Oracle Select AI for Python

Release 1.4.0

Oracle

Jun 11, 2026

CONTENTS

`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.

1.1.1 What you can build

Use `select_ai` to:

- Ask questions about database objects in natural language and generate SQL.
- Run generated SQL and return results as Python objects such as pandas data frames.
- Generate narrative answers, explanations, prompt previews, translations, and summaries.
- Use Retrieval Augmented Generation (RAG) with vector indexes over documents and object storage content.
- Create synthetic data for database tables.
- Build context-aware chat sessions with database-backed conversations.
- Register tools, tasks, agents, and teams for database-backed AI agent workflows.
- Use the optional `select-ai` command line interface for interactive chat and SQL workflows.
- Use synchronous APIs, asynchronous APIs, and connection pools in scripts, services, and web applications.

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*
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*

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. See *Command Line Interface*.

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
```

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. 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.

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

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(
    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"],
```

(continues on next page)

(continued from previous page)

```
        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)
```

(continues on next page)

(continued from previous page)

```
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`	Uses `api.openai.com`.

4.1.1 Examples

OCI Gen AI provider:

```
provider = select_ai.OCIGenAIProvider(
    region="us-chicago-1",
    oci_apiformat="GENERIC",
```

(continues on next page)

(continued from previous page)

```
    model="cohere.command-r-plus",
)
```

OpenAI provider:

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

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,
)
```

(continues on next page)

(continued from previous page)

```
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

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 = None*, *region: str | None = None*)

Anthropic specific attributes

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

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

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 = None*, *region: str | None = None*)

Cohere AI specific attributes

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

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'

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 = None*, *region: str | None = None*)

Google AI specific attributes

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 = None*, *region: str | None = None*)

HuggingFace specific attributes

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 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>", }`
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:

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.

5.1.1 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)
print("Created credential: ", credential["credential_name"])
```

output:

```
Created credential:  my_oci_ai_profile_key
```

5.1.2 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
    )
    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.

5.2.1 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
)
print("Deleted credential: my_oci_ai_profile_key")
```

output:

```
Deleted credential: my_oci_ai_profile_key
```

5.2.2 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
    )
    print("Deleted credential: my_oci_ai_profile_key")


asyncio.run(main())
```

output:

```
Deleted credential: my_oci_ai_profile_key
```

CHAPTER

SIX

PROFILE ATTRIBUTES

6.1 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.

6.1.1 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"},
    ],
)
```

6.1.2 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.
`vector_index_name`, `enable_sources`, `enable_source_offsets`, `enable_custom_source_uri`	Configures retrieval-augmented generation and source reporting for vector index workflows.

6.1.3 Object list examples

Grant access to every supported object in a schema:

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

Grant access to selected tables:

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

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"},
    ],
    enforce_object_list=True,
)
```

6.1.4 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='[";"]',
)
```

class select_ai.**ProfileAttributes**(*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: str | None = None*, *stop_tokens: str | None = None*, *streaming: str | None = None*, *temperature: float | 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.
- **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
- **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.
- **vector_index_name** (*str*) – Name of the vector index

CHAPTER

SEVEN

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*.

7.1 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.
- Update profile attributes when provider settings or object scope changes.
- Delete profiles that are no longer needed.

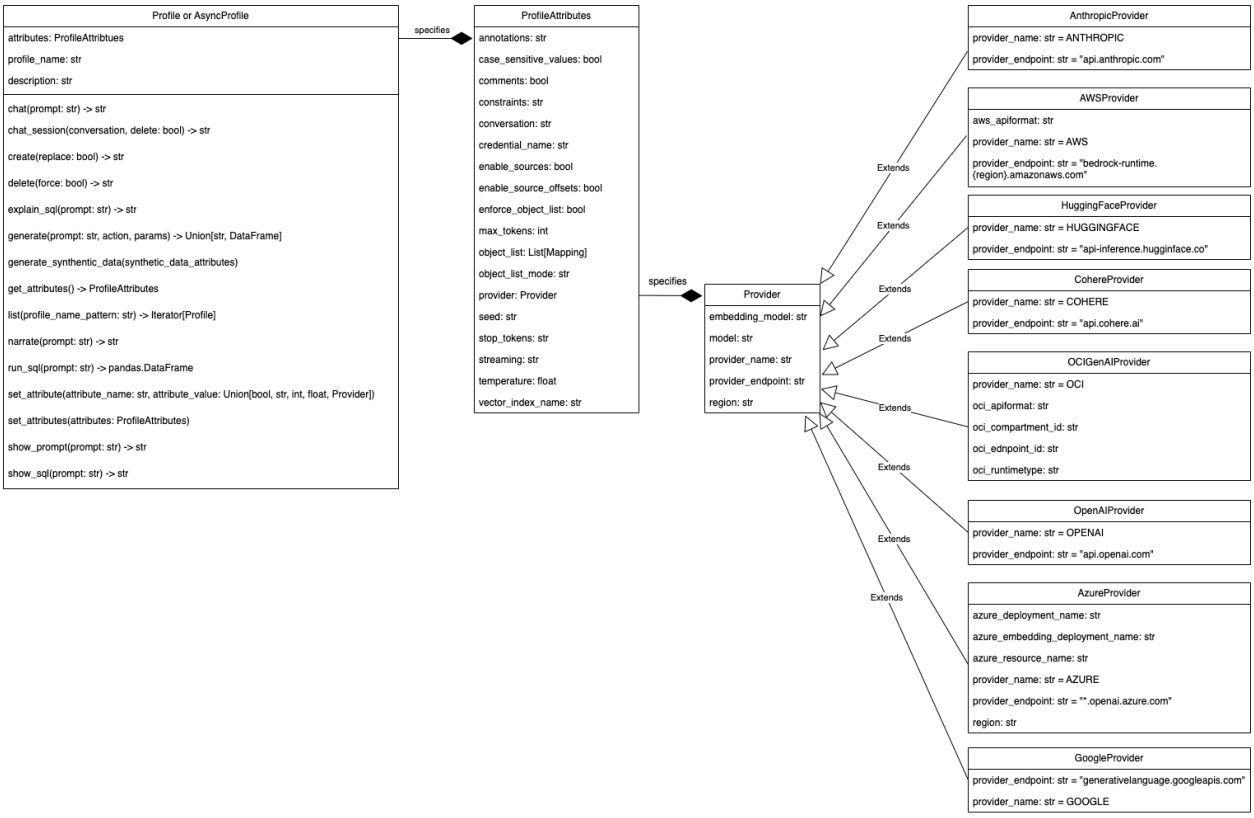
`replace=True` recreates a profile when a profile with the same name already exists. `merge=True` fetches the existing profile and updates it with the non-null attributes passed by the caller.

7.2 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.

7.2.1 Profile Object Model



7.2.2 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*)

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
- **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.

7.2.3 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, Action]*) – First element is the prompt and 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, Action]*) – First element is the prompt and 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*) → str | Generator[str, None, None]

Chat with the 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

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, Action]*) – First element is the prompt and 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

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

Explain 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

classmethod fetch(*profile_name: str*) → *Profile*

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

Parameters
profile_name (*str*) – The name of the AI profile

Returns
select_ai.Profile

Raises
ProfileNotFoundError

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

Perform AI translation using this profile

Parameters

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

classmethod list(*profile_name_pattern: str = '.*'*) → 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.

Returns
Iterator[Profile]

narrate(*prompt: str*, *params: Mapping = None*, *stream: bool = False*, *chunk_size: int = 8192*) → 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

- **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*, *params: 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

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 (*ProviderAttributes*) – Object specifying AI profile attributes

Returns
None

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

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*, *target_language: str*) → str | None

Translate a text using a source language and a target language

Parameters

- **text** (*str*) – Text to translate
- **source_language** (*str*) – Source language
- **target_language** (*str*) – Target language

Returns
str

7.2.4 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"
)
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}
```

7.2.5 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")
```

7.2.6 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,
    )
)
```

7.2.7 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)
```

7.2.8 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.
```

7.2.9 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"
```

7.2.10 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)
```

7.2.11 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)
```

7.2.12 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
```

7.2.13 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.
```

7.2.14 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`.

7.2.15 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.
```

7.2.16 Translate

```
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.translate(
    text="Thank you", source_language="en", target_language="de"
)
print(response)
```

output:

```
Danke
```

7.2.17 List profiles

Profile listing returns profiles visible to the connected database user. Instantiate `Profile` with one of the returned names to reuse the saved profile.

```
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()

# matches all the profiles
for fetched_profile in profile.list():
    print(fetched_profile.profile_name)
```

output:

```
ASYNC_OCI_AI_PROFILE
OCI_VECTOR_AI_PROFILE
ASYNC_OCI_VECTOR_AI_PROFILE
OCI_AI_PROFILE
```

CHAPTER

EIGHT

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*.

8.1 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.
- 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 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.

8.2.1 `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, Action]*) – First element is the prompt and 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, Action]*) – First element is the prompt and 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*) → str | AsyncGenerator[str, None]

Asynchronously chat with the LLM

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM 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, Action]*) – First element is the prompt and 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 explain_sql(*prompt: str*, *params: Mapping = None*, *stream: bool = False*, *chunk_size: int = 8192*)

Explain the generated SQL

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM 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*) → *AsyncProfile*

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

Parameters
profile_name (*str*)

Returns
select_ai.Profile

Raises
ProfileNotFoundError

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

Asynchronously perform AI translation using this profile

Parameters

- **prompt** (*str*) – Natural language prompt to translate
- **action** (*select_ai.profile.Action*)
- **params** – Parameters to include in the LLM request. For e.g. conversation_id for context-aware chats
- **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.provider.ProviderAttributes

Raises
ProfileNotFoundError

classmethod list(*profile_name_pattern: str = '.*'*) → 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.

Returns

Iterator[Profile]

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

Narrate the result of the SQL

Parameters

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

Returns

str

async run_pipeline(*prompt_specifications: List[Tuple[str, Action]]*, *continue_on_error: bool = False*) → List[str | DataFrame]

Send Multiple prompts in a single roundtrip to the Database

Parameters

- **prompt_specifications** (*List[Tuple[str, 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

Returns

List[Union[str, pandas.DataFrame]]

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

Explain the generated SQL

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM 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*)

Show the prompt sent to LLM

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM 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*)

Show the generated SQL

Parameters

- **prompt** (*str*) – Natural language prompt
- **params** – Parameters to include in the LLM 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*, *target_language: str*) → str | None

Translate a text using a source language and a target language

Parameters

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

- **source_language** (*str*) – Source language
- **target_language** (*str*) – Target language

Returns
str

8.2.2 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}
```

8.2.3 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"
)
```

8.2.4 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,
    )
)
```

8.2.5 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,
)
```

8.2.6 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.
```

8.2.7 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
```

8.2.8 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"
```

8.2.9 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)
```

8.2.10 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():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_profile = await select_ai.AsyncProfile(
        profile_name="async_oci_ai_profile",
    )
    sql_tasks = [
        async_profile.show_sql(prompt="How many customers?"),
        async_profile.run_sql(prompt="How many promotions?"),
        async_profile.explain_sql(prompt="How many promotions?"),
    ]

    # Collect results from multiple asynchronous tasks
    for sql_task in asyncio.as_completed(sql_tasks):
        result = await sql_task
        print(result)


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";
```

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"`.
```

(continues on next page)

(continued from previous page)

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

   PROMOTION_COUNT
0              503
```

8.2.11 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)
...
...
```

8.2.12 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`.

8.2.13 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.
```

8.2.14 Translate

```
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.translate(
        text="Thank you",
        source_language="en",
        target_language="de",
    )
    print(response)


asyncio.run(main())
```

output:

```
Danke
```

8.2.15 Async pipeline

```
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 = [
        ("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"),
    ]

    # 1. Multiple prompts are sent in a single roundtrip to the Database
    # 2. Results are returned as soon as Database has executed all prompts
    # 3. Application doesn't have to wait on one response before sending
    #    the next prompts
    # 4. Fewer round trips and database is kept busy
    # 5. Efficient network usage
    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
```

(continues on next page)

(continued from previous page)

```
↪"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
```
```

8.2.16 List profiles asynchronously

Profile listing returns profiles visible to the connected database user. The async list API returns 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_profile = await select_ai.AsyncProfile()
    # matches the start of string
    async for fetched_profile in async_profile.list(
        profile_name_pattern="^oci"
    ):
        print(fetched_profile.profile_name)


asyncio.run(main())
```

output:

```
OCI_VECTOR_AI_PROFILE
OCI_AI_PROFILE
```

CHAPTER

NINE

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.

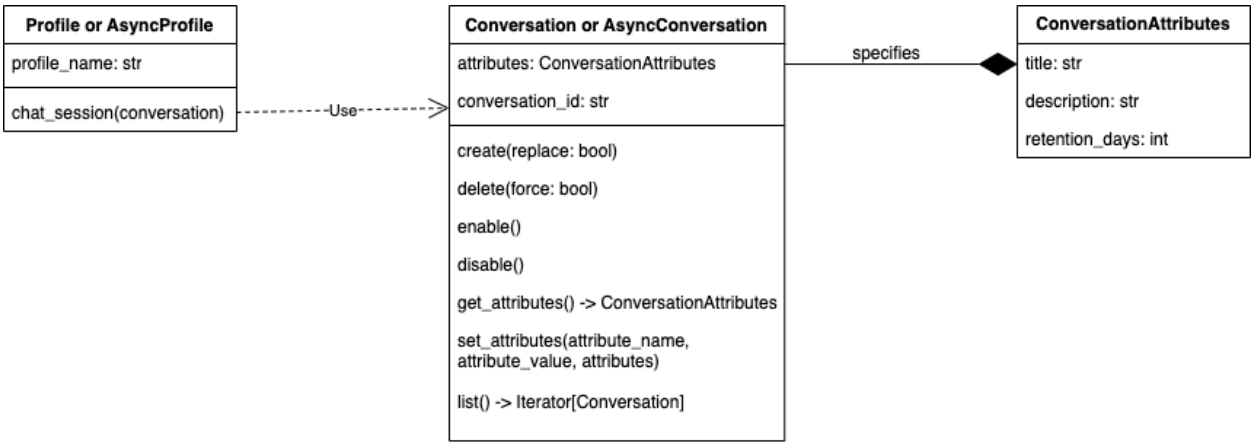
A conversation is stored in the database and identified by a `conversation_id`. Pass that conversation to `Profile.chat_session()` or `AsyncProfile.chat_session()` when follow-up prompts should use prior prompts as context. This is useful for chat workflows where the user asks a question, then asks follow-up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

9.1 Conversation Object model



9.2 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

9.3 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

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

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]

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.
`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.

9.3.1 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
```

9.3.2 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:
...
...
```

9.3.3 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)
```

9.3.4 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
```

9.4 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 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 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 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(...)`
`with profile.chat_session(...)`	`async with async_profile.chat_session(...)`

9.4.1 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:
...
...
```

9.4.2 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

TEN

VECTOR INDEX

10.1 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.

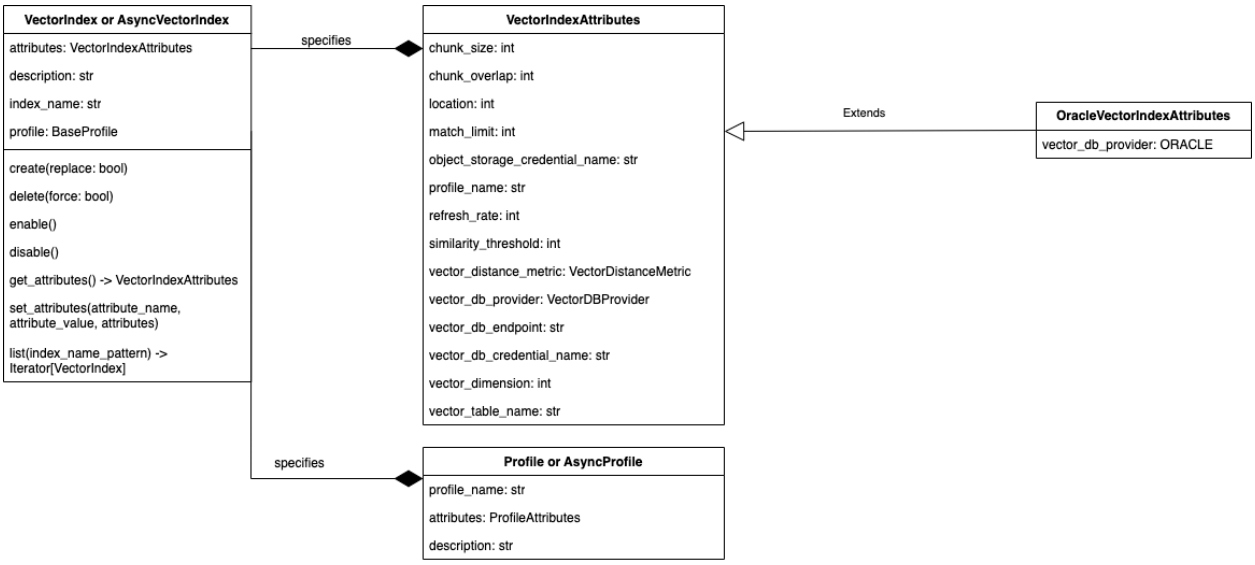
Before creating a vector index, make sure the database user has:

- 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.

10.2 `VectorIndex` Object Model


VectorIndex or AsyncVectorIndex
attributes: VectorIndexAttributes
description: str
index_name: str
profile: BaseProfile
create(replace: bool)
delete(force: bool)
enable()
disable()
get_attributes() -> VectorIndexAttributes
set_attributes(attribute_name, attribute_value, attributes)
list(index_name_pattern) -> Iterator[VectorIndex]
specifies
VectorIndexAttributes
chunk_size: int
chunk_overlap: int
location: int
match_limit: int
object_storage_credential_name: str
profile_name: str
refresh_rate: int
similarity_threshold: int
vector_distance_metric: VectorDistanceMetric
vector_db_provider: VectorDBProvider
vector_db_endpoint: str
vector_db_credential_name: str
vector_dimension: int
vector_table_name: str
Extends
OracleVectorIndexAttributes
vector_db_provider: ORACLE
specifies
Profile or AsyncProfile
profile_name: str
attributes: ProfileAttributes
description: str

10.3 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** (*VectorDistanceMetric*) – Specifies the type of distance calculation used to compare vectors in a database
- **vector_db_provider** (*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

10.3.1 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

10.4 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*)

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)` → *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

`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

`classmethod list(index_name_pattern: str = '.*')` → 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.

Returns
Iterator[VectorIndex]

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)`	Build a `VectorIndex` proxy from database metadata, including attributes and the linked profile when it still exists.
`list(index_name_pa` `*")`	Iterate over vector indexes visible to the current user. 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.

10.4.1 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()`.

```
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)
# Configure an AI provider with an embedding model
# of your choice
provider = select_ai.OCIGenAIProvider(
    region="us-chicago-1",
    oci_apiformat="GENERIC",
    embedding_model="cohere.embed-english-v3.0",
)

# Create an AI profile to use the Vector index with
profile_attributes = select_ai.ProfileAttributes(
    credential_name="my_oci_ai_profile_key",
    provider=provider,
)
profile = select_ai.Profile(
    profile_name="oci_vector_ai_profile",
    attributes=profile_attributes,
    description="MY OCI AI Profile",
    replace=True,
)

# Specify objects to create an embedding for. In this example,
# the objects reside in ObjectStore and the vector database is
# Oracle
vector_index_attributes = select_ai.OracleVectorIndexAttributes(
    location="https://objectstorage.us-ashburn-1.oraclecloud.com/n/dwcsdev/b/conda-
↪environment/o/tenant1-pdb3/graph",
    object_storage_credential_name="my_oci_ai_profile_key",
)

# Create a Vector index object
vector_index = select_ai.VectorIndex(
    index_name="test_vector_index",
    attributes=vector_index_attributes,
    description="Test vector index",
    profile=profile,
)
vector_index.create(replace=True, wait_for_completion=True)
print("Created vector index: test_vector_index")
```

output:

```
Created vector index: test_vector_index
```

10.4.2 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)
```

output:

```
Vector index TEST_VECTOR_INDEX
Vector index profile Profile(profile_name=oci_vector_ai_profile,␣
↪attributes=ProfileAttributes(annotations=None, case_sensitive_values=None,␣
↪comments=None, constraints=None, conversation=None, credential_name='my_oci_ai_profile_
↪key', enable_sources=None, enable_source_offsets=None, enforce_object_list=None, max_
↪tokens=1024, object_list=None, 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='test_vector_index'),␣
↪description=None)
```

10.4.3 Fetch vector index

You can fetch the vector index attributes and associated AI profile using the class method `VectorIndex.fetch(index_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.

```
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())
```

output:

```
OracleVectorIndexAttributes(chunk_size=1024, chunk_overlap=128, location='https://
↪objectstorage.us-ashburn-1.oraclecloud.com/n/dwcsdev/b/conda-environment/o/tenant1-
↪pdb3/graph', match_limit=5, object_storage_credential_name='my_oci_ai_profile_key',␣
↪profile_name='oci_vector_ai_profile', refresh_rate=1450, similarity_threshold=0.5,␣
↪vector_distance_metric='COSINE', vector_db_endpoint=None, vector_db_credential_
↪name=None, vector_db_provider=<VectorDBProvider.ORACLE: 'oracle'>, vector_
↪dimension=None, vector_table_name=None, pipeline_name='TEST_VECTOR_INDEX$VECPIPELINE')

Profile(profile_name=oci_vector_ai_profile,␣
↪attributes=ProfileAttributes(annotations=None, case_sensitive_values=None,␣
↪comments=None, constraints=None, conversation=None, credential_name='my_oci_ai_profile_
↪key', enable_custom_source_uri=None, enable_sources=None, enable_source_offsets=None,␣
↪enforce_object_list=None, max_tokens=1024, object_list=None, object_list_mode=None,␣
↪provider=OCIGenAIProvider(embedding_model='cohere.embed-english-v3.0', 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='test_
↪vector_index'), description=MY OCI AI Profile)
```

10.4.4 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)
```

output:

```
OracleVectorIndexAttributes(chunk_size=1024, chunk_overlap=128, location='https://
↪objectstorage.us-ashburn-1.oraclecloud.com/n/dwcsdev/b/conda-environment/o/tenant1-
↪pdb3/graph', match_limit=5, object_storage_credential_name='my_oci_ai_profile_key',␣
↪profile_name='oci_vector_ai_profile', refresh_rate=1450, similarity_threshold=0.5,␣
↪vector_distance_metric='COSINE', vector_db_endpoint=None, vector_db_credential_
↪name=None, vector_db_provider=<VectorDBProvider.ORACLE: 'oracle'>, vector_
↪dimension=None, vector_table_name=None, pipeline_name='TEST_VECTOR_INDEX$VECPIPELINE')
```

10.4.5 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.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_vector_ai_profile")
r = profile.narrate("list the conda environments in my object store")
print(r)
```

output:

```
The conda environments in your object store are:
 1. fccenv
 2. myrenv
 3. fully-loaded-mlenv
 4. graphenv

 These environments are listed in the provided data as separate JSON documents, each␣
↪containing information about a specific conda environment.

 Sources:
   - fccenv-manifest.json (https://objectstorage.us-ashburn-1.oraclecloud.com/n/dwcsdev/
↪b/conda-environment/o/tenant1-pdb3/graph/fccenv-manifest.json)
   - myrenv-manifest.json (https://objectstorage.us-ashburn-1.oraclecloud.com/n/dwcsdev/
↪b/conda-environment/o/tenant1-pdb3/graph/myrenv-manifest.json)
   - fully-loaded-mlenv-manifest.json (https://objectstorage.us-ashburn-1.oraclecloud.
↪com/n/dwcsdev/b/conda-environment/o/tenant1-pdb3/graph/fully-loaded-mlenv-manifest.
↪json)
   - graphenv-manifest.json (https://objectstorage.us-ashburn-1.oraclecloud.com/n/
↪dwcsdev/b/conda-environment/o/tenant1-pdb3/graph/graphenv-manifest.json)
```

10.4.6 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
```

10.5 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*)

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*) → *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

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

classmethod list(*index_name_pattern: str = '.*'*) → 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.

Returns
AsyncGenerator[VectorIndex]

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(...)`

10.5.1 Async create 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)

    provider = select_ai.OCIGenAIProvider(
        region="us-chicago-1",
        oci_apiformat="GENERIC",
        embedding_model="cohere.embed-english-v3.0",
    )
    profile_attributes = select_ai.ProfileAttributes(
        credential_name="my_oci_ai_profile_key",
        provider=provider,
    )
    async_profile = await select_ai.AsyncProfile(
        profile_name="async_oci_vector_ai_profile",
        attributes=profile_attributes,
        description="MY OCI AI Profile",
        replace=True,
    )

    vector_index_attributes = select_ai.OracleVectorIndexAttributes(
        location="https://objectstorage.us-ashburn-1.oraclecloud.com/n/dwcsdev/b/conda-
↪environment/o/tenant1-pdb3/graph",
        object_storage_credential_name="my_oci_ai_profile_key",
    )

    async_vector_index = select_ai.AsyncVectorIndex(
        index_name="test_vector_index",
        attributes=vector_index_attributes,
        description="Vector index for conda environments",
        profile=async_profile,
    )
    await async_vector_index.create(replace=True, wait_for_completion=True)
    print("Created vector index: test_vector_index")


asyncio.run(main())
```

output:

```
created vector index: test_vector_index
```

10.5.2 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())
```

output:

```
Vector index TEST_VECTOR_INDEX
Vector index profile AsyncProfile(profile_name=oci_vector_ai_profile,␣
↪attributes=ProfileAttributes(annotations=None, case_sensitive_values=None,␣
↪comments=None, constraints=None, conversation=None, credential_name='my_oci_ai_profile_
↪key', enable_sources=None, enable_source_offsets=None, enforce_object_list=None, max_
↪tokens=1024, object_list=None, 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='test_vector_index'),␣
↪description=None)
```

10.5.3 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())
```

output:

```
OracleVectorIndexAttributes(chunk_size=1024, chunk_overlap=128, location='https://
↪objectstorage.us-ashburn-1.oraclecloud.com/n/dwcsdev/b/conda-environment/o/tenant1-
↪pdb3/graph', match_limit=5, object_storage_credential_name='my_oci_ai_profile_key',␣
↪profile_name='oci_vector_ai_profile', refresh_rate=1450, similarity_threshold=0.5,␣
↪vector_distance_metric='COSINE', vector_db_endpoint=None, vector_db_credential_
↪name=None, vector_db_provider=<VectorDBProvider.ORACLE: 'oracle'>, vector_
↪dimension=None, vector_table_name=None, pipeline_name='TEST_VECTOR_INDEX$VECPIPELINE')

AsyncProfile(profile_name=oci_vector_ai_profile,␣
↪attributes=ProfileAttributes(annotations=None, case_sensitive_values=None,␣
↪comments=None, constraints=None, conversation=None, credential_name='my_oci_ai_profile_
↪key', enable_custom_source_uri=None, enable_sources=None, enable_source_offsets=None,␣
↪enforce_object_list=None, max_tokens=1024, object_list=None, object_list_mode=None,␣
↪provider=OCIGenAIProvider(embedding_model='cohere.embed-english-v3.0', 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='test_
↪vector_index'), description=MY OCI AI Profile)
```

10.5.4 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())
```

output:

```
OracleVectorIndexAttributes(chunk_size=1024, chunk_overlap=128, location='https://
↪objectstorage.us-ashburn-1.oraclecloud.com/n/dwcsdev/b/conda-environment/o/tenant1-
↪pdb3/graph', match_limit=5, object_storage_credential_name='my_oci_ai_profile_key',␣
↪profile_name='oci_vector_ai_profile', refresh_rate=1450, similarity_threshold=0.5,␣
↪vector_distance_metric='COSINE', vector_db_endpoint=None, vector_db_credential_
↪name=None, vector_db_provider=<VectorDBProvider.ORACLE: 'oracle'>, vector_
↪dimension=None, vector_table_name=None, pipeline_name='TEST_VECTOR_INDEX$VECPIPELINE')
```

10.5.5 Async RAG using 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")


async def main():
    await select_ai.async_connect(user=user, password=password, dsn=dsn)
    async_profile = await select_ai.AsyncProfile(
        profile_name="async_oci_vector_ai_profile"
    )
    r = await async_profile.narrate(
        "list the conda environments in my object store"
    )
    print(r)


asyncio.run(main())
```

output:

```
The conda environments in your object store are:
1. fccenv
2. myrenv
3. fully-loaded-mlenv
4. graphenv

These environments are listed in the provided data as separate JSON documents, each␣
↪containing information about a specific conda environment.

Sources:
  - fccenv-manifest.json (https://objectstorage.us-ashburn-1.oraclecloud.com/n/dwcsdev/b/
↪conda-environment/o/tenant1-pdb3/graph/fccenv-manifest.json)
  - myrenv-manifest.json (https://objectstorage.us-ashburn-1.oraclecloud.com/n/dwcsdev/b/
↪conda-environment/o/tenant1-pdb3/graph/myrenv-manifest.json)
  - fully-loaded-mlenv-manifest.json (https://objectstorage.us-ashburn-1.oraclecloud.com/
↪n/dwcsdev/b/conda-environment/o/tenant1-pdb3/graph/fully-loaded-mlenv-manifest.json)
  - graphenv-manifest.json (https://objectstorage.us-ashburn-1.oraclecloud.com/n/dwcsdev/
↪b/conda-environment/o/tenant1-pdb3/graph/graphenv-manifest.json)
```

CHAPTER

ELEVEN

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.

11.1 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.

11.2 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`.

11.3 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.

11.3.1 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"

11.3.2 SyntheticDataParams

class select_ai.**SyntheticDataParams**(*sample_rows: int | None = None*, *table_statistics: bool | None = False*, *priority: str | None = 'HIGH'*, *comments: bool | None = False*)

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. Default value is False
- **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
- **comments** (*bool*) – Enable or disable sending comments to the LLM to guide data generation. Default value is False

11.3.3 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
```

11.3.4 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

TWELVE

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.

12.1 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)
```

12.2 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)
```

12.3 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)
```

12.4 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,
)
```

12.5 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)
```

12.6 Validation

`summarize` requires exactly one of `content` or `location_uri`. Passing both, or passing neither, raises an error.

12.6.1 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

12.6.2 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

12.6.3 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

12.6.4 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.

CHAPTER

THIRTEEN

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.

13.1 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.

13.1.1 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`

13.1.2 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`.

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

class select_ai.agent.**Tool**(*tool_name: str | None = None*, *description: str | None = None*, *attributes: ToolAttributes | None = None*)

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.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.

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]

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

13.1.3 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'>)
```

13.1.4 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
```

13.2 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.

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

13.2.1 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)
```

13.2.2 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
```

13.3 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*)

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

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

13.3.1 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)
```

13.4 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'*)

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.

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.

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

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]

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

13.4.1 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)
```

13.4.2 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`.

13.4.3 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)
```

13.5 AI agent examples

13.5.1 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, params={"conversation_id": conversation_id}
    )
    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

FOURTEEN

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 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.

14.1 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.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 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 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

14.1.1 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'>)
```

14.1.2 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
```

14.2 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

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 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)
```

14.2.2 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
```

14.3 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

14.3.1 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)
```

14.3.2 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
```

14.4 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 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 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 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

14.4.1 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)
```

14.4.2 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`.

14.4.3 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)
```

14.4.4 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
```

14.5 Async AI agent examples

14.5.1 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

FIFTEEN

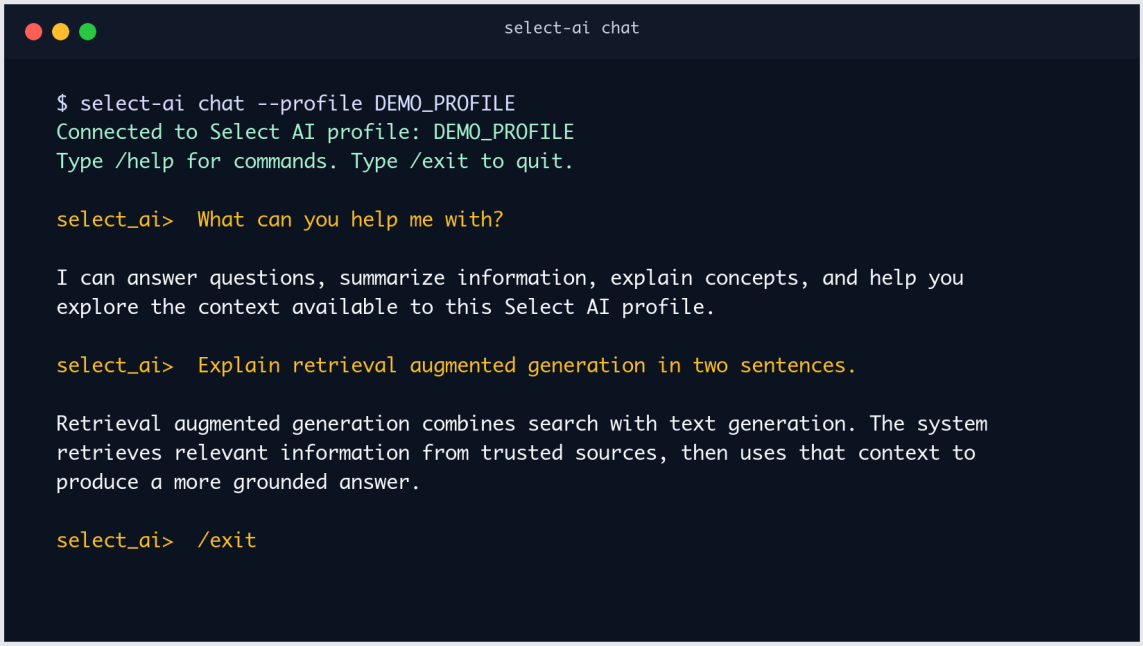
COMMAND LINE INTERFACE

15.1 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. RAG is supported when the selected profile is already configured with a vector index. Additional CLI options and workflows will be added in upcoming releases as the CLI evolves.



The package provides an optional `select-ai` command line tool. Install the CLI extra to use it:

```
pip install 'select_ai[cli]'
```

Use `select-ai --help` to view the available command groups and `select-ai <command> --help` to view options for a specific command.

Set the database connection details as environment variables, or pass them as command line options:

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

15.1.1 Connection options

All CLI commands accept the same database connection options:

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`

If `--password` and `SELECT_AI_PASSWORD` are not set, the CLI prompts for the database password. Wallet options are optional and are only needed for wallet-based database connections.

15.1.2 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.

15.1.3 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.

15.1.4 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 \
    --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.

15.1.5 Command summary

Table 3: 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.

CHAPTER

SIXTEEN

WEB FRAMEWORKS

16.1 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.

16.1.1 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.

16.1.2 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.

16.1.3 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()
```

(continues on next page)

(continued from previous page)

```
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 creates the `Profile` proxy inside each handler. The proxy is lightweight; the database connection is acquired from the pool only when the profile method calls the database.

16.1.4 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,
```

(continues on next page)

(continued from previous page)

```
        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.

16.1.5 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)})
```

(continues on next page)

(continued from previous page)

```
    @atexit.register
    def close_select_ai_pool():
        if select_ai.is_connected():
            select_ai.disconnect()

    return app
```

16.1.6 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")),
            )
```

```
# 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.

16.1.7 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.

16.1.8 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.

```
select_ai.create_pool(
    user=user,
    password=password,
    dsn=dsn,
    min_size=2,
    max_size=8,
    increment=2,
    wait_timeout=10,
)
```

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.

16.1.9 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

SEVENTEEN

CONCURRENT PROMPT PROCESSING

17.1 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.

17.1.1 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 sent in a single database round trip.
Async queue	*async_queue_workers.py*	Use async queue workers for long-running async services or background prompt processors.

17.1.2 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
```

(continues on next page)

(continued from previous page)

```
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.

17.1.3 sync_thread_pool.py

This recipe uses `ThreadPoolExecutor` and `as_completed()`. Results are printed in the order they finish.

```
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)


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]
```

(continues on next page)

(continued from previous page)

```
        for future in as_completed(futures):
            prompt, sql = future.result()
            print(f"\nPrompt: {prompt}")
            print(sql)
finally:
    select_ai.disconnect()
```

17.1.4 sync_ordered_results.py

This recipe uses `ThreadPoolExecutor.map()`. Prompts run concurrently, but results are printed in the same order as the input list.

```
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)


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:
```

(continues on next page)

(continued from previous page)

```
        results = executor.map(show_sql, prompts)

        for prompt, sql in zip(prompts, results):
            print(f"\nPrompt: {prompt}")
            print(sql)
finally:
    select_ai.disconnect()
```

17.1.5 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.

```
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()
```

(continues on next page)

(continued from previous page)

```
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()
```

17.1.6 async_gather.py

This recipe uses `asyncio.gather()`. Prompts run concurrently, and results are returned in the same order as the input task list.

```
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"))
```

(continues on next page)

(continued from previous page)

```
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)


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())
```

17.1.7 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.

```
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")

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


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())
```

17.1.8 async_pipeline.py

This recipe uses `AsyncProfile.run_pipeline()` to send multiple prompt/action pairs in one database round trip. This is different from Python task concurrency: the application submits a batch and receives the batch results when the pipeline completes.

```
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),
]


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())
```

17.1.9 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.

```
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()


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:
```

(continues on next page)

(continued from previous page)

```
        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())
```

17.1.10 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`.

INDEX

E

F

G

H

I

L

N

O

P

R

S

T

V