ENDNUTZERDOKUMENTATION

LLM INTERACTION MANAGEMENT SYSTEM

-

LIMS

DEUTSCH

Marvin Emrich

Dokumentation für:
https://github.com/VoltexRB/LIMS

1. Installation

Die Schnittstelle ist öffentlich über das Versionsverwaltungssystem GitHub verfügbar:

https://github.com/VoltexRB/LIMS

Für die Nutzung der Schnittstelle wird Python ab Version 3.9 vorausgesetzt. Unter bestimmten Umständen kann es vorkommen, dass transitive Abhängigkeiten einzelner verwendeter Pakete die jeweils aktuelle Python-Version noch nicht unterstützen. In solchen Fällen wird empfohlen, bei entsprechenden Fehlermeldungen auf eine ältere, kompatible Python-Version auszuweichen.

Die Schnittstelle kann aus dem GitHub Repository bezogen und zur Verwendung installiert werden. Dies kann entweder direkt über den Python Paketmanager `pip` erfolgen oder durch manuelles Klonen beziehungsweise Herunterladen des Repositories, das anschließend über den Paketmanager installiert wird.

Eine Installation ist jedoch in jedem Fall nötig, da die Schnittstelle benötigte externe Bibliotheken ebenfalls installieren muss, welche nicht im Repository mitgebündelt werden.

Abruf und Installation direkt durch pip

Sofern Python auf dem System installiert ist, kann der Paketmanager `pip` als Umgebungsvariable beispielsweise über die Eingabeaufforderung angesprochen werden. Die Schnittstelle kann anschließend als Paket direkt aus dem Git Repository installiert werden. Der folgende Befehl wird dazu in der Kommandozeile des Betriebssystems ausgeführt:

```
pip install git+https://github.com/VoltexRB/LIMS.git
```

Durch die Ausführung dieses Befehls wird das Repository heruntergeladen, das Paket erstellt und sämtliche erforderlichen Abhängigkeiten installiert. Anschließend kann die Schnittstelle systemweit in Python-Projekten verwendet werden.

Manuelle Installation

Der gleiche Ablauf kann jedoch auch in Einzelschritte zerlegt werden. Wird die Schnittstelle auf eine unterschiedliche Weise auf das System geladen und ist dort in einem Verzeichnis verfügbar, kann sie durch den folgenden Befehl, ausgeführt im Verzeichnis der Schnittstelle, ebenfalls installiert werden. Der Befehl installiert die Schnittstelle für die Python-Standardumgebung des Systems.

```
pip install .
```

Wird eine separate oder isolierte Python-Umgebung verwendet, beispielsweise innerhalb eines Jupyter-Notebooks oder eines virtuellen Environments, muss der Installationsbefehl innerhalb dieser Umgebung ausgeführt werden.

```
!pip install git+https://github.com/VoltexRB/LIMS.git
```

Beispielsweise wird der obige Befehl (mit Beachtung des Ausrufezeichens) im Kontext des ausgewählten Jupyter Notebook-Kernels ausgeführt.

Ob die Schnittstelle in einer bestimmten Umgebung verfügbar ist, lässt sich beispielsweise mit den folgenden Zeilen testen.

```
import llm_interaction_manager
print(llm_interaction_manager.__version__)
```

Wenn die Schnittstelle erfolgreich installiert ist, wird die aktuelle Version der Schnittstelle ausgegeben. Falls die Installation nicht erfolgreich war, wird ein Fehler zurückgegeben.

2. Paketstruktur

Das Hauptpaket `llm_interaction_manager` ist entsprechend der Verwendungszwecke aufgeteilt in 4 Unterpakete. Je nach Tiefe der Verwendung können also Komponenten aus anderen Unterpaketen verwendet werden.

Möchte man eine von objektorientierten Konzepten befreite, einfache Verwendung der Schnittstelle implementieren, befindet sich im Unterpaket `api` die Datei `lims_interface.py`, welche die Funktionalitäten der Schnittstelle isoliert anbietet.

Das Unterpaket `core` enthält die Hauptbestandteile der Schnittstelle. Der `InteractionManager` in `interaction_manager.py` stellt die primäre Instanz der Interaktion mit der Schnittstelle durch den Nutzer dar.

Im Unterpaket `handlers` befinden sich die Schnittstellenklassen der verschiedenen externen Endpunkte, beispielsweise des LLM oder der Speichereinheiten. Diese können ebenfalls wahlweise bei einer tieferen Verwendung einzeln erstellt und verwaltet werden.

Das Unterpaket `utils` enthält Hilfsklassen und Methoden zur Verwaltung der Einstellungen der Schnittstelle sowie beispielhafte Datenstrukturen, welche zur Erweiterung herangezogen werden. Das Paket steht für Endnutzer von minderem Interesse dar.

Das heruntergeladene Verzeichnis enthält außerdem den Ordner `dokumentation`, in dem diese Endnutzerdokumentation sowie die Entwicklerdokumentation zu finden sind.

3. Initialisierung

Die Schnittstelle kann allgemein auf zwei verschiedene Arten verwendet werden. Die High-Level API in `lims_interface.py` ermöglicht eine rein skriptbasierte Verwendung der Schnittstelle, während objektorientierte Konzepte intern verwaltet werden. Möchte der Nutzer eine feinere Granularität über die Abläufe und Objekte der Schnittstelle, kann jedoch ebenfalls selbst ein `InteractionManager`-Objekt erstellt und verwaltet werden.

3.1. High-Level API

Die High-Level-API ermöglicht eine vollständig skriptbasierte Nutzung der Schnittstelle. Sämtliche Methoden, die vom InteractionManager bereitgestellt werden, sind in der Datei `lims_interface.py` als Funktionen verfügbar. Intern wird dabei automatisch eine Instanz des `InteractionManager` erzeugt und verwaltet.

Der Import der High-Level-API erfolgt wie folgt:

```
import llm_interaction_manager.api.lims_interface as api
```

Dabei werden die Funktionen des `InteractionManager` über diese Schnittstelle direkt aufgerufen, jedoch muss zuerst ein `InteractionManager`-Objekt intern erzeugt werden. Dies geschieht mit folgendem Befehlsaufruf:

```
api.initialize()
```

Die Methoden können danach ebenfalls ähnlich aufgerufen werden und gleichen in ihrer Signatur und Verwendung den Methoden des `InteractionManager`-Objekts.

Da bei `initialize()` bereits die externen Schnittstellenobjekte für LLM, persistenten- und Vektordatenspeicher erstellt werden, besitzt die Funktion verschiedene optionale Überladungen, je nachdem welche Kommunikation erwünscht ist.

Die Datei `api.interaction_manager_factory.py` erstellt im Zuge von `initialize()` einen InteractionManager, welcher intern gehalten wird. Sie enthält verschiedene Enum-Klassen für mögliche Überladungen der Schnittstellenklassen während der Erstellung. Diese Enums enthalten Auswahlmöglichkeiten der Schnittstellenklassen oder eine Selektion, dass die zuletzt verwendete Schnittstellenklasse erneut verwendet wird, welche als Standardauswahl dient. Wird also kein Parameter des Schnittstellentyps an `initialize()` übergeben, wird versucht die vorherige Einstellung aus der Konfigurationsdatei wiederherzustellen. Es ist zu beachten, dass eine Schnittstelle explizit angegeben werden muss oder vorherige Verbindungsinformationen in der Konfigurationsdatei vorhanden sein müssen. Sind beide Fälle nicht gegeben, schlägt die Initialisierung fehl. Bei einer ersten Verwendung der Software müssen also alle drei Schnittstellen explizit angegeben werden.

Das Enum zur Auswahl der LLM-Schnittstelle enthält beispielsweise die folgenden Werte:

```
class LLMEnum(Enum):
LANGCHAIN
HUGGINGFACE
SETTINGS
```

Dies bedeutet, dass entweder explizit eine Schnittstellenklasse für Langchain beziehungsweise Huggingface erzeugt wird oder versucht werden kann, die letzte Auswahl aus der Konfigurationsdatei zu laden.

Ein valider Aufruf von `initialize()` und der Initialisierung der Software könnte also wie folgt aussehen:

```
import llm_interaction_manager.api.lims_interface as api
from llm_interaction_manager.api.interaction_manager_factory import *

api.initialize(
llm=LLMEnum.LANGCHAIN, vector=VectorEnum.SETTINGS)
```

Dieser Aufruf führt dann zu folgendem Ablauf:

- Explizite LLM-Zuweisung erstellt ein LangchainHandler-Objekt
- Implizierte Vector-Zuweisung erstellt Handler basierend auf Einstellungen
- Optionale, fehlende Persistent-Zuweisung erstellt Handler basierend auf Einstellungen, da dies die Standardüberladung ist

Finden sich also Werte für den Vector- und Persistent-Handler in den Einstellungen, werden drei Handler-Objekte erstellt und damit ein `InteractionManager`-Objekt erstellt und intern in der High-Level API gespeichert.

3.2. Mit Objektverwaltung

Wenn eine feinere Kontrolle erforderlich ist, kann das `InteractionManager`-Objekt auch selbst erstellt und verwaltet werden. Ähnlich wie bei der Verwendung der High-Level API muss auch hier im Konstruktor das Objekt mit den Schnittstellen-Handlern initialisiert werden, wenn keine vorherigen Einstellungen vorhanden sind.

```
from llm_interaction_manager.core.interaction_manager
import InteractionManager

from llm_interaction_manager.handlers.huggingface_handler
import HuggingfaceHandler

llm = HuggingfaceHandler()

interaction_manager = InteractionManager(llm_handler=llm)
```

3.3. Verbindung aufnehmen

Unabhängig davon, ob der InteractionManager über die High-Level-API oder manuell erstellt wurde, wird bei seiner Instanziierung versucht, zuvor gespeicherte Verbindungsinformationen aus den Einstellungen zu laden und eine Verbindung zu den jeweiligen Handlern herzustellen.

Sind keine Einstellungen vorhanden oder der Nutzer möchte die Handler mit einem anderen Endpunkt verbinden, kann die Funktion `connect()` verwendet werden. Ihr werden als Parameter die ausgewählte Schnittstelle übergeben und die Verbindungsinformationen in einem `dict`.

```
connection_data = {port:8000, host:“localhost”}
interaction_manager.connect(ConnectionType.VECTOR, connection_data)
```

Die in `connection_data` zu übergebenden Schlüssel unterscheiden sich je nach verwendetem Schnittstellenhandler. Rote Schlüssel sind erforderlich.

ChromaDB:

- `[client_type]`: Enum in chromadb_handler.py, für Verbindungsauswahl. Je nach Auswahl können weitere Keys erforderlich sein:
  - „VOLATILE“: Daten nur in der aktuellen Instanz aktiv
    - Keine weiteren Keys
  - „PERSISTENT“: Daten lokal gespeichert
    - `[persistent_client_db_path]`: Pfad zum Verzeichnis
  - „HTTP_SERVER“: Über http angesprochener Server
    - `[hostname]`:
    - `[port]`:
  - „CHROMA_CLOUD“: Cloud-Abonnement von ChromaDB
    - `[cloud_tenant]`: Cloud-Account
    - `[cloud_database]`: Cloud-Datenbank
    - `[cloud_key]`: Cloud-Zugriffsschlüssel

PostgreSQL & MongoDB:

- `[hostname]`: Hostname, unter der die Datenbank erreicht werden kann
- `[port]`: Port, unter der die Datenbank erreicht werden kann
- [`username]`: Nutzername eines Nutzers in der Datenbank
- `[password]`: Passwort des Nutzers
- `[database]`: Datenbank, auf die zugegriffen werden soll.

LangChain & Huggingface:

- `[model]`: Ausgewähltes LLM-Modell für die Kommunikation
- `[token]`: API-Token für Huggingface beziehungsweise TogetherAI

3.4. Beispielhafte erste Initialisierungen

Zur Verständlichkeit werden beispielhaft zwei Codeabschnitte aufgeführt, die eine erste Initialisierung der Schnittstelle durch High-Level API beziehungsweise objektorientiert darstellen.

```
import llm_interaction_manager.api.lims_interface as api
from llm_interaction_manager.api.interaction_manager_factory import *

api.initialize(llm=LLMEnum.LANGCHAIN,
persistent=PersistentEnum.MONGODB, vector = VectorEnum.CHROMADB)
```

```
from llm_interaction_manager.core.interaction_manager import *
from llm_interaction_manager.handlers import *

llm = HuggingfaceHandler()
vector = ChromadbHandler()
persistent = PostgresHandler()

im = InteractionManager(llm_handler=llm, vector_handler=vector,
persistent_handler=persistent)
```

Ab diesem Schritt ist die Verwendung von InteractionManager-Objekt und API identisch. Beispielhaft folgt der Verbindungsaufbau der Schnittstellen am gerade erstellten InteractionManager-Objekt. Das Enum `ConnectionType` stammt hierbei direkt aus der Datei interaction_manager.py, weshalb jedes Objekt der Datei importiert wird.

```
llm_data={“model”:”model_name”, token:”api_token”}

vector_data={“client_type”:”PERSISTENT”,
”persistent_client_db_path”:”/home”}

persistent_data={“hostname”:”host”,”port”:5432,”database”:”db”}

im.connect(ConnectionType.VECTOR, vector_data)

im.connect(ConnectionType.PERSISTENT, persistent_data)

im.connect(ConnectionType.LLM, llm_data)
```

Die Verbindung kann über die `is_connected()` Methode überprüft werden.

```
print(im.is_connected(ConnectionType.LLM)
```

4. Verwendung

Sind alle drei Handler im InteractionManager initialisiert und mit den Endpunkten verbunden, kann mit der eigentlichen Kommunikation und somit der Verwendung der Schnittstelle fortgefahren werden.

Die Codeabschnitte in diesem Abschnitt gehen von einer Verwendung eines `InteractionManager`-Objekts aus, können aber auf die API übertragen werden.

Ausgangspunkt einer jeden Interaktion ist eine bestehende Konversation. Jede Nachricht an ein LLM wird im Kontext einer Konversation gesendet, um eine Verbindung zwischen dem Nachrichtenverlauf zu sichern und vorherige Nachrichten mit in die Konversation einbeziehen zu können, selbst wenn die LLM-Schnittstelle kein Konversationsmanagement anbietet. Eine Konversation wird gestartet durch:

```
interaction_manager.start_conversation()
```

Dieser Methode kann optional ein flaches `dict` an Daten übergeben werden, welches im Persistenten Speicher als Metadaten für die gesamte Konversation gespeichert wird. Die Methode erstellt ein neues `Conversation`-Objekt und speichert es im InteractionManager. Bei der Erstellung wird dem Objekt eine eindeutige ID zugewiesen und ein Zeitstempel der Erstellung festgehalten.

Das eigentliche Senden eines Prompts an das LLM wird ausgeführt mit:

```
interaction_manager.send_prompt(“Example”)
```

Dies führt die folgende Abfolge aus:

1. Es wird in der Konversation ein neues Nachrichtenobjekt angelegt und gespeichert
2. Das Prompt wird zum LLM gesendet und die Antwort erwartet
3. ID, Prompt, Antwort und Metadaten werden persistent gespeichert
4. ID, Prompt und Antwort werden in den Vektordaten gespeichert
5. Die Antwort des LLM wird als Rückgabe an den Nutzer zurückgegeben

Neben dieser Hauptfunktion werden noch weitere Funktionen in der Schnittstelle angeboten, die diesen Ablauf unterstützen. Diese werden jeweils im Folgenden einzeln kurz beschrieben.

```
.connect(ConnectionType, data)
```

Stellt Verbindung mit einem ausgewählten externen Handler her.

```
.is_connected(ConnectionType)
```

Überprüft, ob die Verbindung mit ausgewähltem externen Handler besteht.

`.add_metadata(to_conversation, data)`

Fügt je nach Stand von `to_conversation` der Konversation oder der letzten Nachricht manuelle Metadaten in Form eines flachen dicts hinzu.

`.change_comment("comment")`

Ändert das Benutzerkommentar-Feld der letzten Nachricht zur Eingabe.

`.set_rag_data(data, volatile)`

Fügt ein dict von RAG-Daten an das zu sendende Prompt an, je nach Stand von `volatile` werden die Daten ebenfalls in den Nutzereinstellungen persistent gespeichert. Der RAG-Modus wird zusätzlich so gesetzt, dass die Schnittstelle die Daten verwendet.

`.set_rag_mode(RAGMode)`

Ändert den RAG-Modus zu einem bestimmten anderen Modus. Änderungen zu VOLATILE und PERSISTENT sind nur möglich, wenn dazugehörige Daten hinterlegt sind. Weitere Modi sind:

- NONE: Keine Einbindung von RAG-Daten
- DYNAMIC: Ähnlichkeitssuche in der Vektordatenbank passend zum Prompt

`.delete_rag_data()`

Löscht alle persistenten und flüchtigen RAG-Daten in der Schnittstelle.

`.nearest_search_vector(input, top_k, table)`

Manuelle Ähnlichkeitssuche auf der Vektordatenbank. Liefert `top_k` Ergebnisse aus der Tabelle `table`.

`.export_data(path, filters)`

Exportiert Daten aus der persistenten Datenbank, die den Filterkriterien angegeben als dict entsprechen.

`.add_persistent_data(conversation, messages)`

Manuelles Einfügen von Daten in die persistente Datenbank. Die Eingabe muss dem Datenschema der Konversation-Nachricht-Struktur entsprechen. Es wird entweder eine neue Konversation erstellt oder eine bestehende Konversation mit den übergebenen Daten aktualisiert. Dies bedeutet:

- Neue Konversations-ID → Neues Konversationsobjekt in der Datenbank
- Bestehende Konversations-ID → Falls neue Konversationsdaten vorhanden, Konversation aktualisieren

Wenn Nachrichten im Listenparameter `messages` vorhanden sind, werden sie ebenfalls mit dem gleichen Prinzip eingefügt und mit der Konversation verbunden beziehungsweise aktualisiert, falls sie bereits existieren.

```
.add_vector_data(data, table)
```

Fügt die im `data`-dict angegebene Struktur in einen Vektor in der `table`-Tabelle der Vektordatenbank ein. Bestehen Restriktionen an einen Fremdschlüssel, beispielsweise bei PGVector das Vorhandensein einer Message-ID, so müssen diese ebenfalls erfüllt sein.

```
.read_setting(key)
```

Liest eine Einstellung aus der persistenten Einstellungs-Datei der Anwendung. Welche Einstellungen für die Schnittstelle vorgenommen werden können wird im nächsten Kapitel vorgestellt.

```
.write_setting(key, value)
```

Setzt eine bestimmte Einstellung auf den angegebenen Wert.

5. Einstellungen

Um das Verhalten der Schnittstelle für den Nutzer zu ändern, existieren diverse Einstellungsmöglichkeiten in einer Konfigurationsdatei. Diese speichert vorherige Zugangsdaten und Auswahlen der Schnittstellen zu den drei Endpunkten, jedoch auch einfache Key-Value Paare an Einstellungen, welche das Verhalten oder den Ablauf der Software verändern.

Die Einstellungsdatei `config.json` wird zur persistenten Speicherung dieser Einstellungen und Anwendungsdaten im Arbeitsverzeichnis der Anwendung angelegt. Sie ist aufgeteilt in drei Einstellungsbereiche:

- `default_handlers`: Die Auswahl des letzten Handlertyps jeder Schnittstelle
- `handlers`: Die letzten Einstellungen jedes Handlertyps
- `general`: Einfache Nutzereinstellungen für das Verhalten der Software

Die ersten beiden Bereiche werden weitestgehend von der Software geschrieben, gelesen und verwaltet. Sie werden bei der Initialisierung verwendet, um die Schnittstelle automatisch mit den vorher ausgewählten Schnittstellen verbinden zu können und werden bei einer Änderung der Schnittstellen automatisch vom Programm beschrieben.

Der `general` Abschnitt enthält stattdessen vom Nutzer getroffene Einstellungen, welche das Verhalten der Schnittstelle beeinflussen. Diese Einstellungen können mit `read_setting(key)` ausgelesen werden, beziehungsweise mit `write_setting(key, value)` verändert werden. Es folgt eine Übersicht über die zu treffenden Einstellungen und wie sie die Schnittstelle beeinflussen.

- use_rag_data: Wird ebenfalls von `set_rag_mode()` gesetzt und bestimmt, ob und welche RAG-Daten mit dem Prompt an das LLM gesendet werden. Valide Werte sind, wie bereits bei `set_rag_mode()` angegeben:
  - VOLATILE: Flüchtige, statische Daten im Hauptspeicher hinterlegt.
  - PERSISTENT: Feste, statische Daten in den Einstellungen hinterlegt.
  - NONE: Keine RAG-Daten.
  - DYNAMIC: Dynamische RAG-Daten durch Ähnlichkeitssuche in der Vektordatenbank basierend auf dem Prompt.

- default_rag_data: Gespeicherte RAG-Daten, welche standardmäßig im PERSISTENT-Modus gesendet werden.

- default_system_prompt: Wenn vorhanden, wird der Inhalt mit dem Schlüssel „System Prompt“ an das LLM gesendet. Kann verwendet werden, um einen konstanten Kontext zu setzen.

- wait_for_manual_data: Boolescher Wert, der festlegt, ob der allgemeine Ablauf des Sendens eines Prompts für manuelle Nutzerkommentare angehalten wird. Ist der Wert True, wird in Oberflächen mit möglicher Konsoleneingabe die Antwort des LLM an den Nutzer ausgegeben und anschließend auf einen Kommentar zu dieser Interaktion gewartet, bis die Daten in den Speichereinheiten abgelegt sind.

- send_conversation_history: Boolescher Wert, der festlegt, ob vorherige Konversationen mit Prompt und LLM-Antwort an zukünftige Prompts angefügt werden, um eine Konversationshistorie als Kontext bereitzustellen. Diese Einstellung ermöglicht die Referenz auf frühere Prompts und Antworten innerhalb einer Konversation und unterstützt die Emulation einer kohärenten Konversation.

- default_export_path: Standardpfad, an dem exportiere Dateien gespeichert werden sollen. Wenn nicht anders spezifiziert, ist dies das Arbeitsverzeichnis.

6. Datenstrukturen

Bei einer Erweiterung der Schnittstelle oder wenn weitere Hintergrundinformationen benötigt werden, werden in diesem Kapitel die wichtigsten komplexen Datenstrukturen der Schnittstelle aufgezeigt, welche dem Nutzer übergeben werden können oder vom Nutzer an die Schnittstelle gesendet werden können.

Wie bereits erwähnt, werden bei der Verbindung zu einem externen Interface Verbindungsdaten in einem dict übergeben. Hostname und Port werden dabei bei den persistenten Handlern als Parameter in der Funktion übergeben, diese Trennung wird jedoch vom InteractionManager durchgeführt. Vom Nutzer kann connect() also beispielhaft mit dem folgenden Code für einen persistenten Datenhandler aufgerufen werden:

```
data = {
      “hostname”: “localhost”,
      “port”: 5432,
      “username”: “admin”,
      “password”: “password”,
      “database”: “database”}
interaction_manager.connect(ConnectionType.PERSISTENT, data)
```

Dieses Konstrukt würde dann intern aufgeteilt werden und mit Hostname und Port in den Parametern an den verbunden PersistentDataHandler gesendet werden

```
pers_data = {
“username”: data.username,
“password”: data.password,
“database”: data.database}
persistent_data_handler.connect(host, port, pers_data)
```

Bis zu diesem Punkt deckt die Endnutzerdokumentation die häufigsten Anwendungsszenarien der Schnittstelle ab. Zur Erweiterung werden folgend weitere interne komplexe Datentypen aufgelistet, deren semantische Vorlage befolgt werden muss, wenn neue Schnittstellenklassen für externe Entitäten erstellt werden. Diese Strukturen sind jedoch für Endnutzer von niederem Interesse, weshalb lediglich ihre Struktur und ihre Verwendung aufgeführt werden.

Die Methode `save_record(conversation, messages)` von PersistentDataHandlerBase, der Basisklasse der Schnittstellenklassen zu persistenten Datenspeichern, enthält zwei Parameter zur Speicherung eines Datensatzes. Ein `dict`, welches eine Konversation identifiziert und eine liste von `dicts`, welche Nachrichten in der Konversation enthält. Beide Strukturen müssen dabei jeweils zur Referenz mindestens eine Message- beziehungsweise Conversation-ID enthalten. **Das Enthaltensein muss von jeder implementierenden Klasse überprüft werden.** Valide Parameter für die Funktion wären also beispielsweise:

```
conversation = {
“conversation_id”: “conv_001”,
“created_at”: 1736131.3301}
messages = [
    {
        "conversation_id": msg_001,
        "prompt": "Hello",
        "response": "Hi there"},
    {
        "id": msg_002,
        "prompt": "How are you",
        "response": "Fine"}]
```

Vergleichsweise kann `get_data()` ein Filter-dict hinzugefügt werden, welches Werte enthält, die je nach Key gesucht werden. Suchen wir also die Nachricht mit ID „msg_001“ so kann ein Funktionsaufruf wie folgt aussehen:

```
.get_data({“message_id”:”msg_001”})
```

Wichtiger ist hierbei jedoch die Rückgabe. **Jede implementierende Klasse sollte die Rückgabe so gestalten, dass eine Liste von Konversationen zurückgegeben wird, welche die passenden Daten enthält**, beziehungsweise bei Filtern, welche auf die Konversation wirken, die gesamte Konversation. Der beispielhafte Aufruf muss also eine Struktur zurückgeben, welche der folgenden Form entspricht:

```
results = [
    {
        "conversation_id": "conv_001",
        "created_at": 1736131.3301,
        "messages": [
            {
                "message_id": "msg_001",
                "prompt": "Hello",
                "response": "Hi there"
            }
        ]
    }]
```

Bei der Konversation mit LLM wird ebenfalls von `send_prompt()` ein komplexer Datentyp zurückgegeben. Es wird ein `dict` erstellt, welches bestimmte Felder enthalten muss, welche von Conversation erwartet werden.

- `„response“:` Die Antwort des LLM
- `„prompt“:` Das Nutzerprompt, das zur Konsistenz ebenfalls zurückgegeben wird

Weitere Felder des `dicts` sind optional, werden als Metadaten angesehen und in der persistenten Datenspeicherung unter den Metadaten gespeichert.

Bei den Methoden der Vektorspeicher treten ebenfalls komplexe Datentypen auf. `save_vector(data, table)` erwartet als data-Parameter ein `dict`, welches die folgenden drei Felder enthält:

- `„response“:` Die Antwort des LLM
- `„prompt“:` Das Nutzerprompt, das zur Konsistenz ebenfalls zurückgegeben wird
- `„id“:` Die Nachrichten-ID für eine Referenz zum persistenten Speicher

Die Methode `nearest_search()` enthält als Rückgabewert eine liste von `string`-Objekten, welche Prompt und Antwort in einem Format vereinen, da manche Vektorspeicher nur ein Feld pro ID speichern können. Dies bedeutet, dass bei diesen Speichern diese Speicherkonvention eingehalten werden sollte, und Speicherschnittstellen für Speicher, welche mehrere Felder pro ID erlauben müssen bei nearest_search() das Rückgabekonstrukt manuell erzeugen. Die Rückgabe hat das folgende Format, wobei jeder String die Marker `PROMPT:` und `RESPONSE:` enthalten muss:

```
results = [
“PROMPT: Sag Hallo \n RESPONSE: Hallo!”,
“PROMPT: Zähl bis drei \n RESPONSE: 1,2,3”
]
```