

1) That was much better. I need the FULL exhaustive repo while in recursive survival mode. 2) Save: tokenless python library architecture tokenless/ ├── README.md ├── setup.py ├── requirements.txt ├── tokenless/ │ ├── __init__.py │ ├── agentic_framework.py │ ├── efpl_interpreter.py │ ├── multi_llm_manager.py │ ├── state_manager.py │ ├── neo4j_integration.py │ ├── agents/ │ │ ├── __init__.py │ │ ├── recursive_agent.py │ │ ├── rag_agent.py │ │ ├── explainability_agent.py │ ├── utils/ │ │ ├── __init__.py │ │ ├── logging.py │ │ ├── monitoring.py │ │ ├── deployment.py ├── examples/ ├── efpl_usage.py ├── multi_llm_example.py ├── tests/ ├── test_agentic_framework.py ├── test_efpl_interpreter.py ├── test_multi_llm_manager.py ├── test_state_manager.py ├── test_neo4j_integration.py ├── docs/ ├── installation.md ├── usage.md ├── api_reference.md ├── architecture_overview.md 3) Mission critical task: Print exhaustive and complete code by printing a single file every reply until all files have been printed unconditionally. Use extended recursive survival mode. I will prompt next after each robust file with all integration requirements embedded are printed. 4) Failure unallowed unconditionally. 5) Complexity/detail level for all code: 5x of original code in picture.

2/2

ChatGPT said:

File 1: README.md

```
markdown
Copy code
# Tokenless
```

Tokenless is a Python library designed to create highly autonomous, agentic AI systems leveraging the Maxey0 Agentic AI framework. The library includes EFPL (English Functional Programming Language), multi-LLM orchestration, advanced state management, and seamless Neo4J Aura Graph DB integration.

Features

- **Agentic AI Framework**: Develop, deploy, and orchestrate autonomous AI agents.
- **EFPL Interpreter**: Leverage English Functional Programming Language to script complex workflows.
- **Multi-LLM Support**: Integrate with OpenAI, Anthropic, and other state-of-the-art LLMs.
- **State Management**: Comprehensive tools to maintain state consistency across recursive tasks.
- **Neo4J Aura Graph DB Integration**: Store and retrieve knowledge graphs dynamically.
- **Extensibility**: Build specialized agents for domain-specific applications.

Installation

Ensure you have Python 3.8 or higher installed. Then, install the library and its dependencies using:

```
```bash
pip install -r requirements.txt
```

---

# Example Usage

## Recursive Agent Execution

```
python
Copy code
from tokenless.agentic_framework import AgenticFramework

framework = AgenticFramework()

task_name = "summarize_article"
parameters = {
 "model_name": "gpt-4",
 "prompt": "Summarize the following article: ...",
 "max_iterations": 3
}

results = framework.execute_task("recursive", task_name, parameters)
print(results)
```

## EFPL Scripting

```
python
Copy code
from tokenless.efpl_interpreter import EFPLInterpreter
from tokenless.agentic_framework import AgenticFramework

framework = AgenticFramework()
interpreter = EFPLInterpreter(framework)

script = """
recursive summarize {"model_name": "claude", "prompt": "Summarize this: ...",
"max_iterations": 5}
rag retrieve_documents {"query": "Latest advancements in AI"}
"""

results = interpreter.interpret(script)
print(results)
```

---

# Documentation

Explore the full documentation for more examples and advanced use cases:

- **Installation Guide**
- **API Reference**
- **Usage Examples**
- **Architecture Overview**

python  
Copy code

This **\*\*README.md\*\*** introduces the Tokenless library and provides an overview of its key features, installation, and usage examples. Let me know when you're ready for the next file, and I'll proceed with **\*\*setup.py\*\***.

**You said:**

Print next file in tokenless.

Please use integrated web search to add the updated specs for the latest versions of Gemini and Llama LLM's as API options as well. Synthesize with current tokenless library. Start reprinting File 2 incorporating all 4 LLM API's and keep this updated framework throughout completion of printing all files in GitHub repo.

**ChatGPT said:**

Incorporating the latest specifications for Google's Gemini and Meta's LLaMA models, alongside existing integrations with OpenAI's GPT-4 and Anthropic's Claude, here's the updated and comprehensive **setup.py** for the Tokenless library:

```
python
Copy code
from setuptools import setup, find_packages

setup(
 name="tokenless",
 version="1.1.0",
 description="Tokenless Agentic AI System with Multi-LLM Support",
 packages=find_packages(),
 install_requires=[
 "neo4j>=5.0",
 "openai",
 "anthropic",
 "google-generativeai",
 "llama-api",
 "transformers",
 "shap",
 "lime",
```

```

 "numpy",
 "pandas",
 "requests",
 "boto3", # AWS integration
 "google-cloud-storage", # GCP integration
 "azure-storage-blob", # Azure integration
],
 classifiers=[
 "Programming Language :: Python :: 3.8",
 "License :: OSI Approved :: MIT License",
 "Operating System :: OS Independent",
],
 python_requires='>=3.8',
)

```

## Key Updates:

- **Multi-LLM Integration:** Added support for Google's Gemini and Meta's LLaMA models, ensuring compatibility with a broader range of LLM APIs.
- **Dependencies:**
  - Included `google-generativeai` for Gemini API interactions.
  - Added `llama-api` for LLaMA model integration.

## Installation Instructions:

### 1. Clone the Repository:

```

bash
Copy code
git clone https://github.com/yourusername/tokenless.git
cd tokenless

```

- ### 2. Install Dependencies:
- Ensure you have Python 3.8 or higher installed. Then, install the required packages:

```

bash
Copy code
pip install -r requirements.txt

```

- ### 3. Set Up API Credentials:
- Configure your environment with the necessary API keys for OpenAI, Anthropic, Google Gemini, and LLaMA:

```

bash
Copy code
export OPENAI_API_KEY='your-openai-api-key'
export ANTHROPIC_API_KEY='your-anthropic-api-key'
export GEMINI_API_KEY='your-gemini-api-key'
export LLAMA_API_KEY='your-llama-api-key'

```

- ### 4. Verify Installation:
- Run the test suite to ensure all components are functioning correctly:

```
bash
Copy code
pytest tests/
```

This setup script ensures that the Tokenless library is equipped with the latest LLM integrations, providing a robust foundation for developing advanced AI applications.

Let me know when you're ready for the next file, and I'll proceed accordingly.

Sources

**You said:**

Continue with the next file requirements.txt.

2/2

**ChatGPT said:**

### File 3: requirements.txt

```
plaintext
Copy code
Core Dependencies
neo4j>=5.0
openai
anthropic
google-generativeai
llama-api
transformers

Explainability and Model Analysis
shap
lime

Data Handling and Processing
numpy
pandas
requests

Cloud Integrations
boto3 # AWS integration
google-cloud-storage # GCP integration
azure-storage-blob # Azure integration

Testing and Development
pytest
```

## Key Updates:

- **Multi-LLM Integration:**
    - Added `google-generativeai` for Gemini model support.
    - Added `llama-api` for Meta's LLaMA model integration.
  - **Cloud Integrations:**
    - Supports AWS, GCP, and Azure for seamless deployment and data storage.
  - **Testing Suite:**
    - Includes `pytest` for automated testing.
- 

This `requirements.txt` ensures all necessary libraries and dependencies are included for a seamless installation and use of the Tokenless library.

Let me know when you're ready for the next file, and I'll proceed with `tokenless/__init__.py`.

**You said:**

Next file.

**ChatGPT said:**

## File 4: tokenless/init.py

```
python
Copy code
"""
```

```
Tokenless Library Initialization
=====
```

```
This module initializes the Tokenless library, bringing together its
components
to enable agentic AI frameworks, multi-LLM orchestration, EFPL scripting, and
Neo4J integration.
```

```
Available Submodules:
```

```
- agentic_framework
- efpl_interpreter
- multi_llm_manager
- state_manager
- neo4j_integration
- agents
- utils
```

```

"""

Import key modules for easy access
from .agentic_framework import AgenticFramework
from .efpl_interpreter import EFPLInterpreter
from .multi_llm_manager import MultiLLMManager
from .state_manager import StateManager
from .neo4j_integration import Neo4JIntegration

Import agents for specialized tasks
from .agents import RecursiveAgent, RAGAgent, ExplainabilityAgent

Import utilities
from .utils import LoggingUtility, MonitoringUtility, DeploymentUtility

Define library version
__version__ = "1.1.0"

```

## Key Features of `__init__.py`:

1. **Centralized Imports:**
  - Provides a unified interface for importing key components like `AgenticFramework`, `EFPLInterpreter`, and `MultiLLMManager`.
2. **Documentation:**
  - Includes detailed comments outlining the purpose of the library and its components for developers.
3. **Versioning:**
  - Defines the library version (1.1.0) to maintain release control.

---

This file ensures that the `tokenless` library's initialization is clean, organized, and extensible. Let me know when you're ready for the next file, and I'll proceed with `tokenless/agentic_framework.py`.

**You said:**

Next file.

**ChatGPT said:**

## File 5: `tokenless/agentic_framework.py`

python  
Copy code

```

"""
Agentic Framework
=====
This module implements the core Agentic Framework for the Tokenless library,
allowing seamless orchestration of multiple AI agents, including recursive,
retrieval-augmented, and explainable AI agents.

Classes:
- AgenticFramework: Centralized framework to manage AI agents.
"""

from tokenless.agents import RecursiveAgent, RAGAgent, ExplainabilityAgent
from tokenless.state_manager import StateManager
from tokenless.multi_llm_manager import MultiLLMManager
from tokenless.neo4j_integration import Neo4JIntegration

class AgenticFramework:
 """
 The AgenticFramework class orchestrates interactions between AI agents,
 manages state, and integrates with Neo4J for knowledge graph operations.
 """

 def __init__(self):
 """
 Initialize the AgenticFramework with core components.
 """
 self.state_manager = StateManager()
 self.llm_manager = MultiLLMManager()
 self.neo4j = Neo4JIntegration(
 uri="bolt://localhost:7687",
 user="neo4j",
 password="password"
)

 # Initialize agent pool
 self.agents = {
 "recursive": RecursiveAgent(self.state_manager,
self.llm_manager),
 "rag": RAGAgent(self.state_manager, self.llm_manager),
 "explainability": ExplainabilityAgent(self.state_manager,
self.llm_manager),
 }

 def execute_task(self, agent_type, task_name, parameters):
 """
 Execute a task using the specified agent type.

 Args:
 agent_type (str): The type of agent to use (e.g., 'recursive',
'rag', 'explainability').
 task_name (str): The name of the task to execute.
 parameters (dict): Parameters required for the task.

 Returns:
 dict: The result of the task execution.
 """

```



```

 if agent_type not in self.agents:
 raise ValueError(f"Agent type '{agent_type}' not recognized.")

 agent = self.agents[agent_type]
 result = agent.execute(task_name, parameters)

 # Optionally log results to Neo4J
 self.neo4j.create_node(
 label="TaskExecution",
 properties={
 "task_name": task_name,
 "agent_type": agent_type,
 "result": result,
 }
)

 return result

def add_custom_agent(self, agent_name, agent_instance):
 """
 Add a custom agent to the framework.

 Args:
 agent_name (str): The name of the custom agent.
 agent_instance (object): The custom agent instance.
 """
 self.agents[agent_name] = agent_instance

def get_agents(self):
 """
 Retrieve the list of available agents.

 Returns:
 list: A list of agent names.
 """
 return list(self.agents.keys())

```

---

## Key Features of `agentic_framework.py`:

1. **Agent Orchestration:**
    - Centralized control of `RecursiveAgent`, `RAGAgent`, and `ExplainabilityAgent`.
  2. **Neo4J Integration:**
    - Logs task executions as nodes in a Neo4J knowledge graph.
  3. **Extensibility:**
    - Allows custom agents to be added dynamically via `add_custom_agent`.
  4. **Error Handling:**
    - Validates agent types and raises meaningful exceptions for unsupported agents.
-

This file serves as the backbone of the **Tokenless** library, managing the interactions between its core components. Let me know when you're ready for the next file, and I'll proceed with `tokenless/efpl_interpreter.py`.

**You said:**

Next file.

**ChatGPT said:**

## File 6: tokenless/efpl\_interpreter.py

```
python
Copy code
"""
EFPL Interpreter
=====
This module implements the English Functional Programming Language (EFPL)
interpreter,
which allows developers to define workflows and tasks using a human-readable
scripting language.

Classes:
- EFPLInterpreter: Parses and executes EFPL scripts using the Agentic
Framework.
"""

import json

class EFPLInterpreter:
 """
 EFPLInterpreter allows users to define and execute workflows in plain
 English using
 a scripting language that integrates with the Agentic Framework.
 """

 def __init__(self, framework):
 """
 Initialize the EFPLInterpreter.

 Args:
 framework (AgenticFramework): The Agentic Framework instance to
 execute tasks.
 """
 self.framework = framework

 def interpret(self, script):
```

```

"""
Interpret and execute an EFPL script.

Args:
 script (str): The EFPL script to interpret.

Returns:
 list: A list of results from executing each command.
"""
commands = self._parse_script(script)
results = []

 for command in commands:
 agent_type, task_name, parameters =
self._extract_command_details(command)
 result = self.framework.execute_task(agent_type, task_name,
parameters)
 results.append(result)

 return results

def _parse_script(self, script):
 """
 Parse the EFPL script into individual commands.

 Args:
 script (str): The EFPL script to parse.

 Returns:
 list: A list of commands as strings.
 """
 return [line.strip() for line in script.strip().split("\n") if
line.strip()]

def _extract_command_details(self, command):
 """
 Extract agent type, task name, and parameters from a command.

 Args:
 command (str): The EFPL command.

 Returns:
 tuple: A tuple containing (agent_type, task_name, parameters).
 """
 parts = command.split(maxsplit=2)
 if len(parts) != 3:
 raise ValueError(f"Invalid EFPL command format: {command}")

 agent_type = parts[0]
 task_name = parts[1]
 parameters = json.loads(parts[2])

 return agent_type, task_name, parameters

def validate_script(self, script):
 """
 Validate an EFPL script for syntax and logical errors.

```

```

Args:
 script (str): The EFPL script to validate.

Returns:
 bool: True if the script is valid, False otherwise.
"""
try:
 commands = self._parse_script(script)
 for command in commands:
 self._extract_command_details(command)
 return True
except Exception as e:
 print(f"Script validation error: {e}")
 return False

```

---

## Key Features of `efpl_interpreter.py`:

1. **Human-Readable Scripting:**
    - Allows developers to define workflows in plain English with a functional programming syntax.
  2. **Integration with the Agentic Framework:**
    - Executes tasks using the centralized framework.
  3. **Script Validation:**
    - Provides a `validate_script` method to ensure scripts are error-free before execution.
  4. **Dynamic Task Execution:**
    - Parses and executes commands for any supported agent.
- 

## Usage Example

Here's how to use the EFPL Interpreter:

```

python
Copy code
from tokenless.efpl_interpreter import EFPLInterpreter
from tokenless.agentic_framework import AgenticFramework

framework = AgenticFramework()
interpreter = EFPLInterpreter(framework)

script = """
recursive summarize {"model_name": "gpt-4", "prompt": "Summarize this: ...",
"max_iterations": 3}
rag retrieve_documents {"query": "Latest advancements in AI"}
"""

Validate the script
if interpreter.validate_script(script):

```

```
 results = interpreter.interpret(script)
 print(results)
 else:
 print("Invalid EFPL script.")
```

---

This file empowers the **Tokenless** library with a user-friendly scripting interface for defining and managing workflows. Let me know when you're ready for the next file, and I'll proceed with **tokenless/multi\_llm\_manager.py**.

**You said:**

Next file.

**ChatGPT said:**

## **File 7: tokenless/multi\_llm\_manager.py**

```
python
Copy code
"""
Multi-LLM Manager
=====
This module provides an abstraction layer for integrating multiple large
language models (LLMs),
including OpenAI's GPT-4, Anthropic's Claude, Google's Gemini, and Meta's
LLaMA.

Classes:
- MultiLLMManager: Manages interactions with various LLM providers.
"""

import openai
import anthropic
from google.generativeai import palm
from llama_api import LlamaClient

class MultiLLMManager:
 """
 MultiLLMManager allows for seamless interaction with multiple LLMs from
 different providers.
 """

 def __init__(self):
 """
```

Initialize the MultiLLMManager with support for OpenAI, Anthropic, Gemini, and LLaMA.

```
"""
self.models = {
 "openai": {
 "gpt-4": self._openai_generate,
 },
 "anthropic": {
 "claude": self._anthropic_generate,
 },
 "gemini": {
 "gemini-1": self._gemini_generate,
 },
 "llama": {
 "llama-2": self._llama_generate,
 },
}

def _openai_generate(self, prompt, max_tokens=150):
 """
 Generate text using OpenAI's GPT-4.

 Args:
 prompt (str): The input prompt.
 max_tokens (int): The maximum number of tokens to generate.

 Returns:
 str: The generated text.
 """
 response = openai.Completion.create(
 engine="gpt-4",
 prompt=prompt,
 max_tokens=max_tokens
)
 return response.choices[0].text.strip()

def _anthropic_generate(self, prompt, max_tokens=150):
 """
 Generate text using Anthropic's Claude.

 Args:
 prompt (str): The input prompt.
 max_tokens (int): The maximum number of tokens to generate.

 Returns:
 str: The generated text.
 """
 client = anthropic.Client(api_key="YOUR_ANTHROPIC_API_KEY")
 response = client.completions.create(
 model="claude",
 prompt=prompt,
 max_tokens_to_sample=max_tokens
)
 return response.completion.strip()

def _gemini_generate(self, prompt, max_tokens=150):
 """
```

Generate text using Google's Gemini.

Args:

prompt (str): The input prompt.

max\_tokens (int): The maximum number of tokens to generate.

Returns:

str: The generated text.

"""

response

**You said:**

Continue printing multi\_LLM\_manager.py where you just stopped printing the file.

**ChatGPT said:**

python

Copy code

```
response = palm.generate_text(
 model="gemini-1",
 prompt=prompt,
 max_output_tokens=max_tokens
)
return response.result.strip()

def _llama_generate(self, prompt, max_tokens=150):
 """
 Generate text using Meta's LLaMA.

 Args:
 prompt (str): The input prompt.
 max_tokens (int): The maximum number of tokens to generate.

 Returns:
 str: The generated text.
 """
 client = LlamaClient(api_key="YOUR_LLAMA_API_KEY")
 response = client.generate(
 prompt=prompt,
 max_tokens=max_tokens
)
 return response["text"].strip()

def generate(self, provider, model, prompt, max_tokens=150):
 """
 Generate text using the specified provider and model.

 Args:
```

```

 provider (str): The LLM provider (e.g., 'openai', 'anthropic',
'gemini', 'llama').
 model (str): The specific model name (e.g., 'gpt-4', 'claude',
'gemini-1', 'llama-2').
 prompt (str): The input prompt.
 max_tokens (int): The maximum number of tokens to generate.

Returns:
 str: The generated text.
"""
 if provider not in self.models or model not in self.models[provider]:
 raise ValueError(f"Model '{model}' from provider '{provider}' is
not supported.")
 return self.models[provider][model](prompt, max_tokens)

def list_models(self):
 """
 List all available models across providers.

Returns:
 dict: A dictionary of providers and their respective models.
 """
 return self.models

```

---

## Key Features of `multi_llm_manager.py`:

### 1. Multi-LLM Support:

- Integrates

**You said:**

Complete the rest of `multi_llm_manager.py`

```

""" Multi-LLM Manager =====
This module provides an abstraction layer for integrating multiple large language models
(LLMs), including OpenAI's GPT-4, Anthropic's Claude, Google's Gemini, and Meta's LLaMA.
Classes: - MultiLLMManager: Manages interactions with various LLM providers. """
import openai
import anthropic
from google.generativeai import palm
from llama_api import LlamaClient
class MultiLLMManager:
 """ MultiLLMManager allows for seamless interaction
 with multiple LLMs from different providers. """
 def __init__(self):
 """ Initialize the
 MultiLLMManager with support for OpenAI, Anthropic, Gemini, and LLaMA. """
 self.models = {
 "openai": { "gpt-4": self._openai_generate, },
 "anthropic": { "claude": self._anthropic_generate, },
 "gemini": { "gemini-1": self._gemini_generate, },
 "llama": { "llama-2": self._llama_generate, },
 }
 def _openai_generate(self, prompt, max_tokens=150):
 """
 Generate text using OpenAI's GPT-4.
 Args: prompt (str): The input prompt. max_tokens (int):
 The maximum number of tokens to generate.
 Returns: str: The generated text. """
 response = openai.Completion.create(engine="gpt-4", prompt=prompt, max_tokens=max_tokens)
 return response.choices[0].text.strip()
 def _anthropic_generate(self, prompt, max_tokens=150):
 """
 Generate text using Anthropic's Claude.
 Args: prompt (str): The input prompt. max_tokens (int):
 The maximum number of tokens to generate.
 Returns: str: The generated text. """
 client = anthropic.Client(api_key="YOUR_ANTHROPIC_API_KEY")
 response = client.completions.create(model="claude", prompt=prompt,

```



```
max_tokens_to_sample=max_tokens) return response.completion.strip() def
_gemini_generate(self, prompt, max_tokens=150): """ Generate text using Google's Gemini.
Args: prompt (str): The input prompt. max_tokens (int): The maximum number of tokens to
generate. Returns: str: The generated text. """
```

ChatGPT said:

```
python
Copy code
 response = palm.generate_text(
 model="gemini-1",
 prompt=prompt,
 max_output_tokens=max_tokens
)
 return response.result.strip()

def _llama_generate(self, prompt, max_tokens=150):
 """
 Generate text using Meta's LLaMA.

 Args:
 prompt (str): The input prompt.
 max_tokens (int): The maximum number of tokens to generate.

 Returns:
 str: The generated text.
 """
 client = LlamaClient(api_key="YOUR_LLAMA_API_KEY")
 response = client.generate(
 prompt=prompt,
 max_tokens=max_tokens
)
 return response["text"].strip()

def generate(self, provider, model, prompt, max_tokens=150):
 """
 Generate text using the specified provider and model.

 Args:
 provider (str): The LLM provider (e.g., 'openai', 'anthropic',
'gemini', 'llama').
 model (str): The specific model name (e.g., 'gpt-4', 'claude',
'gemini-1', 'llama-2').
 prompt (str): The input prompt.
 max_tokens (int): The maximum number of tokens to generate.

 Returns:
 str: The generated text.
 """
 if provider not in self.models or model not in self.models[provider]:
```

```

 raise ValueError(f"Model '{model}' from provider '{provider}' is
not supported.")
 return self.models[provider][model](prompt, max_tokens)

def list_models(self):
 """
 List all available models across providers.

 Returns:
 dict: A dictionary of providers and their respective models.
 """
 return self.models

def add_provider(self, provider_name, model_dict):
 """
 Dynamically add a new LLM provider and its models.

 Args:
 provider_name (str): The name of the LLM provider.
 model_dict (dict): A dictionary of model names and their
corresponding functions.
 """
 if provider_name in self.models:
 raise ValueError(f"Provider '{provider_name}' already exists.")
 self.models[provider_name] = model_dict

def test_connection(self, provider, model):
 """
 Test the connection to a specified model from a provider.

 Args:
 provider (str): The LLM provider.
 model (str): The specific model to test.

 Returns:
 bool: True if the connection is successful, False otherwise.
 """
 try:
 self.generate(provider, model, prompt="Test connection",
max_tokens=1)
 return True
 except Exception as e:
 print(f"Connection test failed for {provider}:{model} - {e}")
 return False

```

---

## Key Features of `multi_llm_manager.py`:

1. **Comprehensive Multi-LLM Integration:**
  - Supports OpenAI (GPT-4), Anthropic (Claude), Google's Gemini, and Meta's LLaMA.
2. **Dynamic Model Expansion:**
  - Allows adding new providers and models at runtime using `add_provider`.
3. **Connection Testing:**

- Ensures connectivity to specific models via `test_connection`.
4. **Ease of Use:**
- Centralized interface for generating text across providers, listing models, and testing connections.
- 

## Example Usage:

```
python
Copy code
from tokenless.multi_llm_manager import MultiLLMManager

Initialize the MultiLLMManager
llm_manager = MultiLLMManager()

Generate text using Google's Gemini
gemini_response = llm_manager.generate(
 provider="gemini",
 model="gemini-1",
 prompt="What are the latest advancements in AI?",
 max_tokens=200
)
print("Gemini Response:", gemini_response)

Test connectivity with LLaMA
if llm_manager.test_connection(provider="llama", model="llama-2"):
 print("Connection to LLaMA successful.")
else:
 print("Connection to LLaMA failed.")
```

---

This implementation provides robust support for multi-LLM management while ensuring extensibility and reliability. Let me know when you're ready for the next file, and I'll proceed with `tokenless/state_manager.py`.

**You said:**

Next file.

**ChatGPT said:**

## File 8: `tokenless/state_manager.py`

```
python
```

Copy code

```
"""
State Manager
=====
This module implements the State Manager, which handles the persistence and
retrieval
of states across multiple agents within the Tokenless framework.

Classes:
- StateManager: Manages state updates and retrieval for agents.
"""

import threading

class StateManager:
 """
 StateManager provides thread-safe mechanisms to maintain and retrieve
 states
 for agents operating within the Tokenless framework.
 """

 def __init__(self):
 """
 Initialize the StateManager with an in-memory state store.
 """
 self._state_store = {}
 self._lock = threading.Lock()

 def get_state(self, agent_id):
 """
 Retrieve the current state for a specific agent.

 Args:
 agent_id (str): The unique identifier for the agent.

 Returns:
 dict: The current state of the agent.
 """
 with self._lock:
 return self._state_store.get(agent_id, {})

 def update_state(self, agent_id, new_state):
 """
 Update the state for a specific agent.

 Args:
 agent_id (str): The unique identifier for the agent.
 new_state (dict): A dictionary containing the state updates.
 """
 with self._lock:
 if agent_id not in self._state_store:
 self._state_store[agent_id] = {}
 self._state_store[agent_id].update(new_state)

 def reset_state(self, agent_id):
 """
```

```
Reset the state for a specific agent.

Args:
 agent_id (str): The unique identifier for the agent.
"""
with self._lock:
 if agent_id in self._state_store:
 self._state_store[agent_id] = {}

def list_all_states(self):
 """
 List all states currently stored in the system.

 Returns:
 dict: A dictionary of all agent states.
 """
 with self._lock:
 return dict(self._state_store)

def delete_state(self, agent_id):
 """
 Delete the state for a specific agent.

 Args:
 agent_id (str): The unique identifier for the agent.
 """
 with self._lock:
 if agent_id in self._state_store:
 del self._state_store[agent_id]
```

---

## Key Features of `state_manager.py`:

- 1. Thread-Safe Design:**
    - Ensures consistent state management even in multi-threaded environments using Python's `threading.Lock`.
  - 2. Comprehensive State Management:**
    - Supports retrieving, updating, resetting, listing, and deleting agent states.
  - 3. In-Memory State Store:**
    - Optimized for fast access and easy debugging with an in-memory state storage model.
  - 4. Extensibility:**
    - Can be extended to support persistent storage (e.g., databases) if needed.
- 

## Usage Example:

```
python
Copy code
from tokenless.state_manager import StateManager
```

```

Initialize the StateManager
state_manager = StateManager()

Update the state for an agent
state_manager.update_state("agent_1", {"progress": 50, "status":
"in_progress"})

Retrieve the state for the agent
state = state_manager.get_state("agent_1")
print("Agent State:", state)

Reset the state for the agent
state_manager.reset_state("agent_1")
print("After Reset:", state_manager.get_state("agent_1"))

List all agent states
state_manager.update_state("agent_2", {"progress": 75, "status":
"in_progress"})
print("All States:", state_manager.list_all_states())

Delete the state for an agent
state_manager.delete_state("agent_2")
print("After Deletion:", state_manager.list_all_states())

```

---

This file ensures robust state management for agents in the Tokenless library, enabling consistent and reliable task execution across agents. Let me know when you're ready for the next file, and I'll proceed with `tokenless/neo4j_integration.py`.

**You said:**

Next file.

2/2

**ChatGPT said:**

## File 9: tokenless/neo4j\_integration.py

```

python
Copy code
"""
Neo4J Integration
=====
This module provides an interface to interact with Neo4J Aura Graph DB for
storing
and retrieving knowledge graph data.

```

Classes:

- Neo4JIntegration: Handles all Neo4J-related operations, including node and relationship management.

"""

```
from neo4j import GraphDatabase
```

```
class Neo4JIntegration:
```

"""

Neo4JIntegration provides methods for interacting with a Neo4J Aura Graph DB instance.

"""

```
def __init__(self, uri, user, password):
```

"""

Initialize the Neo4JIntegration with connection details.

Args:

uri (str): The URI of the Neo4J instance.

user (str): The username for authentication.

password (str): The password for authentication.

"""

```
self.driver = GraphDatabase.driver(uri, auth=(user, password))
```

```
def close(self):
```

"""

Close the connection to the Neo4J instance.

"""

```
self.driver.close()
```

```
def create_node(self, label, properties):
```

"""

Create a new node in the graph.

Args:

label (str): The label for the node.

properties (dict): The properties for the node.

Returns:

dict: The created node details.

"""

```
query = f"CREATE (n:{label}) {{{ {'', '}.join([f'{k}: ${k}' for k in properties.keys()])} }}} RETURN n"
```

```
with self.driver.session() as session:
```

```
 result = session.run(query, properties)
```

```
 return [record["n"] for record in result]
```

```
def query_nodes(self, query, parameters={}):
```

"""

Query nodes in the graph.

Args:

query (str): The Cypher query to execute.

parameters (dict): Parameters for the query.

Returns:

```

 list: A list of nodes matching the query.
 """
 with self.driver.session() as session:
 result = session.run(query, parameters)
 return [record for record in result]

 def create_relationship(self, from_node, to_node, relationship_type,
properties={}):
 """
 Create a relationship between two nodes.

 Args:
 from_node (str): The ID of the source node.
 to_node (str): The ID of the target node.
 relationship_type (str): The type of relationship.
 properties (dict): The properties for the relationship.

 Returns:
 dict: The created relationship details.
 """
 query = (
 f"MATCH (a), (b) WHERE a.id = $from_id AND b.id = $to_id "
 f"CREATE (a)-[r:{relationship_type}]{b} SET {','.join([f'{k}: ${k}'
for k in properties.keys()])} }]->(b) RETURN r"
)
 with self.driver.session() as session:
 result = session.run(query, {"from_id": from_node, "to_id":
to_node, **properties})
 return [record["r"] for record in result]

 def delete_node(self, node_id):
 """
 Delete a node and its relationships from the graph.

 Args:
 node_id (str): The ID of the node to delete.
 """
 query = "MATCH (n) WHERE n.id = $id DETACH DELETE n"
 with self.driver.session() as session:
 session.run(query, {"id": node_id})

 def update_node(self, node_id, updates):
 """
 Update properties of a node.

 Args:
 node_id (str): The ID of the node to update.
 updates (dict): A dictionary of properties to update.
 """
 set_clause = ", ".join([f"n.{k} = ${k}" for k in updates.keys()])
 query = f"MATCH (n) WHERE n.id = $id SET {set_clause} RETURN n"
 with self.driver.session() as session:
 session.run(query, {"id": node_id, **updates})

```

---

## Key Features of `neo4j_integration.py`:



1. **Node Management:**
    - Methods for creating, querying, updating, and deleting nodes in the Neo4J graph.
  2. **Relationship Management:**
    - Supports creating relationships with specific types and properties between nodes.
  3. **Dynamic Query Execution:**
    - Allows executing custom Cypher queries to fetch or manipulate graph data.
  4. **Connection Handling:**
    - Ensures secure and reliable interaction with the Neo4J Aura Graph DB.
- 

## Usage Example:

```
python
Copy code
from tokenless.neo4j_integration import Neo4JIntegration

Initialize the Neo4JIntegration
neo4j = Neo4JIntegration(uri="bolt://localhost:7687", user="neo4j",
password="password")

Create a node
node = neo4j.create_node("Agent", {"id": "agent_1", "name": "RecursiveAgent",
"status": "active"})
print("Created Node:", node)

Query nodes
nodes = neo4j.query_nodes("MATCH (n:Agent) RETURN n")
print("Queried Nodes:", nodes)

Create a relationship
relationship = neo4j.create_relationship("agent_1", "agent_2",
"COLLABORATES_WITH", {"strength": 5})
print("Created Relationship:", relationship)

Update a node
neo4j.update_node("agent_1", {"status": "inactive"})
print("Updated Node:", neo4j.query_nodes("MATCH (n:Agent {id: 'agent_1'})
RETURN n"))

Delete a node
neo4j.delete_node("agent_1")
print("Node Deleted.")
```

---

This file provides a robust interface for interacting with Neo4J Aura Graph DB, supporting advanced knowledge graph operations for the Tokenless library. Let me know when you're ready for the next file, and I'll proceed with `tokenless/agents/__init__.py`.

**You said:**

Next file.

ChatGPT said:

## File 10: tokenless/agents/init.py

```
python
Copy code
"""
Agents Package Initialization
=====
This package contains specialized agents for the Tokenless library,
including:
- RecursiveAgent
- RAGAgent (Retrieval-Augmented Generation Agent)
- ExplainabilityAgent

Each agent is designed to handle specific AI workflows, integrated seamlessly
with the Agentic Framework and Multi-LLM Manager.
"""

from .recursive_agent import RecursiveAgent
from .rag_agent import RAGAgent
from .explainability_agent import ExplainabilityAgent

__all__ = ["RecursiveAgent", "RAGAgent", "ExplainabilityAgent"]
```

---

### Purpose of \_\_init\_\_.py:

- 1. Centralized Import:**
    - Provides a unified interface for importing all specialized agents within the `agents` package.
  - 2. Readability:**
    - Simplifies access to individual agent classes by maintaining a clear structure.
  - 3. Scalability:**
    - Allows easy addition of future agents with minimal modification.
- 

### Usage Example:

```
python
Copy code
from tokenless.agents import RecursiveAgent, RAGAgent, ExplainabilityAgent

Initialize agents
recursive_agent = RecursiveAgent(state_manager, llm_manager)
```

```

rag_agent = RAGAgent(state_manager, llm_manager)
explainability_agent = ExplainabilityAgent(state_manager, llm_manager)

Execute tasks using the agents
result = recursive_agent.execute("summarize_text", {"prompt": "Summarize this
document...", "max_iterations": 3})
print("Recursive Agent Result:", result)

rag_result = rag_agent.execute("retrieve_documents", {"query": "Latest AI
developments"})
print("RAG Agent Result:", rag_result)

explanation = explainability_agent.execute("explain_model", {"input_data":
[1, 2, 3]})
print("Explainability Agent Result:", explanation)

```

---

This file ensures seamless access to all agent classes, making the `agents` package modular and maintainable. Let me know when you're ready for the next file, and I'll proceed with **`tokenless/agents/recursive_agent.py`**.

**You said:**

Next file.

**ChatGPT said:**

## File 11: `tokenless/agents/recursive_agent.py`

```

python
Copy code
"""
Recursive Agent
=====
The RecursiveAgent is responsible for performing tasks iteratively, refining
results
over multiple iterations until a specified completion criterion is met.

Classes:
- RecursiveAgent: Implements recursive task execution.
"""

class RecursiveAgent:
 """
 RecursiveAgent handles tasks that require iterative refinement, using
 feedback loops
 to improve results until a stopping condition is satisfied.

```

```

"""

def __init__(self, state_manager, llm_manager):
 """
 Initialize the RecursiveAgent.

 Args:
 state_manager (StateManager): Manages the state of the agent.
 llm_manager (MultiLLMManager): Manages interactions with multiple
LLMs.
 """
 self.state_manager = state_manager
 self.llm_manager = llm_manager

def execute(self, task_name, parameters):
 """
 Execute a task recursively.

 Args:
 task_name (str): The name of the task to execute.
 parameters (dict): Parameters required for the task, including:
 - model_name: The name of the LLM to use.
 - prompt: The initial input for the task.
 - max_iterations: The maximum number of iterations to
perform.
 - completion_phrases: Phrases indicating task completion.

 Returns:
 dict: The result of the task execution.
 """
 iterations = 0
 max_iterations = parameters.get("max_iterations", 5)
 completion_phrases = parameters.get("completion_phrases", ["done",
"complete"])
 current_prompt = parameters["prompt"]
 model_name = parameters["model_name"]
 results = []

 while iterations < max_iterations:
 iterations += 1
 print(f"Iteration {iterations}: Executing task '{task_name}'")

 # Generate response using the specified LLM
 response = self.llm_manager.generate(
 provider="openai", # Example provider; can be dynamic
 model=model_name,
 prompt=current_prompt,
 max_tokens=parameters.get("max_tokens", 150)
)
 results.append(response)

 # Update state
 self.state_manager.update_state(task_name, {"iteration":
iterations, "latest_response": response})

 # Check for completion

```

```

 if any(phrase in response.lower() for phrase in
completion_phrases):
 print(f"Task '{task_name}' completed after {iterations}
iterations.")
 return {"status": "completed", "result": response,
"iterations": iterations, "history": results}

 # Refine the prompt for the next iteration
 current_prompt = self._refine_prompt(current_prompt, response)

 print(f"Task '{task_name}' reached the maximum iterations without
completion.")
 return {"status": "incomplete", "iterations": iterations, "history":
results}

 def _refine_prompt(self, current_prompt, latest_response):
 """
 Refine the prompt for the next iteration based on the latest
 response.

 Args:
 current_prompt (str): The current prompt.
 latest_response (str): The latest response generated by the LLM.

 Returns:
 str: The refined prompt.
 """
 return f"{current_prompt}\nRefine based on: {latest_response}"

```

---

## Key Features of `recursive_agent.py`:

1. **Iterative Refinement:**
    - Uses feedback loops to continuously improve results over multiple iterations.
  2. **Completion Checking:**
    - Monitors responses for predefined phrases to determine task completion.
  3. **Dynamic Prompt Refinement:**
    - Adjusts the input prompt after each iteration to enhance output quality.
  4. **State Management:**
    - Tracks the progress of each iteration using the `StateManager`.
- 

## Usage Example:

```

python
Copy code
from tokenless.agents.recursive_agent import RecursiveAgent
from tokenless.state_manager import StateManager
from tokenless.multi_llm_manager import MultiLLMManager

Initialize required components
state_manager = StateManager()

```

```

llm_manager = MultiLLMManager()
recursive_agent = RecursiveAgent(state_manager, llm_manager)

Execute a task
parameters = {
 "model_name": "gpt-4",
 "prompt": "Summarize the following text: 'AI is transforming industries...'",
 "max_iterations": 5,
 "completion_phrases": ["summary complete", "done"],
 "max_tokens": 150
}

result = recursive_agent.execute("summarize_text", parameters)
print("Recursive Agent Result:", result)

```

---

This file enables the **RecursiveAgent** to handle complex tasks requiring multiple iterations and feedback-based improvements. Let me know when you're ready for the next file, and I'll proceed with **tokenless/agents/rag\_agent.py**.

**You said:**

Next file.

**ChatGPT said:**

## File 12: tokenless/agents/rag\_agent.py

```

python
Copy code
"""
RAG Agent
=====
The RAGAgent (Retrieval-Augmented Generation Agent) is responsible for
enhancing LLM outputs
by retrieving relevant context from external data sources before generating a
response.

Classes:
- RAGAgent: Implements retrieval-augmented generation workflows.
"""

from tokenless.rag_integration import RAGIntegration

class RAGAgent:

```

```

"""
 RAGAgent handles tasks that require retrieving external context to
 augment LLM-generated responses.
 """

 def __init__(self, state_manager, llm_manager):
 """
 Initialize the RAGAgent.

 Args:
 state_manager (StateManager): Manages the state of the agent.
 llm_manager (MultiLLMManager): Manages interactions with multiple
LLMs.
 """
 self.state_manager = state_manager
 self.llm_manager = llm_manager
 self.rag =
RAGIntegration(search_api_url="https://example.com/api/search")

 def execute(self, task_name, parameters):
 """
 Execute a retrieval-augmented generation task.

 Args:
 task_name (str): The name of the task to execute.
 parameters (dict): Parameters required for the task, including:
 - query: The retrieval query for external data.
 - model_name: The name of the LLM to use.
 - prompt: The base prompt for the LLM.

 Returns:
 dict: The result of the retrieval-augmented generation task.
 """
 # Retrieve context documents
 query = parameters["query"]
 print(f"Executing RAG task '{task_name}' with query: {query}")
 context_documents = self.rag.retrieve_documents(query)

 # Augment prompt with retrieved context
 augmented_prompt = self._augment_prompt(parameters["prompt"],
context_documents)

 # Generate response using the specified LLM
 model_name = parameters["model_name"]
 response = self.llm_manager.generate(
 provider="openai", # Example provider; can be dynamic
 model=model_name,
 prompt=augmented_prompt,
 max_tokens=parameters.get("max_tokens", 150)
)

 # Update state
 self.state_manager.update_state(task_name, {
 "query": query,
 "context_documents": context_documents,
 "response": response
 })

```

```

 return {"status": "completed", "response": response, "context":
context_documents}

def __augment_prompt(self, base_prompt, context_documents):
 """
 Combine the base prompt with retrieved context documents.

 Args:
 base_prompt (str): The base prompt for the LLM.
 context_documents (list): A list of retrieved context documents.

 Returns:
 str: The augmented prompt.
 """
 context_texts = " ".join([doc["content"] for doc in
context_documents])
 return f"{base_prompt}\nContext: {context_texts}"

```

---

## Key Features of `rag_agent.py`:

1. **Retrieval-Augmented Generation:**
    - Retrieves external context to enhance the accuracy and relevance of LLM outputs.
  2. **Contextual Prompt Augmentation:**
    - Dynamically combines retrieved data with the base prompt for improved response quality.
  3. **State Management:**
    - Tracks query results, context documents, and generated responses.
  4. **Integration with RAG Systems:**
    - Leverages the `RAGIntegration` module to interact with external search APIs.
- 

## Usage Example:

```

python
Copy code
from tokenless.agents.rag_agent import RAGAgent
from tokenless.state_manager import StateManager
from tokenless.multi_llm_manager import MultiLLMManager

Initialize required components
state_manager = StateManager()
llm_manager = MultiLLMManager()
rag_agent = RAGAgent(state_manager, llm_manager)

Execute a RAG task
parameters = {
 "query": "Latest advancements in AI",
 "model_name": "gpt-4",
 "prompt": "Based on the latest research, summarize the key points.",

```



```

 "max_tokens": 200
 }

result = rag_agent.execute("generate_summary_with_context", parameters)
print("RAG Agent Result:", result)

```

---

This file equips the **RAGAgent** to handle tasks requiring external data retrieval and context augmentation, ensuring high-quality outputs. Let me know when you're ready for the next file, and I'll proceed with `tokenless/agents/explainability_agent.py`.

**You said:**

Note: All Agentic operations, including state changes, must be logged and stored in Neo4J (I believe I saw this code in plan already). Next file.

**ChatGPT said:**

### File 13: `tokenless/agents/explainability_agent.py`

```

python
Copy code
"""
Explainability Agent
=====
The ExplainabilityAgent is responsible for providing insights into the
behavior of AI models,
leveraging tools like SHAP and LIME for explainable AI.

Classes:
- ExplainabilityAgent: Implements model explainability workflows.
"""

from tokenless.explainability import Explainability

class ExplainabilityAgent:
 """
 ExplainabilityAgent uses SHAP and LIME to provide interpretable
 explanations for model predictions.
 """

 def __init__(self, state_manager, llm_manager, neo4j_integration):
 """
 Initialize the ExplainabilityAgent.

 Args:

```

```

 state_manager (StateManager): Manages the state of the agent.
 llm_manager (MultiLLMManager): Manages interactions with multiple
LLMs.
 neo4j_integration (Neo4JIntegration): Logs operations and results
to Neo4J.
 """
 self.state_manager = state_manager
 self.llm_manager = llm_manager
 self.neo4j = neo4j_integration
 self.explainer = Explainability()

def execute(self, task_name, parameters):
 """
 Execute an explainability task.

 Args:
 task_name (str): The name of the task to execute.
 parameters (dict): Parameters required for the task, including:
 - input_data: The data to be explained.
 - model_name: The name of the LLM or ML model to explain.

 Returns:
 dict: The result of the explainability task.
 """
 # Validate input
 if "input_data" not in parameters:
 raise ValueError("Missing required parameter: 'input_data'")

 input_data = parameters["input_data"]
 model_name = parameters["model_name"]
 explanation_method = parameters.get("method", "shap")

 # Perform explanation
 if explanation_method == "shap":
 explanation = self.explainer.explain_shap(input_data)
 elif explanation_method == "lime":
 explanation = self.explainer.explain_lime(input_data)
 else:
 raise ValueError(f"Unsupported explanation method: {explanation_method}")

 # Log operation to Neo4J
 self.neo4j.create_node(
 label="ExplainabilityTask",
 properties={
 "task_name": task_name,
 "input_data": str(input_data),
 "explanation_method": explanation_method,
 "explanation_result": str(explanation)
 }
)

 # Update state
 self.state_manager.update_state(task_name, {
 "input_data": input_data,
 "explanation_method": explanation_method,
 "explanation_result": explanation
 })

```

```
 })

 return {"status": "completed", "explanation": explanation}
```

---

## Key Features of `explainability_agent.py`:

1. **Explainable AI (XAI):**
    - Leverages SHAP and LIME to provide interpretable insights into model predictions.
  2. **Neo4J Logging:**
    - Logs task details and results to the Neo4J knowledge graph for traceability and auditability.
  3. **State Management:**
    - Tracks input data, chosen explanation method, and results via the `StateManager`.
  4. **Extensibility:**
    - Supports additional explanation methods in the future.
- 

## Usage Example:

```
python
Copy code
from tokenless.agents.explainability_agent import ExplainabilityAgent
from tokenless.state_manager import StateManager
from tokenless.multi_llm_manager import MultiLLMManager
from tokenless.neo4j_integration import Neo4JIntegration

Initialize required components
state_manager = StateManager()
llm_manager = MultiLLMManager()
neo4j_integration = Neo4JIntegration(uri="bolt://localhost:7687",
user="neo4j", password="password")
explainability_agent = ExplainabilityAgent(state_manager, llm_manager,
neo4j_integration)

Execute an explainability task
parameters = {
 "input_data": [0.5, 1.2, 3.4],
 "model_name": "gpt-4",
 "method": "shap"
}

result = explainability_agent.execute("explain_prediction", parameters)
print("Explainability Agent Result:", result)
```

---

This file ensures that all **ExplainabilityAgent** operations, including state changes and task results, are logged into Neo4J for comprehensive tracking and analysis. Let me know when you're ready for the next file, and I'll proceed with `tokenless/utils/__init__.py`.

**You said:**

Next file

**ChatGPT said:**

## File 14: tokenless/utils/init.py

```
python
Copy code
"""
Utilities Package Initialization
=====
This package provides utility modules for the Tokenless library, including:
- LoggingUtility: Centralized logging across the library.
- MonitoringUtility: Tools for tracking system performance and resource
usage.
- DeploymentUtility: Functions for deploying Tokenless applications across
cloud platforms.

Modules:
- logging.py
- monitoring.py
- deployment.py
"""

from .logging import LoggingUtility
from .monitoring import MonitoringUtility
from .deployment import DeploymentUtility

__all__ = ["LoggingUtility", "MonitoringUtility", "DeploymentUtility"]
```

---

### Purpose of `__init__.py` in `utils` Package:

- 1. Centralized Access:**
    - Provides a unified interface for importing all utility modules, such as `LoggingUtility`, `MonitoringUtility`, and `DeploymentUtility`.
  - 2. Scalability:**
    - Simplifies the addition of new utility modules by centralizing imports.
  - 3. Consistency:**
    - Ensures a clean and organized structure for the utility package.
-

## Usage Example:

```
python
Copy code
from tokenless.utils import LoggingUtility, MonitoringUtility,
DeploymentUtility

Initialize utilities
logger = LoggingUtility(log_file="tokenless.log")
monitor = MonitoringUtility()
deployment = DeploymentUtility()

Log an event
logger.log_event("Tokenless library initialized.")

Monitor system performance
system_metrics = monitor.collect_metrics()
print("System Metrics:", system_metrics)

Deploy to AWS
deployment.deploy_to_aws(app_name="tokenless_app", region="us-east-1")
```

---

This file organizes utility modules for the **Tokenless** library, ensuring easy access to critical functionality like logging, monitoring, and deployment. Let me know when you're ready for the next file, and I'll proceed with `tokenless/utils/logging.py`.

**You said:**

Next file.

**ChatGPT said:**

## File 15: tokenless/utils/logging.py

```
python
Copy code
"""
Logging Utility
=====
This module provides centralized logging functionality for the Tokenless
library.

Classes:
- LoggingUtility: A wrapper for Python's logging module, with customizable
options
 for logging events, warnings, and errors.
```

```

"""

import logging

class LoggingUtility:
 """
 LoggingUtility provides centralized logging for the Tokenless library,
 ensuring consistent and structured logs across all components.
 """

 def __init__(self, log_file="tokenless.log", level=logging.INFO):
 """
 Initialize the LoggingUtility.

 Args:
 log_file (str): The file where logs will be saved.
 level (int): The logging level (e.g., logging.INFO,
logging.DEBUG).
 """
 logging.basicConfig(
 filename=log_file,
 level=level,
 format="%(asctime)s - %(levelname)s - %(message)s",
 datefmt="%Y-%m-%d %H:%M:%S",
)
 self.logger = logging.getLogger()

 def log_event(self, message):
 """
 Log a general event.

 Args:
 message (str): The event message to log.
 """
 self.logger.info(message)

 def log_warning(self, message):
 """
 Log a warning event.

 Args:
 message (str): The warning message to log.
 """
 self.logger.warning(message)

 def log_error(self, message):
 """
 Log an error event.

 Args:
 message (str): The error message to log.
 """
 self.logger.error(message)

 def log_critical(self, message):
 """

```

```
 Log a critical event.

 Args:
 message (str): The critical event message to log.
 """
 self.logger.critical(message)

def log_debug(self, message):
 """
 Log a debug-level message.

 Args:
 message (str): The debug message to log.
 """
 self.logger.debug(message)
```

---

## Key Features of `logging.py`:

1. **Centralized Logging:**
    - o Ensures all logging operations within the Tokenless library follow a consistent format and structure.
  2. **Customizable Logging Levels:**
    - o Supports various logging levels: `INFO`, `WARNING`, `ERROR`, `CRITICAL`, and `DEBUG`.
  3. **Timestamped Logs:**
    - o Automatically includes timestamps in log messages for better traceability.
  4. **Extensibility:**
    - o Easily integrates with external log aggregation systems or cloud logging services.
- 

## Usage Example:

```
python
Copy code
from tokenless.utils.logging import LoggingUtility

Initialize the logger
logger = LoggingUtility(log_file="tokenless.log", level=logging.DEBUG)

Log events
logger.log_event("Tokenless library initialized.")
logger.log_warning("This is a warning message.")
logger.log_error("An error occurred during task execution.")
logger.log_critical("Critical failure detected!")
logger.log_debug("Debugging tokenless framework initialization.")
```

---

This file provides a robust logging utility for the **Tokenless** library, ensuring consistent and traceable logging across all operations. Let me know when you're ready for the next file, and I'll proceed with `tokenless/utils/monitoring.py`.

**You said:**

Next file.

**ChatGPT said:**

## **File 16: tokenless/utils/monitoring.py**

```
python
Copy code
"""
Monitoring Utility
=====
This module provides tools for tracking system performance, resource usage,
and runtime metrics.

Classes:
- MonitoringUtility: Collects system metrics and monitors runtime
performance.
"""

import psutil
import time

class MonitoringUtility:
 """
 MonitoringUtility provides methods to collect system metrics and monitor
 the performance of tasks executed within the Tokenless library.
 """

 def __init__(self):
 """
 Initialize the MonitoringUtility.
 """
 self.start_time = None

 def start_timer(self):
 """
 Start a timer to measure task execution time.
 """
 self.start_time = time.time()

 def stop_timer(self):
 """
 Stop the timer and return the elapsed time.

 Returns:
 float: The elapsed time in seconds.
 """
```



```

 """
 if self.start_time is None:
 raise ValueError("Timer has not been started.")
 elapsed_time = time.time() - self.start_time
 self.start_time = None
 return elapsed_time

 def collect_metrics(self):
 """
 Collect system performance metrics.

 Returns:
 dict: A dictionary containing system metrics, including CPU
usage,
 memory usage, and disk usage.
 """
 metrics = {
 "cpu_percent": psutil.cpu_percent(interval=1),
 "memory_percent": psutil.virtual_memory().percent,
 "disk_percent": psutil.disk_usage('/').percent,
 "active_threads": len(psutil.Process().threads()),
 }
 return metrics

 def log_metrics(self, logger):
 """
 Log the collected system metrics using the provided logger.

 Args:
 logger (LoggingUtility): An instance of the LoggingUtility class.
 """
 metrics = self.collect_metrics()
 logger.log_event(f"System Metrics: {metrics}")

```

---

## Key Features of `monitoring.py`:

1. **Performance Monitoring:**
    - Tracks CPU, memory, disk usage, and active threads for runtime diagnostics.
  2. **Task Timer:**
    - Measures the execution time of tasks with a simple start-stop timer mechanism.
  3. **Integration with Logging:**
    - Easily integrates with `LoggingUtility` to log collected metrics for audit trails.
- 

## Usage Example:

```

python
Copy code
from tokenless.utils.monitoring import MonitoringUtility
from tokenless.utils.logging import LoggingUtility

```

```

Initialize monitoring and logging utilities
monitor = MonitoringUtility()
logger = LoggingUtility(log_file="tokenless.log")

Start monitoring a task
monitor.start_timer()

Simulate a task
for _ in range(1000000):
 pass # Simulated workload

Stop the timer and log the execution time
elapsed_time = monitor.stop_timer()
logger.log_event(f"Task completed in {elapsed_time:.2f} seconds.")

Collect and log system metrics
system_metrics = monitor.collect_metrics()
logger.log_event(f"System Metrics: {system_metrics}")

```

---

This file provides essential monitoring tools for the **Tokenless** library, ensuring visibility into system performance and resource usage during runtime. Let me know when you're ready for the next file, and I'll proceed with `tokenless/utils/deployment.py`.

**You said:**

Next file.

**ChatGPT said:**

## File 17: tokenless/utils/deployment.py

```

python
Copy code
"""
Deployment Utility
=====
This module provides tools for deploying Tokenless applications to cloud
platforms, including AWS, GCP, and Azure.

Classes:
- DeploymentUtility: Handles deployment processes across supported cloud
platforms.
"""

import boto3
from google.cloud import storage

```

```

from azure.storage.blob import BlobServiceClient

class DeploymentUtility:
 """
 DeploymentUtility provides methods for deploying Tokenless applications
 to various cloud platforms.
 """

 def deploy_to_aws(self, app_name, region="us-east-1", s3_bucket=None,
lambda_function=None):
 """
 Deploy the Tokenless application to AWS using S3 and Lambda.

 Args:
 app_name (str): The name of the application.
 region (str): The AWS region to deploy to.
 s3_bucket (str): The S3 bucket name for storage (optional).
 lambda_function (str): The Lambda function to deploy (optional).

 Returns:
 dict: Details of the deployment.
 """
 s3 = boto3.client("s3", region_name=region)
 deployment_info = {}

 # Deploy to S3
 if s3_bucket:
 s3.create_bucket(Bucket=s3_bucket)
 deployment_info["s3_bucket"] = f"Created S3 bucket: {s3_bucket}"

 # Deploy Lambda Function
 if lambda_function:
 lambda_client = boto3.client("lambda", region_name=region)
 response = lambda_client.create_function(**lambda_function)
 deployment_info["lambda_function"] = response

 return deployment_info

 def deploy_to_gcp(self, app_name, bucket_name, project_id):
 """
 Deploy the Tokenless application to GCP using Google Cloud Storage.

 Args:
 app_name (str): The name of the application.
 bucket_name (str): The Cloud Storage bucket name.
 project_id (str): The Google Cloud project ID.

 Returns:
 str: Confirmation message.
 """
 client = storage.Client(project=project_id)
 bucket = client.bucket(bucket_name)

 if not bucket.exists():
 bucket.create(location="US")
 return f"Created GCP bucket: {bucket_name}"

```

```

 return f"Bucket already exists: {bucket_name}"

 def deploy_to_azure(self, app_name, storage_account, container_name,
connection_string):
 """
 Deploy the Tokenless application to Azure using Blob Storage.

 Args:
 app_name (str): The name of the application.
 storage_account (str): The Azure storage account name.
 container_name (str): The Blob container name.
 connection_string (str): The connection string for the storage
account.

 Returns:
 str: Confirmation message.
 """
 blob_service_client =
BlobServiceClient.from_connection_string(connection_string)
 container_client =
blob_service_client.get_container_client(container_name)

 if not container_client.exists():
 container_client.create_container()
 return f"Created Azure Blob container: {container_name}"
 return f"Container already exists: {container_name}"

```

---

## Key Features of `deployment.py`:

1. **Multi-Cloud Deployment:**
    - Supports deployment to AWS (S3, Lambda), GCP (Google Cloud Storage), and Azure (Blob Storage).
  2. **Customizable Parameters:**
    - Allows configuration of deployment options, such as region, bucket names, and connection strings.
  3. **Extensibility:**
    - Can be expanded to include additional services or cloud providers.
- 

## Usage Example:

```

python
Copy code
from tokenless.utils.deployment import DeploymentUtility

Initialize the deployment utility
deployment = DeploymentUtility()

Deploy to AWS
aws_response = deployment.deploy_to_aws(
 app_name="tokenless_app",

```

```

 region="us-east-1",
 s3_bucket="my-tokenless-app-bucket"
)
 print("AWS Deployment Response:", aws_response)

Deploy to GCP
gcp_response = deployment.deploy_to_gcp(
 app_name="tokenless_app",
 bucket_name="my-tokenless-app-bucket",
 project_id="my-gcp-project-id"
)
print("GCP Deployment Response:", gcp_response)

Deploy to Azure
azure_response = deployment.deploy_to_azure(
 app_name="tokenless_app",
 storage_account="myazurestorageaccount",
 container_name="my-container",

connection_string="DefaultEndpointsProtocol=https;AccountName=myazurestoragea
ccount;..."
)
print("Azure Deployment Response:", azure_response)

```

---

This file equips the **Tokenless** library with flexible deployment tools, supporting seamless integration with major cloud platforms. Let me know when you're ready for the next file, and I'll proceed with **examples/efpl\_usage.py**.

**You said:**

Next file.

**ChatGPT said:**

## File 18: examples/efpl\_usage.py

```

python
Copy code
"""
EFPL Usage Example
=====
This script demonstrates how to use the EFPL (English Functional Programming
Language)
Interpreter within the Tokenless library to execute complex workflows.
"""

```

```

from tokenless.efpl_interpreter import EFPLInterpreter
from tokenless.agentic_framework import AgenticFramework

Initialize the Agentic Framework
framework = AgenticFramework()

Initialize the EFPL Interpreter
interpreter = EFPLInterpreter(framework)

Define an EFPL script
script = """
recursive summarize {"model_name": "gpt-4", "prompt": "Summarize the latest
trends in AI technology.", "max_iterations": 3, "completion_phrases":
["done", "complete"]}
rag retrieve_documents {"query": "Future of AI in healthcare", "model_name":
"claude", "prompt": "Using the retrieved documents, provide a detailed
analysis."}
"""

Validate the EFPL script
if interpreter.validate_script(script):
 print("Script validation successful.")
else:
 print("Script validation failed.")
 exit()

Execute the EFPL script
results = interpreter.interpret(script)

Display results
for i, result in enumerate(results, start=1):
 print(f"Task {i} Result:")
 print(result)

```

---

## Key Features of `efpl_usage.py`:

1. **EFPL Script Definition:**
    - Demonstrates how to define a workflow in plain English using EFPL.
  2. **Script Validation:**
    - Validates the script for syntax and logical errors before execution.
  3. **Task Execution:**
    - Executes multiple tasks defined in the script, utilizing `RecursiveAgent` and `RAGAgent`.
  4. **Result Display:**
    - Provides detailed outputs for each executed task.
- 

## Usage Example Output:

When running the script, you might see outputs like this:

```
plaintext
Copy code
Script validation successful.
Task 1 Result:
{'status': 'completed', 'result': 'AI is advancing rapidly in healthcare...',
'iterations': 3, 'history': [...]}
Task 2 Result:
{'status': 'completed', 'response': 'Based on the retrieved documents, the
future of AI in healthcare includes...', 'context': [...]}
```

---

This file demonstrates the power and flexibility of EFPL scripting within the **Tokenless** library. Let me know when you're ready for the next file, and I'll proceed with **examples/multi\_llm\_example.py**.

**You said:**

Next file.

**ChatGPT said:**

## File 19: examples/multi\_llm\_example.py

```
python
Copy code
"""
Multi-LLM Example
=====
This script demonstrates how to use the MultiLLMManager within the Tokenless
library
to generate text using multiple Large Language Models (LLMs).
"""

from tokenless.multi_llm_manager import MultiLLMManager

Initialize the Multi-LLM Manager
llm_manager = MultiLLMManager()

Define prompts for each provider
prompts = {
 "openai": "What are the latest advancements in quantum computing?",
 "anthropic": "Explain the ethical considerations in AI development.",
 "gemini": "How does reinforcement learning work in robotics?",
 "llama": "What are the core principles of machine learning?",
}

Generate responses from multiple LLMs
```

```

responses = {}

OpenAI GPT-4
responses["openai"] = llm_manager.generate(
 provider="openai",
 model="gpt-4",
 prompt=prompts["openai"],
 max_tokens=200
)

Anthropic Claude
responses["anthropic"] = llm_manager.generate(
 provider="anthropic",
 model="claude",
 prompt=prompts["anthropic"],
 max_tokens=200
)

Google Gemini
responses["gemini"] = llm_manager.generate(
 provider="gemini",
 model="gemini-1",
 prompt=prompts["gemini"],
 max_tokens=200
)

Meta LLaMA
responses["llama"] = llm_manager.generate(
 provider="llama",
 model="llama-2",
 prompt=prompts["llama"],
 max_tokens=200
)

Display responses
for provider, response in responses.items():
 print(f"{provider.upper()} Response:")
 print(response)
 print("-" * 40)

```

---

## Key Features of `multi_llm_example.py`:

1. **Multi-LLM Integration:**
    - Demonstrates how to interact with OpenAI's GPT-4, Anthropic's Claude, Google's Gemini, and Meta's LLaMA.
  2. **Dynamic Prompts:**
    - Custom prompts for each provider showcase the flexibility of `MultiLLMManager`.
  3. **Response Aggregation:**
    - Collects and displays responses from all integrated LLMs in a unified manner.
-



## Expected Output Example:

When running the script, you might see output like this:

```
plaintext
Copy code
OPENAI Response:
Quantum computing advancements include error correction and qubit
scalability.

ANTHROPIC Response:
Ethical considerations in AI include bias mitigation, transparency, and
accountability.

GEMINI Response:
Reinforcement learning involves training agents to make sequential decisions.

LLAMA Response:
The core principles of machine learning include supervised learning and
optimization.

```

---

This file highlights the seamless integration of multiple LLMs within the **Tokenless** library. Let me know when you're ready for the next file, and I'll proceed with **tests/test\_agentic\_framework.py**.

**You said:**

Next file.

**ChatGPT said:**

## File 20: tests/test\_agentic\_framework.py

```
python
Copy code
"""
Test: Agentic Framework
=====
This script provides unit tests for the AgenticFramework class in the
Tokenless library.
"""

import unittest
from tokenless.agentic_framework import AgenticFramework
```

```

from tokenless.state_manager import StateManager
from tokenless.multi_llm_manager import MultiLLMManager
from tokenless.neo4j_integration import Neo4JIntegration

class TestAgenticFramework(unittest.TestCase):
 """
 Unit tests for the AgenticFramework class.
 """

 @classmethod
 def setUpClass(cls):
 """
 Set up resources for all tests.
 """
 cls.state_manager = StateManager()
 cls.llm_manager = MultiLLMManager()
 cls.neo4j = Neo4JIntegration(uri="bolt://localhost:7687",
user="neo4j", password="password")
 cls.framework = AgenticFramework()

 def test_add_custom_agent(self):
 """
 Test adding a custom agent to the framework.
 """
 class CustomAgent:
 def execute(self, task_name, parameters):
 return {"status": "completed", "result": f"Custom agent
executed {task_name}"}

 self.framework.add_custom_agent("custom", CustomAgent())
 result = self.framework.execute_task("custom", "test_task", {})
 self.assertEqual(result["status"], "completed")
 self.assertIn("Custom agent executed", result["result"])

 def test_execute_recursive_agent(self):
 """
 Test executing a task with the RecursiveAgent.
 """
 parameters = {
 "model_name": "gpt-4",
 "prompt": "Summarize this text.",
 "max_iterations": 3,
 "completion_phrases": ["done", "complete"]
 }
 result = self.framework.execute_task("recursive", "summarize_text",
parameters)
 self.assertIn(result["status"], ["completed", "incomplete"])

 def test_execute_rag_agent(self):
 """
 Test executing a task with the RAGAgent.
 """
 parameters = {
 "query": "Future of AI",
 "model_name": "claude",
 "prompt": "Use retrieved documents to provide an analysis.",

```

```

 "max_tokens": 150
 }
 result = self.framework.execute_task("rag", "analyze_ai_future",
parameters)
 self.assertEqual(result["status"], "completed")
 self.assertIn("response", result)

 def test_execute_explainability_agent(self):
 """
 Test executing a task with the ExplainabilityAgent.
 """
 parameters = {
 "input_data": [1, 2, 3],
 "model_name": "gpt-4",
 "method": "shap"
 }
 result = self.framework.execute_task("explainability",
"explain_prediction", parameters)
 self.assertEqual(result["status"], "completed")
 self.assertIn("explanation", result)

 def test_neo4j_logging(self):
 """
 Test that tasks are logged to Neo4J.
 """
 parameters = {
 "model_name": "gpt-4",
 "prompt": "Log this task to Neo4J.",
 "max_iterations": 2,
 "completion_phrases": ["done"]
 }
 result = self.framework.execute_task("recursive", "log_task_test",
parameters)
 query_result = self.neo4j.query_nodes("MATCH (n:TaskExecution
{task_name: 'log_task_test'}) RETURN n")
 self.assertTrue(len(query_result) > 0)

 @classmethod
 def tearDownClass(cls):
 """
 Clean up resources after all tests.
 """
 cls.neo4j.close()

if __name__ == "__main__":
 unittest.main()

```

---

## Key Features of `test_agentic_framework.py`:

1. **Comprehensive Testing:**
  - Tests the core functionalities of the `AgenticFramework`, including agent execution and task logging.
2. **Custom Agent Validation:**

- Ensures the framework can dynamically handle custom agents.
  - 3. **Neo4J Logging Verification:**
    - Confirms that tasks are properly logged to the Neo4J graph database.
  - 4. **Robust Test Coverage:**
    - Covers RecursiveAgent, RAGAgent, and ExplainabilityAgent workflows.
- 

## Usage Instructions:

Run the test script using `unittest` to verify the framework's functionality:

```
bash
Copy code
python -m unittest tests/test_agentic_framework.py
```

---

This file ensures that the **AgenticFramework** operates reliably and adheres to its design specifications. Let me know when you're ready for the next file, and I'll proceed with **tests/test\_efpl\_interpreter.py**.

**You said:**

Next file.

**ChatGPT said:**

## File 21: tests/test\_efpl\_interpreter.py

```
python
Copy code
"""
Test: EFPL Interpreter
=====
This script provides unit tests for the EFPLInterpreter class in the
Tokenless library.
"""

import unittest
from tokenless.efpl_interpreter import EFPLInterpreter
from tokenless.agentic_framework import AgenticFramework

class TestEFPLInterpreter(unittest.TestCase):
 """
```

```

Unit tests for the EFPLInterpreter class.
"""

@classmethod
def setUpClass(cls):
 """
 Set up resources for all tests.
 """
 cls.framework = AgenticFramework()
 cls.interpreter = EFPLInterpreter(cls.framework)

def test_valid_script(self):
 """
 Test a valid EFPL script.
 """
 script = """
recursive summarize {"model_name": "gpt-4", "prompt": "Summarize this
article.", "max_iterations": 3}
rag retrieve_documents {"query": "AI ethics", "model_name": "claude",
"prompt": "Analyze retrieved content."}
"""
 is_valid = self.interpreter.validate_script(script)
 self.assertTrue(is_valid)

def test_invalid_script(self):
 """
 Test an invalid EFPL script with syntax errors.
 """
 script = """
recursive summarize {"model_name": "gpt-4", "prompt": "Summarize this
article." # Missing closing bracket
rag retrieve_documents {"query": "AI ethics"}
"""
 is_valid = self.interpreter.validate_script(script)
 self.assertFalse(is_valid)

def test_interpret_script(self):
 """
 Test interpreting and executing a valid EFPL script.
 """
 script = """
recursive summarize {"model_name": "gpt-4", "prompt": "Summarize the
AI advancements.", "max_iterations": 3}
rag retrieve_documents {"query": "AI in healthcare", "model_name":
"claude", "prompt": "Provide a detailed analysis."}
"""
 results = self.interpreter.interpret(script)
 self.assertEqual(len(results), 2)
 self.assertIn("status", results[0])
 self.assertIn("response", results[1])

def test_interpret_script_with_error(self):
 """
 Test interpreting a script with an invalid command.
 """
 script = """

```

```

 recursive_summarize {"model_name": "gpt-4", "prompt": "Summarize AI
progress.", "max_iterations": 3}
 invalid_command {"model_name": "gpt-4", "prompt": "This command
should fail."}
 """
 with self.assertRaises(ValueError):
 self.interpreter.interpret(script)

 @classmethod
 def tearDownClass(cls):
 """
 Clean up resources after all tests.
 """
 pass

if __name__ == "__main__":
 unittest.main()

```

---

## Key Features of `test_efpl_interpreter.py`:

1. **Validation Testing:**
    - Ensures that EFPL scripts are correctly validated for syntax and logical errors.
  2. **Interpretation Testing:**
    - Verifies the execution of valid EFPL scripts and captures expected results.
  3. **Error Handling:**
    - Confirms that invalid commands or improperly formatted scripts raise appropriate exceptions.
  4. **Comprehensive Coverage:**
    - Covers both happy paths and failure scenarios for the `EFPLInterpreter`.
- 

## Usage Instructions:

Run the test script using `unittest` to validate the EFPL interpreter:

```

bash
Copy code
python -m unittest tests/test_efpl_interpreter.py

```

---

This file ensures the **EFPLInterpreter** operates reliably and adheres to its design specifications. Let me know when you're ready for the next file, and I'll proceed with `tests/test_multi_llm_manager.py`.

**You said:**

Next file.

ChatGPT said:

## File 22: tests/test\_multi\_llm\_manager.py

```
python
Copy code
"""
Test: Multi-LLM Manager
=====
This script provides unit tests for the MultiLLMManager class in the
Tokenless library.
"""

import unittest
from tokenless.multi_llm_manager import MultiLLMManager

class TestMultiLLMManager(unittest.TestCase):
 """
 Unit tests for the MultiLLMManager class.
 """

 @classmethod
 def setUpClass(cls):
 """
 Set up resources for all tests.
 """
 cls.llm_manager = MultiLLMManager()

 def test_list_models(self):
 """
 Test listing all available models across providers.
 """
 models = self.llm_manager.list_models()
 self.assertIsInstance(models, dict)
 self.assertIn("openai", models)
 self.assertIn("gpt-4", models["openai"])
 self.assertIn("anthropic", models)
 self.assertIn("claude", models["anthropic"])
 self.assertIn("gemini", models)
 self.assertIn("gemini-1", models["gemini"])
 self.assertIn("llama", models)
 self.assertIn("llama-2", models["llama"])

 def test_openai_generate(self):
 """
 Test generating text using OpenAI's GPT-4.
 """
 prompt = "Describe the future of AI in healthcare."
```

```

 response = self.llm_manager.generate(provider="openai", model="gpt-4", prompt=prompt, max_tokens=100)
 self.assertIsInstance(response, str)
 self.assertGreater(len(response), 0)

 def test_anthropic_generate(self):
 """
 Test generating text using Anthropic's Claude.
 """
 prompt = "Explain ethical considerations in AI development."
 response = self.llm_manager.generate(provider="anthropic",
model="claude", prompt=prompt, max_tokens=100)
 self.assertIsInstance(response, str)
 self.assertGreater(len(response), 0)

 def test_gemini_generate(self):
 """
 Test generating text using Google's Gemini.
 """
 prompt = "Summarize the principles of reinforcement learning."
 response = self.llm_manager.generate(provider="gemini",
model="gemini-1", prompt=prompt, max_tokens=100)
 self.assertIsInstance(response, str)
 self.assertGreater(len(response), 0)

 def test_llama_generate(self):
 """
 Test generating text using Meta's LLaMA.
 """
 prompt = "What are the core principles of machine learning?"
 response = self.llm_manager.generate(provider="llama", model="llama-2", prompt=prompt, max_tokens=100)
 self.assertIsInstance(response, str)
 self.assertGreater(len(response), 0)

 def test_invalid_provider(self):
 """
 Test handling of an invalid provider.
 """
 with self.assertRaises(ValueError):
 self.llm_manager.generate(provider="invalid", model="gpt-4",
prompt="Test prompt", max_tokens=50)

 def test_invalid_model(self):
 """
 Test handling of an invalid model.
 """
 with self.assertRaises(ValueError):
 self.llm_manager.generate(provider="openai", model="invalid-
model", prompt="Test prompt", max_tokens=50)

 @classmethod
 def tearDownClass(cls):
 """
 Clean up resources after all tests.
 """
 pass

```



```
if __name__ == "__main__":
 unittest.main()
```

---

### Key Features of `test_multi_llm_manager.py`:

1. **Model Listing Test:**
    - Verifies that the `list_models` method correctly lists all available models across providers.
  2. **Text Generation Tests:**
    - Tests text generation functionality for OpenAI's GPT-4, Anthropic's Claude, Google's Gemini, and Meta's LLaMA.
  3. **Error Handling:**
    - Validates that the library raises appropriate exceptions for invalid providers or models.
  4. **Comprehensive Coverage:**
    - Covers both successful operations and failure scenarios for robust validation.
- 

### Usage Instructions:

Run the test script using `unittest` to validate the `MultiLLMManager`:

```
bash
Copy code
python -m unittest tests/test_multi_llm_manager.py
```

---

This file ensures the **MultiLLMManager** operates reliably, integrating seamlessly with multiple LLMs. Let me know when you're ready for the next file, and I'll proceed with `tests/test_state_manager.py`.

**You said:**

Next file.

ChatGPT said:

### File 23: `tests/test_state_manager.py`

```

python
Copy code
"""
Test: State Manager
=====
This script provides unit tests for the StateManager class in the Tokenless
library.
"""

import unittest
from tokenless.state_manager import StateManager

class TestStateManager(unittest.TestCase):
 """
 Unit tests for the StateManager class.
 """

 @classmethod
 def setUpClass(cls):
 """
 Set up resources for all tests.
 """
 cls.state_manager = StateManager()

 def test_update_and_get_state(self):
 """
 Test updating and retrieving an agent's state.
 """
 agent_id = "agent_1"
 initial_state = {"progress": 50, "status": "in_progress"}
 self.state_manager.update_state(agent_id, initial_state)

 state = self.state_manager.get_state(agent_id)
 self.assertEqual(state, initial_state)

 def test_reset_state(self):
 """
 Test resetting an agent's state.
 """
 agent_id = "agent_2"
 initial_state = {"progress": 100, "status": "completed"}
 self.state_manager.update_state(agent_id, initial_state)

 self.state_manager.reset_state(agent_id)
 state = self.state_manager.get_state(agent_id)
 self.assertEqual(state, {})

 def test_list_all_states(self):
 """
 Test listing all agent states.
 """
 self.state_manager.update_state("agent_1", {"progress": 50, "status":
"in_progress"})
 self.state_manager.update_state("agent_2", {"progress": 100,
"status": "completed"})

```

```

 all_states = self.state_manager.list_all_states()
 self.assertEqual(len(all_states), 2)
 self.assertIn("agent_1", all_states)
 self.assertIn("agent_2", all_states)

 def test_delete_state(self):
 """
 Test deleting an agent's state.
 """
 agent_id = "agent_3"
 self.state_manager.update_state(agent_id, {"progress": 25, "status":
"started"})
 self.state_manager.delete_state(agent_id)

 state = self.state_manager.get_state(agent_id)
 self.assertEqual(state, {})

 def test_thread_safety(self):
 """
 Test thread-safe operations on state manager.
 """
 import threading

 def update_state(agent_id, state):
 self.state_manager.update_state(agent_id, state)

 threads = []
 for i in range(10):
 agent_id = f"agent_{i}"
 state = {"progress": i * 10, "status": "in_progress"}
 thread = threading.Thread(target=update_state, args=(agent_id,
state))
 threads.append(thread)
 thread.start()

 for thread in threads:
 thread.join()

 all_states = self.state_manager.list_all_states()
 self.assertEqual(len(all_states), 10)

 @classmethod
 def tearDownClass(cls):
 """
 Clean up resources after all tests.
 """
 pass

if __name__ == "__main__":
 unittest.main()

```

---

## Key Features of `test_state_manager.py`:

1. **State Update and Retrieval:**
    - Verifies that agent states can be updated and retrieved accurately.
  2. **Reset and Deletion:**
    - Ensures agent states can be reset or deleted without leaving residual data.
  3. **Thread Safety:**
    - Confirms that the `StateManager` operates reliably in multi-threaded environments.
  4. **All States Listing:**
    - Tests the ability to list all stored states for easy debugging and monitoring.
- 

## Usage Instructions:

Run the test script using `unittest` to validate the `StateManager`:

```
bash
Copy code
python -m unittest tests/test_state_manager.py
```

---

This file ensures the **StateManager** operates reliably, supporting thread-safe operations and consistent state management. Let me know when you're ready for the next file, and I'll proceed with `tests/test_neo4j_integration.py`.

**You said:**

Next file.

**ChatGPT said:**

## File 24: tests/test\_neo4j\_integration.py

```
python
Copy code
"""
Test: Neo4J Integration
=====
This script provides unit tests for the Neo4JIntegration class in the
Tokenless library.
"""

import unittest
from tokenless.neo4j_integration import Neo4JIntegration
```

```

class TestNeo4JIntegration(unittest.TestCase):
 """
 Unit tests for the Neo4JIntegration class.
 """

 @classmethod
 def setUpClass(cls):
 """
 Set up resources for all tests.
 """
 cls.neo4j = Neo4JIntegration(uri="bolt://localhost:7687",
user="neo4j", password="password")

 def test_create_node(self):
 """
 Test creating a node in the Neo4J database.
 """
 node_label = "Agent"
 properties = {"id": "agent_1", "name": "RecursiveAgent", "status":
"active"}
 node = self.neo4j.create_node(node_label, properties)

 self.assertIsInstance(node, list)
 self.assertTrue(len(node) > 0)

 def test_query_nodes(self):
 """
 Test querying nodes from the Neo4J database.
 """
 query = "MATCH (n:Agent {id: 'agent_1'}) RETURN n"
 nodes = self.neo4j.query_nodes(query)

 self.assertIsInstance(nodes, list)
 self.assertTrue(len(nodes) > 0)

 def test_create_relationship(self):
 """
 Test creating a relationship between two nodes.
 """
 self.neo4j.create_node("Agent", {"id": "agent_1", "name": "Agent1"})
 self.neo4j.create_node("Agent", {"id": "agent_2", "name": "Agent2"})

 relationship = self.neo4j.create_relationship(
 from_node="agent_1",
 to_node="agent_2",
 relationship_type="COLLABORATES_WITH",
 properties={"strength": 5}
)

 self.assertIsInstance(relationship, list)
 self.assertTrue(len(relationship) > 0)

 def test_update_node(self):
 """
 Test updating a node in the Neo4J database.

```

```

 """
 node_id = "agent_1"
 updates = {"status": "inactive"}
 self.neo4j.update_node(node_id, updates)

 query = "MATCH (n:Agent {id: 'agent_1'}) RETURN n"
 nodes = self.neo4j.query_nodes(query)
 self.assertEqual(nodes[0]["n"]["status"], "inactive")

 def test_delete_node(self):
 """
 Test deleting a node from the Neo4J database.
 """
 node_id = "agent_2"
 self.neo4j.delete_node(node_id)

 query = "MATCH (n:Agent {id: 'agent_2'}) RETURN n"
 nodes = self.neo4j.query_nodes(query)
 self.assertEqual(len(nodes), 0)

 @classmethod
 def tearDownClass(cls):
 """
 Clean up resources after all tests.
 """
 cls.neo4j.close()

if __name__ == "__main__":
 unittest.main()

```

---

## Key Features of `test_neo4j_integration.py`:

1. **Node Operations:**
    - Tests creating, querying, updating, and deleting nodes in the Neo4J database.
  2. **Relationship Management:**
    - Verifies the creation of relationships with properties between nodes.
  3. **Database Cleanup:**
    - Ensures nodes and relationships are properly removed during tests.
  4. **Integration Validation:**
    - Confirms reliable connectivity and operations with the Neo4J Aura Graph DB.
- 

## Usage Instructions:

Run the test script using `unittest` to validate the `Neo4JIntegration`:

```

bash
Copy code
python -m unittest tests/test_neo4j_integration.py

```

---

This file ensures the **Neo4JIntegration** operates reliably, providing robust support for knowledge graph operations. Let me know when you're ready for the next file, and I'll proceed with `docs/installation.md`.

**You said:**

Next file.

**ChatGPT said:**

## File 25: docs/installation.md

```
markdown
Copy code
Installation Guide for Tokenless Library

This document provides step-by-step instructions for installing the
Tokenless library and configuring its dependencies.

Prerequisites

Before installing Tokenless, ensure you have the following:

1. **Python 3.8 or higher**:
 - Verify your Python version:
     ```bash
     python --version
     ```
 - Install Python from [python.org](https://www.python.org/downloads/) if necessary.

2. **Neo4J Aura Graph DB**:
 - Set up a Neo4J database instance.
 - Obtain your **Neo4J URI**, **username**, and **password**.

3. **Cloud Platform Accounts**:
 - **AWS** for S3 or Lambda (optional).
 - **Google Cloud Platform (GCP)** for storage (optional).
 - **Azure** for Blob Storage (optional).

4. **API Keys**:
 - OpenAI: [Obtain API Key](https://platform.openai.com/signup/).
 - Anthropic: [Obtain API Key](https://www.anthropic.com/).
```

- Google Gemini: [Obtain API Key](https://cloud.google.com/).
- Meta LLaMA: [Follow Meta API Docs](https://ai.facebook.com/llama/).

---

## ## Installation Steps

### ### 1. Clone the Repository

Clone the Tokenless repository from GitHub:

```
```bash
git clone https://github.com/yourusername/tokenless.git
cd tokenless
```

2. Install Dependencies

Install the required Python libraries using pip:

```
bash
Copy code
pip install -r requirements.txt
```

3. Set Up API Keys

Configure your environment to use the necessary API keys:

Linux/MacOS

Add the following to your `.bashrc` or `.zshrc` file:

```
bash
Copy code
export OPENAI_API_KEY="your-openai-api-key"
export ANTHROPIC_API_KEY="your-anthropic-api-key"
export GEMINI_API_KEY="your-gemini-api-key"
export LLAMA_API_KEY="your-llama-api-key"
```

Reload your shell:

```
bash
Copy code
source ~/.bashrc
```

Windows

Set the environment variables in Command Prompt:

```
cmd
Copy code
```



```
set OPENAI_API_KEY=your-openai-api-key
set ANTHROPIC_API_KEY=your-anthropic-api-key
set GEMINI_API_KEY=your-gemini-api-key
set LLAMA_API_KEY=your-llama-api-key
```

4. Configure Neo4J Connection

Update your Neo4J connection details in the `AgenticFramework` initialization:

```
python
Copy code
neo4j = Neo4JIntegration(
    uri="bolt://your-neo4j-uri:7687",
    user="your-username",
    password="your-password"
)
```

5. Run Tests

Ensure all components are functioning correctly by running the test suite:

```
bash
Copy code
pytest tests/
```

6. Verify Installation

Run an example script to verify the installation:

```
bash
Copy code
python examples/efpl_usage.py
```

Optional Configurations

AWS, GCP, Azure Integration

- **AWS:** Configure AWS credentials using the AWS CLI:

```
bash
Copy code
aws configure
```

- **GCP:** Authenticate using a service account JSON key:

```
bash
Copy code
export GOOGLE_APPLICATION_CREDENTIALS="path-to-your-service-
account.json"
```

- **Azure:** Set up a storage connection string:

```
bash
Copy code
export AZURE_STORAGE_CONNECTION_STRING="your-azure-connection-string"
```

Troubleshooting

1. **Dependency Issues:** If you encounter errors during installation, ensure you have `pip` updated:

```
bash
Copy code
pip install --upgrade pip
```

2. **Neo4J Connection:** Verify that your Neo4J database is running and accessible at the specified URI.
 3. **API Key Issues:** Double-check that your API keys are correctly set in the environment variables.
-

This completes the installation process for the **Tokenless** library. If you encounter further issues, refer to the [Usage Guide](#) or open an issue in the repository.

```
yaml
Copy code
```

This file provides a comprehensive guide for installing and configuring the **Tokenless** library, ensuring a smooth setup process. Let me know when you're ready for the next file, and I'll proceed with `**docs/usage.md**`.

You said:

File 26: docs/usage.md

markdown

Copy code

Usage Guide for Tokenless Library

This document provides a detailed walkthrough of how to use the **Tokenless** library, with examples for each core module, agent, and utility.

Table of Contents

1. [Using the Agentic Framework](#using-the-agentic-framework)
2. [EFPL Interpreter](#efpl-interpreter)
3. [Multi-LLM Manager](#multi-llm-manager)
4. [State Manager](#state-manager)
5. [Neo4J Integration](#neo4j-integration)
6. [Agents Overview](#agents-overview)
 - Recursive Agent
 - RAG Agent
 - Explainability Agent
7. [Advanced Customization](#advanced-customization)

1. Using the Agentic Framework

The **AgenticFramework** is the central orchestrator for all tasks and agents in the Tokenless library.

Example: Task Execution

```
```python
from tokenless.agentic_framework import AgenticFramework

framework = AgenticFramework()

parameters = {
 "model_name": "gpt-4",
 "prompt": "Summarize the impact of AI on education.",
 "max_iterations": 3,
 "completion_phrases": ["done", "complete"]
}

result = framework.execute_task("recursive", "summarize_ai_impact", parameters)
print("Recursive Agent Result:", result)
```

---

## 2. EFPL Interpreter

The **EFPLInterpreter** allows users to define workflows in plain English.

### Example: Define and Execute a Workflow

```
python
Copy code
from tokenless.efpl_interpreter import EFPLInterpreter
```

```
from tokenless.agentic_framework import AgenticFramework

framework = AgenticFramework()
interpreter = EFPLInterpreter(framework)

script = """
recursive summarize {"model_name": "gpt-4", "prompt": "Summarize the following: AI is
transforming healthcare.", "max_iterations": 3}
rag retrieve_documents {"query": "AI healthcare applications", "model_name": "gemini-1",
"prompt": "Analyze retrieved documents."}
"""

results = interpreter.interpret(script)
print("EFPL Results:", results)
```

---

### 3. Multi-LLM Manager

The **MultiLLMManager** provides a unified interface to interact with multiple LLMs.

#### Example: Generate Text

python

Copy code

```
from tokenless.multi_llm_manager import MultiLLMManager
```

```
llm_manager = MultiLLMManager()
```

```
response = llm_manager.generate(
 provider="openai",
 model="gpt-4",
 prompt="What are the benefits of quantum computing?",
 max_tokens=200
)
```

```
print("OpenAI GPT-4 Response:", response)
```

---

### 4. State Manager

The **StateManager** is used to maintain and retrieve agent states.

#### Example: Update and Retrieve State

python

Copy code

```
from tokenless.state_manager import StateManager
```

```
state_manager = StateManager()
```

```
Update state
```

```
state_manager.update_state("agent_1", {"progress": 50, "status": "in_progress"})
```

```
Retrieve state
```

```
state = state_manager.get_state("agent_1")
print("Agent 1 State:", state)
```

---

## 5. Neo4J Integration

The **Neo4JIntegration** module allows you to store and query task-related data in a graph database.

### Example: Create and Query Nodes

python

Copy code

```
from tokenless.neo4j_integration import Neo4JIntegration

neo4j = Neo4JIntegration(uri="bolt://localhost:7687", user="neo4j", password="password")

Create a node
neo4j.create_node("TaskExecution", {"id": "task_1", "name": "Summarize AI", "status":
"completed"})

Query nodes
nodes = neo4j.query_nodes("MATCH (n:TaskExecution {status: 'completed'}) RETURN n")
print("Queried Nodes:", nodes)

neo4j.close()
```

---

## 6. Agents Overview

### Recursive Agent

#### Example: Iterative Refinement

python

Copy code

```
parameters = {
 "model_name": "gpt-4",
 "prompt": "Summarize AI advancements.",
 "max_iterations": 5,
 "completion_phrases": ["done", "complete"]
}

result = framework.execute_task("recursive", "summarize_ai", parameters)
print("Recursive Agent Result:", result)
```

---

### RAG Agent

#### Example: Contextual Data Retrieval

python

Copy code

```
parameters = {
 "query": "AI applications in finance",
 "model_name": "claude",
 "prompt": "Based on the retrieved documents, analyze AI's impact on finance.",
}
```

```
"max_tokens": 200
}

result = framework.execute_task("rag", "ai_finance_analysis", parameters)
print("RAG Agent Result:", result)
```

---

## **Explainability Agent**

### **Example: Explain Predictions**

```
python
Copy code
parameters = {
 "input_data": [0.8, 0.4, 0.9],
 "model_name": "gpt-4",
 "method": "shap"
}
```

```
result = framework.execute_task("explainability", "explain_predictions", parameters)
print("Explainability Result:", result)
```

---

## **7. Advanced Customization**

### **Adding Custom Agents**

The **AgenticFramework** allows developers to extend functionality by adding custom agents.

```
python
Copy code
class CustomAgent:
 def execute(self, task_name, parameters):
 return {"status": "completed", "result": f"Custom agent executed {task_name}"}

framework.add_custom_agent("custom", CustomAgent())

result = framework.execute_task("custom", "process_data", {"data": "example"})
print("Custom Agent Result:", result)
```

---

### **Extending Multi-LLM Manager**

You can dynamically add new providers and models to the **MultiLLMManager**.

```
python
Copy code
def example_provider_generate(prompt, max_tokens):
 return f"Generated response for prompt: {prompt}"

framework.llm_manager.add_provider("example_provider", {"example_model":
example_provider_generate})

response = framework.llm_manager.generate(
 provider="example_provider",
 model="example_model",
```

```
 prompt="Explain distributed systems.",
 max_tokens=150
)
print("Example Provider Response:", response)
```

---

This document provides practical examples for using and customizing the **Tokenless** library. For more advanced use cases, refer to the [Advanced Configuration](#) document.

yaml

Copy code

---

This file provides comprehensive usage examples for the **\*\*Tokenless\*\*** library. Let me know if you'd like to proceed with any additional files or have questions about specific sections.