**AI Agents Inference Benchmarking**

**1. Overview**

The Omdena Knowledge AI Agents Inference Benchmarking project aims to evaluate the inference performance of AI agents using CrewAI, LangGraph, and AutoGen with Llama models. It assesses the efficiency of text generation tasks based on various input queries by measuring key performance metrics such as latency, memory consumption, and token usage and others.

**2. Objectives**

To simplify the benchmarking process, we selected a scenario where an agentic framework generates a paragraph based on a given input keyword. We then measure the time taken, memory consumption, and tokens used by the LLM within each framework. Finally, a larger model evaluates and rates the generated paragraphs, ensuring a structured assessment of text quality.

**3. Key Components**

**3.1 Text Generation Pipeline**

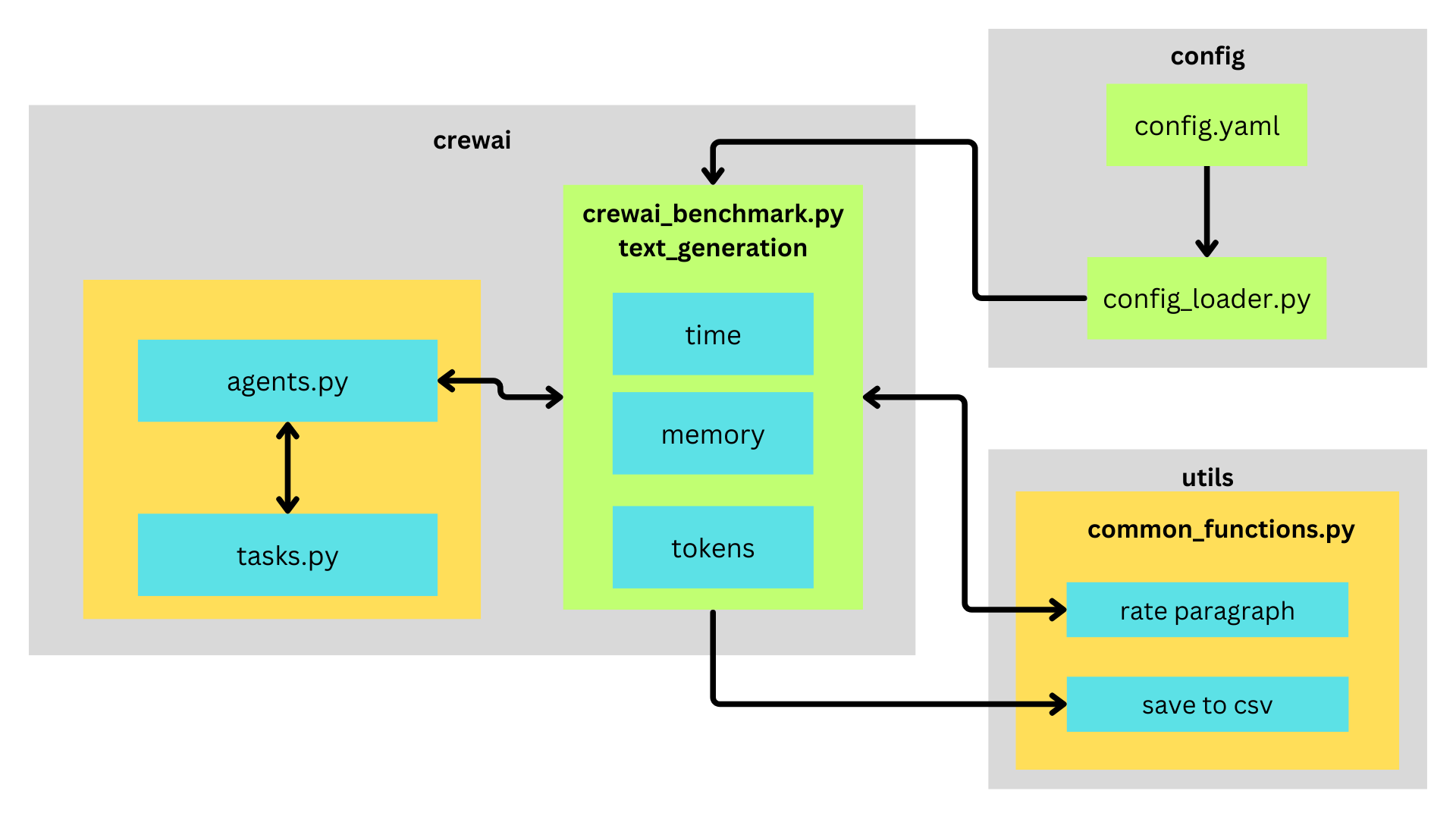
The text\_generation function, implemented in crewai\_benchmark.py, langgraph\_benchmark.py, and autogen\_benchmark.py, orchestrates the text generation benchmarking process. It starts by loading configuration settings and initializing AI models using Groq to optimize performance.

The function executes text generation tasks across CrewAI, LangGraph, and AutoGen, facilitating a comparative analysis of their efficiency. Performance metrics such as latency, memory usage, and token consumption are recorded during execution. Additionally, the generated text is evaluated for quality, ensuring compliance with expected standards. The results are then stored in a CSV file for further comparison and analysis.

**3.2 Framework-Specific Implementations**

**CrewAI Implementation**

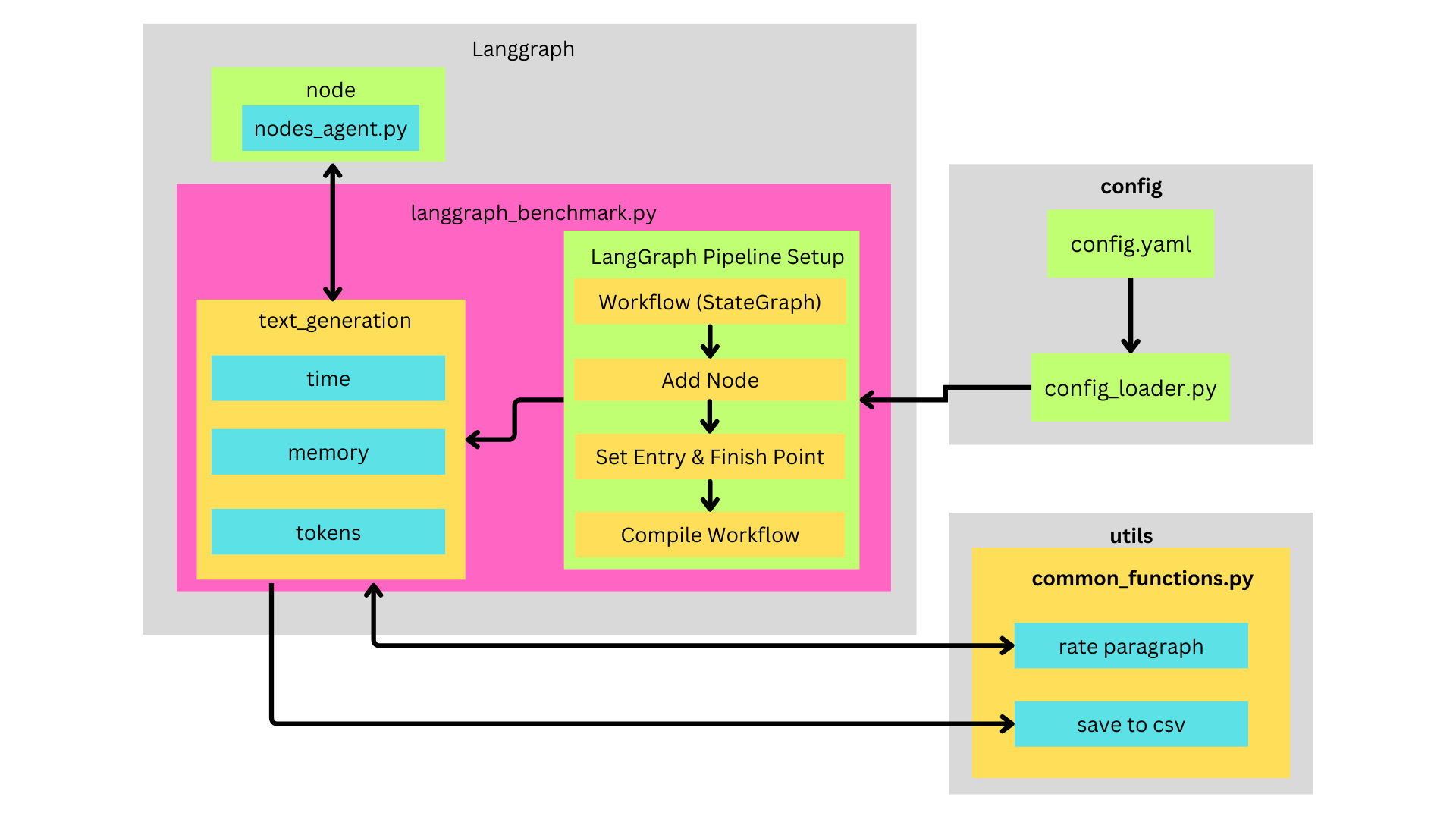
In CrewAI, text generation is handled by a Writer Agent and a Writer Task. A predefined list of input keywords is stored in a configuration file and passed to the text\_generation function. Performance tracking includes measuring total and individual execution time, token consumption, and memory usage.



Once initialized, the Crew component triggers the writer\_agent, powered by the Llama 70B Versatile model, to generate paragraphs for each keyword. Each generated text is stored, while individual time, token count, and memory usage are logged. This process continues until all keywords are processed.

After completing text generation, the system calculates the average memory consumption, total tokens used, and total execution time. The generated paragraphs are then sent to the Llama 90B model for evaluation and rating. This structured process ensures efficient benchmarking of both performance and text quality.

**LangGraph Implementation**

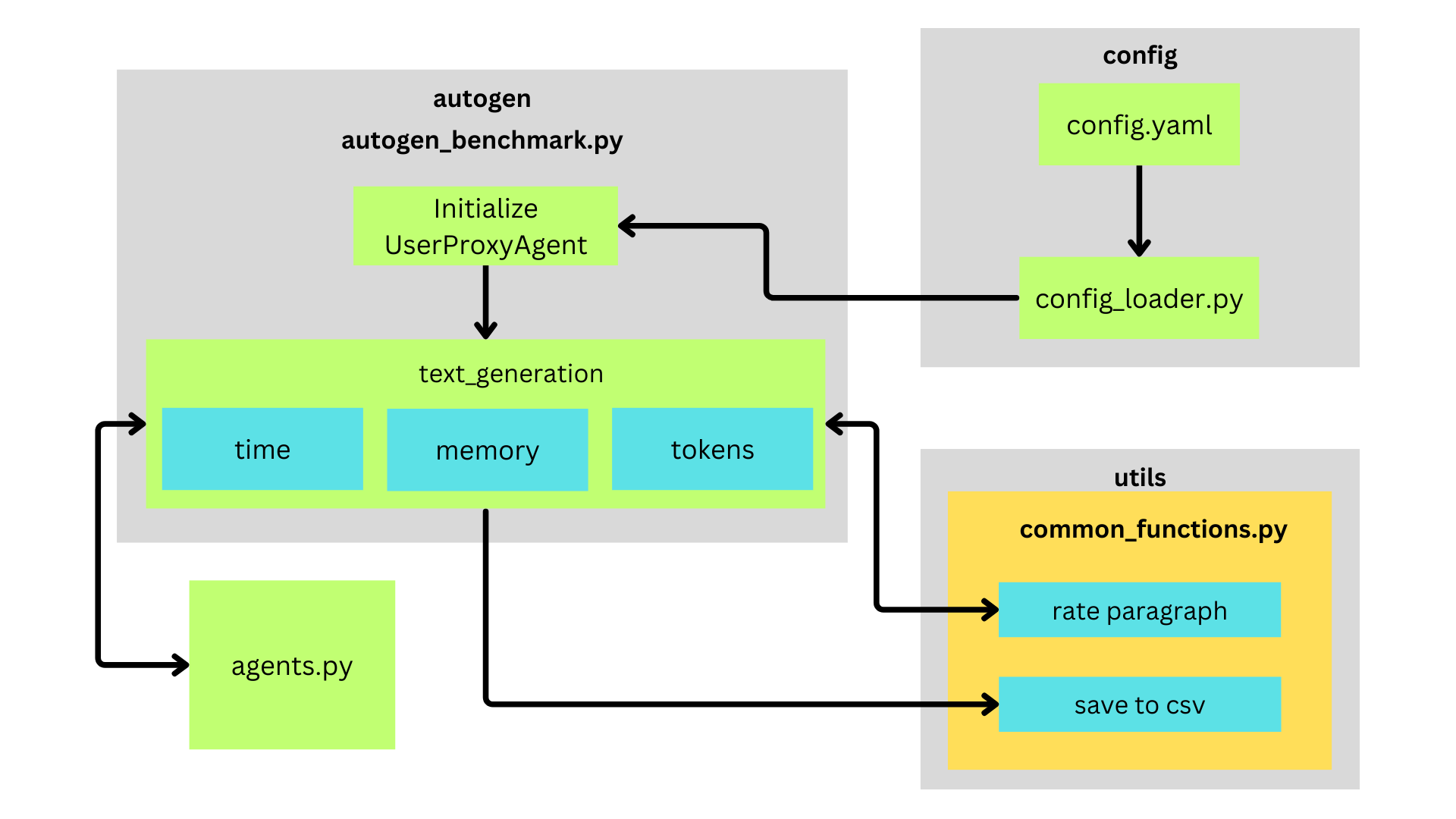


LangGraph follows a structured workflow. First, all configuration settings are loaded to ensure proper initialization. A workflow pipeline is then initiated, determining the sequence of execution for different nodes. Once the workflow is established, the text\_generation function is activated.

Performance tracking involves measuring execution time, token usage, and memory consumption. The writer\_node (equivalent to an agent in other frameworks) powered by Llama 70B Versatile generates paragraphs for each input keyword. Metrics are recorded during this process for accurate benchmarking.

After processing all keywords, the system calculates average memory consumption, total tokens used, and total execution time. The generated paragraphs are finally evaluated and rated by the Llama 90B model, ensuring a systematic benchmarking process.

**AutoGen Implementation**



In AutoGen, the text generation process is structured using a User Proxy Agent and a Writer Agent. It begins by loading all configuration settings, ensuring that parameters such as model selection, API credentials, and text generation constraints are correctly initialized.

Once the configuration is set, a User Proxy Agent (autogen.UserProxyAgent) is created. This agent acts as a controller for the text generation process, handling communication between the user and the AI model. It is configured with a system message, execution settings, and a specified working directory.

The text\_generation function is then activated, processing a list of input keywords stored in the configuration file. Performance tracking includes execution time, token usage, and memory consumption. For each keyword in test\_queries, the function calls the Writer Agent, responsible for generating text.

The Writer Agent is dynamically assigned a role, goal, and backstory based on configuration prompts. Operating with Llama 7B, it processes each keyword by generating a paragraph. The generated text is stored, while individual time, token usage, and memory consumption are recorded for each iteration. This cycle continues until all input keywords have been processed.

Once all paragraphs are generated, the system calculates total and average memory usage, total tokens used, and total execution time. Finally, the Llama 90B model evaluates and rates the generated paragraphs based on predefined criteria. This structured process ensures an efficient benchmarking workflow while enabling performance tracking and text quality assessment.

**3.3 Configuration Management**

The benchmarking process begins by loading configurations from a structured YAML file (config.yaml). This file defines key parameters such as AI models, temperature settings, token limits, and encoding methods. Additionally, it includes task-specific prompts, detailing the roles, objectives, and expected outputs for text generation.

A predefined list of test queries (keywords) is also stored in the configuration, serving as input topics for paragraph generation. The YAML file further defines structured CSV settings, detailing how benchmarking results will be recorded, including latency, memory usage, token consumption, and quality ratings.

The config\_loader.py module handles the retrieval and processing of configurations. It also integrates environment variables to securely load API credentials. By dynamically fetching parameters, the system maintains flexibility while avoiding hardcoded values.

**3.4 Utility Functions**

**Saving Benchmarking Results**

The save\_results\_to\_csv function organizes and stores benchmarking results in a CSV file. It selects the appropriate filename based on the framework, formats the results into a structured table, and appends a summary row capturing total processing time, throughput, and token usage. The final dataset is then saved for further analysis.

**Paragraph Rating**

The rate\_paragraph function evaluates generated text using a larger AI model. It sends each paragraph to the model, which assigns a numerical score (1-10) based on clarity, coherence, and engagement. The function ensures that the rating is valid before returning the result.

**3.5 Setup and Dependencies**

The project setup script configures the Omdena AI Agents Inference Benchmarking package using setuptools. It organizes project files within the src directory, ensuring proper structuring for distribution. Additionally, it enforces a minimum Python version requirement of 3.12 to ensure compatibility with modern dependencies. By defining package metadata, directory structure, and dependencies, the setup script facilitates efficient installation and management of the benchmarking framework.

**4. Test Environment**

The tests for benchmarking the AI agent inference performance across the three frameworks (CrewAI, LangGraph, and AutoGen) were conducted on a user PC with the following specifications and setup:

**System Specifications:**

* RAM: 8 GB
* Operating System: Windows 10
* Age of System: 10+ years
* Internet Speed: Moderate

**Development Environment:**

* IDE Used: Visual Studio Code

**Execution Setup:**

* LLM Execution: The large language models (LLM) were run through Groq Cloud, and the frameworks interacted with the models via API calls.

**Test Timeframe:**

* The tests were conducted in March 2025.

This test environment setup ensures that the results represent a typical user scenario, with a relatively older system configuration and a standard internet speed. The execution through Groq Cloud enabled efficient handling of the resource-intensive tasks despite the limitations of the local machine.

**5. Results**

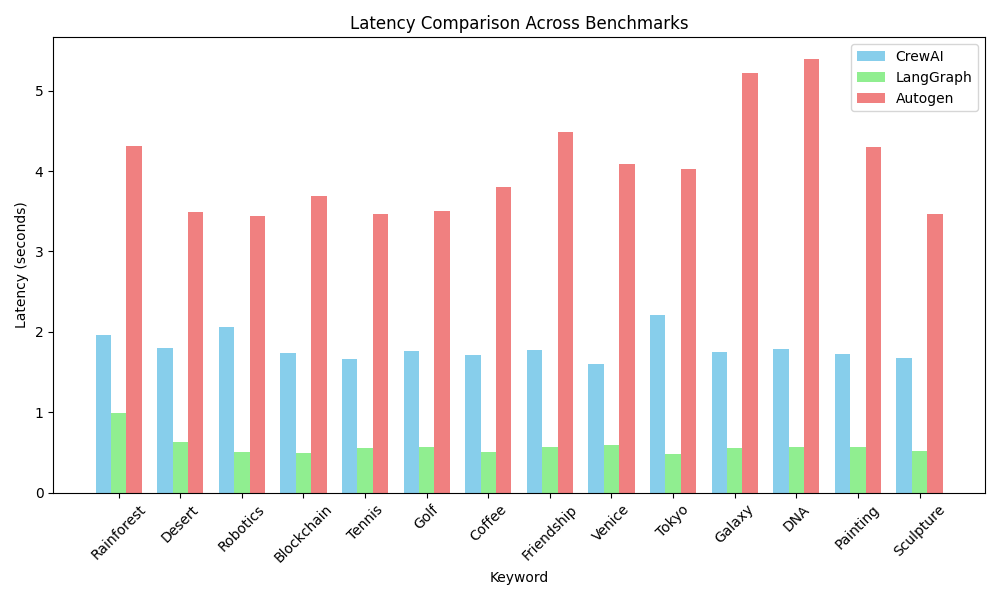
For testing purposes, 14 keywords were provided as input to each of the agentic frameworks. The results demonstrated that all frameworks were able to generate good responses for each keyword. Below is a summary of the performance metrics and observations for each framework:

**5.1 Latency**

**CrewAI:** The average latency for generating a response was around 2 seconds per keyword.

**LangGraph:** LangGraph had a relatively faster response time with an average latency of around 1 second per keyword.

**AutoGen:** The average latency for AutoGen was higher, at around 4 seconds per keyword.

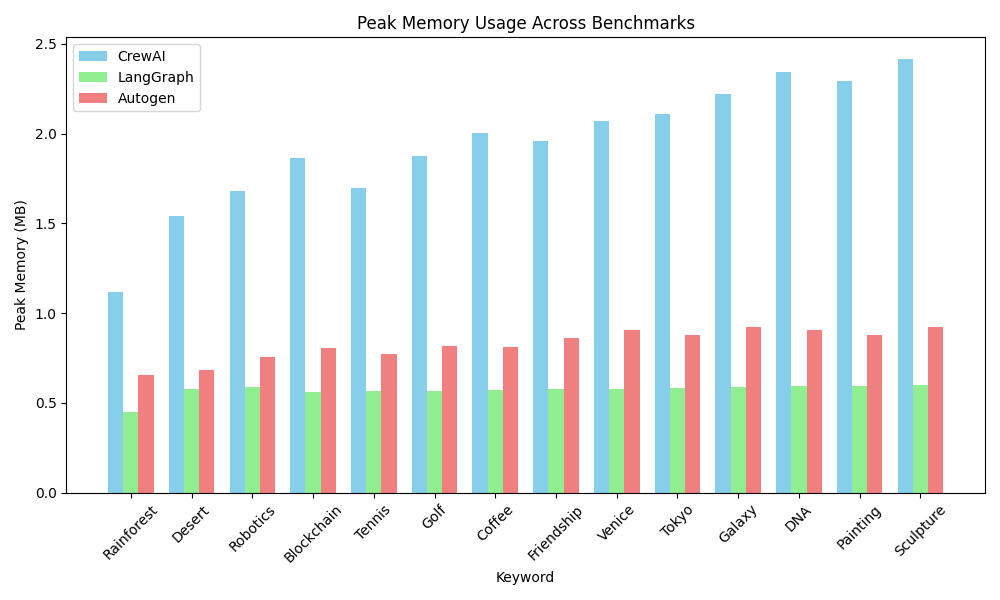


**5.2 Memory Usage**

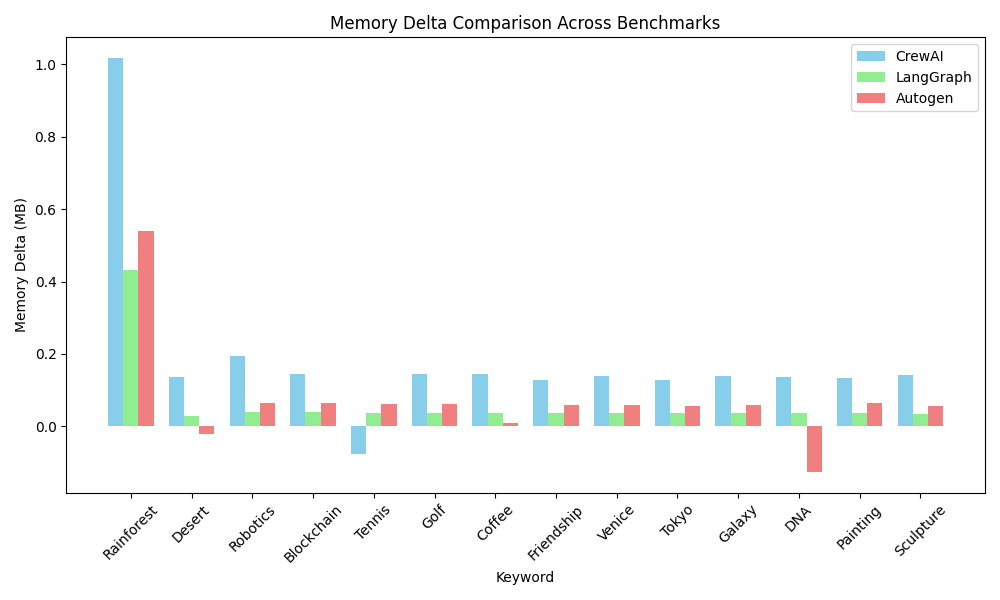
**CrewAI:** The highest memory usage observed during the execution was between 1-2 MB.

**LangGraph:** LangGraph had a relatively lower memory consumption, with the highest usage ranging from 0.4-0.6 MB.

**AutoGen:** AutoGen’s memory usage was slightly higher than LangGraph, with the peak memory usage falling between 0.6-0.8 MB.



**5.3 Memory Usage Change**

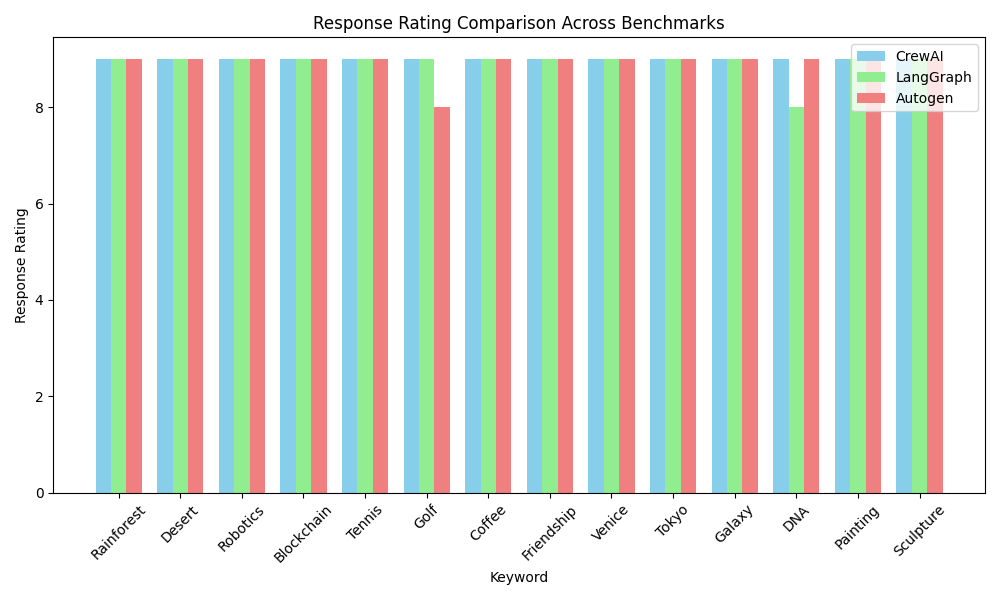


**LangGraph:** The change in memory usage was almost negligible, indicating a stable memory footprint during execution.

**AutoGen:** AutoGen’s memory usage showed moderate fluctuations, positioning it between CrewAI and LangGraph in terms of memory change.

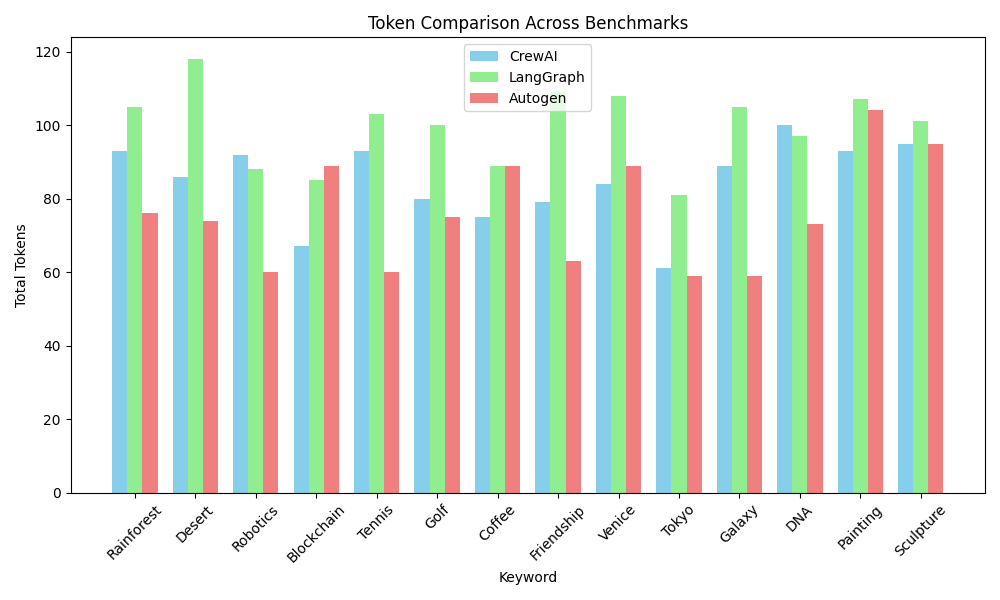
**CrewAI:** CrewAI experienced significant memory fluctuations, showing considerable changes in memory usage over the execution period.

**5.4 Response Ratings**



Almost all generated responses were rated highly, with a 9/10 rating on average across the frameworks. However, there were a few responses that were rated lower in LangGraph and AutoGen due to some minor inconsistencies in the generated content.

**5.5 Token Usage Comparison**



**AutoGen:** The length of responses generated by AutoGen was generally shorter compared to the other frameworks.

**LangGraph:** LangGraph produced responses of optimal length, maintaining a balance between brevity and detail.

**CrewAI:** The length of responses generated by CrewAI was positioned between AutoGen and LangGraph, offering a good balance between short and detailed responses.