.

A2A deployment examples

The *A2A guide* explains how fixed and dynamic connections differ from standalone and clustered topologies. These examples use CLI options for the database settings. Replace the placeholder credentials and connection string with your values.

Fixed standalone connection

Pass all four connection values at startup to use the standalone server's shared connection pool. This needs no Consul or worker and supports streaming:

```
select-ai a2a serve \
    --deployment standalone \
    --team ORACLE_AI_DATABASE_AGENT \
    --user DB_USERNAME \
    --password 'DB_PASSWORD' \
    --dsn 'dbhost.example.com:1521/db_service' \
    --database-tls-ca-file /run/secrets/database-ca.pem \
    --database-call-timeout-seconds 120 \
    --host 0.0.0.0 \
    --port 8000 \
    --public-url https://a2a.example.com
```

The database CA file is optional when the server certificate is already trusted by the operating system. `--wallet-location` and `--wallet-password` are also available for fixed standalone connections that use an Oracle wallet.

Dynamic standalone connection

Omit one or more of `--dsn`, `--user`, `--password`, and `--team` to have the A2A client supply those values through an A2UI form. The server runs the session child processes itself, so this mode needs no Consul or separate worker. Run a single Uvicorn process. Dynamic standalone supports task polling, not streaming:

```
select-ai a2a serve \
    --deployment standalone \
    --database-tls-ca-file /run/secrets/database-ca.pem \
    --host 0.0.0.0 \
    --port 8000 \
    --public-url https://a2a.example.com
```

Remove `--database-tls-ca-file` when the database server certificate is already trusted by the operating system. Any connection values provided on the command line are fixed; the form asks only for the values that are missing.

Clustered gateway and worker

Clustered topology requires Consul and at least one worker. Consul is the worker directory and session-routing service: it helps the gateway find a healthy worker for a new session and send later requests back to that worker. See *Why clustered deployments use Consul* for a worked example. Point the gateway and all workers at the same Consul service with `--consul-url`.

The gateway can run without database credentials; by default, the A2A client supplies them through the connection form. For local development, start Consul and then run the worker and gateway in separate terminals:

```
# Terminal 1: local development Consul agent
consul agent -dev -bind=127.0.0.1 -client=127.0.0.1

# Terminal 2: database worker
select-ai a2a worker \
    --host 127.0.0.1 \
    --port 8081 \
    --consul-url http://127.0.0.1:8500 \
    --worker-id local-worker \
    --worker-endpoint http://127.0.0.1:8081 \
    --database-call-timeout-seconds 120

# Terminal 3: public gateway
select-ai a2a serve \
    --deployment clustered \
    --host 127.0.0.1 \
    --port 8000 \
    --public-url http://127.0.0.1:8000 \
    --consul-url http://127.0.0.1:8500 \
    --database-tls-ca-file /run/secrets/database-ca.pem \
    --database-call-timeout-seconds 120
```

The local Consul command is for development only. In a deployment, point the gateway and workers at the existing Consul service and use network-reachable worker addresses. The database CA option is needed only for a private database CA; the gateway passes its PEM contents to the worker when opening a session. Set the same

`--database-call-timeout-seconds` value on the gateway and worker. See the A2A guide for Consul setup and clustered deployment details.

Important `serve` options include `--team`, `--host`, `--port`, `--public-url`, `--description`, `--pool-max-size`, and `--database-call-timeout-seconds`. The default database call timeout is 120 seconds. On the clustered gateway, this value sets the request deadline basis; the worker applies it to Oracle calls and derives its child-process deadline.

By default, `a2a serve` does not require an application OAuth token. A2A task and context records use a shared owner scope, and each A2A context maps to its own Select AI conversation. Add `--require-oauth` to require `Authorization: Bearer ...` and derive an owner scope as well as context. The SDK does not validate token signatures or claims. A trusted upstream service must validate the end-user token and prevent direct access that bypasses validation. See *the A2A authentication contract*. The Agent Card advertises bearer security only in that mode.

Clustered option reference

Configure discovery first, then connection values and session limits. Explicit CLI values take precedence over environment fallbacks. For a fixed clustered connection, see the *complete gateway command*.

Gateway discovery

`--public-url` (`PUBLIC_URL`)
: Client-visible base URL advertised in the Agent Card.

`--consul-url` (`CONSUL_HTTP_URL`)
: Consul HTTP API URL. Use the same Consul service as the workers.

`--worker-service` (`WORKER_SERVICE`)
: Consul service to discover. Default: `select-ai-a2a-worker`.

Worker registration

`--host` and `--port`
: Worker listen address and port. The port is also registered with Consul.

`--worker-id` (`WORKER_ID`)
: Unique worker identifier. Defaults to the host name.

`--consul-url` (`CONSUL_HTTP_URL`)
: Consul HTTP API URL used to register the worker and send heartbeats.

`--worker-endpoint` (`WORKER_ENDPOINT`)
: Worker URL reachable by the gateway. In cluster manifests, `WORKER_ADDRESS` can supply the pod IP when no endpoint is given.

Connection values and forms

`--dsn`, `--user`, `--password`, and `--team` (gateway)
: Fix the supplied values at startup. Set all four for a fixed connection; otherwise the A2UI form asks only for the missing values. The environment fallbacks are `SELECT_AI_DB_CONNECT_STRING`, `SELECT_AI_USER`, `SELECT_AI_PASSWORD`, and `SELECT_AI_A2A_TEAM`.

`--a2ui-form` (gateway)
: Path to a custom A2UI JSON form. It must submit exactly the missing connection values. See *A2A Integration* for the form contract.

Session lifetime and timeouts

--session-ttl-seconds (gateway and worker)
Lifetime of a temporary database session; default 900 seconds. The gateway also reads SESSION_TTL_SECONDS. Configure the same lifetime on both processes.

--conversation-retention-days (worker)
Days to retain Oracle conversations; default 7. Environment fallback: CONVERSATION_RETENTION_DAYS. Use 0 to retain until manually deleted.

--session-start-timeout-seconds (worker)
Maximum wait for a child process to initialize its database session.

--database-call-timeout-seconds (gateway and worker)
Database call timeout; default 120 seconds. Set the same value on both processes. The worker applies it to Oracle calls; both processes derive their request deadlines from it.

Authentication and TLS

--require-oauth (gateway)
Require a bearer token and derive an owner scope for task and context records. Token validation must occur upstream. Uses the same ownership behavior as standalone mode.

--tls-cert-file and --tls-key-file (gateway)
Enable HTTPS on the public A2A listener. Supply both together; without them, the listener uses HTTP.

--tls-ca-file (gateway)
Optional CA that makes the public listener require and verify client certificates (mTLS). Requires --tls-cert-file and --tls-key-file.

--worker-tls-ca-file, --worker-tls-cert-file, --worker-tls-key-file
Gateway TLS options for worker connections. The CA verifies the worker's server certificate; it can be supplied alone for HTTPS without mTLS. If omitted, Requests uses its default CA bundle. The client certificate and key are optional, but must be supplied together; the worker must require and trust them for mTLS. When any --worker-tls-* option is used, set --worker-endpoint to an HTTPS URL. Environment fallbacks are WORKER_TLS_CA_FILE, WORKER_TLS_CERT_FILE, and WORKER_TLS_KEY_FILE.

--tls-ca-file, --tls-cert-file, --tls-key-file (worker)
The certificate and key enable HTTPS on the worker and must be supplied together. --tls-ca-file is optional; when supplied, the worker requires and verifies client certificates for mTLS. Set --worker-endpoint to an HTTPS URL so the gateway connects over TLS.

--database-tls-ca-file (gateway)
CA for verifying the Oracle database certificate. The gateway passes the PEM contents to the worker session. This is separate from the certificates for the gateway-to-worker HTTP connection and from Oracle wallet mTLS.

Agent Card generation

Print the portable A2A v0.3 card for a standalone public service:

```
select-ai a2a agent-card \
    --team ORACLE_AI_DATABASE_AGENT \
    --public-url https://a2a.example.com
```

--team and --public-url are required; use --description to replace the default description. See *A2A Integration* for protocol compatibility, persistent state, gateway session behavior, wallet support, and Google Cloud deployment details.

17.1.6 Command summary

Table 4: CLI command summary

Command	Purpose
`select-ai chat`	Start an interactive context-aware chat session.
`select-ai sql show`	Generate SQL without executing it.
`select-ai sql run`	Generate SQL, execute it, and print the result table.
`select-ai sql explain`	Explain generated SQL.
`select-ai sql narrate`	Generate and execute SQL, then return a natural language answer.
`select-ai profile list`	List saved profile names.
`select-ai profile summarize`	Summarize inline content, a file, or a URI.
`select-ai profile translate`	Translate text with a saved profile.
`select-ai a2a serve`	Start the public A2A server. The default is standalone; use `--deployment clustered` for a gateway and workers. Omitted connection values are collected through A2UI.
`select-ai a2a worker`	Start a database worker for clustered topology. Standalone deployments do not use a worker.
`select-ai a2a agent-card`	Print an A2A v0.3 Agent Card.

CHAPTER

EIGHTEEN

WEB FRAMEWORKS

18.1 Using `select_ai` with Python web frameworks

Python web applications should create a Select AI connection pool when the application starts and close it when the application shuts down. A pool lets concurrent requests share a bounded set of database connections instead of creating standalone connections per request.

This pattern works with Python Web Server Gateway Interface (WSGI) and Asynchronous Server Gateway Interface (ASGI) frameworks. FastAPI is used below as a concrete example, but the same approach applies to frameworks such as Flask, Django, Starlette, Sanic, and Quart: initialize the pool during application startup, use `select_ai` APIs inside request handlers, and close the pool during application shutdown.

Do not call `select_ai.connect()` or `select_ai.create_pool()` inside every request handler. Creating connections per request adds latency, increases database connection churn, and can exhaust the database connection limit under load. Create one pool per worker process and let `select_ai` acquire and release connections from that pool for each API call.

For background and concurrency measurements, see this connection pooling blog.

18.1.1 Framework patterns

Use the framework lifecycle API that runs once per process:

- FastAPI / Starlette: use a `lifespan` async context manager. Create the pool before `yield` and close it after `yield`.
- Flask: create the pool in `create_app()`. Close the pool from the shutdown hook provided by the process that runs Flask, such as a Gunicorn `worker_exit` hook. For simple local applications, `atexit.register()` can be used for normal interpreter shutdown.
- Django: create the pool in `AppConfig.ready()`. Close the pool from the shutdown hook provided by the process that runs Django, such as a Gunicorn `worker_exit` hook. For simple local applications, `atexit.register()` can be used for normal interpreter shutdown.
- Quart: use `@app.before_serving` / `@app.after_serving`, or `@app.while_serving` with cleanup after `yield`.
- Sanic: use `@app.before_server_start` and `@app.after_server_stop`.

Use `select_ai.create_pool()` for synchronous request handlers and `select_ai.create_pool_async()` for asynchronous request handlers. In general, synchronous routes should call synchronous Select AI methods, and async routes should call async Select AI methods. Avoid mixing blocking synchronous database calls into async routes unless the framework runs them in a worker thread.

For Flask and Django, be careful with hooks that run per request or per application context. A Select AI pool should live for the worker process, not for a single request. For example, Flask's `teardown_appcontext` runs when an application context is popped, so it is not a good place to close a process-wide pool after every request.

18.1.2 Install dependencies

Install `select_ai` and FastAPI server dependencies:

```
python -m pip install select_ai fastapi uvicorn
```

For local development, set the database connection details as environment variables:

```
export SELECT_AI_USER=<select_ai_db_user>
export SELECT_AI_PASSWORD=<select_ai_db_password>
export SELECT_AI_DB_CONNECT_STRING=<db_connect_string>
export SELECT_AI_POOL_MIN=5
export SELECT_AI_POOL_MAX=10
export SELECT_AI_POOL_INCREMENT=5
```

If you use an mTLS wallet, also set `TNS_ADMIN` or pass wallet parameters to `select_ai.create_pool()` / `select_ai.create_pool_async()`.

For production deployments, store these values in your deployment platform's secret manager or environment configuration. Do not hard-code database passwords, wallet passwords, or provider credentials in application source.

18.1.3 FastAPI synchronous endpoints

Create a file named app.py:

```
import os
from contextlib import asynccontextmanager

from fastapi import FastAPI

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

pool_min = int(os.getenv("SELECT_AI_POOL_MIN", "5"))
pool_max = int(os.getenv("SELECT_AI_POOL_MAX", "10"))
pool_increment = int(os.getenv("SELECT_AI_POOL_INCREMENT", "5"))


@asynccontextmanager
async def lifespan(app: FastAPI):
    select_ai.create_pool(
        user=user,
        password=password,
        dsn=dsn,
        min_size=pool_min,
        max_size=pool_max,
        increment=pool_increment,
    )
    yield
    select_ai.disconnect()


app = FastAPI(lifespan=lifespan)


@app.get("/chat")
def chat(prompt: str):
    profile = select_ai.Profile(profile_name="oci_ai_profile")
    return {"response": profile.chat(prompt=prompt)}


@app.get("/show_sql")
def show_sql(prompt: str):
    profile = select_ai.Profile(profile_name="oci_ai_profile")
    return {"sql": profile.show_sql(prompt=prompt)}
```

Start the server:

```
uvicorn app:app --host 0.0.0.0 --port 8000
```

Call the service:

```
curl "http://localhost:8000/chat?prompt=What%20is%20OCI%3F"
```

Stop the server by pressing `Ctrl+C` in the terminal where `uvicorn` is running. FastAPI runs the lifespan shutdown hook and `select_ai.disconnect()` closes the pool.

This example constructs `Profile` inside each handler. Construction reads the saved profile description and attributes from the database, acquiring connections from the pool before the action itself runs. This keeps metadata current but adds database calls to each request. Reusing an initialized profile avoids those reads; refresh it when saved profile settings change, and avoid mutating a shared profile object while requests are using it.

18.1.4 FastAPI asynchronous endpoints

For async endpoints, initialize the async pool with `select_ai.create_pool_async()` and close it with `select_ai.async_disconnect()`.

```
import os
from contextlib import asynccontextmanager

from fastapi import FastAPI

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")


@asynccontextmanager
async def lifespan(app: FastAPI):
    select_ai.create_pool_async(
        user=user,
        password=password,
        dsn=dsn,
        min_size=5,
        max_size=10,
        increment=5,
    )
    yield
    await select_ai.async_disconnect()


app = FastAPI(lifespan=lifespan)


@app.get("/chat")
async def chat(prompt: str):
    profile = await select_ai.AsyncProfile(
        profile_name="async_oci_ai_profile"
    )
    return {"response": await profile.chat(prompt=prompt)}
```

Start and stop the server the same way:

```
uvicorn app:app --host 0.0.0.0 --port 8000
```

Press `Ctrl+C` to stop it.

18.1.5 Flask example

Flask applications normally use synchronous request handlers, so initialize a synchronous Select AI pool when the application is created.

```
import atexit
import os

from flask import Flask, jsonify, request

import select_ai


def create_app():
    app = Flask(__name__)

    select_ai.create_pool(
        user=os.getenv("SELECT_AI_USER"),
        password=os.getenv("SELECT_AI_PASSWORD"),
        dsn=os.getenv("SELECT_AI_DB_CONNECT_STRING"),
        min_size=int(os.getenv("SELECT_AI_POOL_MIN", "1")),
        max_size=int(os.getenv("SELECT_AI_POOL_MAX", "4")),
        increment=int(os.getenv("SELECT_AI_POOL_INCREMENT", "1")),
    )

    @app.get("/show_sql")
    def show_sql():
        prompt = request.args["prompt"]
        profile = select_ai.Profile(profile_name="oci_ai_profile")
        return jsonify({"sql": profile.show_sql(prompt=prompt)})

    @atexit.register
    def close_select_ai_pool():
        if select_ai.is_connected():
            select_ai.disconnect()

    return app
```

18.1.6 Django example

Django has its own database connection management, but `select_ai` uses the Oracle Database connection pool created by this package. Create the Select AI pool once per process, then call `Profile` APIs inside views.

```
# myapp/apps.py
import os

from django.apps import AppConfig

import select_ai


class MyAppConfig(AppConfig):
    name = "myapp"

    def ready(self):
        if not select_ai.is_connected():
            select_ai.create_pool(
                user=os.getenv("SELECT_AI_USER"),
                password=os.getenv("SELECT_AI_PASSWORD"),
                dsn=os.getenv("SELECT_AI_DB_CONNECT_STRING"),
                min_size=int(os.getenv("SELECT_AI_POOL_MIN", "1")),
                max_size=int(os.getenv("SELECT_AI_POOL_MAX", "4")),
                increment=int(os.getenv("SELECT_AI_POOL_INCREMENT", "1")),
            )
```

Add the view in `myapp/views.py`:

```
# myapp/views.py
from django.http import JsonResponse

import select_ai


def show_sql(request):
    profile = select_ai.Profile(profile_name="oci_ai_profile")
    sql = profile.show_sql(prompt=request.GET["prompt"])
    return JsonResponse({"sql": sql})
```

When using Django's development autoreloader, startup hooks may run more than once. The `is_connected()` check prevents this example from creating a second pool in the same process.

18.1.7 Pool sizing

Use connection pooling for concurrent services such as API applications, workloads with mixed fast and slow requests, and applications with tail-latency requirements. Use standalone connections for simple scripts, command-line tools, or low-concurrency batch jobs.

Set pool sizing based on expected request concurrency and database capacity. In multi-worker deployments, each worker process creates its own pool, so total possible database connections are approximately:

```
workers * SELECT_AI_POOL_MAX
```

Choose pool sizes that leave capacity for other database clients and avoid overwhelming small database deployments.

For example, a service running four worker processes with `SELECT_AI_POOL_MAX=10` can open up to forty Select AI database connections. If the database can only spare twenty connections for the application, use fewer workers, reduce `SELECT_AI_POOL_MAX`, or both.

18.1.8 Pool wait behavior

`select_ai.create_pool()` and `select_ai.create_pool_async()` pass pool options through to `python-oracledb`. Use `wait_timeout` and `getmode` to control what happens when all pooled connections are busy.

```
import oracledb

select_ai.create_pool(
    user=user,
    password=password,
    dsn=dsn,
    min_size=2,
    max_size=8,
    increment=2,
    getmode=oracledb.POOL_GETMODE_TIMEDWAIT,
    wait_timeout=10_000,
)
```

`wait_timeout` is in milliseconds and takes effect only with `getmode=oracledb.POOL_GETMODE_TIMEDWAIT`. The example waits up to ten seconds for a connection. The default `POOL_GETMODE_WAIT` waits without this limit. Pool acquisition timeouts do not limit SQL execution or model generation time. See the python-oracledb pool parameters.

Choose a timeout that matches your API latency budget. For public HTTP APIs, it is usually better to fail fast and return an application error than to let requests pile up until every worker is blocked.

18.1.9 Request handling

Validate prompts and profile names before calling Select AI methods. If clients can choose a profile, check the requested profile against an application allowlist instead of passing arbitrary user input directly into `Profile(profile_name=...)`.

For long-running prompts, set HTTP server timeouts and client timeouts deliberately. Text generation and RAG calls can take longer than simple SQL metadata operations, especially when external AI providers or object storage are involved.

For streaming responses, prefer async frameworks and async Select AI methods when the rest of the application is already async. For ordinary JSON responses, either synchronous or asynchronous routes are fine as long as the connection pool matches the route style.

CHAPTER

NINETEEN

CONCURRENT PROMPT PROCESSING

19.1 Concurrent prompt processing

Use concurrent prompt processing when an application needs to send multiple independent prompts without waiting for each prompt to finish before starting the next one. The `select_ai` module supports this pattern with both the synchronous `Profile` API and the asynchronous `AsyncProfile` API.

Create a connection pool before running concurrent work. Use `select_ai.create_pool()` for synchronous recipes and `select_ai.create_pool_async()` for asynchronous recipes.

19.1.1 Recipe summary

Recipe	Script	When to use
Sync completion	*sync_thread_pool.py*	Use `ThreadPoolExecutor` when prompts are independent and results can be handled as soon as each prompt completes.
Sync input order	*sync_ordered_results.py*	Use `ThreadPoolExecutor.map()` when result order must match the input prompt order.
Sync queue	*sync_queue_workers.py*	Use worker threads and a queue for producer-consumer workloads where prompts may arrive over time.
Async input order	*async_gather.py*	Use `asyncio.gather()` when result order must match the input prompt order.
Async completion	*async_as_completed.py*	Use `asyncio.as_completed()` when each result should be processed as soon as it is available.
Async pipeline	*async_pipeline.py*	Use `run_pipeline()` when all prompt/action pairs are known up front and should be submitted as a database pipeline.
Async queue	*async_queue_workers.py*	Use async queue workers for long-running async services or background prompt processors.

19.1.2 Environment variables

The recipes use the same connection environment variables as the other samples:

```
export SELECT_AI_USER=<select_ai_db_user>
export SELECT_AI_PASSWORD=<select_ai_db_password>
export SELECT_AI_DB_CONNECT_STRING=<db_connect_string>
```

Optional environment variables control pool sizing and profile names:

```
export SELECT_AI_POOL_MIN=1
export SELECT_AI_POOL_MAX=4
export SELECT_AI_POOL_INCREMENT=1
export SELECT_AI_PROFILE_NAME=oci_ai_profile
```

Use `SELECT_AI_PROFILE_NAME=async_oci_ai_profile` for the async recipes if that is the async profile name in your environment.

19.1.3 sync_thread_pool.py

This recipe uses `ThreadPoolExecutor` and `as_completed()`. Results are printed in the order they finish.

Set up the imports, connection values, prompts, and any helper functions:

```
import os
from concurrent.futures import ThreadPoolExecutor, as_completed

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

pool_min = int(os.getenv("SELECT_AI_POOL_MIN", "1"))
pool_max = int(os.getenv("SELECT_AI_POOL_MAX", "4"))
pool_increment = int(os.getenv("SELECT_AI_POOL_INCREMENT", "1"))

profile_name = os.getenv("SELECT_AI_PROFILE_NAME", "oci_ai_profile")
max_workers = int(os.getenv("SELECT_AI_MAX_WORKERS", str(pool_max)))

prompts = [
    "How many customers?",
    "How many products?",
    "How many promotions?",
    "List the top 5 customers by sales.",
]


def show_sql(prompt):
    profile = select_ai.Profile(profile_name=profile_name)
    return prompt, profile.show_sql(prompt=prompt)
```

The following code creates the connection pool, processes the prompts, and closes the pool in a `finally` block:

```
select_ai.create_pool(
    user=user,
    password=password,
    dsn=dsn,
    min_size=pool_min,
    max_size=pool_max,
    increment=pool_increment,
)

try:
    with ThreadPoolExecutor(max_workers=max_workers) as executor:
        futures = [executor.submit(show_sql, prompt) for prompt in prompts]

        for future in as_completed(futures):
            prompt, sql = future.result()
            print(f"\nPrompt: {prompt}")
            print(sql)
finally:
    select_ai.disconnect()
```

19.1.4 sync_ordered_results.py

This recipe uses `ThreadPoolExecutor.map()`. Prompts run concurrently, but results are printed in the same order as the input list.

Set up the imports, connection values, prompts, and any helper functions:

```
import os
from concurrent.futures import ThreadPoolExecutor

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

pool_min = int(os.getenv("SELECT_AI_POOL_MIN", "1"))
pool_max = int(os.getenv("SELECT_AI_POOL_MAX", "4"))
pool_increment = int(os.getenv("SELECT_AI_POOL_INCREMENT", "1"))

profile_name = os.getenv("SELECT_AI_PROFILE_NAME", "oci_ai_profile")
max_workers = int(os.getenv("SELECT_AI_MAX_WORKERS", str(pool_max)))

prompts = [
    "How many customers?",
    "How many products?",
    "How many promotions?",
    "List the top 5 customers by sales.",
]


def show_sql(prompt):
    profile = select_ai.Profile(profile_name=profile_name)
    return profile.show_sql(prompt=prompt)
```

The following code creates the connection pool, processes the prompts, and closes the pool in a `finally` block:

```
select_ai.create_pool(
    user=user,
    password=password,
    dsn=dsn,
    min_size=pool_min,
    max_size=pool_max,
    increment=pool_increment,
)

try:
    with ThreadPoolExecutor(max_workers=max_workers) as executor:
        results = executor.map(show_sql, prompts)

        for prompt, sql in zip(prompts, results):
            print(f"\nPrompt: {prompt}")
            print(sql)
finally:
    select_ai.disconnect()
```

19.1.5 sync_queue_workers.py

This recipe uses worker threads and `queue.Queue`. It is useful when prompt producers and prompt processors are separate parts of an application.

Set up the imports, connection values, prompts, and any helper functions:

```
import os
from queue import Queue
from threading import Thread

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

pool_min = int(os.getenv("SELECT_AI_POOL_MIN", "1"))
pool_max = int(os.getenv("SELECT_AI_POOL_MAX", "4"))
pool_increment = int(os.getenv("SELECT_AI_POOL_INCREMENT", "1"))

profile_name = os.getenv("SELECT_AI_PROFILE_NAME", "oci_ai_profile")
worker_count = int(os.getenv("SELECT_AI_WORKER_COUNT", str(pool_max)))

prompts = [
    "How many customers?",
    "How many products?",
    "How many promotions?",
    "List the top 5 customers by sales.",
]


def worker(queue, results):
    while True:
        item = queue.get()
        try:
            if item is None:
                return

            index, prompt = item
            profile = select_ai.Profile(profile_name=profile_name)
            sql = profile.show_sql(prompt=prompt)
            results[index] = (prompt, sql)
        finally:
            queue.task_done()
```

The following code creates the connection pool, processes the prompts, and closes the pool in a `finally` block:

```
select_ai.create_pool(
    user=user,
    password=password,
    dsn=dsn,
    min_size=pool_min,
    max_size=pool_max,
    increment=pool_increment,
)

try:
    queue = Queue()
    results = [None] * len(prompts)

    workers = [
        Thread(target=worker, args=(queue, results))
        for _ in range(worker_count)
    ]
    for thread in workers:
        thread.start()

    for index, prompt in enumerate(prompts):
        queue.put((index, prompt))

    for _ in workers:
        queue.put(None)

    queue.join()
    for thread in workers:
        thread.join()

    for prompt, sql in results:
        print(f"\nPrompt: {prompt}")
        print(sql)
finally:
    select_ai.disconnect()
```

19.1.6 async_gather.py

This recipe uses `asyncio.gather()`. Prompts run concurrently, and results are returned in the same order as the input task list.

Set up the imports, connection values, prompts, and any helper functions:

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

pool_min = int(os.getenv("SELECT_AI_POOL_MIN", "1"))
pool_max = int(os.getenv("SELECT_AI_POOL_MAX", "4"))
pool_increment = int(os.getenv("SELECT_AI_POOL_INCREMENT", "1"))

profile_name = os.getenv("SELECT_AI_PROFILE_NAME", "async_oci_ai_profile")

prompts = [
    "How many customers?",
    "How many products?",
    "How many promotions?",
    "List the top 5 customers by sales.",
]


async def show_sql(profile, prompt):
    return await profile.show_sql(prompt=prompt)
```

The following code creates the connection pool, processes the prompts, and closes the pool in a `finally` block:

```
async def main():
    select_ai.create_pool_async(
        user=user,
        password=password,
        dsn=dsn,
        min_size=pool_min,
        max_size=pool_max,
        increment=pool_increment,
    )

    try:
        profile = await select_ai.AsyncProfile(profile_name=profile_name)

        tasks = [show_sql(profile, prompt) for prompt in prompts]
        results = await asyncio.gather(*tasks)

        for prompt, sql in zip(prompts, results):
            print(f"\nPrompt: {prompt}")
            print(sql)
    finally:
        await select_ai.async_disconnect()


asyncio.run(main())
```

19.1.7 async_as_completed.py

This recipe uses `asyncio.as_completed()`. It is useful for command-line tools or services that can forward each answer as soon as it is ready.

Set up the imports, connection values, prompts, and any helper functions:

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

pool_min = int(os.getenv("SELECT_AI_POOL_MIN", "1"))
pool_max = int(os.getenv("SELECT_AI_POOL_MAX", "4"))
pool_increment = int(os.getenv("SELECT_AI_POOL_INCREMENT", "1"))

profile_name = os.getenv("SELECT_AI_PROFILE_NAME", "async_oci_ai_profile")

prompts = [
    "How many customers?",
    "How many products?",
    "How many promotions?",
    "List the top 5 customers by sales.",
]


async def show_sql(profile, prompt):
    sql = await profile.show_sql(prompt=prompt)
    return prompt, sql
```

The following code creates the connection pool, processes the prompts, and closes the pool in a `finally` block:

```
async def main():
    select_ai.create_pool_async(
        user=user,
        password=password,
        dsn=dsn,
        min_size=pool_min,
        max_size=pool_max,
        increment=pool_increment,
    )

    try:
        profile = await select_ai.AsyncProfile(profile_name=profile_name)
        tasks = [show_sql(profile, prompt) for prompt in prompts]

        for task in asyncio.as_completed(tasks):
            prompt, sql = await task
            print(f"\nPrompt: {prompt}")
            print(sql)
    finally:
        await select_ai.async_disconnect()


asyncio.run(main())
```

19.1.8 async_pipeline.py

This recipe uses `AsyncProfile.run_pipeline()` to submit multiple prompt/action pairs as a batch. Pipeline operations execute sequentially in the database; they do not run in parallel. The application receives the batch results when the pipeline completes. Pass `attributes=...` to apply the same request-level profile overrides to every item in the pipeline.

True pipelining requires a supporting database version. On older databases, the driver executes operations individually, without reducing round trips. Reading returned CLOBs also requires additional database calls. See *Pipeline compatibility* for requirements and result handling.

Set up the imports, connection values, prompts, and any helper functions:

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

pool_min = int(os.getenv("SELECT_AI_POOL_MIN", "1"))
pool_max = int(os.getenv("SELECT_AI_POOL_MAX", "4"))
pool_increment = int(os.getenv("SELECT_AI_POOL_INCREMENT", "1"))

profile_name = os.getenv("SELECT_AI_PROFILE_NAME", "async_oci_ai_profile")

prompt_specifications = [
    ("How many customers?", select_ai.Action.SHOWSQL),
    ("How many promotions?", select_ai.Action.RUNSQL),
    ("Explain how to count products.", select_ai.Action.EXPLAINSQL),
]
```

The following code creates the connection pool, processes the prompts, and closes the pool in a `finally` block:

```
async def main():
    select_ai.create_pool_async(
        user=user,
        password=password,
        dsn=dsn,
        min_size=pool_min,
        max_size=pool_max,
        increment=pool_increment,
    )

    try:
        profile = await select_ai.AsyncProfile(profile_name=profile_name)
        results = await profile.run_pipeline(
            prompt_specifications, continue_on_error=True
        )

        for (prompt, action), result in zip(prompt_specifications, results):
            print(f"\nPrompt: {prompt}")
            print(f"Action: {action}")
            print(result)
    finally:
        await select_ai.async_disconnect()


asyncio.run(main())
```

19.1.9 async_queue_workers.py

This recipe uses `asyncio.Queue` and async worker tasks. It is useful for long-running async applications that receive prompts over time.

Set up the imports, connection values, prompts, and any helper functions:

```
import asyncio
import os

import select_ai

user = os.getenv("SELECT_AI_USER")
password = os.getenv("SELECT_AI_PASSWORD")
dsn = os.getenv("SELECT_AI_DB_CONNECT_STRING")

pool_min = int(os.getenv("SELECT_AI_POOL_MIN", "1"))
pool_max = int(os.getenv("SELECT_AI_POOL_MAX", "4"))
pool_increment = int(os.getenv("SELECT_AI_POOL_INCREMENT", "1"))

profile_name = os.getenv("SELECT_AI_PROFILE_NAME", "async_oci_ai_profile")
worker_count = int(os.getenv("SELECT_AI_WORKER_COUNT", str(pool_max)))

prompts = [
    "How many customers?",
    "How many products?",
    "How many promotions?",
    "List the top 5 customers by sales.",
]


async def worker(name, profile, queue, results):
    while True:
        item = await queue.get()
        try:
            if item is None:
                return

            index, prompt = item
            sql = await profile.show_sql(prompt=prompt)
            results[index] = (prompt, sql)
        finally:
            queue.task_done()
```

The following code creates the connection pool, processes the prompts, and closes the pool in a `finally` block:

```
async def main():
    select_ai.create_pool_async(
        user=user,
        password=password,
        dsn=dsn,
        min_size=pool_min,
        max_size=pool_max,
        increment=pool_increment,
    )

    try:
        profile = await select_ai.AsyncProfile(profile_name=profile_name)
        queue = asyncio.Queue()
        results = [None] * len(prompts)

        workers = [
            asyncio.create_task(worker(i, profile, queue, results))
            for i in range(worker_count)
        ]

        for index, prompt in enumerate(prompts):
            await queue.put((index, prompt))

        for _ in workers:
            await queue.put(None)

        await queue.join()
        await asyncio.gather(*workers)

        for prompt, sql in results:
            print(f"\nPrompt: {prompt}")
            print(sql)
    finally:
        await select_ai.async_disconnect()


asyncio.run(main())
```

19.1.10 Pool sizing

Pool size controls how many database connections the application can use at one time. For thread and worker recipes, keep the worker count close to the pool maximum unless the application intentionally needs additional queued work.

In multi-process deployments, each process creates its own pool. Total possible database connections are approximately:

```
processes * SELECT_AI_POOL_MAX
```

Choose pool sizes that leave capacity for other database clients and for the AI provider calls made by `DBMS_CLOUD_AI`.

CHAPTER

TWENTY

API REFERENCE

20.1 Errors, sessions, and result objects

20.1.1 Error handling

SDK validation and lookup errors are separate from Oracle driver errors. Catch a specific SDK exception when the application can recover from it. Database and provider failures surfaced by PL/SQL can raise `oracledb.DatabaseError`; inspect the driver error code when deciding what the application should do. Avoid retrying authorization or configuration failures without correcting their cause.

```
import oracledb
import select_ai

# Assumes an open connection. Missing profiles are not created here.
try:
    profile = select_ai.Profile.fetch("APP_PROFILE")
    answer = profile.chat("Explain retrieval augmented generation.")
except select_ai.ProfileNotFoundError:
    print("Create APP_PROFILE or choose an existing profile.")
except select_ai.DatabaseNotConnectedError:
    print("Establish a connection before using the profile.")
except oracledb.DatabaseError as exc:
    error, = exc.args
    print(f"Database operation failed (code {error.code}).")
    raise
```

Do not expose raw database errors or connection credentials in public HTTP responses. For web applications, translate expected failures into your application's error response and close pools in the application shutdown hook.

exception select_ai.errors.**AgentAttributesEmptyError**(*agent_name: str*)

Agent attributes not found in the database

add_note(*note*, /)

Add a note to the exception

args

with_traceback(*tb*, /)

Set self.__traceback__ to tb and return self.

exception select_ai.errors.**AgentNotFoundError**(*agent_name: str*)

Agent not found in the database

add_note(*note*, /)

Add a note to the exception

args

with_traceback(*tb*, /)

Set self.__traceback__ to tb and return self.

exception select_ai.errors.**AgentTaskAttributesEmptyError**(*task_name: str*)

Agent task attributes not found in the database

add_note(*note*, /)

Add a note to the exception

args

with_traceback(*tb*, /)

Set self.__traceback__ to tb and return self.

exception select_ai.errors.**AgentTaskNotFoundError**(*task_name: str*)

Agent task not found in the database

add_note(*note*, /)

Add a note to the exception

args

with_traceback(*tb*, /)

Set self.__traceback__ to tb and return self.

exception select_ai.errors.**AgentTeamAttributesEmptyError**(*team_name: str*)

Agent team attributes empty in the database

add_note(*note*, /)

Add a note to the exception

args

with_traceback(*tb*, /)

Set self.__traceback__ to tb and return self.

exception select_ai.errors.**AgentTeamNotFoundError**(*team_name: str*)

Agent team not found in the database

add_note(*note*, /)

Add a note to the exception

args

with_traceback(*tb*, /)

Set self.__traceback__ to tb and return self.

exception select_ai.errors.**AgentToolAttributesEmptyError**(*tool_name: str*)

Agent team attributes empty in the database

add_note(*note*, /)

Add a note to the exception

args

with_traceback(*tb*, /)

Set self.__traceback__ to tb and return self.

exception select_ai.errors.**AgentToolNotFoundError**(*tool_name: str*)

Agent tool not found in the database

add_note(*note*, /)

Add a note to the exception

args

with_traceback(*tb*, /)

Set self.__traceback__ to tb and return self.

exception select_ai.errors.**ConversationNotFoundError**(*conversation_id: str*)

Conversation not found in the database

add_note(*note*, /)

Add a note to the exception

args

with_traceback(*tb*, /)

Set self.__traceback__ to tb and return self.

exception select_ai.errors.**DatabaseNotConnectedError**

Raised when a database is not connected

add_note(*note*, /)

Add a note to the exception

args

with_traceback(*tb*, /)

Set self.__traceback__ to tb and return self.

exception select_ai.errors.**InvalidSQLError**(*error_message: str*)

Invalid SQL generated

add_note(*note*, /)

Add a note to the exception

args

with_traceback(*tb*, /)

Set self.__traceback__ to tb and return self.

exception select_ai.errors.**ProfileAttributesEmptyError**(*profile_name: str*)

Profile attributes empty in the database

add_note(*note*, /)

Add a note to the exception

args

with_traceback(*tb*, /)

Set self.__traceback__ to tb and return self.

exception select_ai.errors.**ProfileExistsError**(*profile_name: str*)

Profile already exists in the database

add_note(*note*, /)

Add a note to the exception

args

with_traceback(*tb*, /)

Set self.__traceback__ to tb and return self.

exception select_ai.errors.**ProfileNotFoundError**(*profile_name: str*)

Profile not found in the database

add_note(*note*, /)

Add a note to the exception

args

with_traceback(*tb*, /)

Set self.__traceback__ to tb and return self.

exception select_ai.errors.**SelectAIError**

Base class for any SelectAIErrors

add_note(*note*, /)

Add a note to the exception

args

with_traceback(*tb*, /)

Set self.__traceback__ to tb and return self.

exception select_ai.errors.**VectorIndexNotFoundError**(*index_name: str*, *profile_name: str = None*)

VectorIndex not found in the database

add_note(*note*, /)

Add a note to the exception

args

with_traceback(*tb*, /)

Set self.__traceback__ to tb and return self.

20.1.2 Session objects

Obtain sessions through `Profile.chat_session()` or `AsyncProfile.chat_session()`. A session holds a connection for the duration of its context. Consume and close streaming iterators inside that context.

class select_ai.profile.**Session**(*profile: Profile*, *params: Mapping*)

Session lets you persist request parameters across DBMS_CLOUD_AI requests. This is useful in context-aware conversations

chat(*prompt: str*, *stream: bool = False*, *chunk_size: int = 8192*, ***, *attributes: Mapping = None*) → str | Generator[str, None, None]

explain_sql(*prompt: str*, *stream: bool = False*, *chunk_size: int = 8192*, ***, *attributes: Mapping = None*) → str | Generator[str, None, None]

Explain the generated SQL

Parameters

- **prompt** (*str*) – Natural language prompt
- **stream** (*bool*) – Return an iterator of response chunks
- **chunk_size** (*int*) – Number of characters to read per stream chunk

Returns

str

narrate(*prompt: str*, *stream: bool = False*, *chunk_size: int = 8192*, ***, *attributes: Mapping = None*) → str | Generator[str, None, None]

Narrate the result of the SQL

Parameters

- **prompt** (*str*) – Natural language prompt
- **stream** (*bool*) – Return an iterator of response chunks
- **chunk_size** (*int*) – Number of characters to read per stream chunk

Returns

str

run_sql(*prompt: str*, ***, *attributes: Mapping = None*) → DataFrame

Run the generate SQL statement and return a pandas Dataframe built using the result set

Parameters

prompt (*str*) – Natural language prompt

Returns

pandas.DataFrame

show_prompt(*prompt: str*, *stream: bool = False*, *chunk_size: int = 8192*, ***, *attributes: Mapping = None*) → str | Generator[str, None, None]

Show the prompt sent to LLM

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM request
- **stream** (*bool*) – Return an iterator of response chunks
- **chunk_size** (*int*) – Number of characters to read per stream chunk

Returns

str

show_sql(*prompt: str*, *stream: bool = False*, *chunk_size: int = 8192*, ***, *attributes: Mapping = None*) → str | Generator[str, None, None]

Show the generated SQL

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM request

- **stream** (*bool*) – Return an iterator of response chunks
- **chunk_size** (*int*) – Number of characters to read per stream chunk

Returns
str

class select_ai.async_profile.**AsyncSession**(*async_profile: AsyncProfile*, *params: Mapping*)

AsyncSession lets you persist request parameters across DBMS_CLOUD_AI requests. This is useful in context-aware conversations

async **chat**(*prompt: str*, *stream: bool = False*, *chunk_size: int = 8192*, ***, *attributes: Mapping = None*) → str | AsyncGenerator[str, None]

async **explain_sql**(*prompt: str*, *stream: bool = False*, *chunk_size: int = 8192*, ***, *attributes: Mapping = None*) → str | AsyncGenerator[str, None]

Explain the generated SQL

Parameters

- **prompt** (*str*) – Natural language prompt
- **stream** (*bool*) – Return an async iterator of response chunks
- **chunk_size** (*int*) – Number of characters to read per stream chunk

Returns
str

async **narrate**(*prompt*, *stream: bool = False*, *chunk_size: int = 8192*, ***, *attributes: Mapping = None*) → str | AsyncGenerator[str, None]

Narrate the result of the SQL

Parameters

- **prompt** (*str*) – Natural language prompt
- **stream** (*bool*) – Return an async iterator of response chunks
- **chunk_size** (*int*) – Number of characters to read per stream chunk

Returns
str

async **run_sql**(*prompt: str*, ***, *attributes: Mapping = None*) → DataFrame

Explain the generated SQL

Parameters
prompt (*str*) – Natural language prompt

Returns
pandas.DataFrame

async **show_prompt**(*prompt: str*, *stream: bool = False*, *chunk_size: int = 8192*, ***, *attributes: Mapping = None*) → str | AsyncGenerator[str, None]

Show the prompt sent to LLM

Parameters

- **prompt** (*str*) – Natural language prompt
- **stream** (*bool*) – Return an async iterator of response chunks
- **chunk_size** (*int*) – Number of characters to read per stream chunk

Returns
str

async show_sql(*prompt*, *stream: bool = False*, *chunk_size: int = 8192*, ***, *attributes: Mapping = None*) → str | AsyncGenerator[str, None]

Show the generated SQL

Parameters

- **prompt** (*str*) – Natural language prompt
- **stream** (*bool*) – Return an async iterator of response chunks
- **chunk_size** (*int*) – Number of characters to read per stream chunk

Returns
str

20.1.3 Agent history results

History APIs yield these typed event objects. Availability and values depend on the current user's database history views.

class select_ai.agent.history.**TeamHistoryEvent**(*team_exec_id: str*, *team_name: str*, *state: str*, *start_date: datetime | None = None*, *end_date: datetime | None = None*, *conversation_id: str | None = None*, *params: str | None = None*)

One row from C##CLOUD$SERVICE.USER_AI_AGENT_TEAM_HISTORY.

conversation_id: str | None = None

dict(*exclude_null=True*)

end_date: datetime | None = None

json(*exclude_null=True*)

classmethod keys()

params: str | None = None

start_date: datetime | None = None

state: str

team_exec_id: str

team_name: str

class select_ai.agent.history.**TaskHistoryEvent**(*team_exec_id: str*, *team_name: str*, *task_order: int | None*, *agent_name: str*, *task_name: str | None*, *conversation_params: str | None*, *input: str | None*, *result: str | None*, *state: str*, *start_date: datetime | None*, *end_date: datetime | None*)

One row from C##CLOUD$SERVICE.USER_AI_AGENT_TASK_HISTORY.

agent_name: str

conversation_params: str | None

dict(*exclude_null=True*)

end_date: datetime | None

input: str | None

json(*exclude_null=True*)

classmethod keys()

result: str | None

start_date: datetime | None

state: str

task_name: str | None

task_order: int | None

team_exec_id: str

team_name: str

class select_ai.agent.history.**ToolHistoryEvent**(*invocation_id: int*, *team_exec_id: str*, *task_order: int | None*, *tool_name: str | None*, *agent_name: str | None*, *task_name: str | None*, *start_date: datetime | None*, *end_date: datetime | None*, *input: Any | None*, *output: Any | None*, *tool_output: str | None*)

One row from C##CLOUD$SERVICE.USER_AI_AGENT_TOOL_HISTORY.

agent_name: str | None

dict(*exclude_null=True*)

end_date: datetime | None

input: Any | None

invocation_id: int

json(*exclude_null=True*)

classmethod keys()

output: Any | None

start_date: datetime | None

task_name: str | None

task_order: int | None

team_exec_id: str

tool_name: str | None

tool_output: str | None

20.1.4 Enumerations

class select_ai.action.**Action**(**values*)

Supported Select AI actions

CHAT = 'chat'

EXPLAINSQL = 'explainsql'

FEEDBACK = 'feedback'

NARRATE = 'narrate'

RUNSQL = 'runsql'

SHOWPROMPT = 'showprompt'

SHOWSQL = 'showsql'

SUMMARIZE = 'summarize'

TRANSLATE = 'translate'

capitalize()

Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold()

Return a version of the string suitable for caseless comparisons.

center(*width*, *fillchar=' '*, */*)

Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

count()

Return the number of non-overlapping occurrences of substring sub in string S[start:end].

Optional arguments start and end are interpreted as in slice notation.

encode(*encoding='utf-8'*, *errors='strict'*)

Encode the string using the codec registered for encoding.

encoding
The encoding in which to encode the string.

errors
The error handling scheme to use for encoding errors. The default is 'strict' meaning that encoding errors raise a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith()

Return True if the string ends with the specified suffix, False otherwise.

suffix
A string or a tuple of strings to try.

start
Optional start position. Default: start of the string.

end
Optional stop position. Default: end of the string.

expandtabs(*tabsize=8*)

Return a copy where all tab characters are expanded using spaces.

If tabsize is not given, a tab size of 8 characters is assumed.

find()

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.

format(**args*, ***kwargs*)

Return a formatted version of the string, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map(*mapping*, /)

Return a formatted version of the string, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

index()

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Raises ValueError when the substring is not found.

isalnum()

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha()

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii()

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal()

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit()

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier()

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as "def" or "class".

islower()

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

isnumeric()

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

isprintable()

Return True if all characters in the string are printable, False otherwise.

A character is printable if repr() may use it in its output.

isspace()

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

istitle()

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

isupper()

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

join(*iterable*, /)

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'

ljust(*width*, *fillchar=' '*, /)

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

lower()

Return a copy of the string converted to lowercase.

lstrip(*chars=None*, /)

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

static maketrans()

Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be

mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

partition(*sep*, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original string and two empty strings.

removeprefix(*prefix*, /)

Return a str with the given prefix string removed if present.

If the string starts with the prefix string, return string[len(prefix):]. Otherwise, return a copy of the original string.

removesuffix(*suffix*, /)

Return a str with the given suffix string removed if present.

If the string ends with the suffix string and that suffix is not empty, return string[:-len(suffix)]. Otherwise, return a copy of the original string.

replace(*old*, *new*, /, *count=-1*)

Return a copy with all occurrences of substring old replaced by new.

count
Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

rfind()

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.

rindex()

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Raises ValueError when the substring is not found.

rjust(*width*, *fillchar=' '*, /)

Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

rpartition(*sep*, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty strings and the original string.

rsplit(*sep=None*, *maxsplit=-1*)

Return a list of the substrings in the string, using sep as the separator string.

sep
The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including n r t f and spaces) and will discard empty strings from the result.

maxsplit
Maximum number of splits. -1 (the default value) means no limit.

Splitting starts at the end of the string and works to the front.

`rstrip`(*chars=None*, */*)

Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

`split`(*sep=None*, *maxsplit=-1*)

Return a list of the substrings in the string, using sep as the separator string.

sep
The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including n r t f and spaces) and will discard empty strings from the result.

maxsplit
Maximum number of splits. -1 (the default value) means no limit.

Splitting starts at the front of the string and works to the end.

Note, str.split() is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

`splitlines`(*keepends=False*)

Return a list of the lines in the string, breaking at line boundaries.

Line breaks are not included in the resulting list unless keepends is given and true.

`startswith`()

Return True if the string starts with the specified prefix, False otherwise.

prefix
A string or a tuple of strings to try.

start
Optional start position. Default: start of the string.

end
Optional stop position. Default: end of the string.

`strip`(*chars=None*, */*)

Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

`swapcase`()

Convert uppercase characters to lowercase and lowercase characters to uppercase.

`title`()

Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate(*table*, /)

Replace each character in the string using the given translation table.

table
Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via __getitem__, for instance a dictionary or list. If this operation raises LookupError, the character is left untouched. Characters mapped to None are deleted.

upper()

Return a copy of the string converted to uppercase.

zfill(*width*, /)

Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

class select_ai.agent.tool.**ToolType**(**values*)

Built-in Tool Types

HTTP = 'HTTP'

HUMAN = 'HUMAN'

NOTIFICATION = 'NOTIFICATION'

RAG = 'RAG'

SQL = 'SQL'

WEBSEARCH = 'WEBSEARCH'

capitalize()

Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold()

Return a version of the string suitable for caseless comparisons.

center(*width*, *fillchar=' '*, /)

Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

count()

Return the number of non-overlapping occurrences of substring sub in string S[start:end].

Optional arguments start and end are interpreted as in slice notation.

encode(*encoding='utf-8'*, *errors='strict'*)

Encode the string using the codec registered for encoding.

encoding
The encoding in which to encode the string.

errors
The error handling scheme to use for encoding errors. The default is 'strict' meaning that encoding errors raise a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith()

Return True if the string ends with the specified suffix, False otherwise.

suffix
A string or a tuple of strings to try.

start
Optional start position. Default: start of the string.

end
Optional stop position. Default: end of the string.

expandtabs(*tabsize=8*)

Return a copy where all tab characters are expanded using spaces.

If tabsize is not given, a tab size of 8 characters is assumed.

find()

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.

format(**args*, ***kwargs*)

Return a formatted version of the string, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map(*mapping*, */*)

Return a formatted version of the string, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

index()

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Raises ValueError when the substring is not found.

isalnum()

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha()

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii()

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal()

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit()

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier()

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as "def" or "class".

islower()

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

isnumeric()

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

isprintable()

Return True if all characters in the string are printable, False otherwise.

A character is printable if repr() may use it in its output.

isspace()

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

istitle()

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

isupper()

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

join(*iterable*, /)

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'

ljust(*width*, *fillchar=' '*, /)

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

lower()

Return a copy of the string converted to lowercase.

lstrip(*chars=None*, */*)

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

static maketrans()

Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

partition(*sep*, */*)

Partition the string into three parts using the given separator.

This will search for the separator in the string. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original string and two empty strings.

removeprefix(*prefix*, */*)

Return a str with the given prefix string removed if present.

If the string starts with the prefix string, return string[len(prefix):]. Otherwise, return a copy of the original string.

removesuffix(*suffix*, */*)

Return a str with the given suffix string removed if present.

If the string ends with the suffix string and that suffix is not empty, return string[:-len(suffix)]. Otherwise, return a copy of the original string.

replace(*old*, *new*, */*, *count=-1*)

Return a copy with all occurrences of substring old replaced by new.

count
Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

rfind()

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.

rindex()

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Raises ValueError when the substring is not found.

rjust(*width*, *fillchar=' '*, */*)

Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

`rpartition`(*sep*, */*)

Partition the string into three parts using the given separator.

This will search for the separator in the string, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty strings and the original string.

`rsplit`(*sep=None*, *maxsplit=-1*)

Return a list of the substrings in the string, using sep as the separator string.

sep
The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including n r t f and spaces) and will discard empty strings from the result.

maxsplit
Maximum number of splits. -1 (the default value) means no limit.

Splitting starts at the end of the string and works to the front.

`rstrip`(*chars=None*, */*)

Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

`split`(*sep=None*, *maxsplit=-1*)

Return a list of the substrings in the string, using sep as the separator string.

sep
The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including n r t f and spaces) and will discard empty strings from the result.

maxsplit
Maximum number of splits. -1 (the default value) means no limit.

Splitting starts at the front of the string and works to the end.

Note, str.split() is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

`splitlines`(*keepends=False*)

Return a list of the lines in the string, breaking at line boundaries.

Line breaks are not included in the resulting list unless keepends is given and true.

`startswith`()

Return True if the string starts with the specified prefix, False otherwise.

prefix
A string or a tuple of strings to try.

start
Optional start position. Default: start of the string.

end
Optional stop position. Default: end of the string.

strip(*chars=None*, /)

Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

swapcase()

Convert uppercase characters to lowercase and lowercase characters to uppercase.

title()

Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate(*table*, /)

Replace each character in the string using the given translation table.

table
Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via __getitem__, for instance a dictionary or list. If this operation raises LookupError, the character is left untouched. Characters mapped to None are deleted.

upper()

Return a copy of the string converted to uppercase.

zfill(*width*, /)

Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

class select_ai.agent.tool.**NotificationType**(**values*)

Notification Types

EMAIL = 'email'

SLACK = 'slack'

capitalize()

Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold()

Return a version of the string suitable for caseless comparisons.

center(*width*, *fillchar=' '*, /)

Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

count()

Return the number of non-overlapping occurrences of substring sub in string S[start:end].

Optional arguments start and end are interpreted as in slice notation.

encode(*encoding='utf-8'*, *errors='strict'*)

Encode the string using the codec registered for encoding.

encoding
The encoding in which to encode the string.

errors
The error handling scheme to use for encoding errors. The default is 'strict' meaning that encoding errors raise a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith()

Return True if the string ends with the specified suffix, False otherwise.

suffix
A string or a tuple of strings to try.

start
Optional start position. Default: start of the string.

end
Optional stop position. Default: end of the string.

expandtabs(*tabsize=8*)

Return a copy where all tab characters are expanded using spaces.

If tabsize is not given, a tab size of 8 characters is assumed.

find()

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.

format(**args*, ***kwargs*)

Return a formatted version of the string, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map(*mapping*, */*)

Return a formatted version of the string, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

index()

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Raises ValueError when the substring is not found.

isalnum()

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha()

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii()

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal()

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit()

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier()

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as "def" or "class".

islower()

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

isnumeric()

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

isprintable()

Return True if all characters in the string are printable, False otherwise.

A character is printable if repr() may use it in its output.

isspace()

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

istitle()

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

isupper()

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

join(*iterable*, /)

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'

ljust(*width*, *fillchar=' '*, */*)

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

lower()

Return a copy of the string converted to lowercase.

lstrip(*chars=None*, */*)

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

static maketrans()

Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

partition(*sep*, */*)

Partition the string into three parts using the given separator.

This will search for the separator in the string. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original string and two empty strings.

removeprefix(*prefix*, */*)

Return a str with the given prefix string removed if present.

If the string starts with the prefix string, return string[len(prefix):]. Otherwise, return a copy of the original string.

removesuffix(*suffix*, */*)

Return a str with the given suffix string removed if present.

If the string ends with the suffix string and that suffix is not empty, return string[:-len(suffix)]. Otherwise, return a copy of the original string.

replace(*old*, *new*, */*, *count=-1*)

Return a copy with all occurrences of substring old replaced by new.

count
Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

rfind()

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.

rindex()

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Raises ValueError when the substring is not found.

rjust(*width*, *fillchar=' '*, */*)

Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

rpartition(*sep*, */*)

Partition the string into three parts using the given separator.

This will search for the separator in the string, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty strings and the original string.

rsplit(*sep=None*, *maxsplit=-1*)

Return a list of the substrings in the string, using sep as the separator string.

sep
The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including n r t f and spaces) and will discard empty strings from the result.

maxsplit
Maximum number of splits. -1 (the default value) means no limit.

Splitting starts at the end of the string and works to the front.

rstrip(*chars=None*, */*)

Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

split(*sep=None*, *maxsplit=-1*)

Return a list of the substrings in the string, using sep as the separator string.

sep
The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including n r t f and spaces) and will discard empty strings from the result.

maxsplit
Maximum number of splits. -1 (the default value) means no limit.

Splitting starts at the front of the string and works to the end.

Note, str.split() is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

splitlines(*keepends=False*)

Return a list of the lines in the string, breaking at line boundaries.

Line breaks are not included in the resulting list unless keepends is given and true.

startswith()

Return True if the string starts with the specified prefix, False otherwise.

prefix
A string or a tuple of strings to try.

start
Optional start position. Default: start of the string.

end
Optional stop position. Default: end of the string.

`strip`(*chars=None*, /)

Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

`swapcase`()

Convert uppercase characters to lowercase and lowercase characters to uppercase.

`title`()

Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

`translate`(*table*, /)

Replace each character in the string using the given translation table.

table
Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via __getitem__, for instance a dictionary or list. If this operation raises LookupError, the character is left untouched. Characters mapped to None are deleted.

`upper`()

Return a copy of the string converted to uppercase.

`zfill`(*width*, /)

Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

`class` select_ai.vector_index.`VectorDistanceMetric`(**values*)

`COSINE = 'COSINE'`

`DOT = 'DOT'`

`EUCLIDEAN = 'EUCLIDEAN'`

`HAMMING = 'HAMMING'`

`L2_SQUARED = 'L2_SQUARED'`

`MANHATTAN = 'MANHATTAN'`

`capitalize`()

Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

`casefold`()

Return a version of the string suitable for caseless comparisons.

`center`(*width*, *fillchar=' '*, /)

Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

count()

Return the number of non-overlapping occurrences of substring sub in string S[start:end].

Optional arguments start and end are interpreted as in slice notation.

encode(*encoding='utf-8'*, *errors='strict'*)

Encode the string using the codec registered for encoding.

encoding
The encoding in which to encode the string.

errors
The error handling scheme to use for encoding errors. The default is 'strict' meaning that encoding errors raise a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith()

Return True if the string ends with the specified suffix, False otherwise.

suffix
A string or a tuple of strings to try.

start
Optional start position. Default: start of the string.

end
Optional stop position. Default: end of the string.

expandtabs(*tabsize=8*)

Return a copy where all tab characters are expanded using spaces.

If tabsize is not given, a tab size of 8 characters is assumed.

find()

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.

format(**args*, ***kwargs*)

Return a formatted version of the string, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map(*mapping*, */*)

Return a formatted version of the string, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

index()

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Raises ValueError when the substring is not found.

isalnum()

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

`isalpha()`

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

`isascii()`

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

`isdecimal()`

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

`isdigit()`

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

`isidentifier()`

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as "def" or "class".

`islower()`

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

`isnumeric()`

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

`isprintable()`

Return True if all characters in the string are printable, False otherwise.

A character is printable if repr() may use it in its output.

`isspace()`

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

`istitle()`

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

`isupper()`

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

join(*iterable*, /)

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'

ljust(*width*, *fillchar=' '*, /)

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

lower()

Return a copy of the string converted to lowercase.

lstrip(*chars=None*, /)

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

static maketrans()

Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

partition(*sep*, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original string and two empty strings.

removeprefix(*prefix*, /)

Return a str with the given prefix string removed if present.

If the string starts with the prefix string, return string[len(prefix):]. Otherwise, return a copy of the original string.

removesuffix(*suffix*, /)

Return a str with the given suffix string removed if present.

If the string ends with the suffix string and that suffix is not empty, return string[:-len(suffix)]. Otherwise, return a copy of the original string.

replace(*old*, *new*, /, *count=-1*)

Return a copy with all occurrences of substring old replaced by new.

count
Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

`rfind`()

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.

`rindex`()

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Raises ValueError when the substring is not found.

`rjust`(*width*, *fillchar=' '*, */*)

Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

`rpartition`(*sep*, */*)

Partition the string into three parts using the given separator.

This will search for the separator in the string, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty strings and the original string.

`rsplit`(*sep=None*, *maxsplit=-1*)

Return a list of the substrings in the string, using sep as the separator string.

sep
The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including n r t f and spaces) and will discard empty strings from the result.

maxsplit
Maximum number of splits. -1 (the default value) means no limit.

Splitting starts at the end of the string and works to the front.

`rstrip`(*chars=None*, */*)

Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

`split`(*sep=None*, *maxsplit=-1*)

Return a list of the substrings in the string, using sep as the separator string.

sep
The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including n r t f and spaces) and will discard empty strings from the result.

maxsplit
Maximum number of splits. -1 (the default value) means no limit.

Splitting starts at the front of the string and works to the end.

Note, str.split() is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

`splitlines`(*keepends=False*)

Return a list of the lines in the string, breaking at line boundaries.

Line breaks are not included in the resulting list unless keepends is given and true.

startswith()

Return True if the string starts with the specified prefix, False otherwise.

prefix
A string or a tuple of strings to try.

start
Optional start position. Default: start of the string.

end
Optional stop position. Default: end of the string.

strip(*chars=None*, /)

Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

swapcase()

Convert uppercase characters to lowercase and lowercase characters to uppercase.

title()

Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate(*table*, /)

Replace each character in the string using the given translation table.

table
Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via __getitem__, for instance a dictionary or list. If this operation raises LookupError, the character is left untouched. Characters mapped to None are deleted.

upper()

Return a copy of the string converted to uppercase.

zfill(*width*, /)

Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

class select_ai.vector_index.**VectorDBProvider**(**values*)

ORACLE = 'oracle'

capitalize()

Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold()

Return a version of the string suitable for caseless comparisons.

center(*width*, *fillchar=' '*, /)

Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

count()

Return the number of non-overlapping occurrences of substring sub in string S[start:end].

Optional arguments start and end are interpreted as in slice notation.

encode(*encoding='utf-8'*, *errors='strict'*)

Encode the string using the codec registered for encoding.

encoding
The encoding in which to encode the string.

errors
The error handling scheme to use for encoding errors. The default is 'strict' meaning that encoding errors raise a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith()

Return True if the string ends with the specified suffix, False otherwise.

suffix
A string or a tuple of strings to try.

start
Optional start position. Default: start of the string.

end
Optional stop position. Default: end of the string.

expandtabs(*tabsize=8*)

Return a copy where all tab characters are expanded using spaces.

If tabsize is not given, a tab size of 8 characters is assumed.

find()

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.

format(**args*, ***kwargs*)

Return a formatted version of the string, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map(*mapping*, */*)

Return a formatted version of the string, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

index()

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Raises ValueError when the substring is not found.

isalnum()

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha()

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii()

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal()

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit()

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier()

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as "def" or "class".

islower()

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

isnumeric()

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

isprintable()

Return True if all characters in the string are printable, False otherwise.

A character is printable if repr() may use it in its output.

isspace()

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

istitle()

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

isupper()

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

join(*iterable*, /)

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'

ljust(*width*, *fillchar=' '*, /)

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

lower()

Return a copy of the string converted to lowercase.

lstrip(*chars=None*, /)

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

static maketrans()

Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

partition(*sep*, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original string and two empty strings.

removeprefix(*prefix*, /)

Return a str with the given prefix string removed if present.

If the string starts with the prefix string, return string[len(prefix):]. Otherwise, return a copy of the original string.

removesuffix(*suffix*, /)

Return a str with the given suffix string removed if present.

If the string ends with the suffix string and that suffix is not empty, return string[:-len(suffix)]. Otherwise, return a copy of the original string.

replace(*old*, *new*, /, *count=-1*)

Return a copy with all occurrences of substring old replaced by new.

count
Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

rfind()

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.

rindex()

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Raises ValueError when the substring is not found.

rjust(*width*, *fillchar=' '*, */*)

Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

rpartition(*sep*, */*)

Partition the string into three parts using the given separator.

This will search for the separator in the string, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty strings and the original string.

rsplit(*sep=None*, *maxsplit=-1*)

Return a list of the substrings in the string, using sep as the separator string.

sep
The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including n r t f and spaces) and will discard empty strings from the result.

maxsplit
Maximum number of splits. -1 (the default value) means no limit.

Splitting starts at the end of the string and works to the front.

rstrip(*chars=None*, */*)

Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

split(*sep=None*, *maxsplit=-1*)

Return a list of the substrings in the string, using sep as the separator string.

sep
The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including n r t f and spaces) and will discard empty strings from the result.

maxsplit
Maximum number of splits. -1 (the default value) means no limit.

Splitting starts at the front of the string and works to the end.

Note, str.split() is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

splitlines(*keepends=False*)

Return a list of the lines in the string, breaking at line boundaries.

Line breaks are not included in the resulting list unless keepends is given and true.

`startswith`()

Return True if the string starts with the specified prefix, False otherwise.

prefix
A string or a tuple of strings to try.

start
Optional start position. Default: start of the string.

end
Optional stop position. Default: end of the string.

`strip`(*chars=None*, /)

Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

`swapcase`()

Convert uppercase characters to lowercase and lowercase characters to uppercase.

`title`()

Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

`translate`(*table*, /)

Replace each character in the string using the given translation table.

table
Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via __getitem__, for instance a dictionary or list. If this operation raises LookupError, the character is left untouched. Characters mapped to None are deleted.

`upper`()

Return a copy of the string converted to uppercase.

`zfill`(*width*, /)

Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

PYTHON MODULE INDEX

S

select_ai.errors, 349

INDEX

A

Action (*class in select_ai.action*), 357
add_negative_feedback() (*select_ai.AsyncProfile method*), 104
add_negative_feedback() (*select_ai.Profile method*), 76
add_note() (*select_ai.errors.AgentAttributesEmptyError method*), 349
add_note() (*select_ai.errors.AgentNotFoundError method*), 349
add_note() (*select_ai.errors.AgentTaskAttributesEmptyError method*), 350
add_note() (*select_ai.errors.AgentTaskNotFoundError method*), 350
add_note() (*select_ai.errors.AgentTeamAttributesEmptyError method*), 350
add_note() (*select_ai.errors.AgentTeamNotFoundError method*), 350
add_note() (*select_ai.errors.AgentToolAttributesEmptyError method*), 350
add_note() (*select_ai.errors.AgentToolNotFoundError method*), 351
add_note() (*select_ai.errors.ConversationNotFoundError method*), 351
add_note() (*select_ai.errors.DatabaseNotConnectedError method*), 351
add_note() (*select_ai.errors.InvalidSQLError method*), 351
add_note() (*select_ai.errors.ProfileAttributesEmptyError method*), 351
add_note() (*select_ai.errors.ProfileExistsError method*), 352
add_note() (*select_ai.errors.ProfileNotFoundError method*), 352
add_note() (*select_ai.errors.SelectAIError method*), 352
add_note() (*select_ai.errors.VectorIndexNotFoundError method*), 352
add_positive_feedback() (*select_ai.AsyncProfile method*), 104
add_positive_feedback() (*select_ai.Profile method*), 76
add_tag() (*select_ai.AsyncConversation method*), 147
add_tag() (*select_ai.Conversation method*), 138
additional_instructions (*select_ai.ProfileAttributes attribute*), 70
Agent (*class in select_ai.agent*), 232
agent_name (*select_ai.agent.history.TaskHistoryEvent attribute*), 355
agent_name (*select_ai.agent.history.ToolHistoryEvent attribute*), 356
AgentAttributes (*class in select_ai.agent*), 231
AgentAttributesEmptyError, 349
AgentNotFoundError, 349
agents (*select_ai.agent.TeamAttributes attribute*), 235
AgentTaskAttributesEmptyError, 350
AgentTaskNotFoundError, 350
AgentTeamAttributesEmptyError, 350
AgentTeamNotFoundError, 350
AgentToolAttributesEmptyError, 350
AgentToolNotFoundError, 351
annotations (*select_ai.ProfileAttributes attribute*), 70
AnthropicProvider (*class in select_ai*), 33
args (*select_ai.errors.AgentAttributesEmptyError attribute*), 349
args (*select_ai.errors.AgentNotFoundError attribute*), 350
args (*select_ai.errors.AgentTaskAttributesEmptyError attribute*), 350
args (*select_ai.errors.AgentTaskNotFoundError attribute*), 350
args (*select_ai.errors.AgentTeamAttributesEmptyError attribute*), 350
args (*select_ai.errors.AgentTeamNotFoundError attribute*), 350
args (*select_ai.errors.AgentToolAttributesEmptyError attribute*), 350
args (*select_ai.errors.AgentToolNotFoundError attribute*), 351
args (*select_ai.errors.ConversationNotFoundError attribute*), 351
args (*select_ai.errors.DatabaseNotConnectedError attribute*), 351
args (*select_ai.errors.InvalidSQLError attribute*), 351

args (*select_ai.errors.ProfileAttributesEmptyError attribute*), 351
args (*select_ai.errors.ProfileExistsError attribute*), 352
args (*select_ai.errors.ProfileNotFoundError attribute*), 352
args (*select_ai.errors.SelectAIError attribute*), 352
args (*select_ai.errors.VectorIndexNotFoundError attribute*), 352
async_create() (*select_ai.ProfileAttributes class method*), 70
AsyncAgent (*class in select_ai.agent*), 269
AsyncConversation (*class in select_ai*), 147
AsyncProfile (*class in select_ai*), 104
AsyncSession (*class in select_ai.async_profile*), 354
AsyncTask (*class in select_ai.agent*), 265
AsyncTaskHistory (*class in select_ai.agent*), 288
AsyncTeam (*class in select_ai.agent*), 273
AsyncTeamHistory (*class in select_ai.agent*), 288
AsyncTool (*class in select_ai.agent*), 256
AsyncToolHistory (*class in select_ai.agent*), 288
AsyncVectorIndex (*class in select_ai*), 170
aws_apiformat (*select_ai.AWSProvider attribute*), 35
AWSProvider (*class in select_ai*), 35
azure_deployment_name (*select_ai.AzureProvider attribute*), 34
azure_embedding_deployment_name (*select_ai.AzureProvider attribute*), 34
azure_resource_name (*select_ai.AzureProvider attribute*), 34
AzureProvider (*class in select_ai*), 34

B

BaseProfile (*class in select_ai*), 75

C

capitalize() (*select_ai.action.Action method*), 357
capitalize() (*select_ai.agent.tool.NotificationType method*), 367
capitalize() (*select_ai.agent.tool.ToolType method*), 362
capitalize() (*select_ai.summary.ChunkProcessingMethod method*), 194
capitalize() (*select_ai.summary.ExtractivenessLevel method*), 200
capitalize() (*select_ai.summary.Style method*), 206
capitalize() (*select_ai.vector_index.VectorDBProvider method*), 377
capitalize() (*select_ai.vector_index.VectorDistanceMetric method*), 372
case_sensitive_values (*select_ai.ProfileAttributes attribute*), 70
casefold() (*select_ai.action.Action method*), 357
casefold() (*select_ai.agent.tool.NotificationType method*), 367
casefold() (*select_ai.agent.tool.ToolType method*), 362
casefold() (*select_ai.summary.ChunkProcessingMethod method*), 194
casefold() (*select_ai.summary.ExtractivenessLevel method*), 200
casefold() (*select_ai.summary.Style method*), 206
casefold() (*select_ai.vector_index.VectorDBProvider method*), 377
casefold() (*select_ai.vector_index.VectorDistanceMetric method*), 372
center() (*select_ai.action.Action method*), 357
center() (*select_ai.agent.tool.NotificationType method*), 367
center() (*select_ai.agent.tool.ToolType method*), 362
center() (*select_ai.summary.ChunkProcessingMethod method*), 194
center() (*select_ai.summary.ExtractivenessLevel method*), 200
center() (*select_ai.summary.Style method*), 206
center() (*select_ai.vector_index.VectorDBProvider method*), 377
center() (*select_ai.vector_index.VectorDistanceMetric method*), 372
channel (*select_ai.agent.ToolParams attribute*), 215
CHAT (*select_ai.action.Action attribute*), 357
chat() (*select_ai.async_profile.AsyncSession method*), 354
chat() (*select_ai.AsyncProfile method*), 104
chat() (*select_ai.Profile method*), 76
chat() (*select_ai.profile.Session method*), 352
chat_session() (*select_ai.AsyncProfile method*), 105
chat_session() (*select_ai.Profile method*), 76
chunk_overlap (*select_ai.OracleVectorIndexAttributes attribute*), 157
chunk_overlap (*select_ai.VectorIndexAttributes attribute*), 156
chunk_processing_method (*select_ai.summary.SummaryParams attribute*), 193
chunk_size (*select_ai.OracleVectorIndexAttributes attribute*), 157
chunk_size (*select_ai.VectorIndexAttributes attribute*), 156
ChunkProcessingMethod (*class in select_ai.summary*), 194
client_identifier (*select_ai.ConversationPrompt attribute*), 143
client_ip (*select_ai.ConversationPrompt attribute*), 143
CohereProvider (*class in select_ai*), 36
comments (*select_ai.ProfileAttributes attribute*), 70
comments (*select_ai.SyntheticDataParams attribute*), 184
constraints (*select_ai.ProfileAttributes attribute*), 70

Conversation (*class in select_ai*), 138
conversation (*select_ai.ProfileAttributes attribute*), 70
conversation_id (*select_ai.agent.history.TeamHistoryEvent attribute*), 355
conversation_id (*select_ai.ConversationPrompt attribute*), 143
conversation_length (*select_ai.ConversationAttributes attribute*), 137
conversation_params (*select_ai.agent.history.TaskHistoryEvent attribute*), 355
conversation_prompt_id (*select_ai.ConversationPrompt attribute*), 143
conversation_title (*select_ai.ConversationPrompt attribute*), 143
ConversationAttributes (*class in select_ai*), 137
ConversationNotFoundError, 351
ConversationPrompt (*class in select_ai*), 143
COSINE (*select_ai.vector_index.VectorDistanceMetric attribute*), 372
count() (*select_ai.action.Action method*), 357
count() (*select_ai.agent.tool.NotificationType method*), 367
count() (*select_ai.agent.tool.ToolType method*), 362
count() (*select_ai.summary.ChunkProcessingMethod method*), 194
count() (*select_ai.summary.ExtractivenessLevel method*), 200
count() (*select_ai.summary.Style method*), 206
count() (*select_ai.vector_index.VectorDBProvider method*), 377
count() (*select_ai.vector_index.VectorDistanceMetric method*), 372
create() (*select_ai.agent.Agent method*), 232
create() (*select_ai.agent.AsyncAgent method*), 269
create() (*select_ai.agent.AsyncTask method*), 265
create() (*select_ai.agent.AsyncTeam method*), 273
create() (*select_ai.agent.AsyncTool method*), 256
create() (*select_ai.agent.Task method*), 227
create() (*select_ai.agent.Team method*), 240
create() (*select_ai.agent.Tool method*), 217
create() (*select_ai.agent.ToolAttributes class method*), 215
create() (*select_ai.agent.ToolParams class method*), 215
create() (*select_ai.AnthropicProvider class method*), 33
create() (*select_ai.AsyncConversation method*), 147
create() (*select_ai.AsyncProfile method*), 105
create() (*select_ai.AsyncVectorIndex method*), 170
create() (*select_ai.AWSProvider class method*), 35
create() (*select_ai.AzureProvider class method*), 34
create() (*select_ai.CohereProvider class method*), 36
create() (*select_ai.Conversation method*), 138
create() (*select_ai.GoogleProvider class method*), 39
create() (*select_ai.HuggingFaceProvider class method*), 40
create() (*select_ai.OCIGenAIProvider class method*), 38
create() (*select_ai.OpenAIProvider class method*), 37
create() (*select_ai.OracleVectorIndexAttributes class method*), 157
create() (*select_ai.Profile method*), 76
create() (*select_ai.ProfileAttributes class method*), 70
create() (*select_ai.Provider class method*), 32
create() (*select_ai.VectorIndex method*), 159
create() (*select_ai.VectorIndexAttributes class method*), 156
create_built_in_tool() (*select_ai.agent.AsyncTool class method*), 256
create_built_in_tool() (*select_ai.agent.Tool class method*), 217
create_email_notification_tool() (*select_ai.agent.AsyncTool class method*), 257
create_email_notification_tool() (*select_ai.agent.Tool class method*), 217
create_pl_sql_tool() (*select_ai.agent.AsyncTool class method*), 257
create_pl_sql_tool() (*select_ai.agent.Tool class method*), 217
create_rag_tool() (*select_ai.agent.AsyncTool class method*), 257
create_rag_tool() (*select_ai.agent.Tool class method*), 218
create_slack_notification_tool() (*select_ai.agent.AsyncTool class method*), 258
create_slack_notification_tool() (*select_ai.agent.Tool class method*), 218
create_sql_tool() (*select_ai.agent.AsyncTool class method*), 258
create_sql_tool() (*select_ai.agent.Tool class method*), 218
create_websearch_tool() (*select_ai.agent.AsyncTool class method*), 258
create_websearch_tool() (*select_ai.agent.Tool class method*), 218
created (*select_ai.ConversationPrompt attribute*), 143
credential_name (*select_ai.agent.ToolParams attribute*), 216
credential_name (*select_ai.ProfileAttributes attribute*), 70

D

DatabaseNotConnectedError, 351
delete() (*select_ai.agent.Agent method*), 232
delete() (*select_ai.agent.AsyncAgent method*), 269

delete() (*select_ai.agent.AsyncTask method*), 265
delete() (*select_ai.agent.AsyncTeam method*), 273
delete() (*select_ai.agent.AsyncTool method*), 258
delete() (*select_ai.agent.Task method*), 227
delete() (*select_ai.agent.Team method*), 240
delete() (*select_ai.agent.Tool method*), 219
delete() (*select_ai.AsyncConversation method*), 147
delete() (*select_ai.AsyncProfile method*), 105
delete() (*select_ai.AsyncVectorIndex method*), 170
delete() (*select_ai.Conversation method*), 138
delete() (*select_ai.Profile method*), 77
delete() (*select_ai.VectorIndex method*), 159
delete_agent() (*select_ai.agent.Agent class method*), 232
delete_agent() (*select_ai.agent.AsyncAgent class method*), 269
delete_feedback() (*select_ai.AsyncProfile method*), 105
delete_feedback() (*select_ai.Profile method*), 77
delete_index() (*select_ai.AsyncVectorIndex class method*), 170
delete_index() (*select_ai.VectorIndex class method*), 159
delete_profile() (*select_ai.AsyncProfile class method*), 105
delete_profile() (*select_ai.Profile class method*), 77
delete_prompt() (*select_ai.AsyncConversation method*), 147
delete_prompt() (*select_ai.Conversation method*), 138
delete_task() (*select_ai.agent.AsyncTask class method*), 265
delete_task() (*select_ai.agent.Task class method*), 227
delete_team() (*select_ai.agent.AsyncTeam class method*), 273
delete_team() (*select_ai.agent.Team class method*), 240
delete_tool() (*select_ai.agent.AsyncTool class method*), 259
delete_tool() (*select_ai.agent.Tool class method*), 219
describe_team() (*select_ai.agent.AsyncTeam method*), 273
describe_team() (*select_ai.agent.Team method*), 240
describe_tool() (*select_ai.agent.AsyncTool method*), 259
describe_tool() (*select_ai.agent.Tool method*), 219
description (*select_ai.ConversationAttributes attribute*), 137
dict() (*select_ai.agent.AgentAttributes method*), 231
dict() (*select_ai.agent.history.TaskHistoryEvent method*), 355
dict() (*select_ai.agent.history.TeamHistoryEvent method*), 355
dict() (*select_ai.agent.history.ToolHistoryEvent method*), 356
dict() (*select_ai.agent.TaskAttributes method*), 226
dict() (*select_ai.agent.TeamAttributes method*), 235
dict() (*select_ai.agent.ToolAttributes method*), 215
dict() (*select_ai.agent.ToolParams method*), 216
dict() (*select_ai.AnthropicProvider method*), 33
dict() (*select_ai.AWSProvider method*), 35
dict() (*select_ai.AzureProvider method*), 34
dict() (*select_ai.CohereProvider method*), 36
dict() (*select_ai.ConversationAttributes method*), 137
dict() (*select_ai.GoogleProvider method*), 39
dict() (*select_ai.HuggingFaceProvider method*), 40
dict() (*select_ai.OCIGenAIProvider method*), 38
dict() (*select_ai.OpenAIProvider method*), 37
dict() (*select_ai.OracleVectorIndexAttributes method*), 157
dict() (*select_ai.ProfileAttributes method*), 70
dict() (*select_ai.Provider method*), 32
dict() (*select_ai.summary.SummaryParams method*), 193
dict() (*select_ai.SyntheticDataAttributes method*), 183
dict() (*select_ai.SyntheticDataParams method*), 184
dict() (*select_ai.VectorIndexAttributes method*), 156
disable() (*select_ai.agent.Agent method*), 232
disable() (*select_ai.agent.AsyncAgent method*), 269
disable() (*select_ai.agent.AsyncTask method*), 265
disable() (*select_ai.agent.AsyncTeam method*), 273
disable() (*select_ai.agent.AsyncTool method*), 259
disable() (*select_ai.agent.Task method*), 227
disable() (*select_ai.agent.Team method*), 240
disable() (*select_ai.agent.Tool method*), 219
disable() (*select_ai.AsyncProfile method*), 105
disable() (*select_ai.AsyncVectorIndex method*), 170
disable() (*select_ai.Profile method*), 77
disable() (*select_ai.VectorIndex method*), 160
DOT (*select_ai.vector_index.VectorDistanceMetric attribute*), 372

E

EMAIL (*select_ai.agent.tool.NotificationType attribute*), 367
embedding_model (*select_ai.AnthropicProvider attribute*), 33
embedding_model (*select_ai.AWSProvider attribute*), 35
embedding_model (*select_ai.AzureProvider attribute*), 34
embedding_model (*select_ai.CohereProvider attribute*), 36
embedding_model (*select_ai.GoogleProvider attribute*), 39
embedding_model (*select_ai.HuggingFaceProvider attribute*), 40
embedding_model (*select_ai.OCIGenAIProvider attribute*), 38

embedding_model (*select_ai.OpenAIProvider attribute*), 37
embedding_model (*select_ai.Provider attribute*), 32
enable() (*select_ai.agent.Agent method*), 232
enable() (*select_ai.agent.AsyncAgent method*), 269
enable() (*select_ai.agent.AsyncTask method*), 265
enable() (*select_ai.agent.AsyncTeam method*), 273
enable() (*select_ai.agent.AsyncTool method*), 259
enable() (*select_ai.agent.Task method*), 227
enable() (*select_ai.agent.Team method*), 240
enable() (*select_ai.agent.Tool method*), 219
enable() (*select_ai.AsyncProfile method*), 106
enable() (*select_ai.AsyncVectorIndex method*), 171
enable() (*select_ai.Profile method*), 77
enable() (*select_ai.VectorIndex method*), 160
enable_custom_source_uri (*select_ai.ProfileAttributes attribute*), 70
enable_human_tool (*select_ai.agent.AgentAttributes attribute*), 231
enable_human_tool (*select_ai.agent.TaskAttributes attribute*), 226
enable_source_offsets (*select_ai.ProfileAttributes attribute*), 70
enable_sources (*select_ai.OracleVectorIndexAttributes attribute*), 157
enable_sources (*select_ai.ProfileAttributes attribute*), 70
enable_sources (*select_ai.VectorIndexAttributes attribute*), 156
encode() (*select_ai.action.Action method*), 357
encode() (*select_ai.agent.tool.NotificationType method*), 367
encode() (*select_ai.agent.tool.ToolType method*), 362
encode() (*select_ai.summary.ChunkProcessingMethod method*), 194
encode() (*select_ai.summary.ExtractivenessLevel method*), 200
encode() (*select_ai.summary.Style method*), 206
encode() (*select_ai.vector_index.VectorDBProvider method*), 378
encode() (*select_ai.vector_index.VectorDistanceMetric method*), 373
end_date (*select_ai.agent.history.TaskHistoryEvent attribute*), 356
end_date (*select_ai.agent.history.TeamHistoryEvent attribute*), 355
end_date (*select_ai.agent.history.ToolHistoryEvent attribute*), 356
endpoint (*select_ai.agent.ToolParams attribute*), 216
endswith() (*select_ai.action.Action method*), 357
endswith() (*select_ai.agent.tool.NotificationType method*), 368
endswith() (*select_ai.agent.tool.ToolType method*), 362
endswith() (*select_ai.summary.ChunkProcessingMethod method*), 194
endswith() (*select_ai.summary.ExtractivenessLevel method*), 200
endswith() (*select_ai.summary.Style method*), 206
endswith() (*select_ai.vector_index.VectorDBProvider method*), 378
endswith() (*select_ai.vector_index.VectorDistanceMetric method*), 373
enforce_object_list (*select_ai.ProfileAttributes attribute*), 70
EUCLIDEAN (*select_ai.vector_index.VectorDistanceMetric attribute*), 372
expandtabs() (*select_ai.action.Action method*), 358
expandtabs() (*select_ai.agent.tool.NotificationType method*), 368
expandtabs() (*select_ai.agent.tool.ToolType method*), 363
expandtabs() (*select_ai.summary.ChunkProcessingMethod method*), 194
expandtabs() (*select_ai.summary.ExtractivenessLevel method*), 200
expandtabs() (*select_ai.summary.Style method*), 206
expandtabs() (*select_ai.vector_index.VectorDBProvider method*), 378
expandtabs() (*select_ai.vector_index.VectorDistanceMetric method*), 373
explain_sql() (*select_ai.async_profile.AsyncSession method*), 354
explain_sql() (*select_ai.AsyncProfile method*), 106
explain_sql() (*select_ai.Profile method*), 77
explain_sql() (*select_ai.profile.Session method*), 352
EXPLAINSQL (*select_ai.action.Action attribute*), 357
export() (*select_ai.agent.AsyncTeam method*), 273
export() (*select_ai.agent.Team method*), 240
export_team() (*select_ai.agent.AsyncTeam class method*), 274
export_team() (*select_ai.agent.Team class method*), 241
extractiveness_level (*select_ai.summary.SummaryParams attribute*), 193
ExtractivenessLevel (*class in select_ai.summary*), 200

F

FEEDBACK (*select_ai.action.Action attribute*), 357
fetch() (*select_ai.agent.Agent class method*), 232
fetch() (*select_ai.agent.AsyncAgent class method*), 269
fetch() (*select_ai.agent.AsyncTask class method*), 265
fetch() (*select_ai.agent.AsyncTeam class method*), 274
fetch() (*select_ai.agent.AsyncTool class method*), 259
fetch() (*select_ai.agent.Task class method*), 227
fetch() (*select_ai.agent.Team class method*), 241
fetch() (*select_ai.agent.Tool class method*), 219

fetch() (*select_ai.AsyncConversation class method*), 147
fetch() (*select_ai.AsyncProfile class method*), 106
fetch() (*select_ai.AsyncVectorIndex class method*), 171
fetch() (*select_ai.Conversation class method*), 138
fetch() (*select_ai.Profile class method*), 78
fetch() (*select_ai.VectorIndex class method*), 160
find() (*select_ai.action.Action method*), 358
find() (*select_ai.agent.tool.NotificationType method*), 368
find() (*select_ai.agent.tool.ToolType method*), 363
find() (*select_ai.summary.ChunkProcessingMethod method*), 194
find() (*select_ai.summary.ExtractivenessLevel method*), 200
find() (*select_ai.summary.Style method*), 206
find() (*select_ai.vector_index.VectorDBProvider method*), 378
find() (*select_ai.vector_index.VectorDistanceMetric method*), 373
format() (*select_ai.action.Action method*), 358
format() (*select_ai.agent.tool.NotificationType method*), 368
format() (*select_ai.agent.tool.ToolType method*), 363
format() (*select_ai.summary.ChunkProcessingMethod method*), 194
format() (*select_ai.summary.ExtractivenessLevel method*), 201
format() (*select_ai.summary.Style method*), 206
format() (*select_ai.vector_index.VectorDBProvider method*), 378
format() (*select_ai.vector_index.VectorDistanceMetric method*), 373
format_map() (*select_ai.action.Action method*), 358
format_map() (*select_ai.agent.tool.NotificationType method*), 368
format_map() (*select_ai.agent.tool.ToolType method*), 363
format_map() (*select_ai.summary.ChunkProcessingMethod method*), 195
format_map() (*select_ai.summary.ExtractivenessLevel method*), 201
format_map() (*select_ai.summary.Style method*), 207
format_map() (*select_ai.vector_index.VectorDBProvider method*), 378
format_map() (*select_ai.vector_index.VectorDistanceMetric method*), 373
function (*select_ai.agent.ToolAttributes attribute*), 215

G

generate() (*select_ai.AsyncProfile method*), 106
generate() (*select_ai.Profile method*), 78
generate_synthetic_data() (*select_ai.AsyncProfile method*), 107
generate_synthetic_data() (*select_ai.Profile method*), 78
get_attributes() (*select_ai.AsyncConversation method*), 147
get_attributes() (*select_ai.AsyncProfile method*), 107
get_attributes() (*select_ai.AsyncVectorIndex method*), 171
get_attributes() (*select_ai.Conversation method*), 138
get_attributes() (*select_ai.Profile method*), 79
get_attributes() (*select_ai.VectorIndex method*), 160
get_next_refresh_timestamp() (*select_ai.AsyncVectorIndex method*), 171
get_next_refresh_timestamp() (*select_ai.VectorIndex method*), 160
get_profile() (*select_ai.AsyncVectorIndex method*), 171
get_profile() (*select_ai.VectorIndex method*), 160
GoogleProvider (*class in select_ai*), 39
grant_access() (*select_ai.agent.AsyncTeam method*), 274
grant_access() (*select_ai.agent.Team method*), 241
grant_access() (*select_ai.AsyncProfile method*), 107
grant_access() (*select_ai.AsyncVectorIndex method*), 171
grant_access() (*select_ai.Profile method*), 79
grant_access() (*select_ai.VectorIndex method*), 160

H

HAMMING (*select_ai.vector_index.VectorDistanceMetric attribute*), 372
HIGH (*select_ai.summary.ExtractivenessLevel attribute*), 200
HTTP (*select_ai.agent.tool.ToolType attribute*), 362
HuggingFaceProvider (*class in select_ai*), 40
HUMAN (*select_ai.agent.tool.ToolType attribute*), 362

I

import_team() (*select_ai.agent.AsyncTeam class method*), 274
import_team() (*select_ai.agent.Team class method*), 241
index() (*select_ai.action.Action method*), 358
index() (*select_ai.agent.tool.NotificationType method*), 368
index() (*select_ai.agent.tool.ToolType method*), 363
index() (*select_ai.summary.ChunkProcessingMethod method*), 195
index() (*select_ai.summary.ExtractivenessLevel method*), 201
index() (*select_ai.summary.Style method*), 207
index() (*select_ai.vector_index.VectorDBProvider method*), 378

index() (*select_ai.vector_index.VectorDistanceMetric method*), 373
input (*select_ai.agent.history.TaskHistoryEvent attribute*), 356
input (*select_ai.agent.history.ToolHistoryEvent attribute*), 356
input (*select_ai.agent.TaskAttributes attribute*), 226
instruction (*select_ai.agent.TaskAttributes attribute*), 226
instruction (*select_ai.agent.ToolAttributes attribute*), 215
InvalidSQLError, 351
invocation_id (*select_ai.agent.history.ToolHistoryEvent attribute*), 356
isalnum() (*select_ai.action.Action method*), 358
isalnum() (*select_ai.agent.tool.NotificationType method*), 368
isalnum() (*select_ai.agent.tool.ToolType method*), 363
isalnum() (*select_ai.summary.ChunkProcessingMethod method*), 195
isalnum() (*select_ai.summary.ExtractivenessLevel method*), 201
isalnum() (*select_ai.summary.Style method*), 207
isalnum() (*select_ai.vector_index.VectorDBProvider method*), 378
isalnum() (*select_ai.vector_index.VectorDistanceMetric method*), 373
isalpha() (*select_ai.action.Action method*), 358
isalpha() (*select_ai.agent.tool.NotificationType method*), 368
isalpha() (*select_ai.agent.tool.ToolType method*), 363
isalpha() (*select_ai.summary.ChunkProcessingMethod method*), 195
isalpha() (*select_ai.summary.ExtractivenessLevel method*), 201
isalpha() (*select_ai.summary.Style method*), 207
isalpha() (*select_ai.vector_index.VectorDBProvider method*), 378
isalpha() (*select_ai.vector_index.VectorDistanceMetric method*), 373
isascii() (*select_ai.action.Action method*), 358
isascii() (*select_ai.agent.tool.NotificationType method*), 368
isascii() (*select_ai.agent.tool.ToolType method*), 363
isascii() (*select_ai.summary.ChunkProcessingMethod method*), 195
isascii() (*select_ai.summary.ExtractivenessLevel method*), 201
isascii() (*select_ai.summary.Style method*), 207
isascii() (*select_ai.vector_index.VectorDBProvider method*), 379
isascii() (*select_ai.vector_index.VectorDistanceMetric method*), 374
isdecimal() (*select_ai.action.Action method*), 358
isdecimal() (*select_ai.agent.tool.NotificationType method*), 368
isdecimal() (*select_ai.agent.tool.ToolType method*), 363
isdecimal() (*select_ai.summary.ChunkProcessingMethod method*), 195
isdecimal() (*select_ai.summary.ExtractivenessLevel method*), 201
isdecimal() (*select_ai.summary.Style method*), 207
isdecimal() (*select_ai.vector_index.VectorDBProvider method*), 379
isdecimal() (*select_ai.vector_index.VectorDistanceMetric method*), 374
isdigit() (*select_ai.action.Action method*), 358
isdigit() (*select_ai.agent.tool.NotificationType method*), 369
isdigit() (*select_ai.agent.tool.ToolType method*), 363
isdigit() (*select_ai.summary.ChunkProcessingMethod method*), 195
isdigit() (*select_ai.summary.ExtractivenessLevel method*), 201
isdigit() (*select_ai.summary.Style method*), 207
isdigit() (*select_ai.vector_index.VectorDBProvider method*), 379
isdigit() (*select_ai.vector_index.VectorDistanceMetric method*), 374
isidentifier() (*select_ai.action.Action method*), 358
isidentifier() (*select_ai.agent.tool.NotificationType method*), 369
isidentifier() (*select_ai.agent.tool.ToolType method*), 364
isidentifier() (*select_ai.summary.ChunkProcessingMethod method*), 195
isidentifier() (*select_ai.summary.ExtractivenessLevel method*), 201
isidentifier() (*select_ai.summary.Style method*), 207
isidentifier() (*select_ai.vector_index.VectorDBProvider method*), 379
isidentifier() (*select_ai.vector_index.VectorDistanceMetric method*), 374
islower() (*select_ai.action.Action method*), 358
islower() (*select_ai.agent.tool.NotificationType method*), 369
islower() (*select_ai.agent.tool.ToolType method*), 364
islower() (*select_ai.summary.ChunkProcessingMethod method*), 195
islower() (*select_ai.summary.ExtractivenessLevel method*), 201
islower() (*select_ai.summary.Style method*), 207
islower() (*select_ai.vector_index.VectorDBProvider method*), 379
islower() (*select_ai.vector_index.VectorDistanceMetric method*), 374
isnumeric() (*select_ai.action.Action method*), 359

isnumeric() (*select_ai.agent.tool.NotificationType method*), 369
isnumeric() (*select_ai.agent.tool.ToolType method*), 364
isnumeric() (*select_ai.summary.ChunkProcessingMethod method*), 195
isnumeric() (*select_ai.summary.ExtractivenessLevel method*), 201
isnumeric() (*select_ai.summary.Style method*), 207
isnumeric() (*select_ai.vector_index.VectorDBProvider method*), 379
isnumeric() (*select_ai.vector_index.VectorDistanceMetric method*), 374
isprintable() (*select_ai.action.Action method*), 359
isprintable() (*select_ai.agent.tool.NotificationType method*), 369
isprintable() (*select_ai.agent.tool.ToolType method*), 364
isprintable() (*select_ai.summary.ChunkProcessingMethod method*), 195
isprintable() (*select_ai.summary.ExtractivenessLevel method*), 202
isprintable() (*select_ai.summary.Style method*), 207
isprintable() (*select_ai.vector_index.VectorDBProvider method*), 379
isprintable() (*select_ai.vector_index.VectorDistanceMetric method*), 374
isspace() (*select_ai.action.Action method*), 359
isspace() (*select_ai.agent.tool.NotificationType method*), 369
isspace() (*select_ai.agent.tool.ToolType method*), 364
isspace() (*select_ai.summary.ChunkProcessingMethod method*), 196
isspace() (*select_ai.summary.ExtractivenessLevel method*), 202
isspace() (*select_ai.summary.Style method*), 208
isspace() (*select_ai.vector_index.VectorDBProvider method*), 379
isspace() (*select_ai.vector_index.VectorDistanceMetric method*), 374
istitle() (*select_ai.action.Action method*), 359
istitle() (*select_ai.agent.tool.NotificationType method*), 369
istitle() (*select_ai.agent.tool.ToolType method*), 364
istitle() (*select_ai.summary.ChunkProcessingMethod method*), 196
istitle() (*select_ai.summary.ExtractivenessLevel method*), 202
istitle() (*select_ai.summary.Style method*), 208
istitle() (*select_ai.vector_index.VectorDBProvider method*), 379
istitle() (*select_ai.vector_index.VectorDistanceMetric method*), 374
isupper() (*select_ai.action.Action method*), 359
isupper() (*select_ai.agent.tool.NotificationType method*), 369
isupper() (*select_ai.agent.tool.ToolType method*), 364
isupper() (*select_ai.summary.ChunkProcessingMethod method*), 196
isupper() (*select_ai.summary.ExtractivenessLevel method*), 202
isupper() (*select_ai.summary.Style method*), 208
isupper() (*select_ai.vector_index.VectorDBProvider method*), 379
isupper() (*select_ai.vector_index.VectorDistanceMetric method*), 374
ITERATIVE_REFINEMENT (*select_ai.summary.ChunkProcessingMethod attribute*), 194

J

join() (*select_ai.action.Action method*), 359
join() (*select_ai.agent.tool.NotificationType method*), 369
join() (*select_ai.agent.tool.ToolType method*), 364
join() (*select_ai.summary.ChunkProcessingMethod method*), 196
join() (*select_ai.summary.ExtractivenessLevel method*), 202
join() (*select_ai.summary.Style method*), 208
join() (*select_ai.vector_index.VectorDBProvider method*), 379
join() (*select_ai.vector_index.VectorDistanceMetric method*), 374
json() (*select_ai.agent.AgentAttributes method*), 231
json() (*select_ai.agent.history.TaskHistoryEvent method*), 356
json() (*select_ai.agent.history.TeamHistoryEvent method*), 355
json() (*select_ai.agent.history.ToolHistoryEvent method*), 356
json() (*select_ai.agent.TaskAttributes method*), 226
json() (*select_ai.agent.TeamAttributes method*), 235
json() (*select_ai.agent.ToolAttributes method*), 215
json() (*select_ai.agent.ToolParams method*), 216
json() (*select_ai.AnthropicProvider method*), 33
json() (*select_ai.AWSProvider method*), 35
json() (*select_ai.AzureProvider method*), 34
json() (*select_ai.CohereProvider method*), 36
json() (*select_ai.ConversationAttributes method*), 137
json() (*select_ai.GoogleProvider method*), 39
json() (*select_ai.HuggingFaceProvider method*), 40
json() (*select_ai.OCIGenAIProvider method*), 38
json() (*select_ai.OpenAIProvider method*), 37
json() (*select_ai.OracleVectorIndexAttributes method*), 157
json() (*select_ai.ProfileAttributes method*), 70
json() (*select_ai.Provider method*), 32

json() (*select_ai.summary.SummaryParams method*), 193
json() (*select_ai.SyntheticDataAttributes method*), 183
json() (*select_ai.SyntheticDataParams method*), 184
json() (*select_ai.VectorIndexAttributes method*), 157

K

key_alias() (*select_ai.AnthropicProvider class method*), 33
key_alias() (*select_ai.AWSProvider class method*), 35
key_alias() (*select_ai.AzureProvider class method*), 34
key_alias() (*select_ai.CohereProvider class method*), 36
key_alias() (*select_ai.GoogleProvider class method*), 39
key_alias() (*select_ai.HuggingFaceProvider class method*), 40
key_alias() (*select_ai.OCIGenAIProvider class method*), 38
key_alias() (*select_ai.OpenAIProvider class method*), 37
key_alias() (*select_ai.Provider class method*), 32
keys() (*select_ai.agent.AgentAttributes class method*), 231
keys() (*select_ai.agent.history.TaskHistoryEvent class method*), 356
keys() (*select_ai.agent.history.TeamHistoryEvent class method*), 355
keys() (*select_ai.agent.history.ToolHistoryEvent class method*), 356
keys() (*select_ai.agent.TaskAttributes class method*), 226
keys() (*select_ai.agent.TeamAttributes class method*), 235
keys() (*select_ai.agent.ToolAttributes class method*), 215
keys() (*select_ai.agent.ToolParams class method*), 216
keys() (*select_ai.AnthropicProvider class method*), 33
keys() (*select_ai.AWSProvider class method*), 35
keys() (*select_ai.AzureProvider class method*), 34
keys() (*select_ai.CohereProvider class method*), 36
keys() (*select_ai.ConversationAttributes class method*), 137
keys() (*select_ai.GoogleProvider class method*), 39
keys() (*select_ai.HuggingFaceProvider class method*), 40
keys() (*select_ai.OCIGenAIProvider class method*), 38
keys() (*select_ai.OpenAIProvider class method*), 37
keys() (*select_ai.OracleVectorIndexAttributes class method*), 158
keys() (*select_ai.ProfileAttributes class method*), 71
keys() (*select_ai.Provider class method*), 32
keys() (*select_ai.summary.SummaryParams class method*), 193
keys() (*select_ai.SyntheticDataAttributes class method*), 183
keys() (*select_ai.SyntheticDataParams class method*), 184
keys() (*select_ai.VectorIndexAttributes class method*), 157

L

L2_SQUARED (*select_ai.vector_index.VectorDistanceMetric attribute*), 372
LIST (*select_ai.summary.Style attribute*), 206
list() (*select_ai.agent.Agent class method*), 232
list() (*select_ai.agent.AsyncAgent class method*), 269
list() (*select_ai.agent.AsyncTask class method*), 265
list() (*select_ai.agent.AsyncTaskHistory class method*), 288
list() (*select_ai.agent.AsyncTeam class method*), 275
list() (*select_ai.agent.AsyncTeamHistory class method*), 288
list() (*select_ai.agent.AsyncTool class method*), 259
list() (*select_ai.agent.AsyncToolHistory class method*), 288
list() (*select_ai.agent.Task class method*), 227
list() (*select_ai.agent.TaskHistory class method*), 250
list() (*select_ai.agent.Team class method*), 242
list() (*select_ai.agent.TeamHistory class method*), 250
list() (*select_ai.agent.Tool class method*), 219
list() (*select_ai.agent.ToolHistory class method*), 250
list() (*select_ai.AsyncConversation class method*), 147
list() (*select_ai.AsyncProfile class method*), 107
list() (*select_ai.AsyncVectorIndex class method*), 171
list() (*select_ai.Conversation class method*), 138
list() (*select_ai.Profile class method*), 79
list() (*select_ai.VectorIndex class method*), 160
list_prompts() (*select_ai.AsyncConversation method*), 147
list_prompts() (*select_ai.Conversation method*), 138
list_tools() (*select_ai.agent.AsyncTeam method*), 275
list_tools() (*select_ai.agent.Team method*), 242
ljust() (*select_ai.action.Action method*), 359
ljust() (*select_ai.agent.tool.NotificationType method*), 369
ljust() (*select_ai.agent.tool.ToolType method*), 364
ljust() (*select_ai.summary.ChunkProcessingMethod method*), 196
ljust() (*select_ai.summary.ExtractivenessLevel method*), 202
ljust() (*select_ai.summary.Style method*), 208
ljust() (*select_ai.vector_index.VectorDBProvider method*), 380
ljust() (*select_ai.vector_index.VectorDistanceMetric method*), 375
location (*select_ai.OracleVectorIndexAttributes attribute*), 158

location (*select_ai.VectorIndexAttributes attribute*), 157
LOW (*select_ai.summary.ExtractivenessLevel attribute*), 200
lower() (*select_ai.action.Action method*), 359
lower() (*select_ai.agent.tool.NotificationType method*), 370
lower() (*select_ai.agent.tool.ToolType method*), 364
lower() (*select_ai.summary.ChunkProcessingMethod method*), 196
lower() (*select_ai.summary.ExtractivenessLevel method*), 202
lower() (*select_ai.summary.Style method*), 208
lower() (*select_ai.vector_index.VectorDBProvider method*), 380
lower() (*select_ai.vector_index.VectorDistanceMetric method*), 375
lstrip() (*select_ai.action.Action method*), 359
lstrip() (*select_ai.agent.tool.NotificationType method*), 370
lstrip() (*select_ai.agent.tool.ToolType method*), 364
lstrip() (*select_ai.summary.ChunkProcessingMethod method*), 196
lstrip() (*select_ai.summary.ExtractivenessLevel method*), 202
lstrip() (*select_ai.summary.Style method*), 208
lstrip() (*select_ai.vector_index.VectorDBProvider method*), 380
lstrip() (*select_ai.vector_index.VectorDistanceMetric method*), 375

M

maketrans() (*select_ai.action.Action static method*), 359
maketrans() (*select_ai.agent.tool.NotificationType static method*), 370
maketrans() (*select_ai.agent.tool.ToolType static method*), 365
maketrans() (*select_ai.summary.ChunkProcessingMethod static method*), 196
maketrans() (*select_ai.summary.ExtractivenessLevel static method*), 202
maketrans() (*select_ai.summary.Style static method*), 208
maketrans() (*select_ai.vector_index.VectorDBProvider static method*), 380
maketrans() (*select_ai.vector_index.VectorDistanceMetric static method*), 375
MANHATTAN (*select_ai.vector_index.VectorDistanceMetric attribute*), 372
MAP_REDUCE (*select_ai.summary.ChunkProcessingMethod attribute*), 194
match_limit (*select_ai.OracleVectorIndexAttributes attribute*), 158
match_limit (*select_ai.VectorIndexAttributes attribute*), 157
max_tokens (*select_ai.ProfileAttributes attribute*), 71
max_words (*select_ai.summary.SummaryParams attribute*), 193
MEDIUM (*select_ai.summary.ExtractivenessLevel attribute*), 200
min_words (*select_ai.summary.SummaryParams attribute*), 193
model (*select_ai.AnthropicProvider attribute*), 33
model (*select_ai.AWSProvider attribute*), 35
model (*select_ai.AzureProvider attribute*), 34
model (*select_ai.CohereProvider attribute*), 36
model (*select_ai.GoogleProvider attribute*), 39
model (*select_ai.HuggingFaceProvider attribute*), 40
model (*select_ai.OCIGenAIProvider attribute*), 38
model (*select_ai.OpenAIProvider attribute*), 37
model (*select_ai.Provider attribute*), 32
modified (*select_ai.ConversationPrompt attribute*), 143
module
    select_ai.errors, 349

N

NARRATE (*select_ai.action.Action attribute*), 357
narrate() (*select_ai.async_profile.AsyncSession method*), 354
narrate() (*select_ai.AsyncProfile method*), 107
narrate() (*select_ai.Profile method*), 79
narrate() (*select_ai.profile.Session method*), 353
NOTIFICATION (*select_ai.agent.tool.ToolType attribute*), 362
notification_type (*select_ai.agent.ToolParams attribute*), 216
NotificationType (*class in select_ai.agent.tool*), 367

O

object_list (*select_ai.ProfileAttributes attribute*), 71
object_list (*select_ai.SyntheticDataAttributes attribute*), 183
object_list_mode (*select_ai.ProfileAttributes attribute*), 71
object_name (*select_ai.SyntheticDataAttributes attribute*), 183
object_storage_credential_name (*select_ai.OracleVectorIndexAttributes attribute*), 158
object_storage_credential_name (*select_ai.VectorIndexAttributes attribute*), 157
oci_apiformat (*select_ai.OCIGenAIProvider attribute*), 38
oci_compartment_id (*select_ai.OCIGenAIProvider attribute*), 38

oci_endpoint_id (*select_ai.OCIGenAIProvider attribute*), 38
oci_runtimetype (*select_ai.OCIGenAIProvider attribute*), 38
OCIGenAIProvider (*class in select_ai*), 38
OpenAIProvider (*class in select_ai*), 37
ORACLE (*select_ai.vector_index.VectorDBProvider attribute*), 377
OracleVectorIndexAttributes (*class in select_ai*), 157
output (*select_ai.agent.history.ToolHistoryEvent attribute*), 356
owner_name (*select_ai.SyntheticDataAttributes attribute*), 183

P

PARAGRAPH (*select_ai.summary.Style attribute*), 206
params (*select_ai.agent.history.TeamHistoryEvent attribute*), 355
params (*select_ai.SyntheticDataAttributes attribute*), 183
partition() (*select_ai.action.Action method*), 360
partition() (*select_ai.agent.tool.NotificationType method*), 370
partition() (*select_ai.agent.tool.ToolType method*), 365
partition() (*select_ai.summary.ChunkProcessingMethod method*), 196
partition() (*select_ai.summary.ExtractivenessLevel method*), 202
partition() (*select_ai.summary.Style method*), 208
partition() (*select_ai.vector_index.VectorDBProvider method*), 380
partition() (*select_ai.vector_index.VectorDistanceMetric method*), 375
pipeline_name (*select_ai.OracleVectorIndexAttributes attribute*), 158
pipeline_name (*select_ai.VectorIndexAttributes attribute*), 157
prepare() (*select_ai.SyntheticDataAttributes method*), 183
priority (*select_ai.SyntheticDataParams attribute*), 184
process (*select_ai.agent.TeamAttributes attribute*), 235
Profile (*class in select_ai*), 76
profile_dict() (*select_ai.AnthropicProvider method*), 33
profile_dict() (*select_ai.AWSProvider method*), 35
profile_dict() (*select_ai.AzureProvider method*), 34
profile_dict() (*select_ai.CohereProvider method*), 36
profile_dict() (*select_ai.GoogleProvider method*), 39
profile_dict() (*select_ai.HuggingFaceProvider method*), 40
profile_dict() (*select_ai.OCIGenAIProvider method*), 38
profile_dict() (*select_ai.OpenAIProvider method*), 37
profile_dict() (*select_ai.Provider method*), 32
profile_name (*select_ai.agent.AgentAttributes attribute*), 231
profile_name (*select_ai.agent.ToolParams attribute*), 216
profile_name (*select_ai.ConversationPrompt attribute*), 143
profile_name (*select_ai.OracleVectorIndexAttributes attribute*), 158
profile_name (*select_ai.VectorIndexAttributes attribute*), 157
ProfileAttributes (*class in select_ai*), 69
ProfileAttributesEmptyError, 351
ProfileExistsError, 351
ProfileNotFoundError, 352
prompt (*select_ai.ConversationPrompt attribute*), 143
prompt_action (*select_ai.ConversationPrompt attribute*), 143
prompt_response (*select_ai.ConversationPrompt attribute*), 143
Provider (*class in select_ai*), 32
provider (*select_ai.ProfileAttributes attribute*), 71
provider_endpoint (*select_ai.AnthropicProvider attribute*), 33
provider_endpoint (*select_ai.AWSProvider attribute*), 35
provider_endpoint (*select_ai.AzureProvider attribute*), 34
provider_endpoint (*select_ai.CohereProvider attribute*), 36
provider_endpoint (*select_ai.GoogleProvider attribute*), 39
provider_endpoint (*select_ai.HuggingFaceProvider attribute*), 40
provider_endpoint (*select_ai.OCIGenAIProvider attribute*), 38
provider_endpoint (*select_ai.OpenAIProvider attribute*), 37
provider_endpoint (*select_ai.Provider attribute*), 32
provider_name (*select_ai.AnthropicProvider attribute*), 33
provider_name (*select_ai.AWSProvider attribute*), 35
provider_name (*select_ai.AzureProvider attribute*), 34
provider_name (*select_ai.CohereProvider attribute*), 36
provider_name (*select_ai.GoogleProvider attribute*), 39
provider_name (*select_ai.HuggingFaceProvider attribute*), 40
provider_name (*select_ai.OCIGenAIProvider attribute*), 38
provider_name (*select_ai.OpenAIProvider attribute*), 37
provider_name (*select_ai.Provider attribute*), 32

Q

qualified_name (*select_ai.AsyncProfile property*), 107
qualified_name (*select_ai.AsyncVectorIndex property*), 172
qualified_name (*select_ai.BaseProfile property*), 75
qualified_name (*select_ai.Profile property*), 79
qualified_name (*select_ai.VectorIndex property*), 161

R

RAG (*select_ai.agent.tool.ToolType attribute*), 362
recipient (*select_ai.agent.ToolParams attribute*), 216
record_count (*select_ai.SyntheticDataAttributes attribute*), 183
refresh_rate (*select_ai.OracleVectorIndexAttributes attribute*), 158
refresh_rate (*select_ai.VectorIndexAttributes attribute*), 157
region (*select_ai.AnthropicProvider attribute*), 33
region (*select_ai.AWSProvider attribute*), 35
region (*select_ai.AzureProvider attribute*), 34
region (*select_ai.CohereProvider attribute*), 36
region (*select_ai.GoogleProvider attribute*), 39
region (*select_ai.HuggingFaceProvider attribute*), 40
region (*select_ai.OCIGenAIProvider attribute*), 38
region (*select_ai.OpenAIProvider attribute*), 37
region (*select_ai.Provider attribute*), 32
remove_tag() (*select_ai.AsyncConversation method*), 147
remove_tag() (*select_ai.Conversation method*), 139
removeprefix() (*select_ai.action.Action method*), 360
removeprefix() (*select_ai.agent.tool.NotificationType method*), 370
removeprefix() (*select_ai.agent.tool.ToolType method*), 365
removeprefix() (*select_ai.summary.ChunkProcessingMethod method*), 196
removeprefix() (*select_ai.summary.ExtractivenessLevel method*), 203
removeprefix() (*select_ai.summary.Style method*), 208
removeprefix() (*select_ai.vector_index.VectorDBProvider method*), 380
removeprefix() (*select_ai.vector_index.VectorDistanceMetric method*), 375
removesuffix() (*select_ai.action.Action method*), 360
removesuffix() (*select_ai.agent.tool.NotificationType method*), 370
removesuffix() (*select_ai.agent.tool.ToolType method*), 365
removesuffix() (*select_ai.summary.ChunkProcessingMethod method*), 197
removesuffix() (*select_ai.summary.ExtractivenessLevel method*), 203
removesuffix() (*select_ai.summary.Style method*), 209
removesuffix() (*select_ai.vector_index.VectorDBProvider method*), 380
removesuffix() (*select_ai.vector_index.VectorDistanceMetric method*), 375
replace() (*select_ai.action.Action method*), 360
replace() (*select_ai.agent.tool.NotificationType method*), 370
replace() (*select_ai.agent.tool.ToolType method*), 365
replace() (*select_ai.summary.ChunkProcessingMethod method*), 197
replace() (*select_ai.summary.ExtractivenessLevel method*), 203
replace() (*select_ai.summary.Style method*), 209
replace() (*select_ai.vector_index.VectorDBProvider method*), 380
replace() (*select_ai.vector_index.VectorDistanceMetric method*), 375
result (*select_ai.agent.history.TaskHistoryEvent attribute*), 356
retention_days (*select_ai.ConversationAttributes attribute*), 137
revoke_access() (*select_ai.agent.AsyncTeam method*), 275
revoke_access() (*select_ai.agent.Team method*), 242
revoke_access() (*select_ai.AsyncProfile method*), 108
revoke_access() (*select_ai.AsyncVectorIndex method*), 172
revoke_access() (*select_ai.Profile method*), 79
revoke_access() (*select_ai.VectorIndex method*), 161
rfind() (*select_ai.action.Action method*), 360
rfind() (*select_ai.agent.tool.NotificationType method*), 370
rfind() (*select_ai.agent.tool.ToolType method*), 365
rfind() (*select_ai.summary.ChunkProcessingMethod method*), 197
rfind() (*select_ai.summary.ExtractivenessLevel method*), 203
rfind() (*select_ai.summary.Style method*), 209
rfind() (*select_ai.vector_index.VectorDBProvider method*), 380
rfind() (*select_ai.vector_index.VectorDistanceMetric method*), 375
rindex() (*select_ai.action.Action method*), 360
rindex() (*select_ai.agent.tool.NotificationType method*), 370
rindex() (*select_ai.agent.tool.ToolType method*), 365
rindex() (*select_ai.summary.ChunkProcessingMethod method*), 197
rindex() (*select_ai.summary.ExtractivenessLevel method*), 203
rindex() (*select_ai.summary.Style method*), 209
rindex() (*select_ai.vector_index.VectorDBProvider method*), 381
rindex() (*select_ai.vector_index.VectorDistanceMetric*

method), 376
rjust() (*select_ai.action.Action method*), 360
rjust() (*select_ai.agent.tool.NotificationType method*), 370
rjust() (*select_ai.agent.tool.ToolType method*), 365
rjust() (*select_ai.summary.ChunkProcessingMethod method*), 197
rjust() (*select_ai.summary.ExtractivenessLevel method*), 203
rjust() (*select_ai.summary.Style method*), 209
rjust() (*select_ai.vector_index.VectorDBProvider method*), 381
rjust() (*select_ai.vector_index.VectorDistanceMetric method*), 376
role (*select_ai.agent.AgentAttributes attribute*), 231
rpartition() (*select_ai.action.Action method*), 360
rpartition() (*select_ai.agent.tool.NotificationType method*), 371
rpartition() (*select_ai.agent.tool.ToolType method*), 365
rpartition() (*select_ai.summary.ChunkProcessingMethod method*), 197
rpartition() (*select_ai.summary.ExtractivenessLevel method*), 203
rpartition() (*select_ai.summary.Style method*), 209
rpartition() (*select_ai.vector_index.VectorDBProvider method*), 381
rpartition() (*select_ai.vector_index.VectorDistanceMetric method*), 376
rsplit() (*select_ai.action.Action method*), 360
rsplit() (*select_ai.agent.tool.NotificationType method*), 371
rsplit() (*select_ai.agent.tool.ToolType method*), 366
rsplit() (*select_ai.summary.ChunkProcessingMethod method*), 197
rsplit() (*select_ai.summary.ExtractivenessLevel method*), 203
rsplit() (*select_ai.summary.Style method*), 209
rsplit() (*select_ai.vector_index.VectorDBProvider method*), 381
rsplit() (*select_ai.vector_index.VectorDistanceMetric method*), 376
rstrip() (*select_ai.action.Action method*), 361
rstrip() (*select_ai.agent.tool.NotificationType method*), 371
rstrip() (*select_ai.agent.tool.ToolType method*), 366
rstrip() (*select_ai.summary.ChunkProcessingMethod method*), 197
rstrip() (*select_ai.summary.ExtractivenessLevel method*), 204
rstrip() (*select_ai.summary.Style method*), 209
rstrip() (*select_ai.vector_index.VectorDBProvider method*), 381
rstrip() (*select_ai.vector_index.VectorDistanceMetric method*), 376
run() (*select_ai.agent.AsyncTeam method*), 275
run() (*select_ai.agent.Team method*), 242
run_pipeline() (*select_ai.AsyncProfile method*), 108
run_sql() (*select_ai.async_profile.AsyncSession method*), 354
run_sql() (*select_ai.AsyncProfile method*), 108
run_sql() (*select_ai.Profile method*), 80
run_sql() (*select_ai.profile.Session method*), 353
run_tool() (*select_ai.agent.AsyncTool method*), 259
run_tool() (*select_ai.agent.Tool method*), 219
RUNSQL (*select_ai.action.Action attribute*), 357

S

sample_rows (*select_ai.SyntheticDataParams attribute*), 184
seed (*select_ai.ProfileAttributes attribute*), 71
select_ai.errors
    module, 349
SelectAIError, 352
sender (*select_ai.agent.ToolParams attribute*), 216
serial (*select_ai.ConversationPrompt attribute*), 143
Session (*class in select_ai.profile*), 352
set_attribute() (*select_ai.agent.Agent method*), 233
set_attribute() (*select_ai.agent.AsyncAgent method*), 270
set_attribute() (*select_ai.agent.AsyncTask method*), 266
set_attribute() (*select_ai.agent.AsyncTeam method*), 275
set_attribute() (*select_ai.agent.AsyncTool method*), 259
set_attribute() (*select_ai.agent.Task method*), 227
set_attribute() (*select_ai.agent.Team method*), 242
set_attribute() (*select_ai.agent.Tool method*), 220
set_attribute() (*select_ai.AsyncProfile method*), 108
set_attribute() (*select_ai.AsyncVectorIndex method*), 172
set_attribute() (*select_ai.Profile method*), 80
set_attribute() (*select_ai.ProfileAttributes method*), 71
set_attribute() (*select_ai.VectorIndex method*), 161
set_attributes() (*select_ai.agent.Agent method*), 233
set_attributes() (*select_ai.agent.AsyncAgent method*), 270
set_attributes() (*select_ai.agent.AsyncTask method*), 266
set_attributes() (*select_ai.agent.AsyncTeam method*), 275
set_attributes() (*select_ai.agent.AsyncTool method*), 259
set_attributes() (*select_ai.agent.Task method*), 228
set_attributes() (*select_ai.agent.Team method*), 242
set_attributes() (*select_ai.agent.Tool method*), 220

set_attributes() (*select_ai.AsyncConversation method*), 148
set_attributes() (*select_ai.AsyncProfile method*), 108
set_attributes() (*select_ai.AsyncVectorIndex method*), 172
set_attributes() (*select_ai.Conversation method*), 139
set_attributes() (*select_ai.Profile method*), 80
set_attributes() (*select_ai.VectorIndex method*), 161
should_include_provider_endpoint() (*select_ai.AnthropicProvider method*), 33
should_include_provider_endpoint() (*select_ai.AWSProvider method*), 35
should_include_provider_endpoint() (*select_ai.AzureProvider method*), 34
should_include_provider_endpoint() (*select_ai.CohereProvider method*), 36
should_include_provider_endpoint() (*select_ai.GoogleProvider method*), 39
should_include_provider_endpoint() (*select_ai.HuggingFaceProvider method*), 40
should_include_provider_endpoint() (*select_ai.OCIGenAIProvider method*), 38
should_include_provider_endpoint() (*select_ai.OpenAIProvider method*), 37
should_include_provider_endpoint() (*select_ai.Provider method*), 32
show_prompt() (*select_ai.async_profile.AsyncSession method*), 354
show_prompt() (*select_ai.AsyncProfile method*), 108
show_prompt() (*select_ai.Profile method*), 80
show_prompt() (*select_ai.profile.Session method*), 353
show_sql() (*select_ai.async_profile.AsyncSession method*), 355
show_sql() (*select_ai.AsyncProfile method*), 109
show_sql() (*select_ai.Profile method*), 80
show_sql() (*select_ai.profile.Session method*), 353
SHOWPROMPT (*select_ai.action.Action attribute*), 357
SHOWSQL (*select_ai.action.Action attribute*), 357
sid (*select_ai.ConversationPrompt attribute*), 143
similarity_threshold (*select_ai.OracleVectorIndexAttributes attribute*), 158
similarity_threshold (*select_ai.VectorIndexAttributes attribute*), 157
SLACK (*select_ai.agent.tool.NotificationType attribute*), 367
smtp_host (*select_ai.agent.ToolParams attribute*), 216
source_language (*select_ai.ProfileAttributes attribute*), 71
split() (*select_ai.action.Action method*), 361
split() (*select_ai.agent.tool.NotificationType method*), 371
split() (*select_ai.agent.tool.ToolType method*), 366
split() (*select_ai.summary.ChunkProcessingMethod method*), 197
split() (*select_ai.summary.ExtractivenessLevel method*), 204
split() (*select_ai.summary.Style method*), 209
split() (*select_ai.vector_index.VectorDBProvider method*), 381
split() (*select_ai.vector_index.VectorDistanceMetric method*), 376
splitlines() (*select_ai.action.Action method*), 361
splitlines() (*select_ai.agent.tool.NotificationType method*), 371
splitlines() (*select_ai.agent.tool.ToolType method*), 366
splitlines() (*select_ai.summary.ChunkProcessingMethod method*), 198
splitlines() (*select_ai.summary.ExtractivenessLevel method*), 204
splitlines() (*select_ai.summary.Style method*), 210
splitlines() (*select_ai.vector_index.VectorDBProvider method*), 381
splitlines() (*select_ai.vector_index.VectorDistanceMetric method*), 376
SQL (*select_ai.agent.tool.ToolType attribute*), 362
start_date (*select_ai.agent.history.TaskHistoryEvent attribute*), 356
start_date (*select_ai.agent.history.TeamHistoryEvent attribute*), 355
start_date (*select_ai.agent.history.ToolHistoryEvent attribute*), 356
startswith() (*select_ai.action.Action method*), 361
startswith() (*select_ai.agent.tool.NotificationType method*), 371
startswith() (*select_ai.agent.tool.ToolType method*), 366
startswith() (*select_ai.summary.ChunkProcessingMethod method*), 198
startswith() (*select_ai.summary.ExtractivenessLevel method*), 204
startswith() (*select_ai.summary.Style method*), 210
startswith() (*select_ai.vector_index.VectorDBProvider method*), 382
startswith() (*select_ai.vector_index.VectorDistanceMetric method*), 377
state (*select_ai.agent.history.TaskHistoryEvent attribute*), 356
state (*select_ai.agent.history.TeamHistoryEvent attribute*), 355
stop_tokens (*select_ai.ProfileAttributes attribute*), 71
streaming (*select_ai.ProfileAttributes attribute*), 71
strip() (*select_ai.action.Action method*), 361
strip() (*select_ai.agent.tool.NotificationType method*),

372
strip() (*select_ai.agent.tool.ToolType method*), 366
strip() (*select_ai.summary.ChunkProcessingMethod method*), 198
strip() (*select_ai.summary.ExtractivenessLevel method*), 204
strip() (*select_ai.summary.Style method*), 210
strip() (*select_ai.vector_index.VectorDBProvider method*), 382
strip() (*select_ai.vector_index.VectorDistanceMetric method*), 377
Style (*class in select_ai.summary*), 206
subject (*select_ai.agent.ToolParams attribute*), 216
SUMMARIZE (*select_ai.action.Action attribute*), 357
summarize() (*select_ai.AsyncProfile method*), 109
summarize() (*select_ai.Profile method*), 81
summary_style (*select_ai.summary.SummaryParams attribute*), 193
SummaryParams (*class in select_ai.summary*), 193
supervisor (*select_ai.agent.AgentAttributes attribute*), 231
supervisor_agent (*select_ai.agent.TeamAttributes attribute*), 235
supervisor_task (*select_ai.agent.TeamAttributes attribute*), 235
swapcase() (*select_ai.action.Action method*), 361
swapcase() (*select_ai.agent.tool.NotificationType method*), 372
swapcase() (*select_ai.agent.tool.ToolType method*), 367
swapcase() (*select_ai.summary.ChunkProcessingMethod method*), 198
swapcase() (*select_ai.summary.ExtractivenessLevel method*), 204
swapcase() (*select_ai.summary.Style method*), 210
swapcase() (*select_ai.vector_index.VectorDBProvider method*), 382
swapcase() (*select_ai.vector_index.VectorDistanceMetric method*), 377
SyntheticDataAttributes (*class in select_ai*), 183
SyntheticDataParams (*class in select_ai*), 184

T

table_statistics (*select_ai.SyntheticDataParams attribute*), 184
target_language (*select_ai.ProfileAttributes attribute*), 71
Task (*class in select_ai.agent*), 227
task_name (*select_ai.agent.history.TaskHistoryEvent attribute*), 356
task_name (*select_ai.agent.history.ToolHistoryEvent attribute*), 356
task_order (*select_ai.agent.history.TaskHistoryEvent attribute*), 356
task_order (*select_ai.agent.history.ToolHistoryEvent attribute*), 356
TaskAttributes (*class in select_ai.agent*), 226
TaskHistory (*class in select_ai.agent*), 250
TaskHistoryEvent (*class in select_ai.agent.history*), 355
Team (*class in select_ai.agent*), 240
team_exec_id (*select_ai.agent.history.TaskHistoryEvent attribute*), 356
team_exec_id (*select_ai.agent.history.TeamHistoryEvent attribute*), 355
team_exec_id (*select_ai.agent.history.ToolHistoryEvent attribute*), 356
team_name (*select_ai.agent.history.TaskHistoryEvent attribute*), 356
team_name (*select_ai.agent.history.TeamHistoryEvent attribute*), 355
TeamAttributes (*class in select_ai.agent*), 235
TeamHistory (*class in select_ai.agent*), 250
TeamHistoryEvent (*class in select_ai.agent.history*), 355
temperature (*select_ai.ProfileAttributes attribute*), 71
title (*select_ai.ConversationAttributes attribute*), 137
title() (*select_ai.action.Action method*), 361
title() (*select_ai.agent.tool.NotificationType method*), 372
title() (*select_ai.agent.tool.ToolType method*), 367
title() (*select_ai.summary.ChunkProcessingMethod method*), 198
title() (*select_ai.summary.ExtractivenessLevel method*), 204
title() (*select_ai.summary.Style method*), 210
title() (*select_ai.vector_index.VectorDBProvider method*), 382
title() (*select_ai.vector_index.VectorDistanceMetric method*), 377
Tool (*class in select_ai.agent*), 217
tool_inputs (*select_ai.agent.ToolAttributes attribute*), 215
tool_name (*select_ai.agent.history.ToolHistoryEvent attribute*), 356
tool_output (*select_ai.agent.history.ToolHistoryEvent attribute*), 356
tool_params (*select_ai.agent.ToolAttributes attribute*), 215
tool_type (*select_ai.agent.ToolAttributes attribute*), 215
ToolAttributes (*class in select_ai.agent*), 215
ToolHistory (*class in select_ai.agent*), 250
ToolHistoryEvent (*class in select_ai.agent.history*), 356
ToolParams (*class in select_ai.agent*), 215
tools (*select_ai.agent.TaskAttributes attribute*), 226
ToolType (*class in select_ai.agent.tool*), 362

TRANSLATE (*select_ai.action.Action attribute*), 357
translate() (*select_ai.action.Action method*), 361
translate() (*select_ai.agent.tool.NotificationType method*), 372
translate() (*select_ai.agent.tool.ToolType method*), 367
translate() (*select_ai.AsyncProfile method*), 109
translate() (*select_ai.Profile method*), 81
translate() (*select_ai.summary.ChunkProcessingMethod method*), 198
translate() (*select_ai.summary.ExtractivenessLevel method*), 204
translate() (*select_ai.summary.Style method*), 210
translate() (*select_ai.vector_index.VectorDBProvider method*), 382
translate() (*select_ai.vector_index.VectorDistanceMetric method*), 377

U

upper() (*select_ai.action.Action method*), 362
upper() (*select_ai.agent.tool.NotificationType method*), 372
upper() (*select_ai.agent.tool.ToolType method*), 367
upper() (*select_ai.summary.ChunkProcessingMethod method*), 198
upper() (*select_ai.summary.ExtractivenessLevel method*), 205
upper() (*select_ai.summary.Style method*), 210
upper() (*select_ai.vector_index.VectorDBProvider method*), 382
upper() (*select_ai.vector_index.VectorDistanceMetric method*), 377
user_prompt (*select_ai.SyntheticDataAttributes attribute*), 183

V

vector_db_credential_name (*select_ai.OracleVectorIndexAttributes attribute*), 158
vector_db_credential_name (*select_ai.VectorIndexAttributes attribute*), 157
vector_db_endpoint (*select_ai.OracleVectorIndexAttributes attribute*), 158
vector_db_endpoint (*select_ai.VectorIndexAttributes attribute*), 157
vector_db_provider (*select_ai.OracleVectorIndexAttributes attribute*), 158
vector_db_provider (*select_ai.VectorIndexAttributes attribute*), 157
vector_dimension (*select_ai.OracleVectorIndexAttributes attribute*), 158
vector_dimension (*select_ai.VectorIndexAttributes attribute*), 157
vector_distance_metric (*select_ai.OracleVectorIndexAttributes attribute*), 158
vector_distance_metric (*select_ai.VectorIndexAttributes attribute*), 157
vector_index_name (*select_ai.ProfileAttributes attribute*), 71
vector_table_name (*select_ai.OracleVectorIndexAttributes attribute*), 158
vector_table_name (*select_ai.VectorIndexAttributes attribute*), 157
VectorDBProvider (*class in select_ai.vector_index*), 377
VectorDistanceMetric (*class in select_ai.vector_index*), 372
VectorIndex (*class in select_ai*), 159
VectorIndexAttributes (*class in select_ai*), 155
VectorIndexNotFoundError, 352

W

WEBSEARCH (*select_ai.agent.tool.ToolType attribute*), 362
with_traceback() (*select_ai.errors.AgentAttributesEmptyError method*), 349
with_traceback() (*select_ai.errors.AgentNotFoundError method*), 350
with_traceback() (*select_ai.errors.AgentTaskAttributesEmptyError method*), 350
with_traceback() (*select_ai.errors.AgentTaskNotFoundError method*), 350
with_traceback() (*select_ai.errors.AgentTeamAttributesEmptyError method*), 350
with_traceback() (*select_ai.errors.AgentTeamNotFoundError method*), 350
with_traceback() (*select_ai.errors.AgentToolAttributesEmptyError method*), 350
with_traceback() (*select_ai.errors.AgentToolNotFoundError method*), 351
with_traceback() (*select_ai.errors.ConversationNotFoundError method*), 351

with_traceback() (*select_ai.errors.DatabaseNotConnectedError method*), 351
with_traceback() (*select_ai.errors.InvalidSQLError method*), 351
with_traceback() (*select_ai.errors.ProfileAttributesEmptyError method*), 351
with_traceback() (*select_ai.errors.ProfileExistsError method*), 352
with_traceback() (*select_ai.errors.ProfileNotFoundError method*), 352
with_traceback() (*select_ai.errors.SelectAIError method*), 352
with_traceback() (*select_ai.errors.VectorIndexNotFoundError method*), 352

Z

zfill() (*select_ai.action.Action method*), 362
zfill() (*select_ai.agent.tool.NotificationType method*), 372
zfill() (*select_ai.agent.tool.ToolType method*), 367
zfill() (*select_ai.summary.ChunkProcessingMethod method*), 198
zfill() (*select_ai.summary.ExtractivenessLevel method*), 205
zfill() (*select_ai.summary.Style method*), 210
zfill() (*select_ai.vector_index.VectorDBProvider method*), 382
zfill() (*select_ai.vector_index.VectorDistanceMetric method*), 377