Select AI for Python

Release 1.3.0

Oracle

Mar 24, 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` supports both synchronous and concurrent(asynchronous) programming styles.

The Select AI Python API supports Python versions 3.9, 3.10, 3.11, 3.12 and 3.13.

1.2 Installing `select_ai`

1.2.1 Installation requirements

To use `select_ai` you need:

- Python 3.9, 3.10, 3.11, 3.12, 3.13 or 3.14

> **Warning**
>
> For async APIs, use Python 3.11 or higher. Python 3.11 stabilized the async event loop management and introduced better-structured APIs

- `python-oracledb` - This package is automatically installed as a dependency requirement
- `pandas` - This package is automatically installed as a dependency requirement

1.2.2 `select_ai` installation

`select_ai` can be installed from Python's package repository PyPI using pip.

1. Install Python 3 if it is not already available. Use any version from Python 3.9 through 3.14.
2. Install `select_ai`:

```
python3 -m pip install select_ai --upgrade --user
```

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

```
python3 -m pip install select_ai --upgrade --user --proxy=http://proxy.example.
↪com:80
```

4. Create a file `select_ai_connection_test.py` such as:

```
import select_ai

user = "<your_db_user>"
password = "<your_db_password>"
dsn = "<your_db_dsn>"
select_ai.connect(user=user, password=password, dsn=dsn)
print("Connected to the Database")
```

5. Run `select_ai_connection_test.py`

```
python3 select_ai_connection_test.py
```

Enter the database password when prompted and message will be shown:

```
Connected to the Database
```

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.

1.3.1 Synchronous connection

To connect to an Oracle Database synchronously, use `select_ai.connect()` method 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)
```

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)
```

1.3.3 Connection Pool

You can create a connection pool using the `select_ai.create_pool` and `select_ai.create_pool_async` methods

```
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,
```

(continues on next page)

(continued from previous page)

```
    max_size=10,
    increment=5
)
```

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

Note

For m-TLS (wallet) based connection, additional parameters like `wallet_location`, `wallet_password`, `https_proxy`, `https_proxy_port` can be passed to the `connect` and `async_connect` methods

CHAPTER

TWO

ACTIONS

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

2.1 Supported Actions

Following list of actions can be performed using `select_ai`

Table 1: Select AI Actions

Actions	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`	Explain the generated SQL query
narrate	`select_ai.Action.NARRATE`	Explains the output of the query in natural language, making the results accessible to users without deep technical expertise.
runsql	`select_ai.Action.RUNSQL`	Executes a SQL query generated from a natural language prompt. This is the default action.
showprompt	`select_ai.Action.SHOWPROMPT`	Show the details of the prompt sent to LLM
showsql	`select_ai.Action.SHOWSQL`	Displays the generated SQL statement without executing it.
summarize	`select_ai.Action.SUMMARIZE`	Generate summary of your large texts
feedback	`select_ai.Action.FEEDBACK`	Provide feedback to improve accuracy of the generated SQL
translate	`select_ai.Action.TRANSLATE`	Translate text from one language to another

CHAPTER

THREE

PRIVILEGES

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`

Note

All sample scripts in this documentation read Oracle database connection details from the environment. Create a dotenv file `.env`, export the 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 admin and run the method `select_ai.grant_privileges(users=select_ai_user)` to grant relevant select ai privileges to other users

```
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_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:

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

CHAPTER

FOUR

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

4.1 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

4.2 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

4.3 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

4.4 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

4.5 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

4.6 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

4.7 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'

4.8 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

4.9 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

4.10 Enable AI service provider

4.10.1 Enable using Sync API

This method adds ACL allowing database users to invoke AI provider's HTTP endpoint

```
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>
```

4.10.2 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>
```

4.11 Disable AI service provider

This method removes ACL blocking database users to invoke AI provider's HTTP endpoint

4.11.1 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>
```

4.11.2 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

Credential object securely stores API key from your AI provider for use by Oracle Database. The following table shows AI Provider and corresponding credential object format

Table 1: AI Provider and expected credential format

AI Provider	Credential format
Anthropic	`{"username": "anthropic", "password": "sk-xxx"}`
HuggingFace	`{"username": "hf", "password": "hf_xxx"}`
OCI Gen AI	`{"user_ocid": "", "tenancy_ocid": "", "private_key": "", "fingerprint": ""}`
OpenAI	`{"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:

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
```

CHAPTER

SIX

PROFILE ATTRIBUTES

6.1 ProfileAttributes

This class defines attributes to manage and configure the behavior of the AI profile. The `ProfileAttributes` objects are created by `select_ai.ProfileAttributes()`.

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

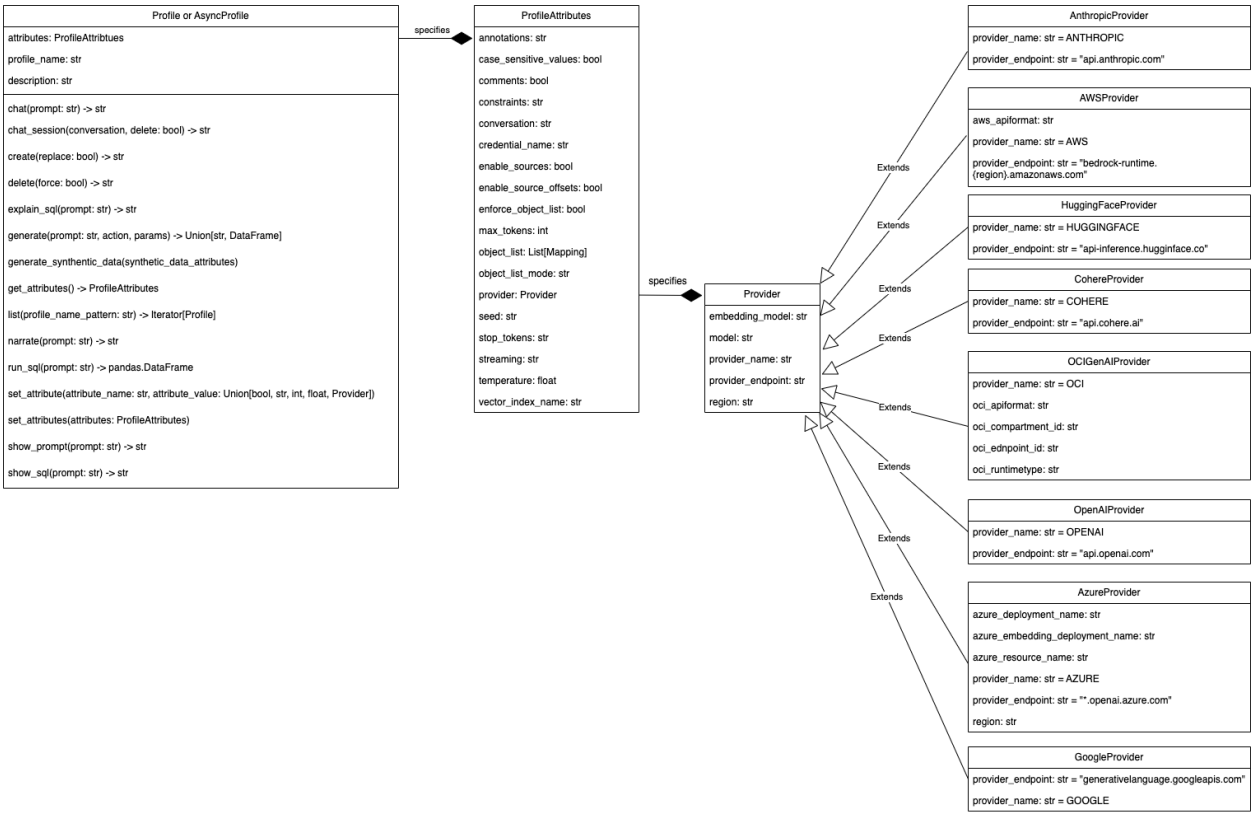
SEVEN

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()`

7.1 Profile Object Model



7.2 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.

7.3 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*) → str

Chat with the LLM

Parameters

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

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*) → str

Explain the generated SQL

Parameters

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

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*) → DataFrame | str | 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

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*) → str

Narrate the result of the SQL

Parameters

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

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*) → str

Show the prompt sent to LLM

Parameters

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

Returns
str

show_sql(*prompt: str*, *params: Mapping = None*) → str

Show the generated SQL

Parameters

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

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

7.4 Create Profile

```
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,
                              region='us-chicago-1',
```

(continues on next page)

(continued from previous page)

```
                              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}
```

7.5 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.
```

7.6 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"
```

7.7 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
```

7.8 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.
```

7.9 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.
```

7.10 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
```

7.11 List profiles

```
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
```

7.12 Async Profile

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

7.12.1 `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*) → str

Asynchronously chat with the LLM

Parameters

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

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*)

Explain the generated SQL

Parameters

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

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*) → DataFrame | str | 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

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*) → str

Narrate the result of the SQL

Parameters

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

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*)

Show the prompt sent to LLM

Parameters

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

Returns
str

async show_sql(*prompt*, *params: Mapping = None*)

Show the generated SQL

Parameters

- **prompt** (*str*) – Natural language prompt

- **params** – Parameters to include in the LLM request

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

7.12.2 Async Profile creation

```
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,
 'max_tokens': '1024',
 'object_list': '[{"owner":"SH"}]',
```

(continues on next page)

(continued from previous page)

```
'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}
```

7.12.3 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.
```

7.12.4 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
```

7.12.5 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"
```

7.12.6 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
```

7.12.7 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)
...
...
```

7.12.8 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.
```

7.12.9 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
```

7.12.10 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
```
```

7.12.11 List profiles asynchronously

```
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
```

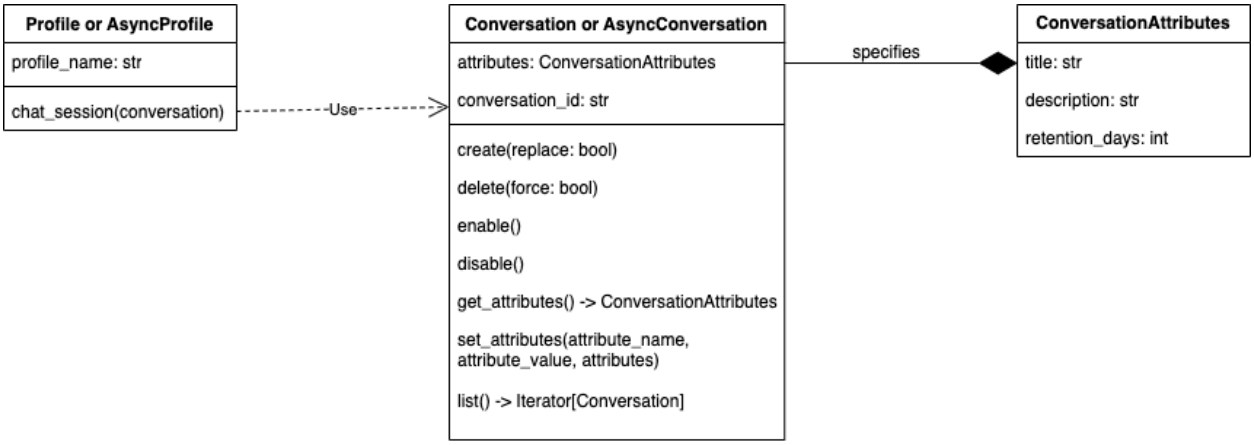
CHAPTER

EIGHT

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.

8.1 Conversation Object model



8.2 ConversationAttributes

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

8.3 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[VectorIndex]

set_attributes(*attributes:* ConversationAttributes)

Updates the attributes of the conversation in the database

8.3.1 Create conversion

```
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
```

8.3.2 Chat session

```
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␣
↪several reasons:
...
...
```

8.3.3 List conversations

```
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)
```

8.3.4 Delete 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 = 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
```

8.4 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
Iterator[VectorIndex]

async set_attributes(*attributes:* ConversationAttributes)

Updates the attributes of the conversation

8.4.1 Async chat session

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

..
```

(continues on next page)

(continued from previous page)

```
..

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:
...
...
```

8.4.2 Async list conversations

```
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)
```

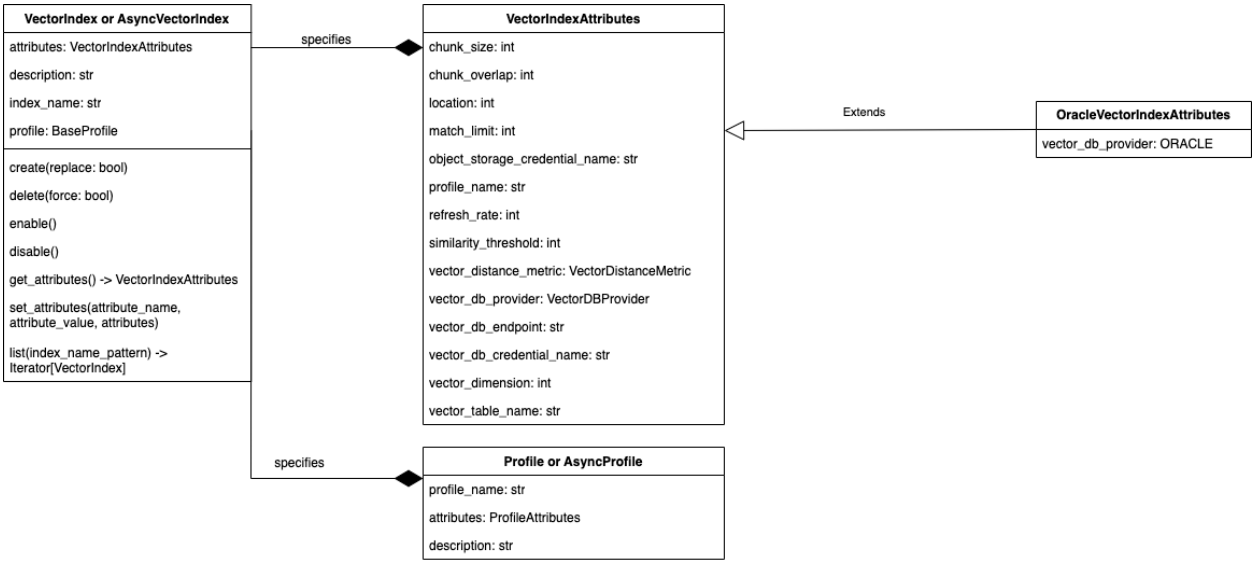
CHAPTER

NINE

VECTOR INDEX

`VectorIndex` supports Retrieval Augmented Generation (RAG). For e.g., you can convert text into vector embeddings and store them in a vector store. Select AI will augment the natural language prompt by retrieving content from the vector store using semantic similarity search.

9.1 `VectorIndex` Object Model



9.2 VectorIndexAttributes

A `VectorIndexAttributes` object can be created with `select_ai.VectorIndexAttributes()`. Also check vector index attributes

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

9.2.1 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

9.3 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

Check the examples below to understand how to create vector indexes

9.3.1 Create vector index

In the following example, vector database provider is Oracle and objects (to create embedding for) reside in OCI's object store

```
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
```

9.3.2 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)
```

9.3.3 Fetch vector index

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

```
import os

import select_ai

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

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

vector_index = select_ai.VectorIndex.fetch(index_name="test_vector_index")
print(vector_index.attributes)
print(vector_index.profile)
print(vector_index.get_next_refresh_timestamp())
```

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)
```

9.3.4 Update vector index attributes

To update attributes, use either `vector_index.set_attribute()` or `vector_index.set_attributes()`

```
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')
```

9.3.5 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")

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)
```

9.3.6 Delete 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(index_name="test_vector_index")
vector_index.delete(force=True)
print("Deleted vector index: test_vector_index")
```

output:

```
Deleted vector index: test_vector_index
```

9.4 AsyncVectorIndex API

A 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

9.4.1 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
```

9.4.2 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)
```

9.4.3 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)
```

9.4.4 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')
```

9.4.5 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

TEN

SYNTHETIC DATA

10.1 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"

10.2 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

Also, check the generate_synthetic_data PL/SQL API for attribute details

10.3 Single table synthetic data

The below example shows single table synthetic data generation

10.3.1 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
```

10.3.2 Singe 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
```

10.4 Multi table synthetic data

The below example shows multitable synthetic data generation

10.4.1 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
```

10.4.2 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

ELEVEN

SUMMARY

11.1 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

11.2 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

11.3 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

11.4 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

TWELVE

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

12.1 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.

12.1.1 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`
`HTTP`	`select_ai.agent.Tool.create_http_tool`	• `tool_name` • `credential_name` • `endpoint`
`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` • `slack_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`

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. For example: "tool_inputs": [

  {
  "name": "data_guard", "description": "Only supported values are "Enabled" and "Disabled""

  }

  ]

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

12.1.2 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,
                                         slack_channel=None,
                                         smtp_host=None),
               tool_inputs=None,
               tool_type=<ToolType.SQL: 'SQL'>)
```

12.1.3 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
```

12.2 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

12.2.1 Create Task

In the following task, we use the MOVIE_SQL_TOOL created in the previous step

```
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)
```

12.2.2 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
```

12.3 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.

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

12.3.1 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)
```

12.4 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.

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

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 **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. Currently, the following parameters are supported:
  - conversation_id: Identifies the conversation session associated

    with the agent team
  - variables: key-value pairs that provide additional 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

12.4.1 Run Team

```
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,
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)
```

(continues on next page)

(continued from previous page)

```
10. Biographical (Biography, 2019-01-10)
... (list continues up to 100 movies)
```

12.5 AI agent examples

12.5.1 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.
```

12.6 Async AI Agent

`select_ai.agent` also provides async interfaces to be used with `async` / `await` keywords

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
HTTP	select_ai.agent.AsyncTool.create_http_tool	• tool_name • credential_name • endpoint
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 • slack_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

12.6.1 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 :param Optional[bool] enabled: Whether the tool should be enabled.

Default: True

Parameters

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

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,
                                         slack_channel=None,
                                         smtp_host=None),
               tool_inputs=None,
               tool_type=<ToolType.SQL: 'SQL'>)
```

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
```

12.6.2 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

Create Task

In the following task, we use the MOVIE_SQL_TOOL created in the previous step

```
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)
```

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
```

12.6.3 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

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)
```

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
```

12.6.4 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 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

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. Currently, the following parameters are supported:
  - conversation_id: Identifies the conversation session associated

    with the agent team
  - variables: key-value pairs that provide additional 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

Run Team

```
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,
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)
```

(continues on next page)

(continued from previous page)

```
9. Historical Epic (Historical, 2019-01-09)
10. Biographical (Biography, 2019-01-10)
... (list continues up to 100 movies)
```

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
```

12.6.5 Async AI agent examples

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,
    )
    print("Created credential: ", OPEN_AI_CREDENTIAL_NAME)
```

(continues on next page)

(continued from previous page)

```
    # # 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,
        description="Web Search Tool using OpenAI",
```

(continues on next page)

(continued from previous page)

```
        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.
```

INDEX

E

F

G

H

L

N

O

P

R

S

T

V