Menu

Orchestration Script

Relevant source files

Purpose and Scope

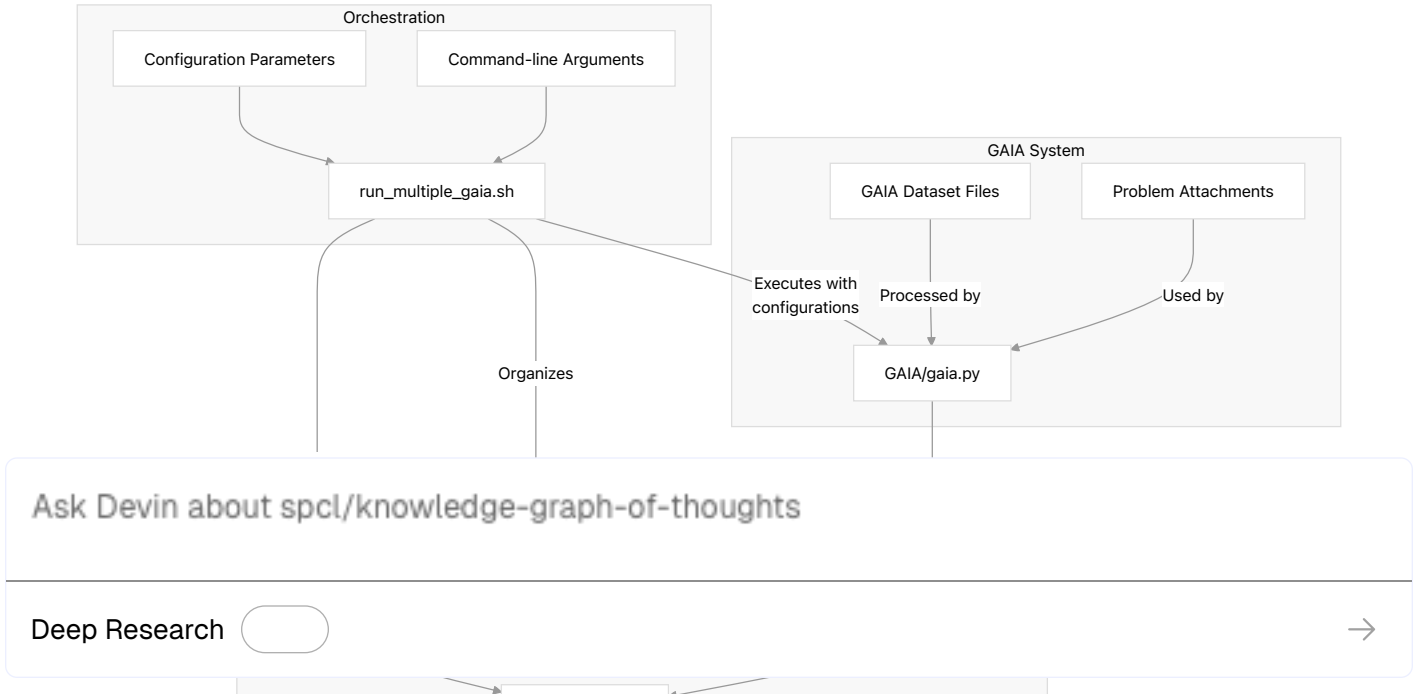
The orchestration script (**`run_multiple_gaia.sh`**) is responsible for executing the GAIA system with multiple configurations and generating analysis plots. It serves as the primary entry point for running experiments across different problem sets, controller strategies, and database implementations. This document covers the script's configuration options, execution flow, and integration with the overall KGoT system.

For information about the Docker environment that hosts this script, see Docker Environment.

Sources: `run_multiple_gaia.sh` `1-5`

Script Architecture

The orchestration script sits at the top level of the execution hierarchy, coordinating the execution of GAIA across different configurations and datasets.



Sources: `run_multiple_gaia.sh` 7–35 `run_multiple_gaia.sh` 242–252

Configuration Options

The orchestration script provides extensive configuration options that can be defined directly in the script or passed as command-line arguments.

Default Configuration

The script defines several default configuration values that are used if not explicitly overridden:

Parameter	Default Value	Description
`PYTHON_SCRIPT`	`GAIA/gaia.py`	Path to the GAIA Python script
`ATTACHMENT_FOLDER`	`GAIA/dataset/attachments/validation`	Path to GAIA problems attachments folder
`CONTROLLER_CHOICE`	`queryRetrieve`	Controller implementation to use
`DB_CHOICE`	`neo4j`	Database implementation to use
`TOOL_CHOICE`	`tools_v2_3`	Tool set implementation to use
`MAX_ITERATIONS`	`7`	Maximum iterations for KGoT
`NEO4J_URI`	`bolt://localhost:7687`	Docker URI for Neo4j
`PYTHON_EXECUTOR_URI`	`http://localhost:16000/run`	URI for Python tool executor
`LLM_EXECUTION_MODEL`	`gpt-4o-mini`	LLM execution model
`LLM_EXECUTION_TEMPERATURE`	`0.0`	LLM execution temperature

Sources: `run_multiple_gaia.sh` 80–90 `run_multiple_gaia.sh` 155–173

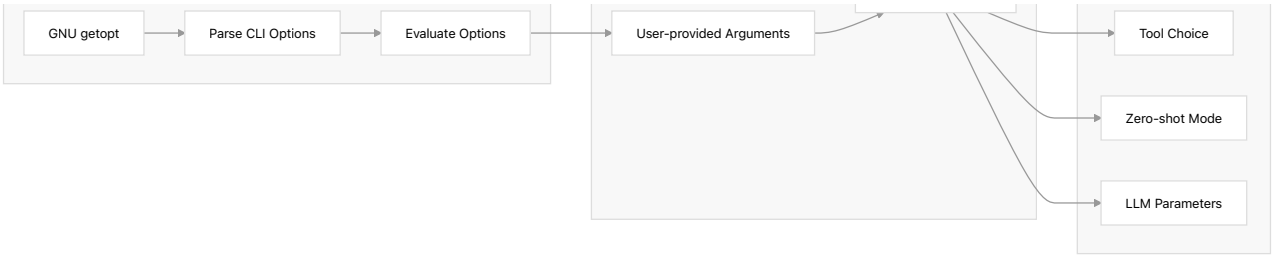
```
# Define an array of GAIA JSON file paths
gaia_files=(
    "GAIA/dataset/validation_subsets/dummy.json"
)
```

By default, it uses a single dummy dataset, but this can be extended to run multiple datasets in sequence.

Sources: `run_multiple_gaia.sh` | `15–18`

Command-line Arguments

The script supports a comprehensive set of command-line arguments that allow overriding the default configuration values.



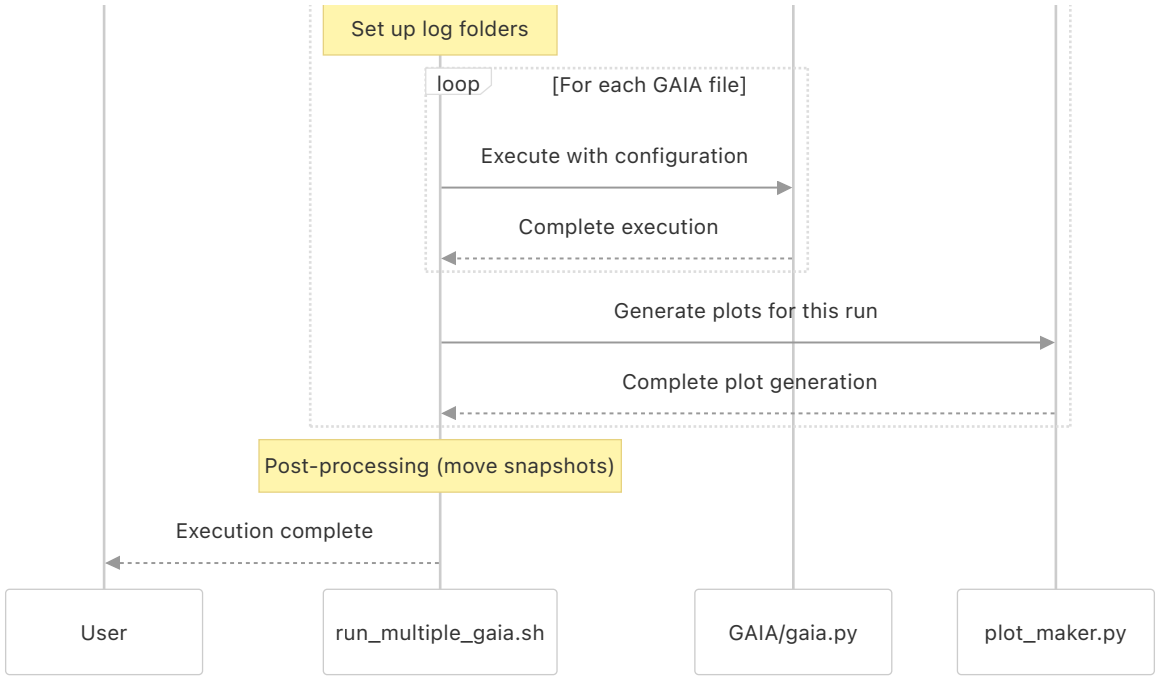
The help message (**`--help`** or **`-h`**) provides a comprehensive list of all available options:

Sources: `run_multiple_gaia.sh` | `40–75` `run_multiple_gaia.sh` | `104–146`

Execution Flow

Multi-Run Loop Structure

The script is designed to support multiple runs of the same configurations for statistical validity:



For each run, the script:

1. Creates a unique log folder based on the configuration
2. Processes each GAIA file in the configured array
3. Builds and executes the Python command with all arguments
4. Generates plots for the run using **`plot_maker.py`**

Sources: `run_multiple_gaia.sh` | `187-243`

Command Building

```
SCRIPT="$PYTHON_SCRIPT --log_folder_base $log_folder \
--gaia_file $gaia_file \
--attachment_folder $ATTACHMENT_FOLDER \
--neo4j_uri $NEO4J_URI \
--python_executor_uri $PYTHON_EXECUTOR_URI \
--controller_choice $CONTROLLER_CHOICE \
--db_choice $DB_CHOICE \
--tool_choice $TOOL_CHOICE \
--max_iterations $MAX_ITERATIONS \
--llm_execution_model $LLM_EXECUTION_MODEL \
--llm_execution_temperature $LLM_EXECUTION_TEMPERATURE"
```

Additional arguments from the command line are appended to this base command.

Sources: `run_multiple_gaia.sh` `217–235`

Output and Analysis

The orchestration script organizes outputs and generates analysis plots to facilitate the evaluation of different configurations.

Log Folder Structure

The script creates a hierarchical log folder structure:

- Base folder: **`logs/{DB_CHOICE}_{CONTROLLER_CHOICE}_{TOOL_CHOICE}`** (or **`logs/{LLM_EXECUTION_MODEL}_{LLM_EXECUTION_TEMPERATURE}_zero_shot`** for zero-shot mode)
- For multiple runs: **`{BASE_FOLDER}/run_{RUN_NUMBER}`**
- For each dataset: **`{RUN_FOLDER}/{DATASET_CATEGORY}`**

Sources: `run_multiple_gaia.sh` `165–172` `run_multiple_gaia.sh` `191–202`

Plot Generation

After each run completes, the script invokes the plot maker to generate analysis visualizations:

Sources: `run_multiple_gaia.sh` | `242`

Neo4j Snapshots

For non-zero-shot executions, the script also handles the movement of Neo4j database snapshots to the log folder for later analysis:

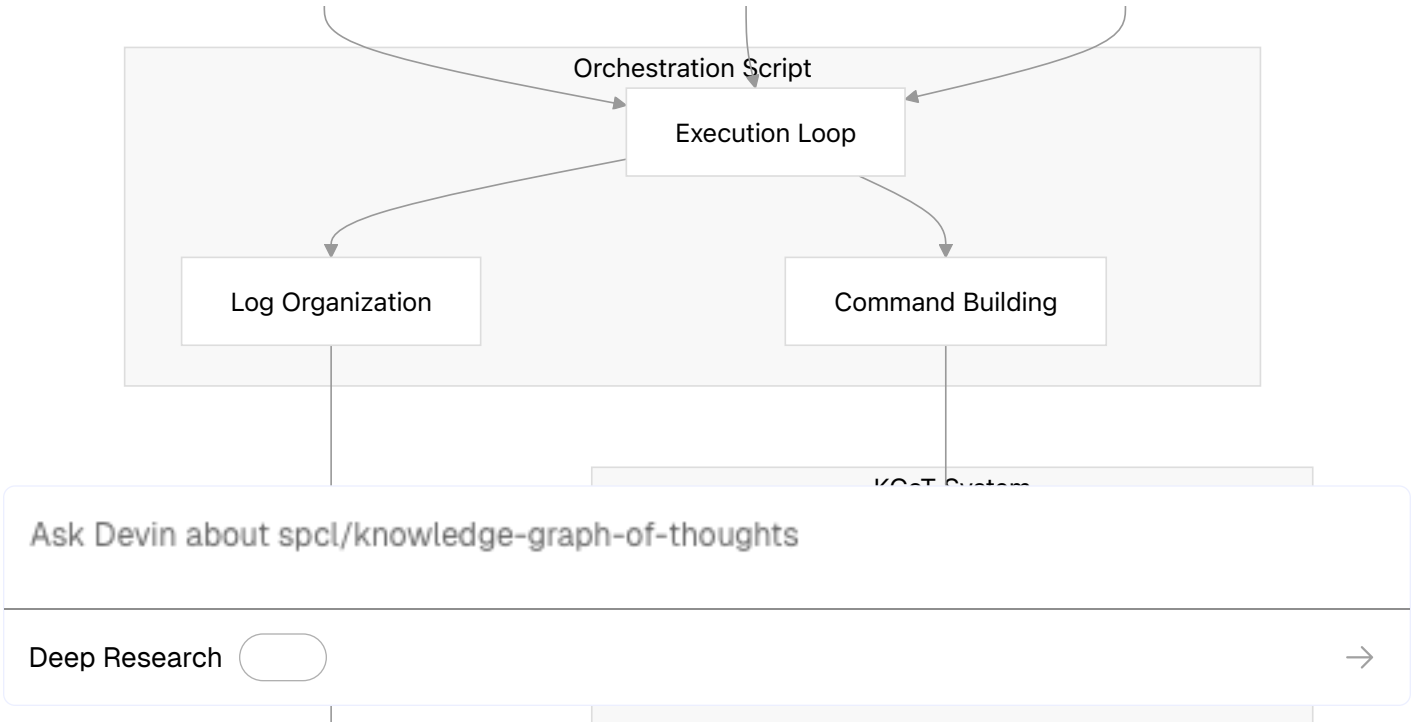
```
if [ "$ZERO_SHOT" = false ]; then
    mv docker_instances/neo4j_docker/snapshots/$LOG_FOLDER_BASE $LOG_FOLDER_BASE/snapsho
fi
```

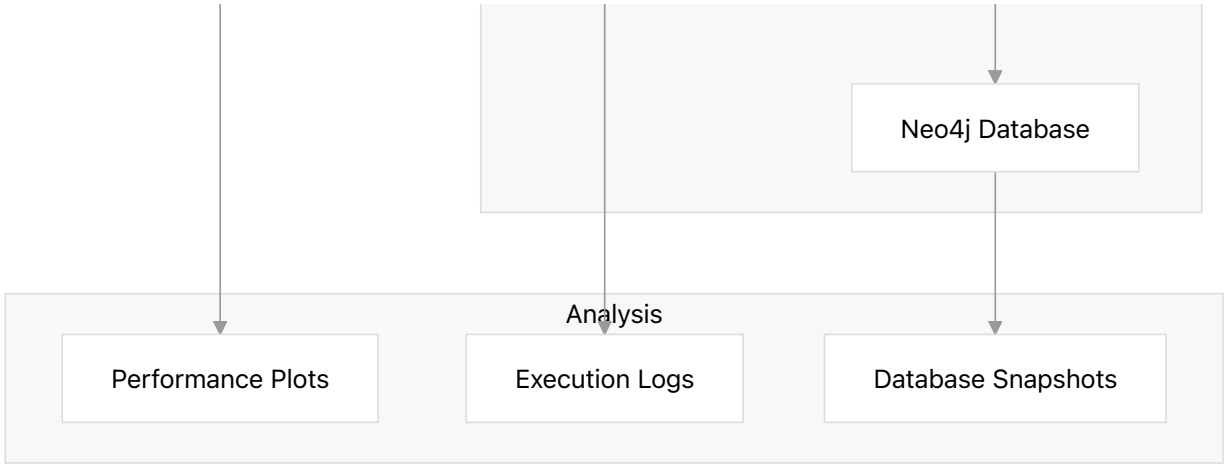
Sources: `run_multiple_gaia.sh` | `249-252`

Use Cases and Integration

The orchestration script serves several key purposes in the KGoT system:

1. **Experimentation**: Running different configurations to compare performance
2. **Benchmarking**: Executing multiple runs to generate statistically significant results
3. **Dataset Testing**: Processing different datasets to evaluate generalization
4. **Visualization**: Automatically generating plots for analysis





Sources: `run_multiple_gaia.sh` `1-5` `run_multiple_gaia.sh` `187-243`

Conclusion

The **`run_multiple_gaia.sh`** orchestration script is a central component in the KGoT system's execution environment. It provides a flexible and configurable way to run experiments, compare different implementations, and analyze results. By automating the execution process and output organization, it streamlines the process of working with the GAIA and KGoT systems.

Menu

Docker Environment

Relevant source files

Purpose and Overview

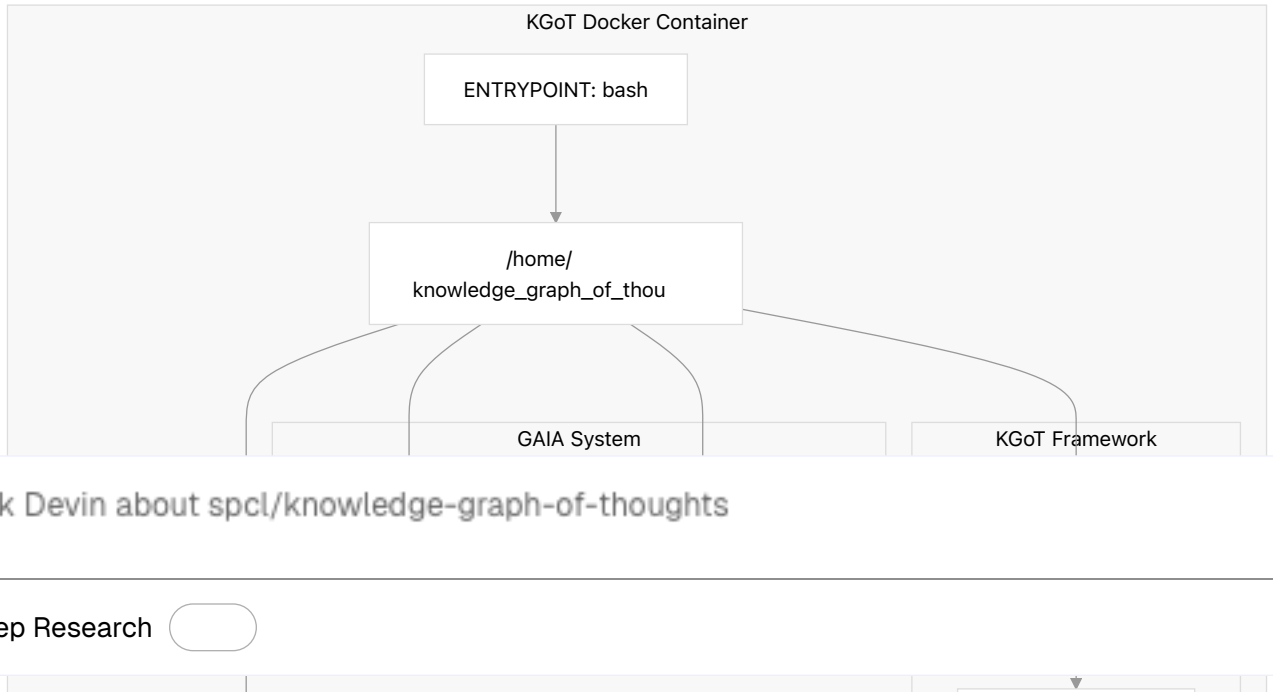
The Docker environment serves as the containerized deployment solution for the Knowledge Graph of Thoughts (KGoT) system. This document explains how the Docker container is structured, built, and configured to provide a consistent and isolated execution environment for KGoT and its components.

For information about executing multiple GAIA instances, see Orchestration Script.

Sources: `docker_instances/kgot_docker/Dockerfile` `1–30`

Container Architecture

The KGoT Docker container encapsulates all necessary components to run the complete Knowledge Graph of Thoughts system, including both the core KGoT framework and the GAIA system built on top of it.



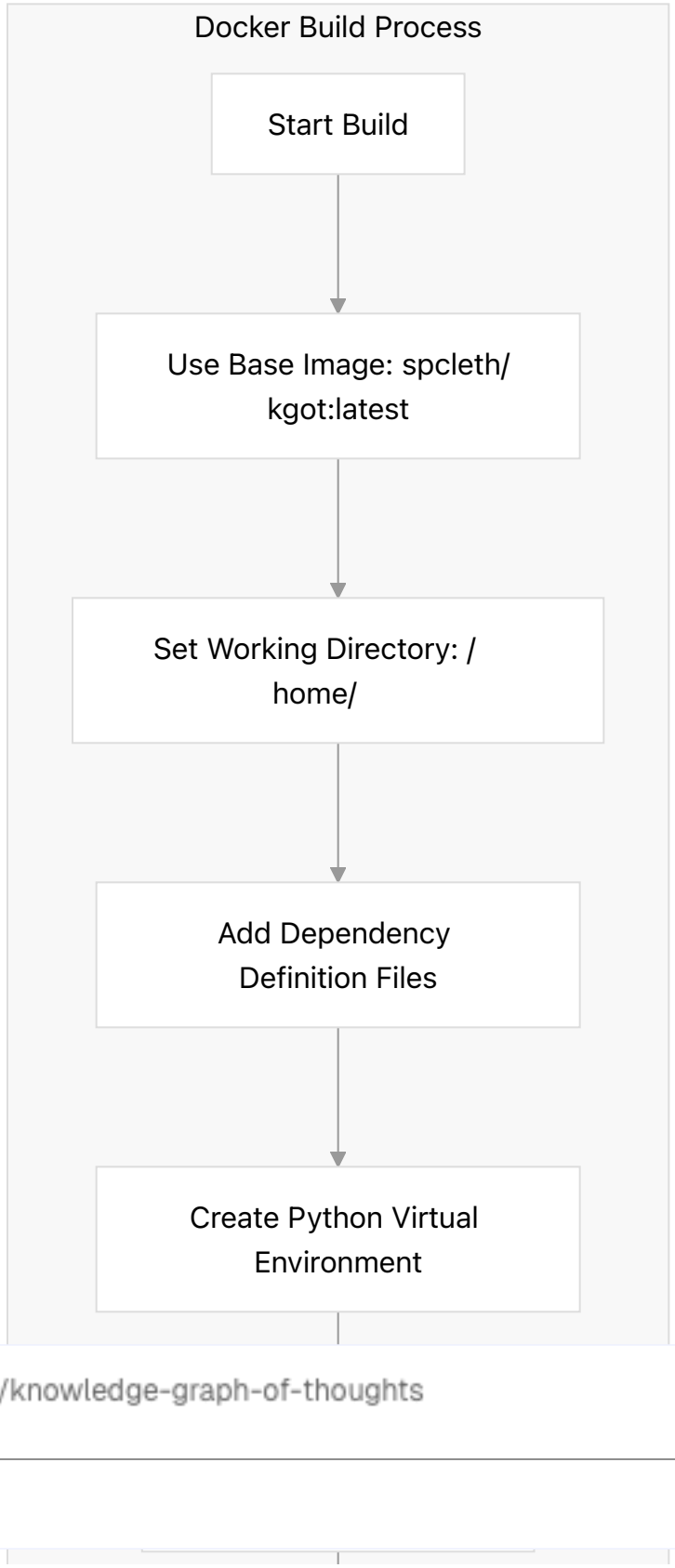
Sources: `docker_instances/kgot_docker/Dockerfile` `1–30`

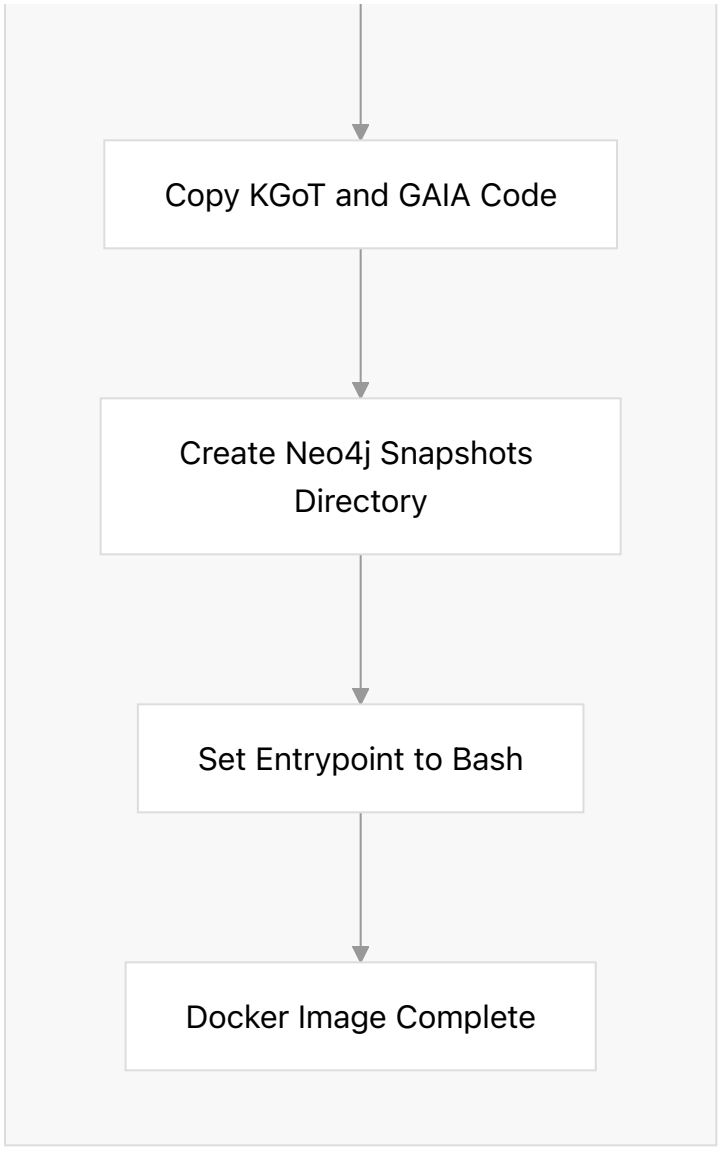
Container Configuration Details

The KGoT Docker container is built with the following key characteristics:

Component	Description
Base Image	**`spcleth/kgot:latest`**
Working Directory	**`/home/knowledge_graph_of_thoughts`**
Python Environment	Python 3.11 with virtual environment
Key Libraries	KGoT package and dependencies, Playwright
Code Components	KGoT framework, GAIA system
Execution Scripts	**`run_multiple_gaia.sh`** and variants
Neo4j Integration	Directory for Neo4j snapshots
Entrypoint	Bash shell

The Dockerfile implements a carefully optimized build process that leverages Docker's layer caching to improve build efficiency. This is particularly helpful when making incremental changes to the code without changing dependencies.



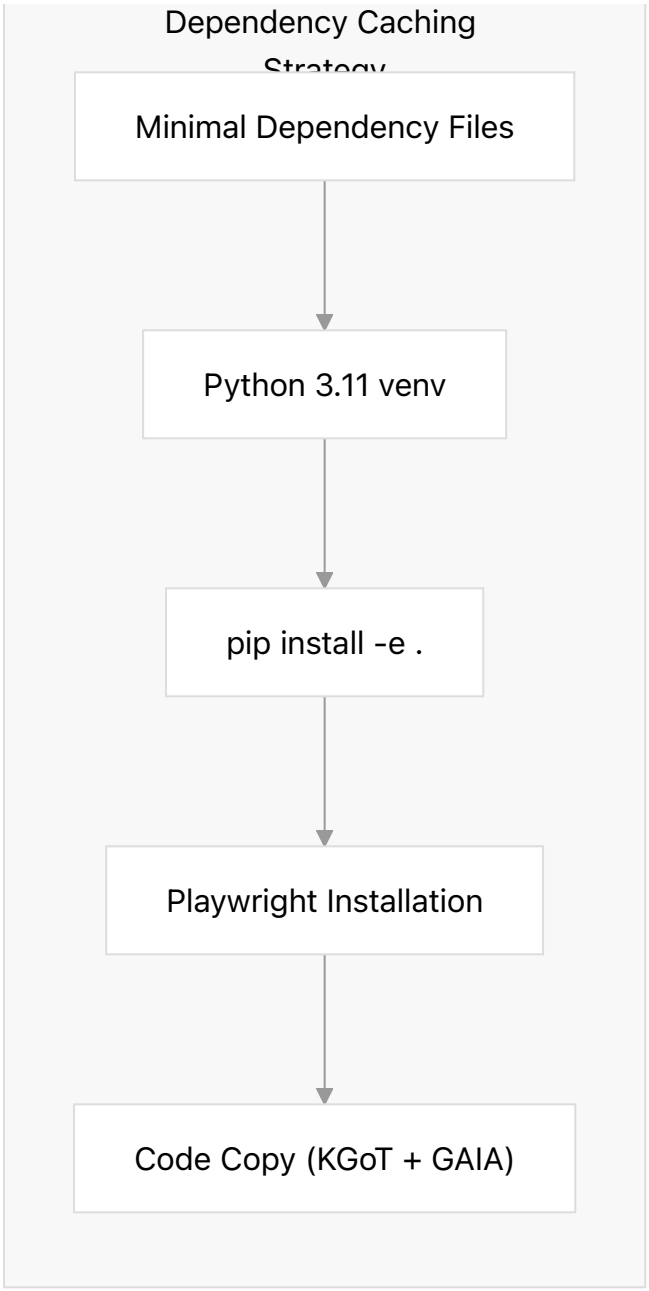


Sources: `docker_instances/kgot_docker/Dockerfile` `1–30`

Dependency Management

The Docker build process prioritizes efficient dependency management by:

1. First adding only the minimal files needed for dependency installation (**`pyproject.toml`** and key Python files)
2. Creating a Python virtual environment



Sources: `docker_instances/kgot_docker/Dockerfile` `6–18`

`docker_instances/kgot_docker/Dockerfile` `20–25`

File Organization

The container's file organization is structured as follows:

Sources: `docker_instances/kgot_docker/Dockerfile` | `3–25`

`docker_instances/kgot_docker/Dockerfile` | `27`

Runtime Configuration

At runtime, the container uses the following configuration:

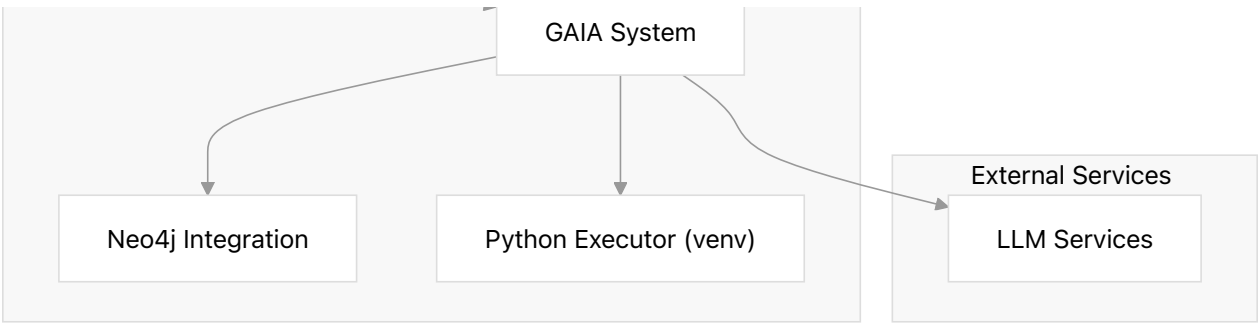
1. The working directory is set to **`/home/knowledge_graph_of_thoughts`**
2. The default entrypoint is a bash shell, allowing flexible command execution
3. The **`run_multiple_gaia.sh`** script is executable and available in the working directory
4. A directory for Neo4j snapshots is available at **`./docker_instances/neo4j_docker/snapshots`**

The container has access to all KGoT and GAIA functionality and is designed to work seamlessly with external services like LLM providers.

Sources: `docker_instances/kgot_docker/Dockerfile` | `24–30`

KGoT Components in Docker

The Docker container integrates the following KGoT system components:



Sources: `docker_instances/kgot_docker/Dockerfile` `1–30`

Usage Guidelines

The Docker container is primarily designed to be used in the following ways:

1. As a development environment for working with KGoT and GAIA
2. For running experiments with the KGoT framework and GAIA system
3. For deploying the system in a consistent, isolated environment
4. For integrating with the orchestration script to run multiple GAIA instances

When running the container, users would typically:

1. Mount volumes for data input/output if needed
2. Configure necessary environment variables for LLM services
3. Execute the **`run_multiple_gaia.sh`** script or directly use the KGoT CLI

Sources: `docker_instances/kgot_docker/Dockerfile` `24–30`

Menu

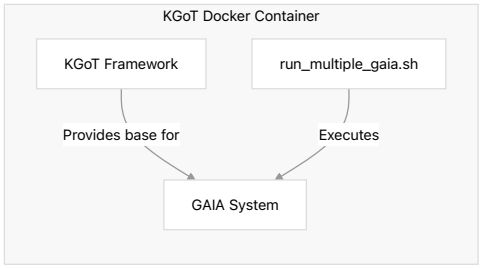
Deployment and Execution

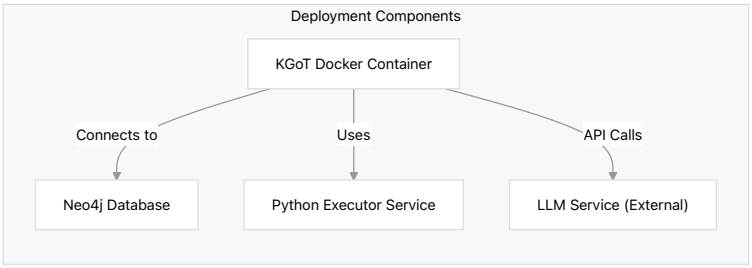
Relevant source files

This page documents how to deploy and run the Knowledge Graph of Thoughts (KGoT) system in various environments. It covers the deployment options, execution methods, configuration parameters, and execution flow of the system. For specific details about the Docker environment setup, see Docker Environment. For in-depth information about the orchestration script, see Orchestration Script.

Deployment Architecture

The KGoT system can be deployed in a containerized environment using Docker, which ensures consistent execution across different platforms. The deployment architecture consists of several components that work together to provide a complete execution environment.





Sources: `docker_instances/kgot_docker/Dockerfile` `run_multiple_gaia.sh`

Deployment Options

Docker Deployment

The Docker deployment is the recommended method for running KGoT, as it provides a self-

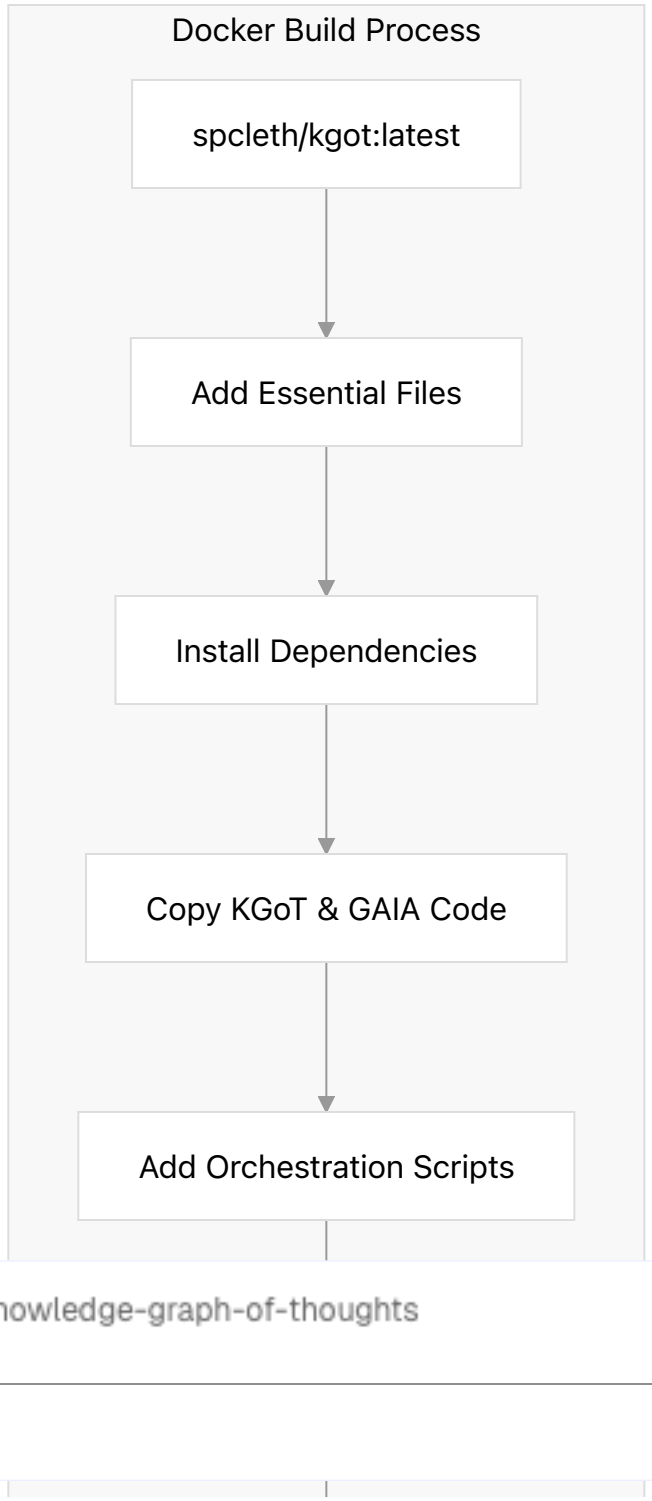
Ask Devin about spcl/knowledge-graph-of-thoughts

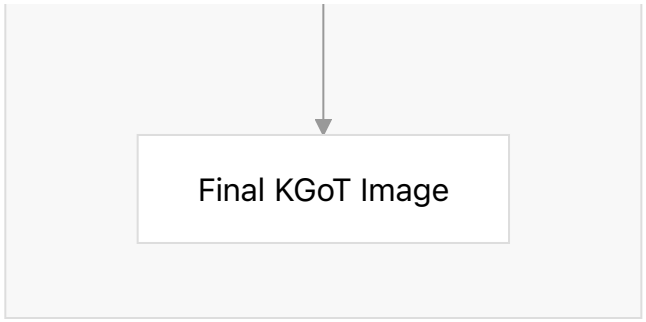
Deep Research

1. Building the Docker image using the provided Dockerfile

1. ~~Building the Docker image using the provided Dockerfile~~
2. Running the Docker container with appropriate port mappings
3. Connecting the container to external services (Neo4j, Python Executor, LLM services)

The Docker image is built from a base image **`spcleth/kgot:latest`** and includes the KGoT codebase, GAIA system, and orchestration scripts. It sets up a Python virtual environment and installs all required dependencies.





Sources: `docker_instances/kgot_docker/Dockerfile` `1–30`

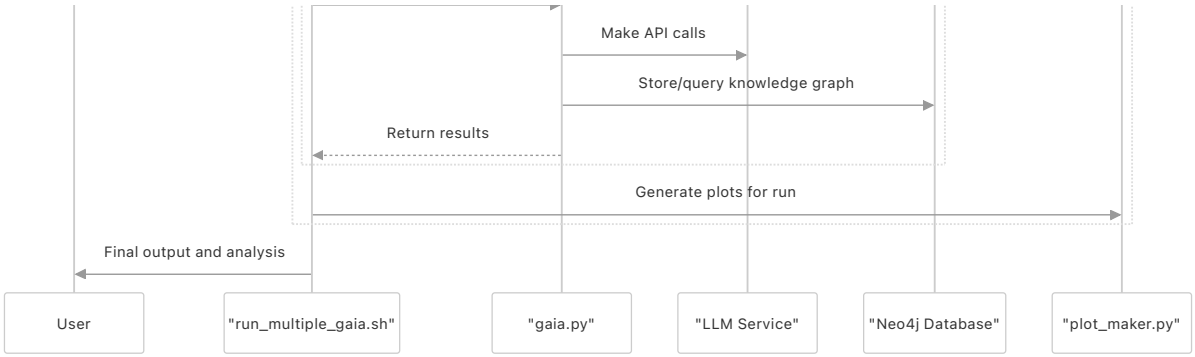
Execution Methods

The KGoT system can be executed in two primary ways:

1. **Direct Execution**: Running GAIA directly for a single configuration
2. **Orchestrated Execution**: Using the `run_multiple_gaia.sh` script to execute GAIA with multiple configurations and generate comparison plots

Orchestrated Execution

The `run_multiple_gaia.sh` script provides a flexible way to run GAIA with different configurations, compare results, and generate analysis plots. It supports running multiple GAIA datasets, with multiple configurations, and for multiple iterations.



Sources: `run_multiple_gaia.sh` | `1-253`

Configuration Parameters

The KGoT system supports a wide range of configuration parameters that can be set when executing the system. The most important parameters include:

Essential Parameters

Parameter	Description	Default Value
`log_folder_base`	Directory where logs will be stored	`logs/[DB_CHOICE]_[CONTROLLER_CHOICE]_[TOOL_CHOICE]`
`attachment_folder`	Path to GAIA problems attachments folder	`GAIA/dataset/attachments/validation`
`gaia_file`	Path to the GAIA dataset file	Various options from `gaia_files` array
`controller_choice`	Controller selection for reasoning strategy	`queryRetrieve`
`db_choice`	Database backend choice	`neo4j`
`tool_choice`	Tool system selection	`tools_v2_3`
`max_iterations`	Maximum iterations for KGoT reasoning	`7`
`neo4j_uri`	URI for Neo4j database	`bolt://localhost:7687`
`python_executor_uri`	URI for Python tool executor	`http://localhost:16000/run`

LLM Configuration Parameters

Parameter	Description	Default Value
`llm_execution_model`	LLM model for execution	`gpt-4o-mini`

`zero_shot`	Use zero-shot mode instead of KGoT	`false`

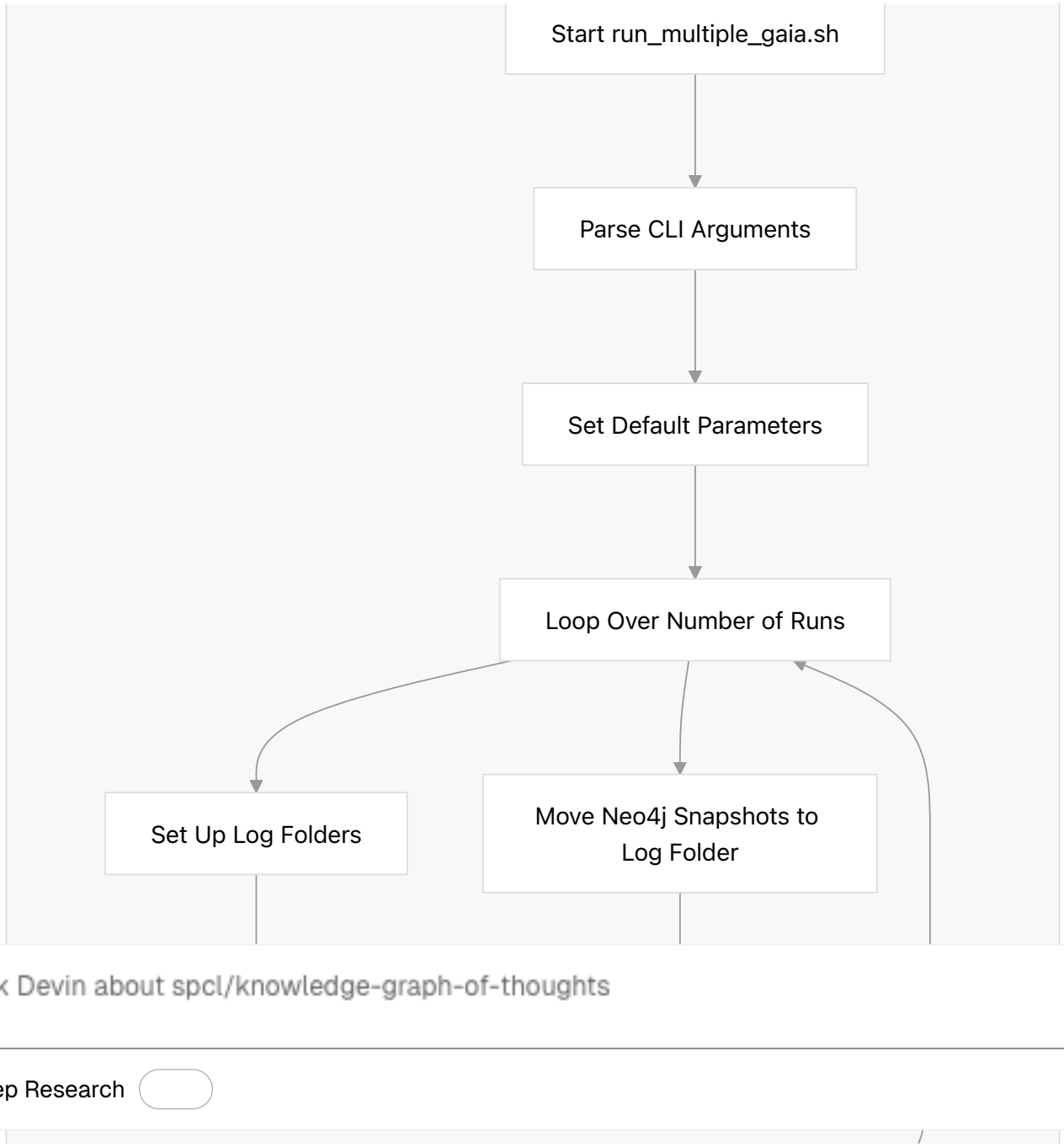
`zero_shot`	Use zero-shot mode instead of KGoT	`false`

Sources: `run_multiple_gaia.sh` 42–73 `run_multiple_gaia.sh` 80–90

Execution Flow

The execution flow of the KGoT system varies depending on whether it's run directly or using the orchestration script.

Orchestrated Execution Flow



Sources: `run_multiple_gaia.sh` `185–253`

Docker Container Structure

The Docker container for KGoT includes the following components:

Sources: `docker_instances/kgot_docker/Dockerfile` `3–27`

Command Line Interface

To execute the KGoT system using the orchestration script, the following command format is used:

```
./run_multiple_gaia.sh --help
```

The orchestration script builds the command for executing GAIA for each run and each GAIA dataset file, processing all the options and setting appropriate default values.

Sources: `run_multiple_gaia.sh` `217–227`

Integration with External Services

The KGoT system integrates with several external services during execution:

Sources: `run_multiple_gaia.sh` `50–56` `run_multiple_gaia.sh` `216–228`

Debug and Monitoring

During execution, the KGoT system generates detailed logs and statistics that can be used to monitor the progress and performance of the system. These are stored in the specified log folder and can be analyzed using the plot maker utility.

The system also supports saving Neo4j snapshots for later analysis, which are automatically moved to the log folder after execution completes.

Sources: `run_multiple_gaia.sh` `242` `run_multiple_gaia.sh` `249–252`

Menu

Dataset Processing

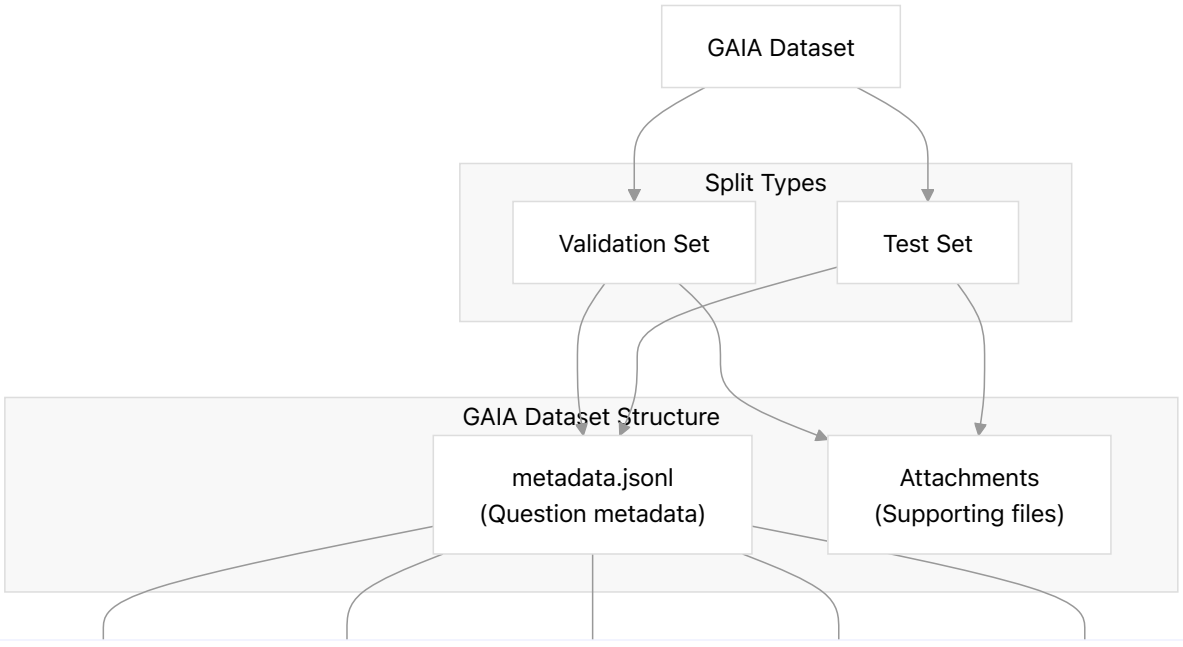
Relevant source files

Purpose and Scope

This document explains how the GAIA dataset is fetched, processed, and organized within the Knowledge Graph of Thoughts (KGoT) framework. It covers the automated download process from Hugging Face, the transformation of dataset files, and the organization of dataset content for use by the GAIA system. For information about how the GAIA system uses this processed data during execution, see Architecture and Execution.

Dataset Overview

The GAIA dataset consists of complex reasoning questions with associated metadata and attachments. It is hosted on the Hugging Face Hub and requires authentication to access.



Ask Devin about spcl/knowledge-graph-of-thoughts

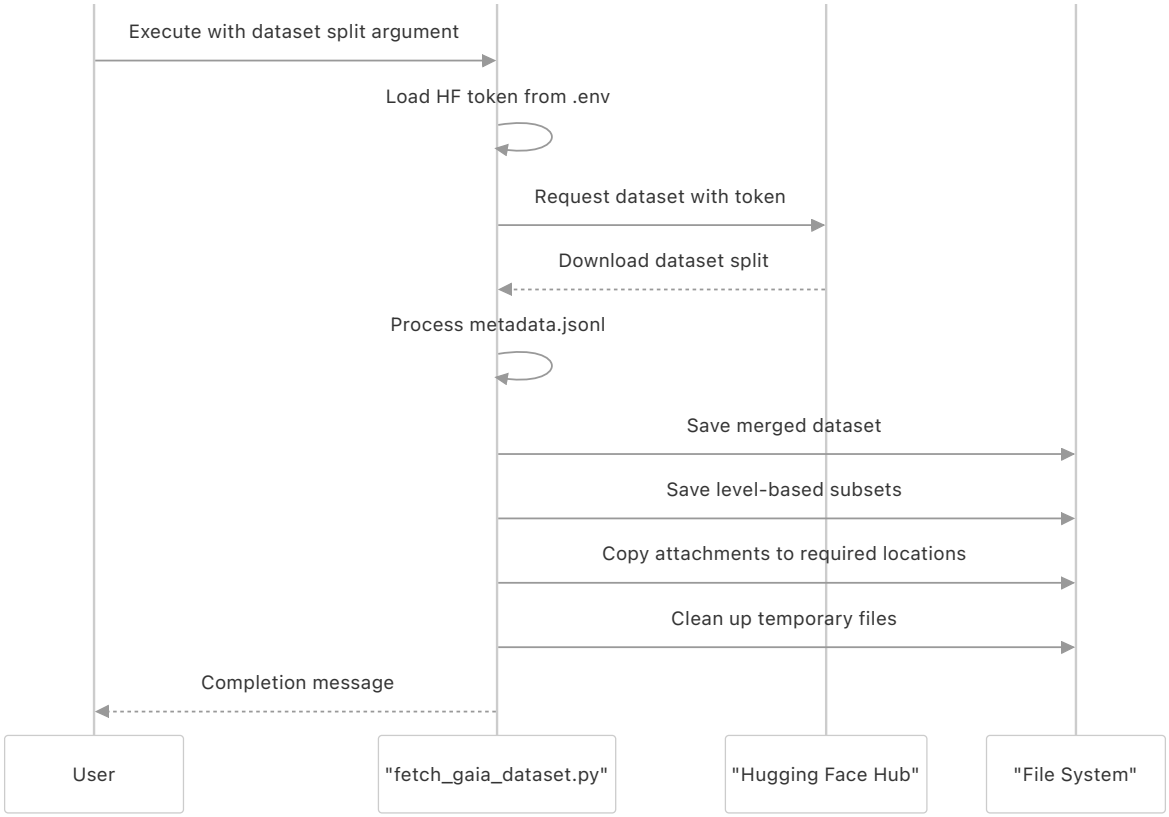
Deep Research

Sources: `GAIA/dataset/fetch_gaia_dataset.py` 18–27

`GAIA/dataset/fetch_gaia_dataset.py` 42–49

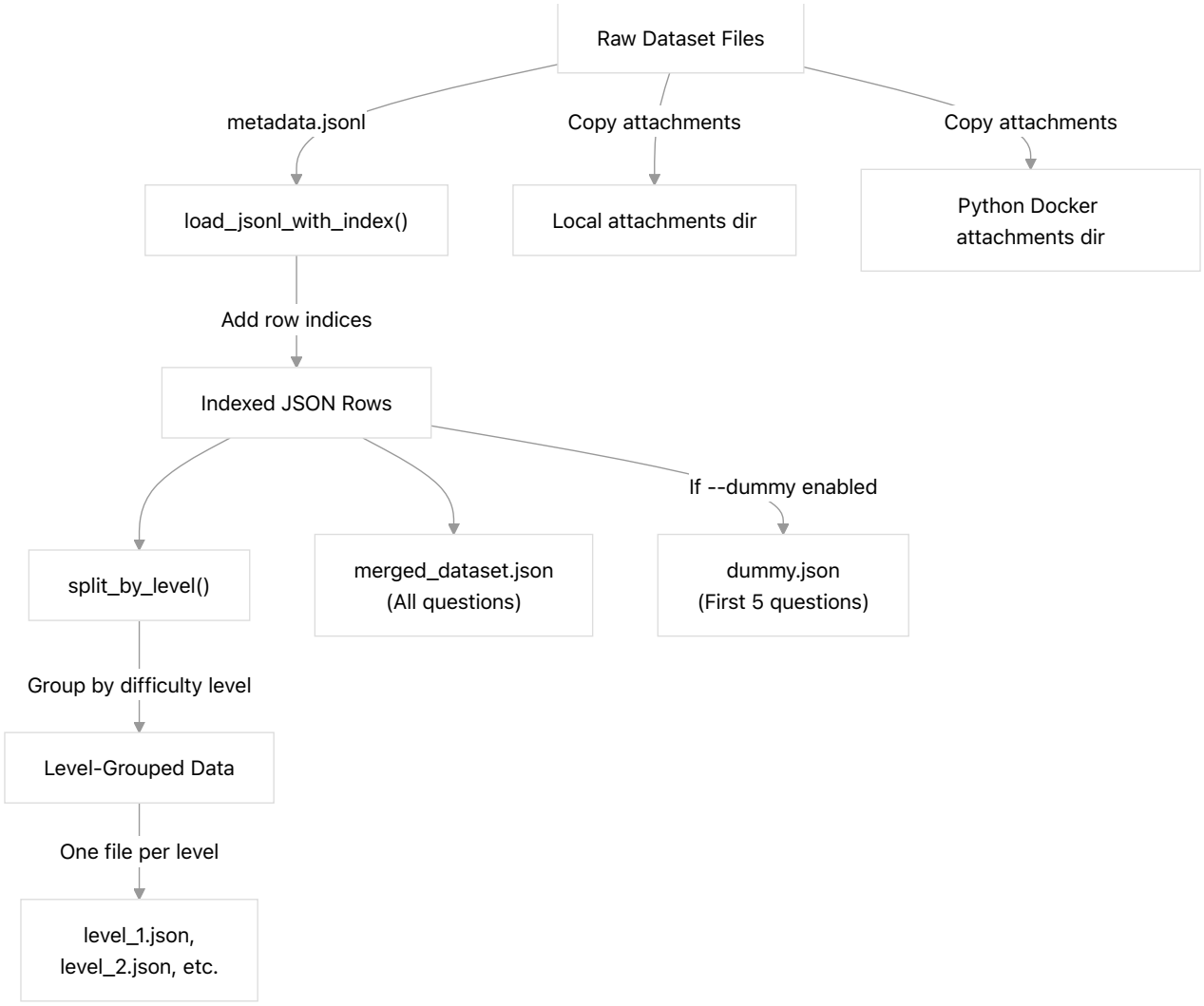
Dataset Fetching Process

The dataset fetching process is handled by the **`fetch_gaia_dataset.py`** script, which downloads a specified split (validation or test) from Hugging Face and processes it for use by the GAIA system.



Sources: `GAIA/dataset/fetch_gaia_dataset.py` 98–119

Processing Workflow Details



Sources: `GAIA/dataset/fetch_gaia_dataset.py` | `30-68`

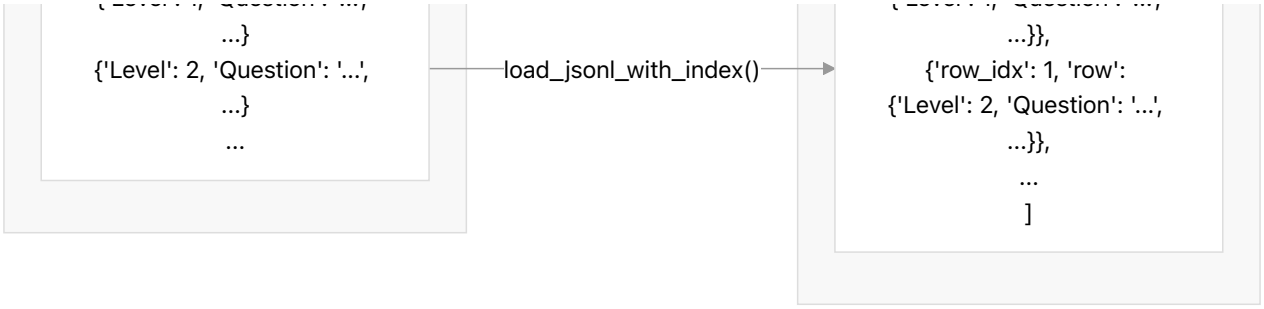
`GAIA/dataset/fetch_gaia_dataset.py` | `109-113`

Data Transformations

The dataset undergoes specific transformations during processing:

1. From .JSONL to .JSON with row indices

JSONL to Indexed JSON Transformation



Sources: `GAIA/dataset/fetch_gaia_dataset.py` `30–33`

Directory Structure

The dataset processing script organizes files into the following directory structure:

Directory	Purpose	Contents
`tmp/`	Temporary storage during download	Raw GAIA dataset files
`{split}/`	Merged dataset	**`merged_dataset.json`** (all questions)
`{split}_subsets/`	Level-specific subsets	**`level_1.json`**, **`level_2.json`**, etc., optional **`dummy.json`**
`attachments/{split}/`	Attachment files for GAIA	Raw attachment files from dataset
`../../docker_instances/python_docker/files/GAIA/dataset/attachments/{split}/`	Attachments for Python Docker	Copy of attachment files for Docker environment

Where **`{split}`** is either "validation" or "test" depending on the dataset split requested.

The dataset fetching script can be run with command-line arguments to control which dataset split to download and whether to create a dummy subset:

```
python GAIA/dataset/fetch_gaia_dataset.py --dataset [validation|test] [--dummy|--no-dummy]
```

Parameters:

- **`--dataset`** : Specifies which dataset split to download (either "validation" or "test", default is "validation")
- **`--dummy`** : Whether to create a small dummy subset with the first 5 questions (enabled by default)
- **`--no-dummy`** : Disables creation of dummy subset

A Hugging Face authentication token must be provided in a **`.env`** file with the key **`HUGGINGFACE_TOKEN`**.

Sources: `GAIA/dataset/fetch_gaia_dataset.py` `79-95`

`GAIA/dataset/fetch_gaia_dataset.py` `71-76`

Integration with GAIA System

The processed dataset is utilized by the GAIA system during execution. When the GAIA entrypoint script runs, it loads the appropriate dataset files based on configuration parameters.

Dataset Metadata Structure

The GAIA dataset questions include metadata with the following key fields:

Field	Description
Level	Difficulty level (1-5)
Question	The main question text
Answer	The correct answer (for evaluation)
Attachments	List of supporting files needed for the question
Additional metadata	Various fields depending on question type

After processing, each question is augmented with a **`row_idx`** field to maintain ordering and allow for easy reference.

Sources: `GAIA/dataset/fetch_gaia_dataset.py` `30–33`

`GAIA/dataset/fetch_gaia_dataset.py` `42–49`

Menu

Architecture and Execution

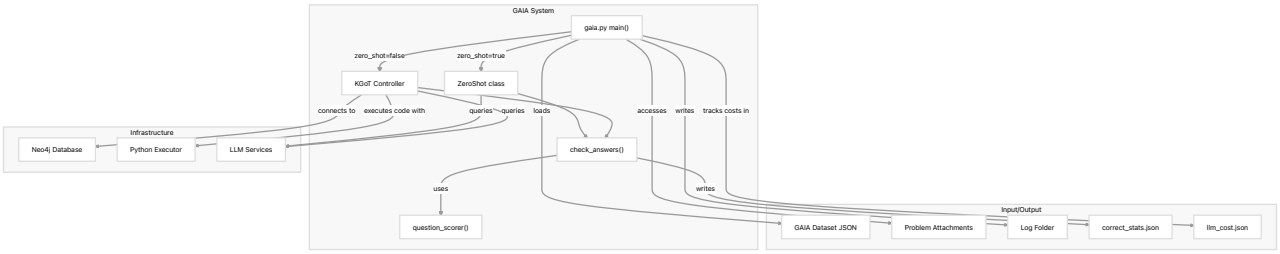
Relevant source files

This document provides a detailed overview of the GAIA (General AI Assistant) system's architecture and execution modes. It explains how GAIA integrates with the Knowledge Graph of Thoughts (KGoT) framework to solve complex reasoning tasks, the two execution modes it supports, and the flow of data and control through the system.

For information about the GAIA dataset processing, see Dataset Processing.

GAIA System Overview

The GAIA system represents a high-level assistant that can solve complex reasoning problems by either using direct LLM queries (zero-shot mode) or leveraging the Knowledge Graph of Thoughts framework for more sophisticated reasoning (KGoT mode).



Sources: GAIA/gaia.py 106–212

Execution Modes

GAIA supports two distinct execution modes that can be selected based on the complexity of the problem and the desired approach:

KGoT Mode

In KGoT mode, GAIA leverages the full capabilities of the Knowledge Graph of Thoughts framework, including:

- Building a dynamic knowledge graph
- Using controller strategies for planning
- Executing tools to gather information
- Iteratively refining the solution

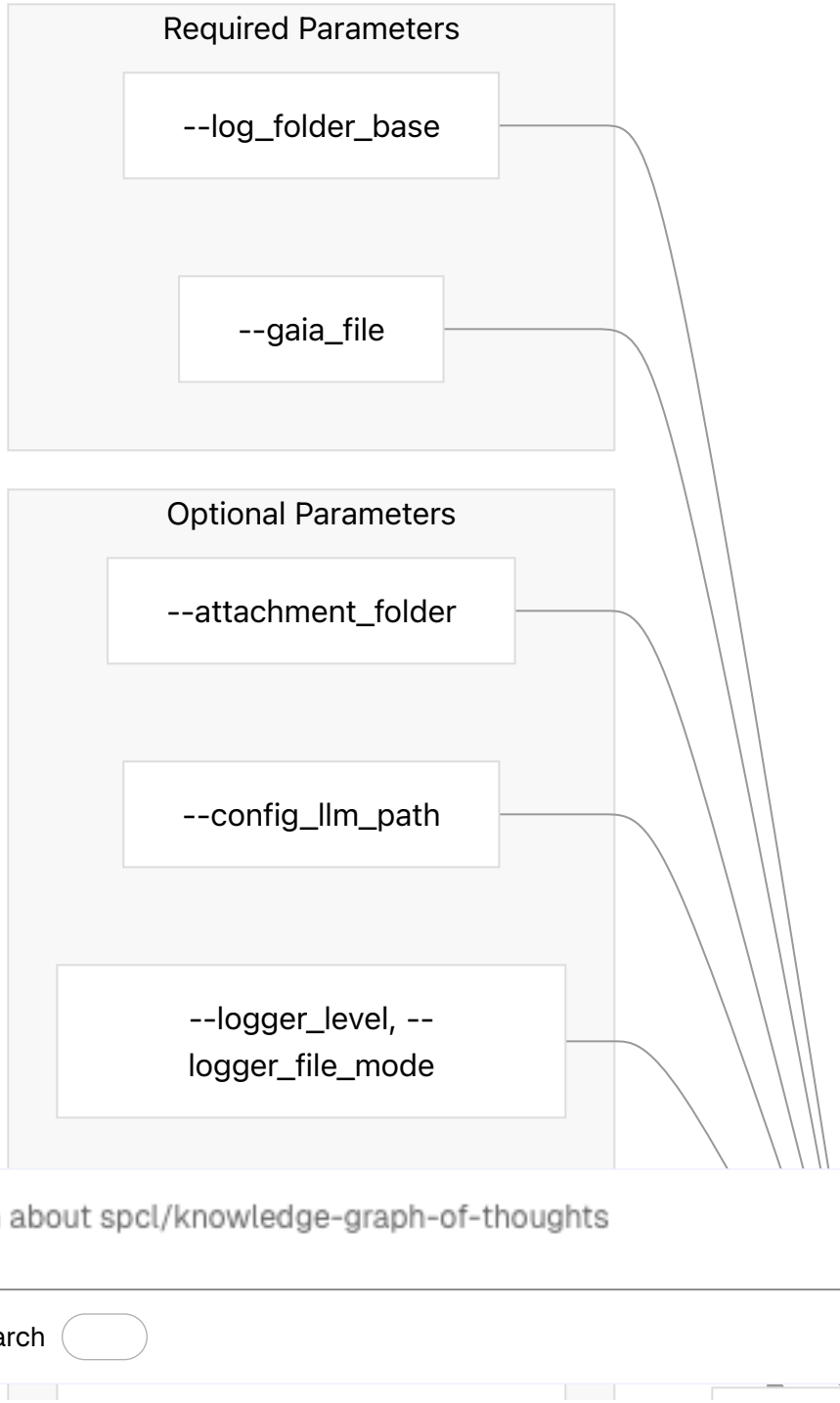
The choice between these modes is controlled by the `zero_shot` parameter when executing GAIA.

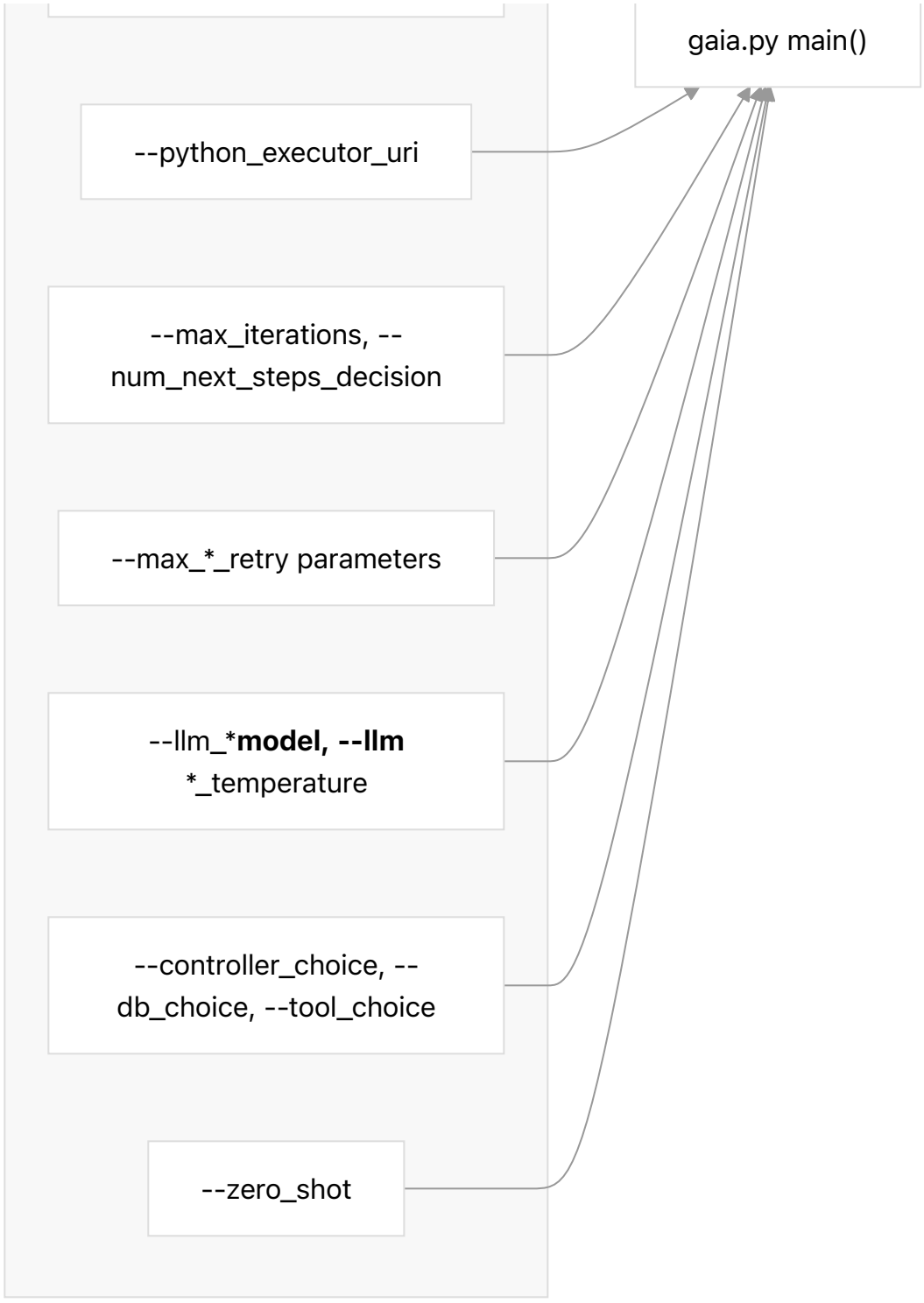
Configuration and Initialization

GAIA is highly configurable with numerous parameters that control its behavior, connections to external services, and execution settings.

Command-Line Interface

GAIA can be executed directly from the command line with a rich set of parameters:





Sources: `GAIA/gaia.py` | `214–276`

Ask Devin about spcl/knowledge-graph-of-thoughts

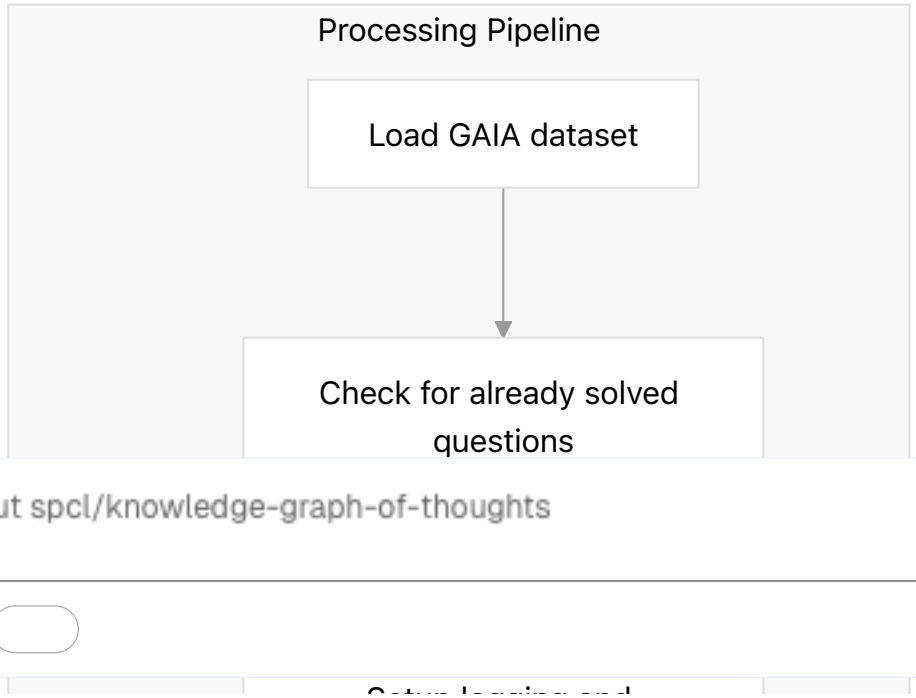
Deep Research

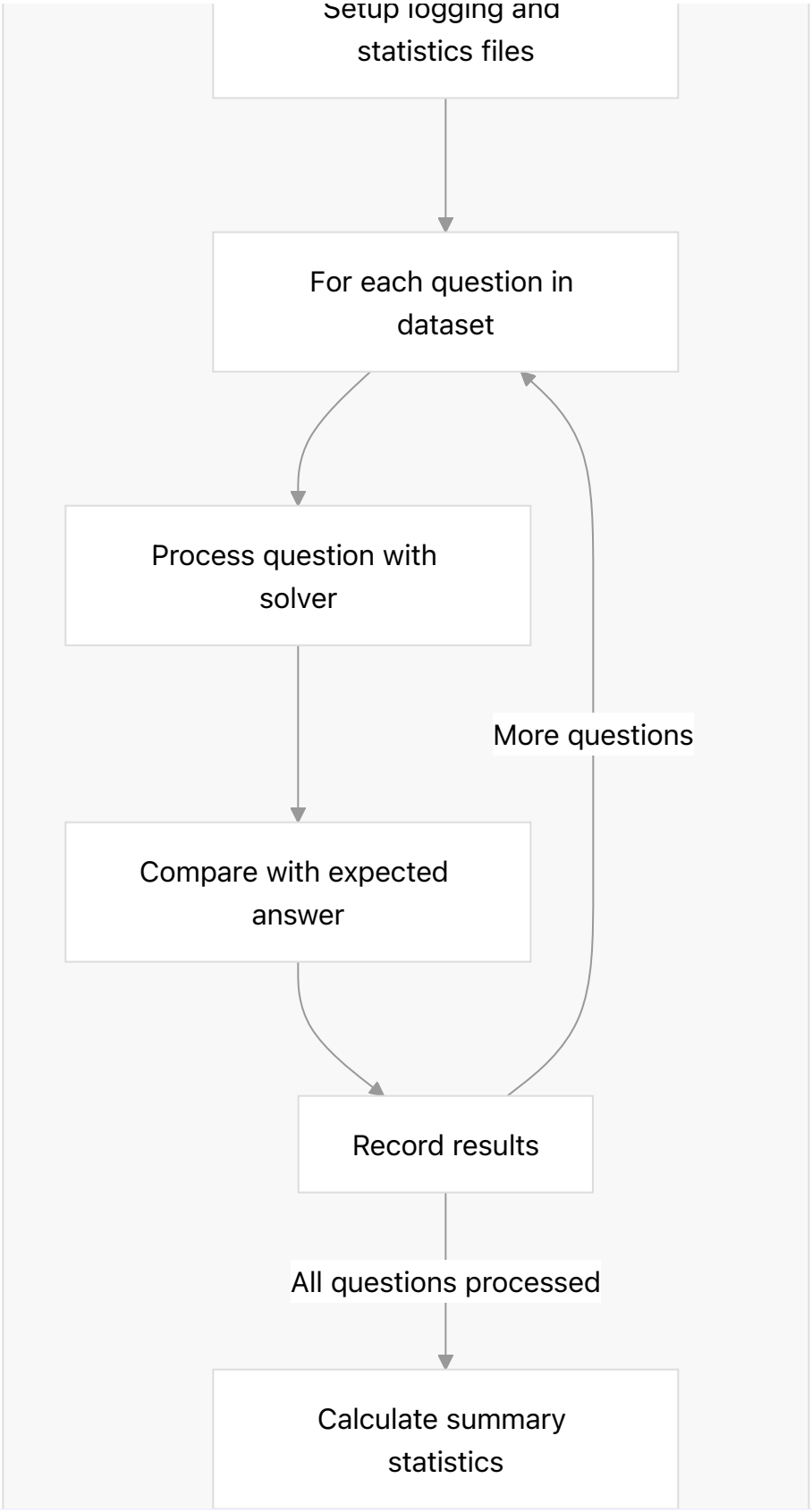
	`attachment_folder`		output locations
Infrastructure	`neo4j_uri`, `neo4j_username`, `neo4j_password`, `python_executor_uri`	`bolt://localhost:7687`, `neo4j`, `password`, `http://localhost:16000/run`	Connect to services
LLM	`llm_planning_model`, `llm_execution_model`, `llm_*_temperature`	`gpt-4o-mini`, `0.0`	Control LLM usage
Execution	`max_iterations`, `num_next_steps_decision`	`7`, `5`	Control execution flow
System Components	`controller_choice`, `db_choice`, `tool_choice`	`queryRetrieve`, `neo4j`, `tools_v2_3`	Select system components
Mode	`zero_shot`	`False`	Toggle execution mode

Sources: `GAIA/gaia.py` `106–132` `GAIA/gaia.py` `214–276`

Question Processing Flow

GAIA processes questions from the dataset sequentially, keeping track of progress to support resuming:





Question Execution

For each question, GAIA:

1. Extracts question details and metadata
2. Calls the appropriate solver function based on execution mode
   - `ZeroShot.answer_query()` for zero-shot mode
   - `Controller.run()` for KGoT mode
3. Compares the returned answer with the expected answer
4. Records detailed statistics including:
   - Correctness information
   - Problem metadata
   - Number of iterations taken
   - Tool usage information

Sources: `GAIA/gaia.py` `25–90`

Results and Statistics

GAIA maintains comprehensive records of execution results and costs:

Correctness Statistics

For each question processed, GAIA records:

- Question number and details
- Expected and returned answers
- Success status (correct, close call, or incorrect)
- Problem metadata (level, expected steps, tools)
- Actual iterations taken

GAIA tracks detailed statistics about LLM usage including:

GAIA tracks detailed statistics about LLM usage including:

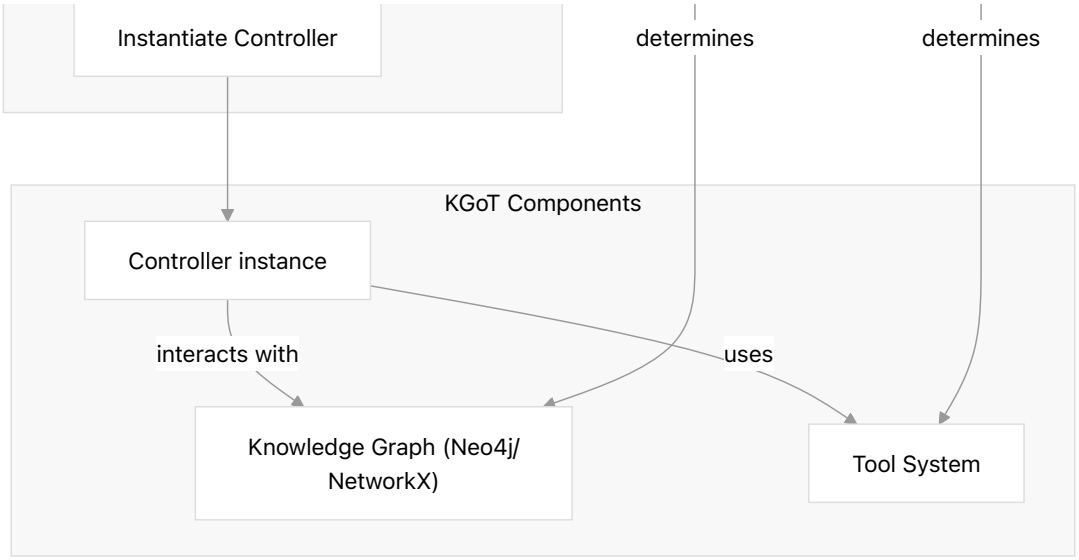
- Total tokens used (prompt and completion)
- Cost calculations
- Model usage information

This information is saved in the **`llm_cost.json`** and **`llm_cost_total.json`** files.

Sources: `GAIA/gaia.py` | `65–77` `GAIA/gaia.py` | `156–159` `GAIA/gaia.py` | `177` `GAIA/gaia.py` | `211`

Integration with KGoT Framework

When operating in KGoT mode, GAIA dynamically loads and initializes the appropriate components from the KGoT framework:



Sources: `GAIA/gaia.py` | `184-190` `GAIA/gaia.py` | `180-210`

The controller is dynamically imported based on the selected database and controller strategy:

```
controller_object = importlib.import_module(f"kgot.controller.{db_choice}.{controller_ch
```

This flexible architecture allows GAIA to leverage different reasoning strategies and knowledge graph implementations while maintaining a consistent interface.

Summary

The GAIA system provides a flexible architecture for solving complex reasoning problems using both direct LLM queries and the more sophisticated Knowledge Graph of Thoughts approach. Its key features include:

1. Dual execution modes (zero-shot and KGoT)
2. Highly configurable parameters
3. Integration with external services (Neo4j, Python executor, LLMs)
4. Comprehensive logging and statistics tracking
5. Dynamic component loading

This architecture enables GAIA to handle a wide range of problem types and complexity levels while maintaining detailed records of execution for analysis and improvement.

Menu

GAIA System

Relevant source files

Purpose and Scope

The GAIA (General AI Assistant) System is a question-answering framework built on top of the Knowledge Graph of Thoughts (KGoT) framework. It is designed to process and answer complex reasoning questions from the GAIA benchmark dataset. This document covers the architecture, components, and operation of the GAIA system, including its dataset processing capabilities and execution modes. For information about the underlying KGoT framework itself, see KGoT Framework.

System Overview

The GAIA system provides a high-level interface for answering complex reasoning questions. It supports two primary execution modes:

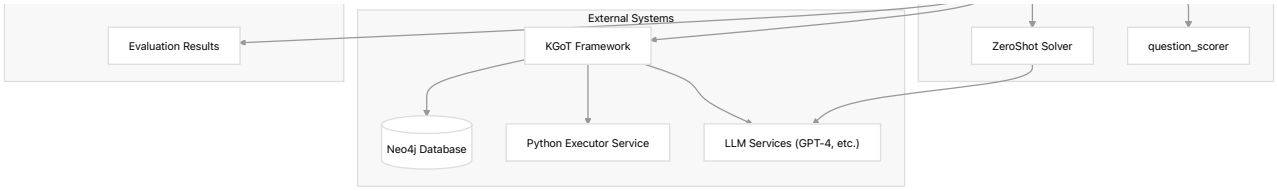
1. **Zero-shot mode**: Directly queries a Large Language Model (LLM) to answer questions without constructing a knowledge graph
2. **KGoT mode**: Utilizes the full Knowledge Graph of Thoughts framework for complex reasoning, building a knowledge graph dynamically while solving problems

GAIA serves as a practical application of the KGoT framework, specifically tailored for the GAIA benchmark tasks, which involve multi-step reasoning across various cognitive tasks.

Sources: `GAIA/gaia.py` `106–276` `GAIA/gaia.py` `10–27`

System Architecture

The GAIA system consists of several interconnected components that work together to process



The main component is **`gaia.py`**, which serves as the entry point for the system. It interfaces with either the KGoT framework (for complex reasoning) or the ZeroShot solver (for direct LLM queries). The system processes questions from the GAIA dataset, which is fetched and organized by the dataset fetcher component.

Sources: `GAIA/gaia.py` 1–276 `GAIA/dataset/fetch_gaia_dataset.py` 1–119

Execution Modes

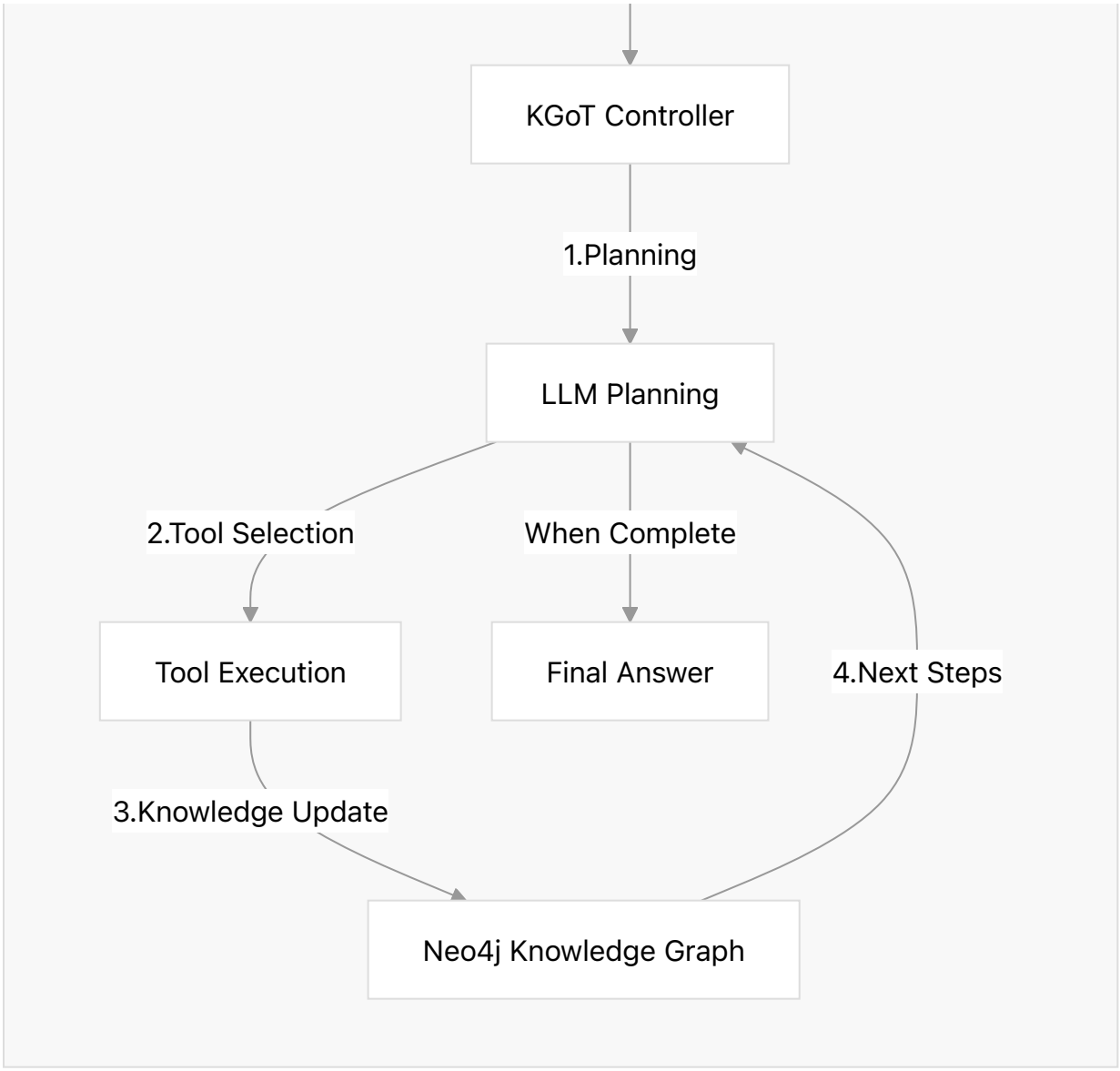
Zero-Shot Mode

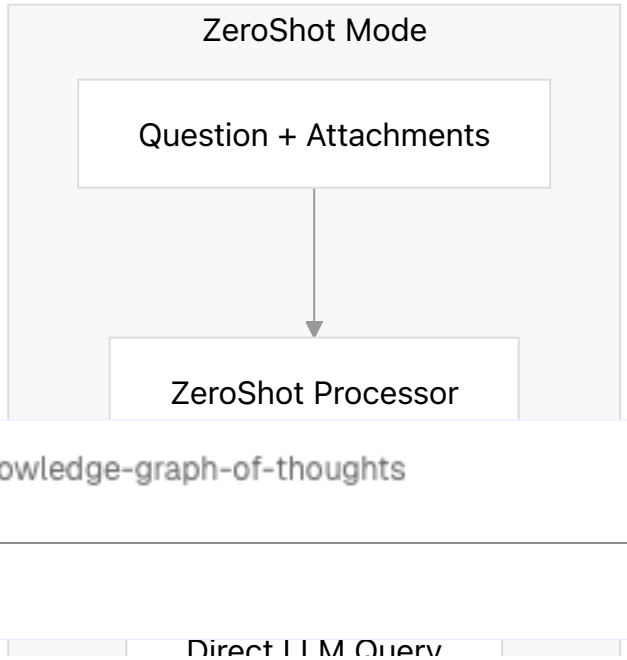
In Zero-Shot mode, the system bypasses the knowledge graph construction and directly sends the question to an LLM for answering. This serves as a baseline approach.

KGoT Mode

In KGoT mode, the system leverages the full capabilities of the Knowledge Graph of Thoughts framework, including:

- Dynamic construction of a knowledge graph
- Iterative reasoning through planning and execution steps
- Utilization of specialized tools for data manipulation
- Integration with Neo4j for knowledge storage and retrieval

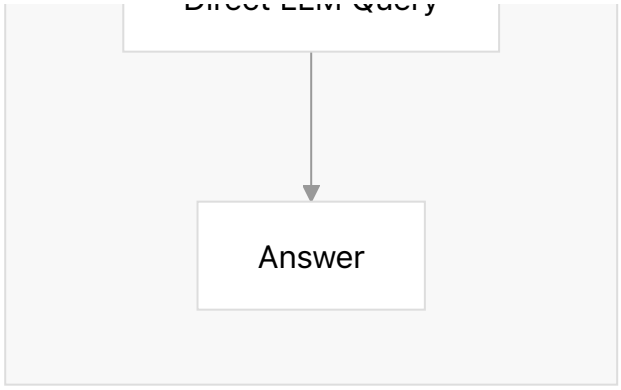




Ask Devin about spcl/knowledge-graph-of-thoughts

Deep Research

Direct LLM Query

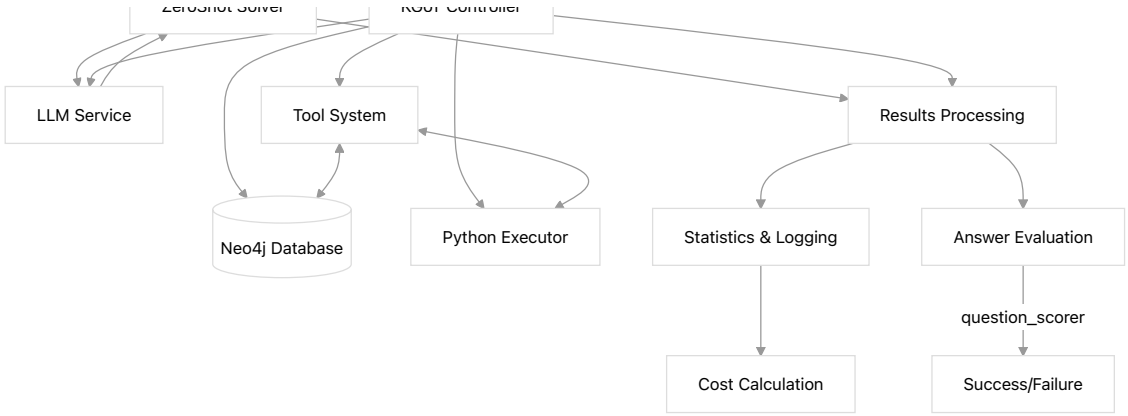


The execution mode is selected via the `--zero_shot` flag when running the GAIA system.

Sources: `GAIA/gaia.py` | `163-176` `GAIA/gaia.py` | `180-211`

Data Flow

The following diagram illustrates how data flows through the GAIA system:



Sources: `GAIA/gaia.py` | `106-212` `GAIA/dataset/fetch_gaia_dataset.py` | `18-119`

Dataset Processing

The GAIA system includes a dataset fetcher that downloads and processes the GAIA benchmark dataset from Hugging Face. This component:

1. Downloads the specified dataset split (validation or test)
2. Processes the metadata and organizes it into different levels
3. Sets up attachments for use by the system

The dataset fetcher handles the complexities of downloading and organizing the dataset, making it ready for use by the main GAIA system.

Sources: `GAIA/dataset/fetch_gaia_dataset.py` | `18-119`

Evaluation and Logging

The GAIA system includes comprehensive evaluation and logging capabilities:

1. **Answer Evaluation**: Compares the system's answers with expected answers
2. **Statistics Tracking**: Monitors LLM usage, costs, and performance metrics
3. **Logging**: Records detailed information about the system's operation

All results are stored in the specified log folder, including:

- Output logs (**`output.log`**)

This information enables detailed analysis of the system's performance and costs.

Sources: `GAIA/gaia.py` | `25–89` `GAIA/gaia.py` | `157–160`

Configuration Parameters

The GAIA system provides extensive configuration options through command-line arguments:

Parameter	Description	Default
`--log_folder_base`	Base folder for logging results	(Required)
`--gaia_file`	Path to GAIA JSON file	(Required)
`--attachment_folder`	Path to GAIA problems attachments folder	`GAIA/dataset/attachments/validation`
`--config_llm_path`	Path to LLM configuration file	`kgot/config_llms.json`
`--logger_level`	Logging level	`logging.INFO`
`--logger_file_mode`	Log file mode	`"a"`
`--neo4j_uri`	URI for Neo4j	`"bolt://localhost:7687"`
`--neo4j_username`	Neo4j username	`"neo4j"`
`--neo4j_password`	Neo4j password	`"password"`
`--python_executor_uri`	URI for Python tool executor	`"http://localhost:16000/run"`
`--max_iterations`	Max iterations for KGoT	`7`
`--llm_planning_model`	LLM planning model	`"gpt-4o-mini"`
`--llm_execution_model`	LLM execution model	`"gpt-4o-mini"`
`--controller_choice`	Controller choice	`"queryRetrieve"`
`--db_choice`	Database choice	`"neo4j"`

Additional parameters are available for fine-tuning the system's behavior, including retry limits,

temperature settings, and other execution parameters.

Sources: `GAIA/gaia.py` | `214–276`

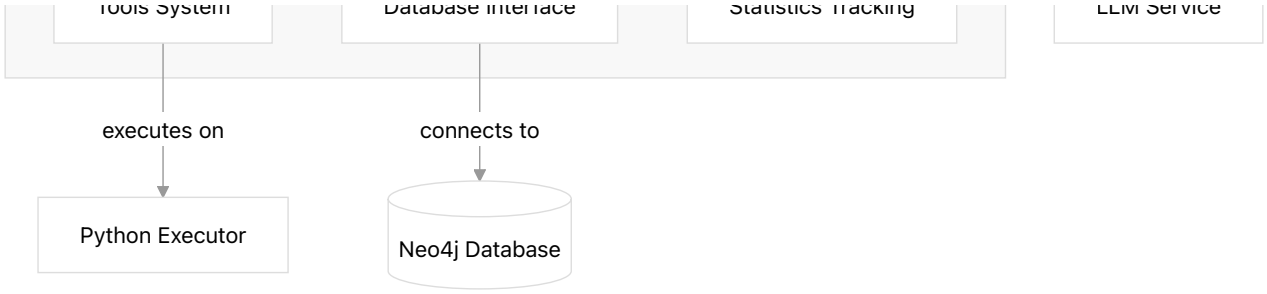
Execution Sequence

The following diagram illustrates the sequence of events when running the GAIA system:

This sequence illustrates how the GAIA system handles questions, showing the different paths for zero-shot versus KGoT modes, and highlighting the iterative nature of the KGoT approach.

Sources: `GAIA/gaia.py` | `106–212` `GAIA/gaia.py` | `25–89`

1. **Controller Selection**: GAIA dynamically imports and initializes the appropriate KGoT controller based on configuration
2. **Tool Integration**: GAIA configures and uses the KGoT tool system for data manipulation
3. **Knowledge Graph Management**: GAIA utilizes the KGoT framework's knowledge graph capabilities for persistent storage



This integration allows the GAIA system to leverage the full capabilities of the KGoT framework while providing a simplified interface focused on the GAIA benchmark tasks.

Sources: `GAIA/gaia.py` | `180–212` `GAIA/gaia.py` | `184–209`

Usage Example

To run the GAIA system, use the **`gaia.py`** script with appropriate parameters:

For KGoT mode:

```
python GAIA/gaia.py --log_folder_base ./results/kgot --gaia_file ./GAIA/validation/merge
```

The system will process each question in the dataset, generate answers, evaluate results, and produce detailed logs and statistics in the specified log folder.

Sources: `GAIA/gaia.py` | `214-276`

DeepWiki spcl/knowledge-graph-of-thoughts

Share

Ask Devin about spcl/knowledge-graph-of-thoughts

Deep Research

Menu

ExtractZipTool

Relevant source files

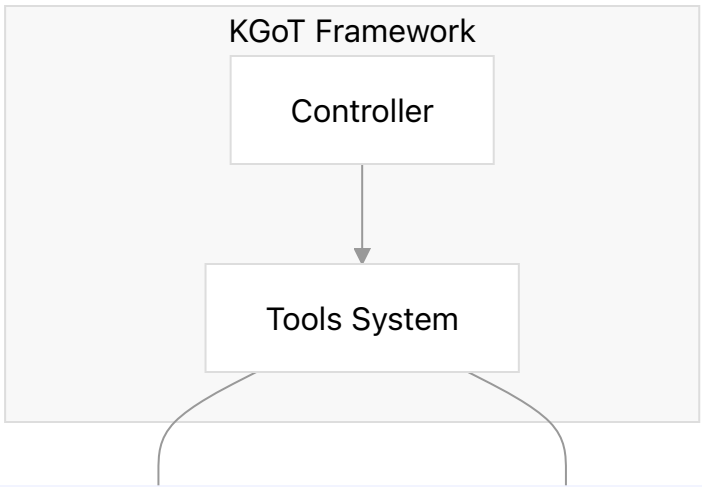
Purpose and Scope

This document provides technical documentation for the `ExtractZipTool` component, which is a utility tool in the Knowledge Graph of Thoughts (KGoT) framework that extracts the contents of zip files. This tool is part of the KGoT tools system and is used by the Controller component to manage and access compressed file contents. For general information about the KGoT Tools System, see Tools System.

Sources: `kgot/tools/tools_v2_3/ExtractZipTool.py` `1–77`

Component Overview

The `ExtractZipTool` provides a standardized interface for extracting zip archives within the KGoT ecosystem. It allows the AI controller to unpack compressed files and access their contents through other file inspection tools.

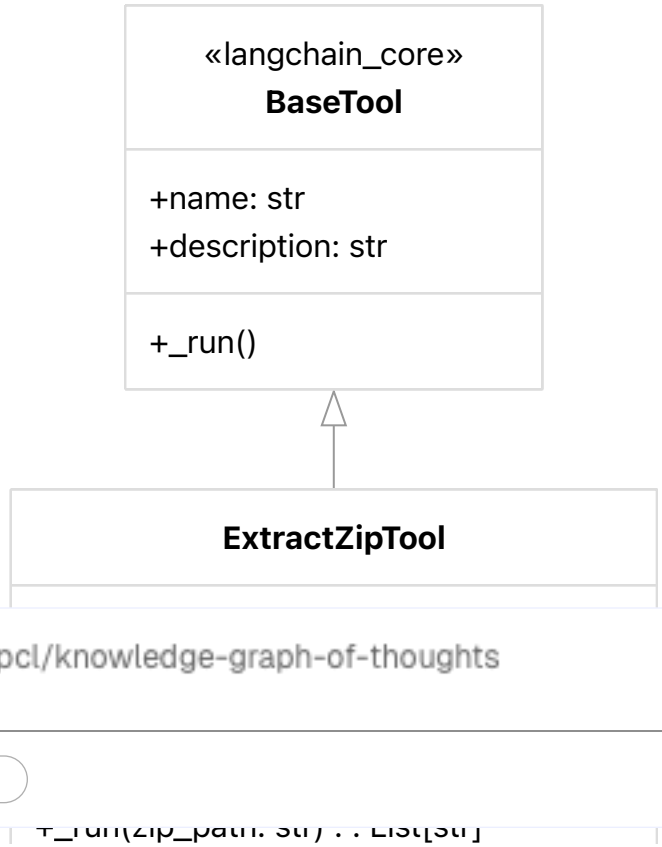


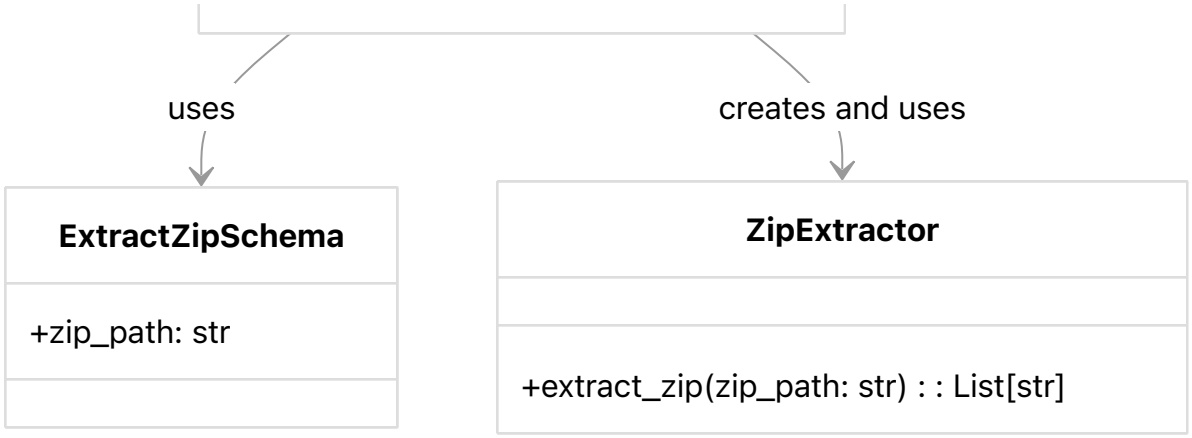
Sources: `kgot/tools/tools_v2_3/ExtractZipTool.py` `11–19`

`kgot/tools/tools_v2_3/ExtractZipTool.py` `63–77`

Class Structure

The **`ExtractZipTool`** implementation consists of three main classes working together:





Sources: `kgot/tools/tools_v2_3/ExtractZipTool.py` | `21–77`

Component Details

ZipExtractor

The **`ZipExtractor`** class implements the core functionality for extracting zip files:

- **Purpose**: Handles the actual extraction of zip files to a designated folder
- **Main Method**: **`extract_zip(zip_path: str) -> List[str]`**
- **Validation**: Checks if the provided file is actually a zip file and not an image or other file type
- **Extraction Location**: Creates a directory named after the zip file with "_EXTRACTED" appended
- **Output**: Returns a list of paths to all extracted files or appropriate error messages

Sources: `kgot/tools/tools_v2_3/ExtractZipTool.py` | `21–57`

ExtractZipSchema

This Pydantic model defines the input schema for the tool:

- **Purpose**: Validates the input parameters for the **`ExtractZipTool`**
- **Fields**:

ExtractZipTool

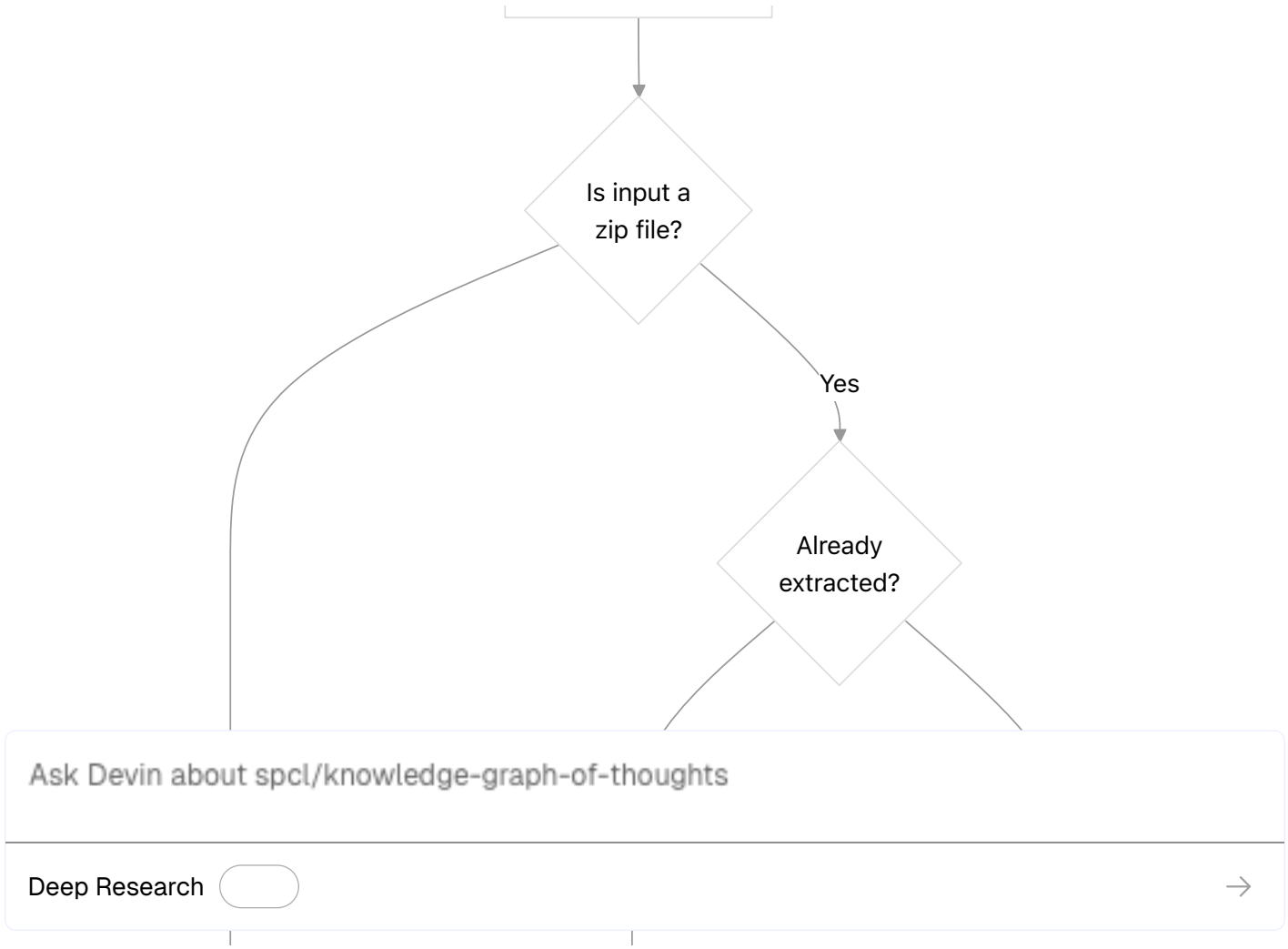
The main tool class that integrates with Langchain's tool system:

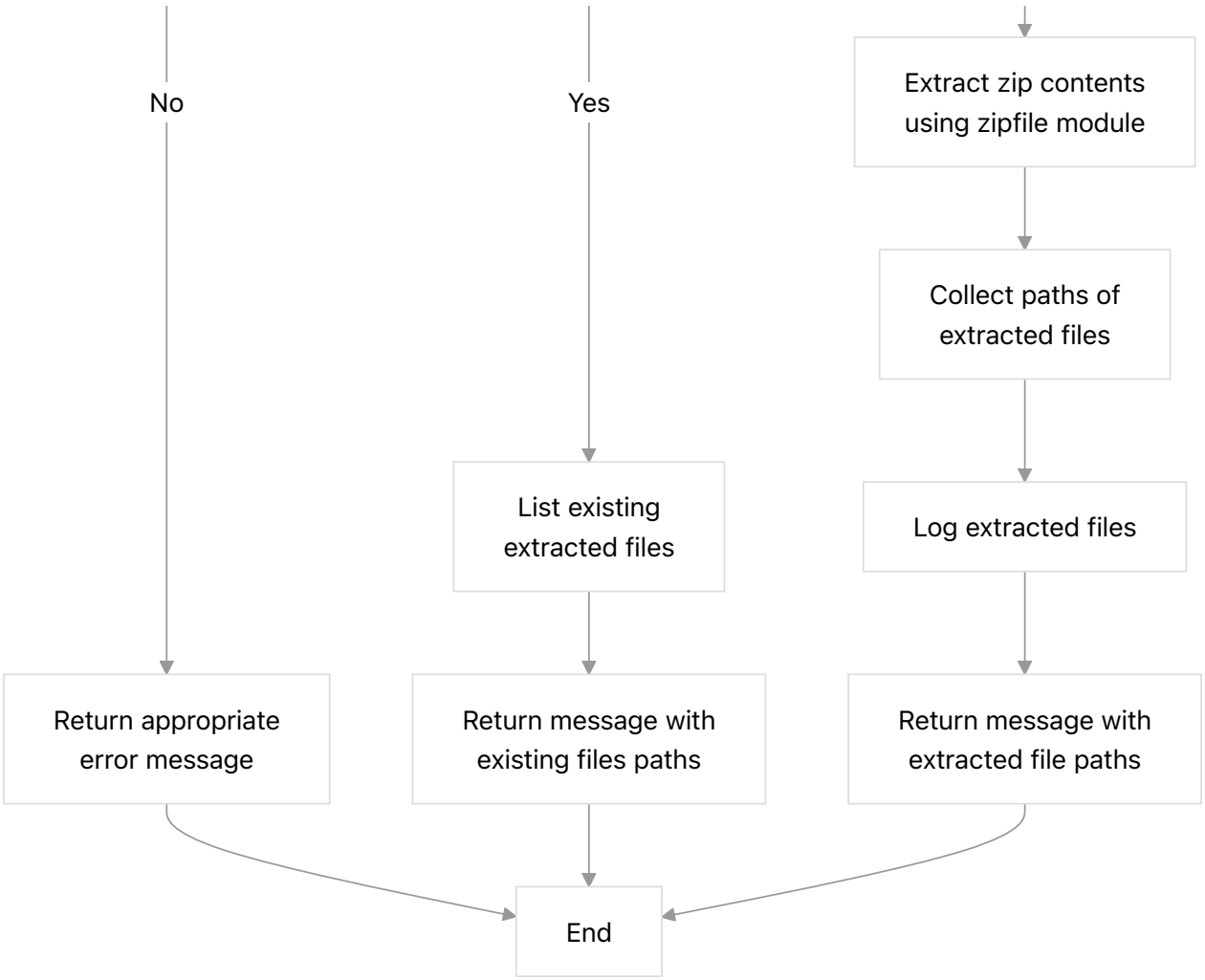
- **Purpose**: Provides a standardized interface for zip extraction within the KGoT framework
- **Base Class**: Inherits from Langchain's `BaseTool`
- **Name**: "extract_zip" (used to invoke the tool)
- **Description**: Detailed explanation of tool purpose and usage constraints
- **Args Schema**: Uses the `ExtractZipSchema` for input validation
- **Runtime Method**: `_run(zip_path: str) -> List[str]` delegates extraction to `ZipExtractor`

Sources: `kgot/tools/tools_v2_3/ExtractZipTool.py` | `63–77`

Execution Flow

The extraction process follows a specific workflow:





Sources: `kgot/tools/tools_v2_3/ExtractZipTool.py` | `22–57`

Validation and Error Handling

The **`ExtractZipTool`** includes several validation checks to handle edge cases:

Input Type	Behavior	Response
Image file (.png, .jpg, etc.)	Rejects the input	Suggests using **`image_inspector`** tool

Sources: `kgot/tools/tools_v2_3/ExtractZipTool.py` | `23–38`

Integration with Other Tools

The **`ExtractZipTool`** is designed to work alongside other tools in the KGoT ecosystem:

1. It handles only the extraction of zip files, not the inspection of file contents
2. It explicitly refers users to other tools for content inspection:
   - **`inspect_file_as_text`** for text-based files
   - **`image_inspector`** for image files

This separation of concerns allows the Controller to orchestrate a multi-step process of extracting and then examining file contents.

Sources: `kgot/tools/tools_v2_3/ExtractZipTool.py` | `53–57`

`kgot/tools/tools_v2_3/ExtractZipTool.py` | `66–70`

Tool Limitations

The tool has several intentional limitations:

1. Only processes files with **`.zip`** extension
2. Does not directly return file contents - only paths to extracted files
3. Does not handle image files even if they are in a zip format
4. Requires additional tool invocations to inspect extracted file contents

These limitations maintain a clear separation of responsibilities between different tools in the KGoT framework.

Sources: `kgot/tools/tools_v2_3/ExtractZipTool.py` | `23–27`

`kgot/tools/tools_v2_3/ExtractZipTool.py` | `66–72`

Menu

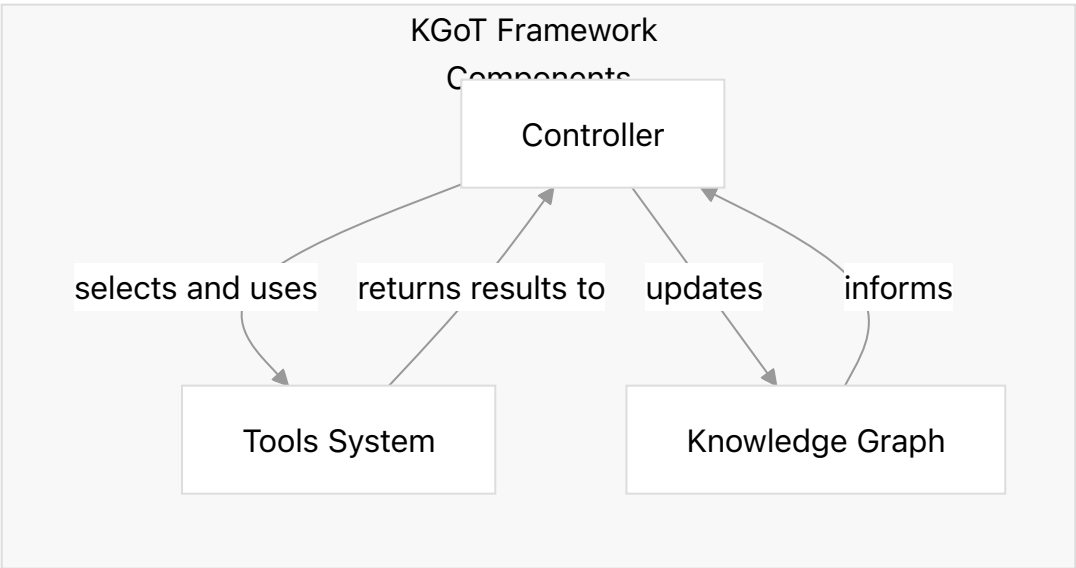
Tools System

Relevant source files

The Tools System in the Knowledge Graph of Thoughts (KGoT) framework provides specialized functionalities that enable the controller to manipulate data, analyze information, and interact with external systems during the reasoning process. This system is a critical part of the KGoT architecture, allowing the framework to perform complex operations through a standardized interface.

Purpose and Architecture

The Tools System provides a collection of modular tools that extend the capabilities of the KGoT framework beyond what the language model can do directly. These tools enable file operations, code execution, information extraction, and other tasks that require interaction with the external environment.



Ask Devin about spcl/knowledge-graph-of-thoughts

Deep Research

The tools are organized into versioned modules, with the current version used by default being

"tools_v2_3" as specified in the CLI interface. This versioning approach allows for evolution of the tools system while maintaining backward compatibility.

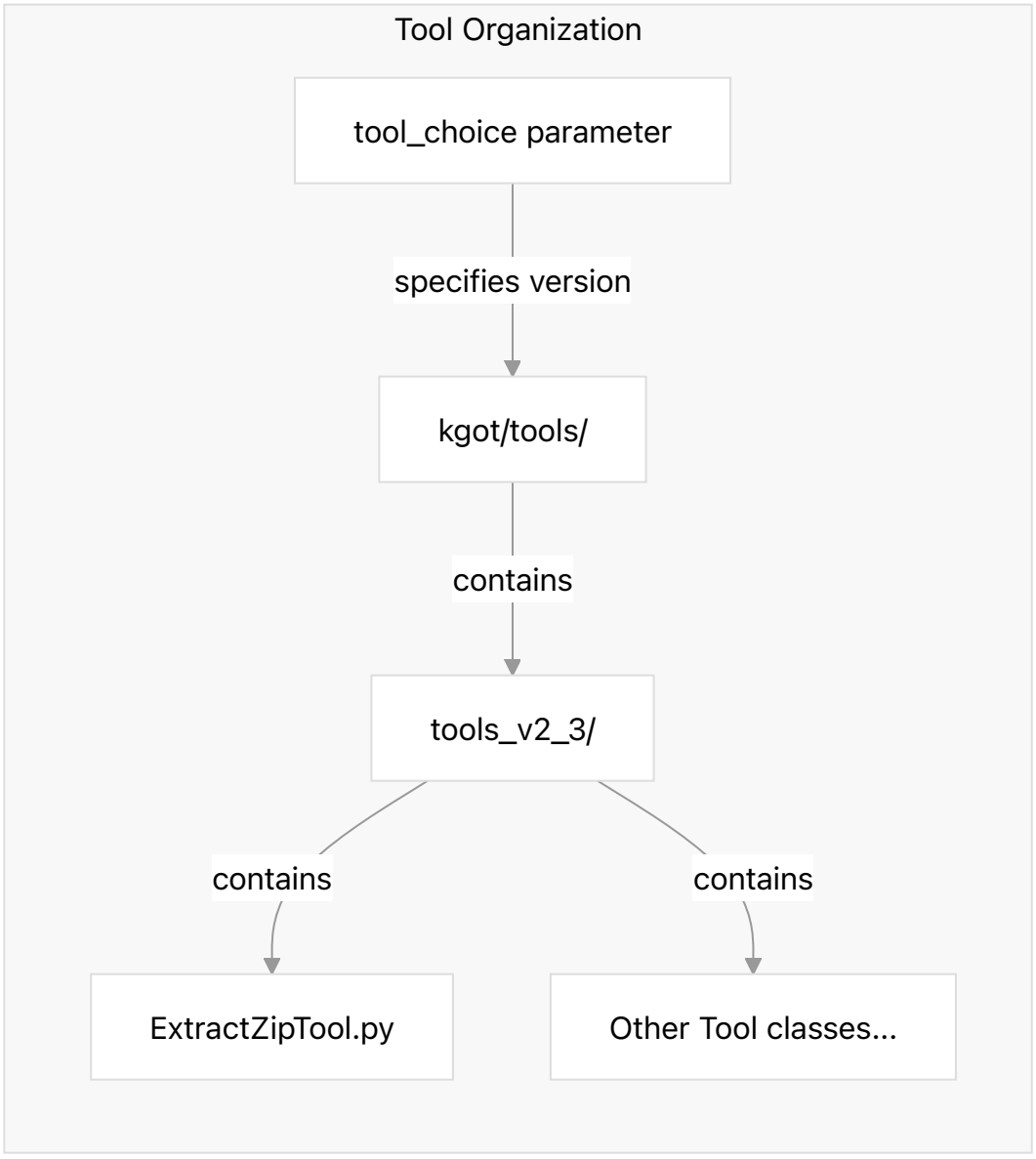


Diagram: Tools System Directory Structure

Sources: `kgot/__main__.py` | `188`

across all tools.

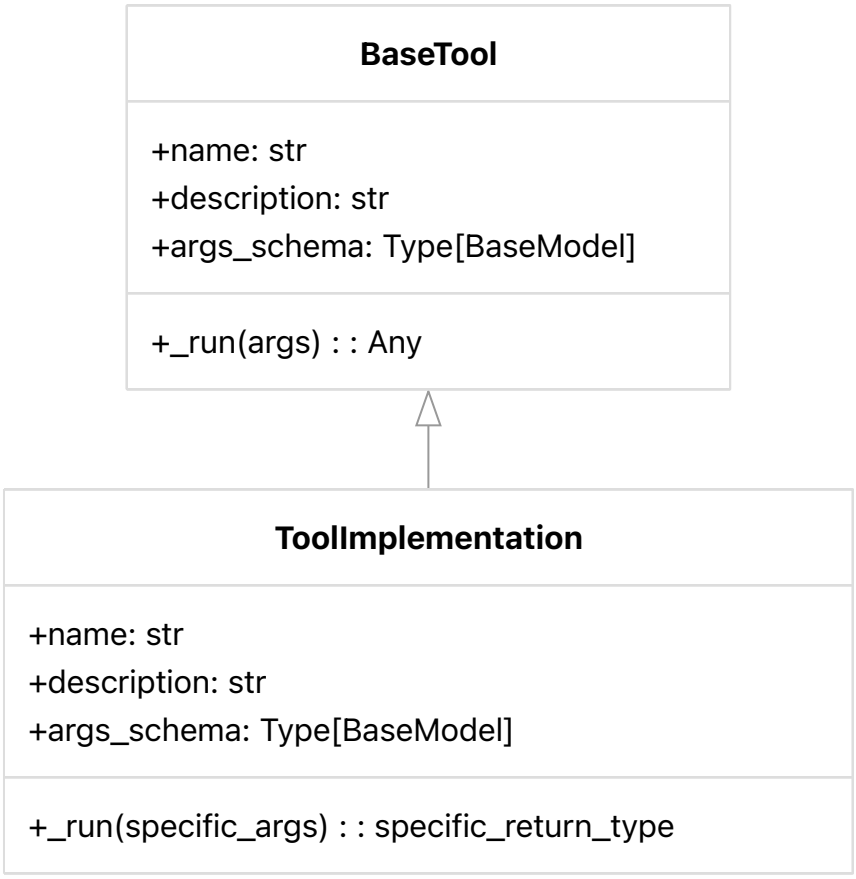


Diagram: Basic Tool Implementation Pattern

Sources: `kgot/tools/tools_v2_3/ExtractZipTool.py` | `63–77`

Every tool must implement:

1. A unique **`name`** property that identifies the tool
2. A detailed **`description`** property that explains the tool's functionality and usage
3. An **`args_schema`** property that defines expected arguments using Pydantic models
4. A **`_run`** method that contains the tool's core functionality

ExtractZipTool: A Detailed Example

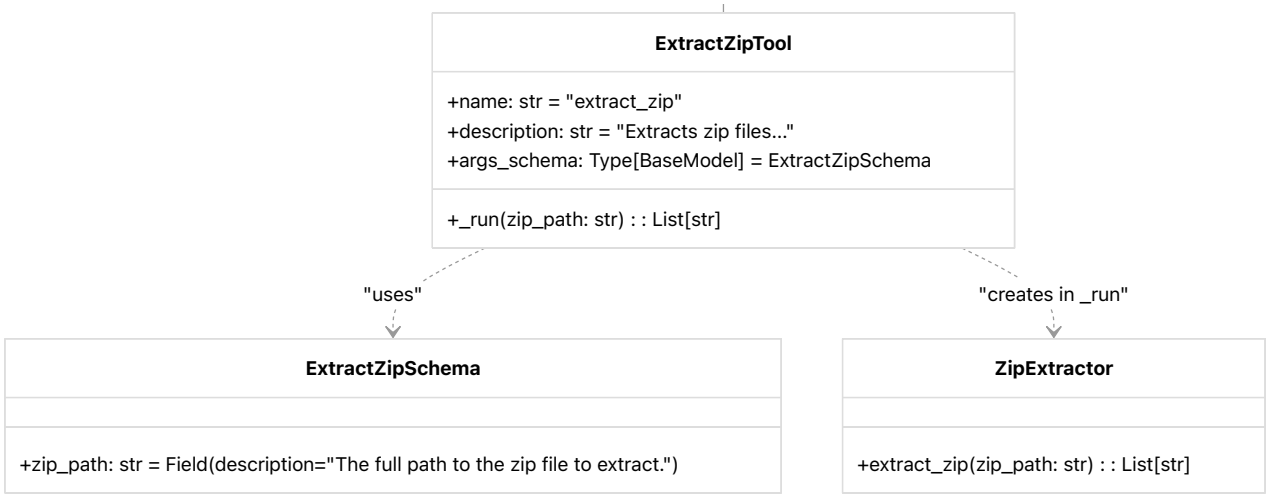


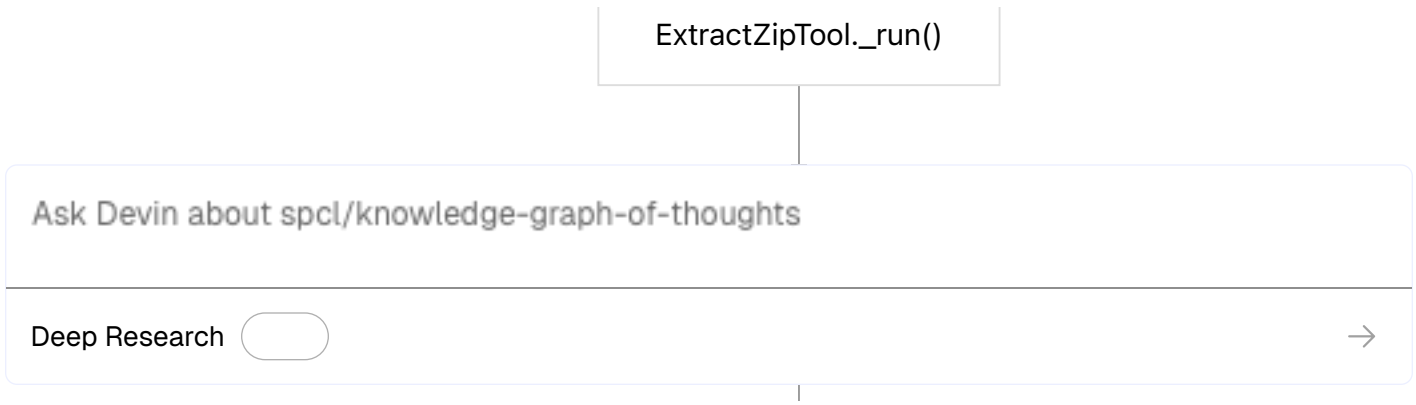
Diagram: ExtractZipTool Implementation Structure

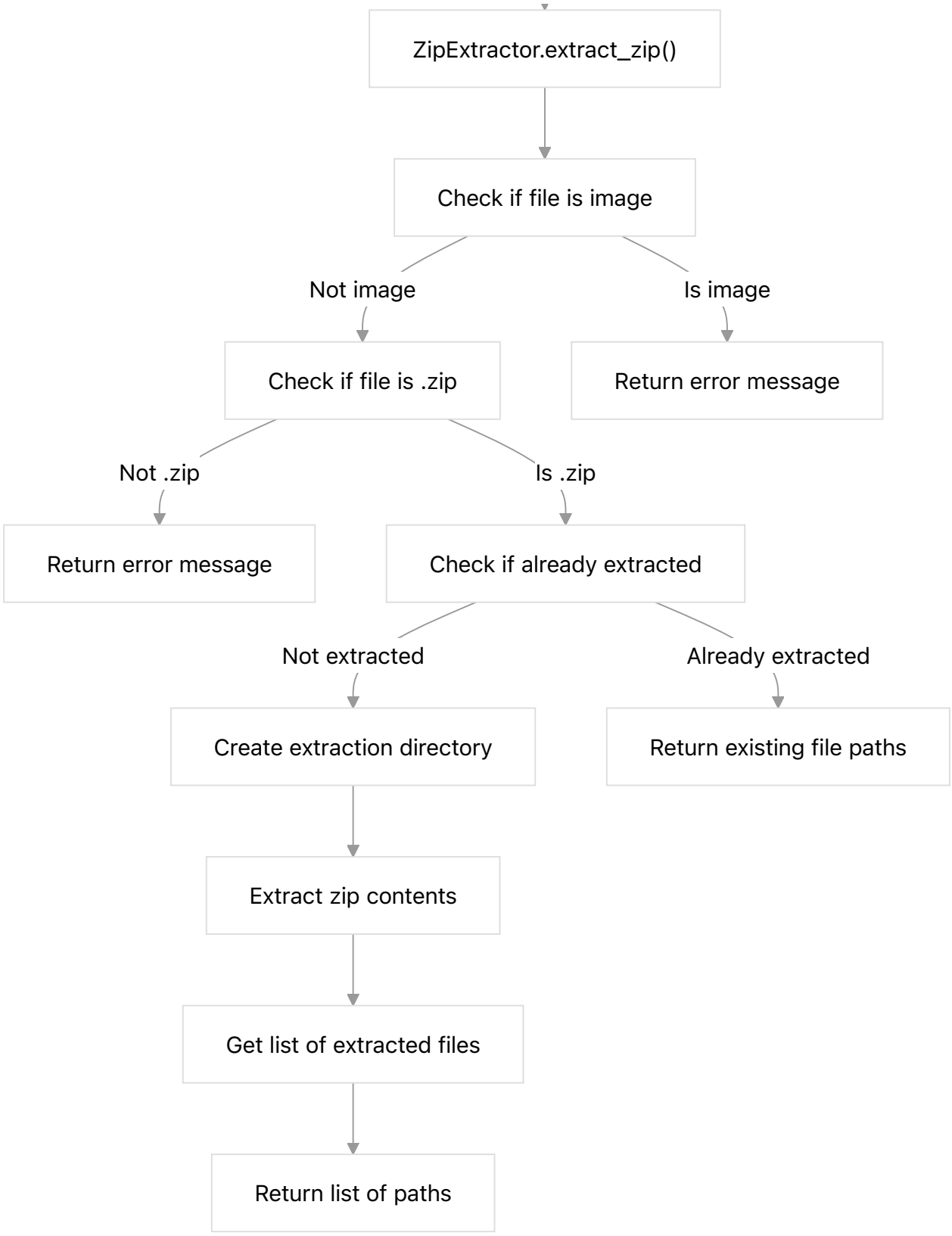
Sources: `kgot/tools/tools_v2_3/ExtractZipTool.py` | `60–77`

The implementation follows a clean separation of concerns:

- **`ExtractZipSchema`** defines the expected input (a zip file path)
- **`ExtractZipTool`** extends BaseTool and defines the interface
- **`ZipExtractor`** contains the actual zip extraction logic
- The **`_run`** method connects these components

Here's the execution flow of the ExtractZipTool:





Tool Configuration and Usage

In the KGoT framework, tools are configured during controller initialization via the `tool_choice` parameter. This parameter determines which version of the tools module to load.

Diagram: Tool Usage in Controller

Sources: `kgot/__main__.py` 97–99 `kgot/__main__.py` 188

During execution of the KGoT framework:

1. The controller is initialized with the specified `tool_choice` (default: "tools_v2_3")
2. When the controller needs to perform a file operation, it selects and calls the appropriate tool
3. The tool executes its functionality and returns results
4. The controller incorporates these results into the reasoning process

Tool Selection and Integration

The selection of which tool to use for a specific task is made by the controller based on the current

3. Invokes the tool with these arguments

4. Processes the results and updates the knowledge graph accordingly

This integration of specialized tools with the reasoning process is a key feature of the KGoT framework, allowing it to combine the capabilities of language models with practical system operations.

Sources: `kgot/__main__.py` `80–106`

Table of Core Properties

Property	Description
`name`	Unique identifier used to invoke the tool
`description`	Detailed explanation of the tool's functionality and usage
`args_schema`	Pydantic model defining expected arguments
`_run` method	Core implementation that executes the tool's functionality

The Tools System can be extended with new tools by creating new classes that follow this implementation pattern and adding them to the appropriate tools module.

Sources: `kgot/tools/tools_v2_3/ExtractZipTool.py` `63–77`

For detailed information about specific tools like the ExtractZipTool, see ExtractZipTool.

Menu

Statistics and Logging

Relevant source files

Overview

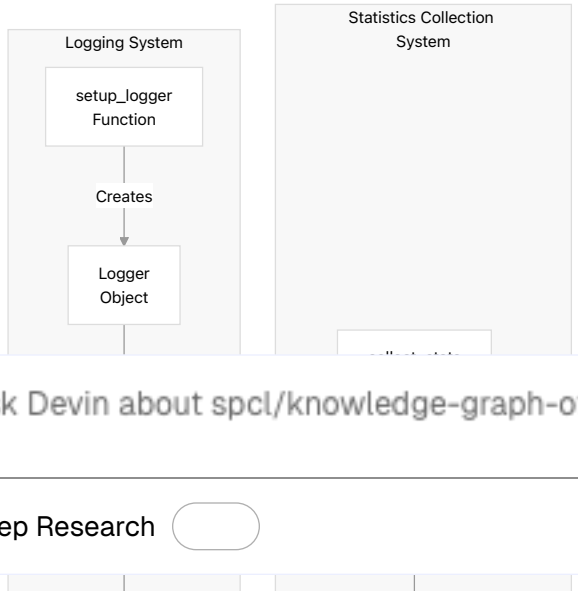
The Statistics and Logging system in the Knowledge Graph of Thoughts (KGoT) framework provides comprehensive utilities for tracking LLM usage metrics, calculating costs, and maintaining logs throughout the system's operation. This component enables performance monitoring, cost analysis, and debugging capabilities essential for both development and production use.

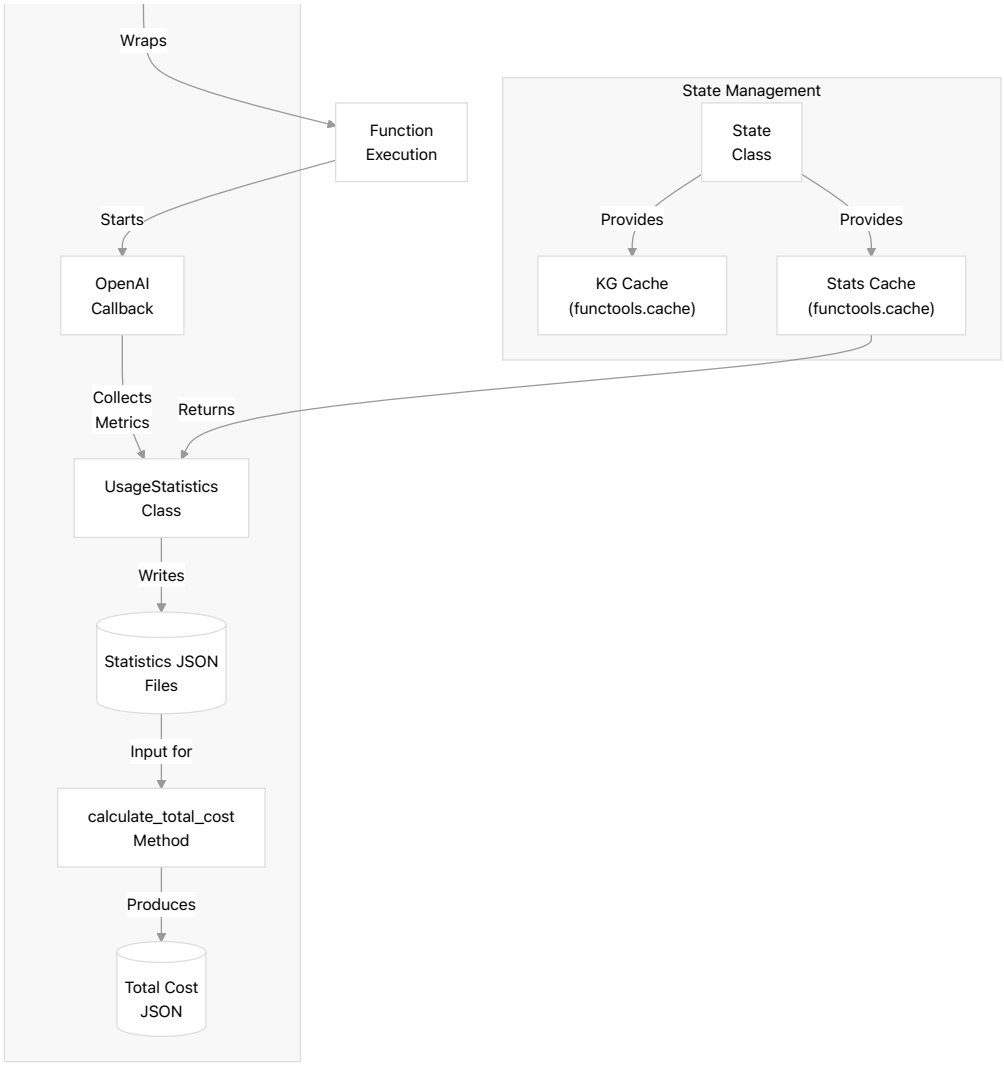
This document details the architecture and usage of the statistics collection and logging mechanisms. For information about the tools system that uses these capabilities, see Tools System.

Sources: `kgot/utils/log_and_statistics.py` `1–246`

Architecture

The Statistics and Logging component consists of several interconnected parts that work together to collect, store, and analyze usage data and logs. The system is designed to track LLM interactions with minimal impact on the core functionality.





Sources: `kgot/utils/log_and_statistics.py` 26–152 `kgot/utils/log_and_statistics.py` 155–198 `kgot/utils/log_and_statistics.py` 202–246

Key Components

UsageStatistics Class

The **`UsageStatistics`** class is responsible for tracking and recording LLM usage statistics,

- Execution time

- Model information

+str statistics_file_name
+DataFrame stats_df

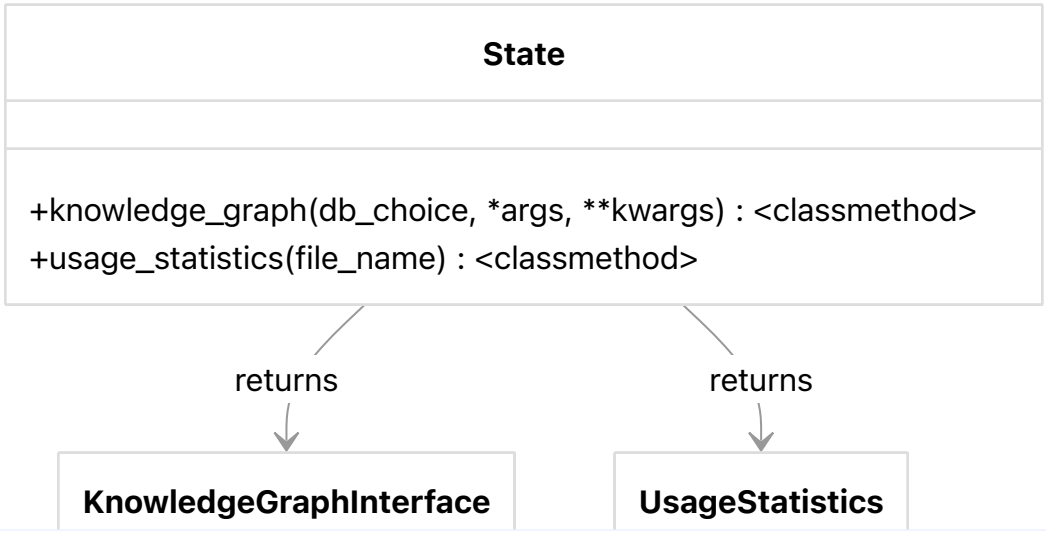
+**init**(file_name)
+log_statistic(function_name, start_time, end_time, model, prompt_tokens, completion_tokens, cost)
+calculate_total_cost(input_log_file, output_log_file) : <static>

The statistics are stored in a JSON file (default: `llm_cost.json`) with each function call recorded as a separate entry. The class also provides functionality to calculate aggregated statistics across all entries.

Sources: `kgot/utils/log_and_statistics.py` | `26–119`

State Class

The `State` class provides a global access point for the knowledge graph and usage statistics. It uses `functools.cache` to ensure efficient access to these resources.



recreate them each time.

Sources: `kgot/utils/log_and_statistics.py` `122-152`

Collect_stats Decorator

The **`collect_stats`** decorator wraps functions to collect statistics about their execution, particularly focusing on LLM calls. It:

1. Determines the model name from the function arguments
2. Uses the OpenAI callback mechanism to track token usage and cost
3. Measures execution time
4. Logs the collected statistics

```
@collect_stats(func_name="my_function_name")
def my_function(llm, ...):
    # Function implementation
```

Sources: `kgot/utils/log_and_statistics.py` `155-198`

Logging System

The **`setup_logger`** function creates customized loggers that can write to files or the console. These loggers support:

- Custom log levels
- File or console output
- Custom formatting
- Propagation control

The logging system complements the statistics tracking by providing qualitative information about system operation.

The following diagram illustrates how data flows through the Statistics and Logging system:

The following diagram illustrates how data flows through the Statistics and Logging system:

The statistics collection process is designed to be minimally intrusive to the main system operation while still capturing all relevant metrics.

Sources: `kgot/utils/log_and_statistics.py` 155–198

`kgot/utils/log_and_statistics.py` 26–119

Usage

Statistics Collection

To track LLM usage in a function, apply the **`collect_stats`** decorator:

```
from kgot.utils.log_and_statistics import collect_stats

@collect_stats(func_name="generate_response")
def generate_response(llm, prompt):
    # Function implementation using LLM
    return llm.generate(prompt)
```

After collecting statistics, you can calculate totals using:

```
from kgot.utils.log_and_statistics import UsageStatistics

# Calculate totals from collected statistics
UsageStatistics.calculate_total_cost(
    input_log_file="llm_cost.json",
    output_log_file="total_cost.json"
)
```

The output file will contain:

- Aggregated statistics by function name
- Total token usage (prompt and completion)
- Total cost
- Total execution time

Setting Up Logging

To create a custom logger:

```
from kgot.utils.log_and_statistics import setup_logger
import logging

# Create a logger that writes to a file
logger = setup_logger(
    name="my_module",
    level=logging.INFO,
    log_file="system.log",
    log_format="%(asctime)s - %(name)s - %(levelname)s - %(message)s"
)

# Use the logger
logger.info("Operation completed successfully")
logger.error("An error occurred: %s", error_message)
```


Statistics JSON Format

The statistics are stored in a line-delimited JSON format, with each line representing a single function call:

```
{"FunctionName": "generate_response", "StartTime": 1678901234.567, "EndTime": 1678901235
{"FunctionName": "answer_question", "StartTime": 1678901236.789, "EndTime": 1678901237.8
```

Total Cost JSON Format

The aggregated statistics are stored in a structured JSON format:

```
{
    "generate_response": {
        "TotalPromptTokens": 450,
        "TotalCompletionTokens": 150,
        "TotalCost": 0.036,
        "TotalDuration": 3.333
    },
    "answer_question": {
        "TotalPromptTokens": 300,
        "TotalCompletionTokens": 90,
        "TotalCost": 0.0045,
        "TotalDuration": 3.303
    },
    "FinalTotal": {
        "TotalPromptTokens": 750,
        "TotalCompletionTokens": 240,
        "TotalCost": 0.0405,
        "TotalDuration": 6.636
    }
}
```

Sources: `kgot/utils/log_and_statistics.py` 67–69

`kgot/utils/log_and_statistics.py` 72–119

The integration ensures that:

1. All LLM interactions are properly tracked for cost and usage analysis
2. Components can easily access the knowledge graph and statistics through the `State` class
3. Comprehensive logging is available throughout the system

Sources: `kgot/utils/log_and_statistics.py` `122–152`

`kgot/utils/log_and_statistics.py` `155–198`

Conclusion

The Statistics and Logging component provides essential infrastructure for monitoring, debugging, and analyzing the KGoT system's operation. By tracking LLM usage, calculating costs, and maintaining logs, it enables developers to optimize performance and understand system behavior.

This component acts as a cross-cutting concern that touches all parts of the system, providing

Menu

CLI Interface

Relevant source files

The Knowledge Graph of Thoughts (KGoT) Command Line Interface (CLI) provides a user-friendly way to interact with the KGoT system from the command line. It serves as the primary entry point for solving problems using the KGoT framework, which integrates Large Language Models (LLMs) with dynamically constructed knowledge graphs.

For information about the broader KGoT framework, see KGoT Framework. For details about deployment and execution, see Deployment and Execution.

Command Structure

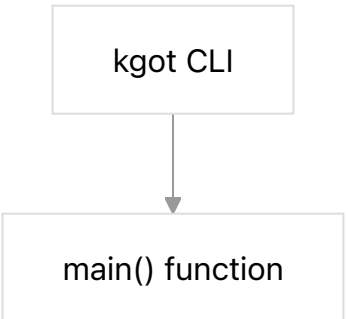
The KGoT CLI follows a command-based structure:

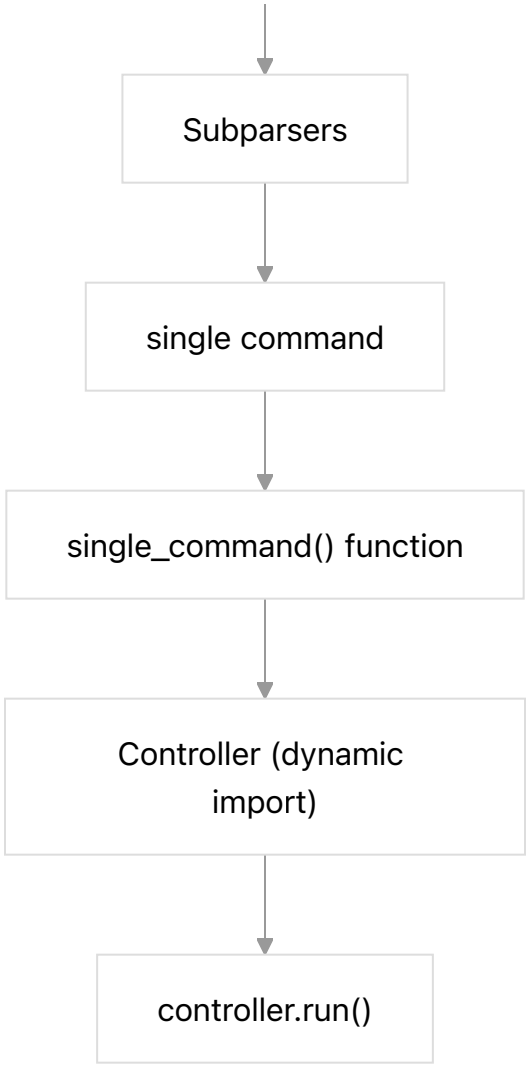
```
kgot [global options] <command> [command options]
```

Currently, the CLI supports one primary command:

- **`single`** : Solve a single problem given a statement and optional associated files

The CLI architecture is shown in the following diagram:





Sources: `kgot/__main__.py` `123–217`

Global Options

The CLI provides several global options to customize the behavior of the KGoT system:

Option	Description	Default
`-v` , **`--version`**	Show program's version number and exit	-
`-i` , **`--iterations`**	Maximum number of iterations to run	7

`--logger_file_mode`	Log file mode	"a"
`--statistics_file`	Path to store LLM usage statistics	"llm_cost.json"
`--num_next_steps_decision`	Number of next steps decision	5
`--max_retrieve_query_retry`	Maximum number of retries for retrieve query	3
`--max_cypher_fixing_retry`	Maximum number of retries for Cypher fixing	3
`--max_final_solution_parsing`	Maximum number of retries for final solution parsing	3
`--max_tool_retries`	Maximum number of retries for tools	6
`--max_llm_retries`	Maximum number of retries for LLM	6

Sources: `kgot/__main__.py` `142–172`

LLM Configuration Options

Option	Description	Default
`--llm-plan`	LLM model used for planning	"gpt-4o-mini"
`--llm-plan-temp`	Temperature for the planning LLM model	0.0
`--llm-exec`	LLM model used for tool execution	"gpt-4o-mini"
`--llm-exec-temp`	Temperature for the execution LLM model	0.0

Sources: `kgot/__main__.py` `174–181`

Component Selection Options

Option	Description	Default
`--controller_choice`	Controller implementation to use	"queryRetrieve"
`--db_choice`	Database implementation to use	"neo4j"

Sources: `kgot/__main__.py` `183–191`

Sources: kgot/__main__.py 183-191

Command: `single`

The **`single`** command is used to solve a single problem given a problem statement and optional associated files.

Usage

```
kgot [global options] single -p "Problem statement" [--files file1.txt file2.py ...]
```

Options

Option	Description	Required
`-p`, **`--problem`**	The problem statement to solve	Yes
`--files`	List of file paths associated with the problem	No

Sources: `kgot/__main__.py` `195-204`

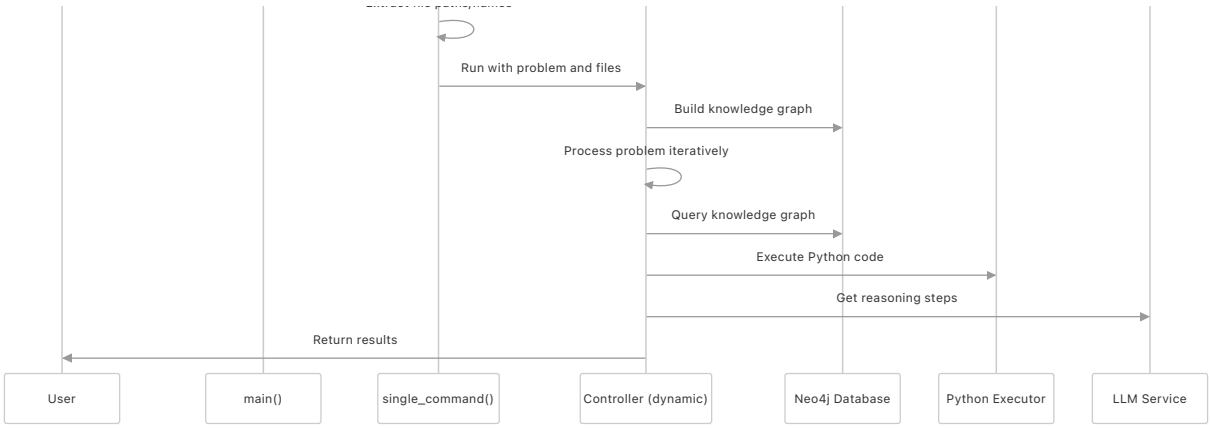
Environment Variables

The KGoT CLI uses the following environment variables:

Variable	Description	Default
`NEO4J_URI`	Neo4j database URI	"bolt://localhost:7687"
`NEO4J_USER`	Neo4j database user	"neo4j"
`NEO4J_PASSWORD`	Neo4j database password	"password"
`PYTHON_EXECUTOR_URI`	Python execution server URI	"http://localhost:16000/run"

These environment variables can be set in the shell or in a **`.env`** file in the current directory, which is loaded by the **`load_variables()`** function.

When executing a command using the KGoT CLI, the following steps occur:



Sources: `kgot/__main__.py` 61–122 `kgot/__main__.py` 123–217

Dynamic Component Loading

One of the key features of the KGoT CLI is its ability to dynamically load components based on user-specified options:

The CLI dynamically imports the controller module based on the specified database and controller choices:

```
controller_object = importlib.import_module(f"kgot.controller.{args.db_choice}.{args.con
```

Sources: `kgot/__main__.py` `80`

CLI Implementation Details

The KGoT CLI is implemented using Python's **`argparse`** module. The main parser handles global options, while subparsers handle command-specific options.

A custom formatter class (**`CustomFormatter`**) is used to format the help text in a user-friendly way:

```
class CustomFormatter(argparse.RawTextHelpFormatter, argparse.ArgumentDefaultsHelpFormat
    """Custom help formatter to align default values nicely."""
    # Implementation details...
```

Sources: `kgot/__main__.py` `19-50`

Examples

Basic Usage

```
kgot single -p "What is the time complexity of binary search?" --iterations 5
```

This command solves the given problem using the default configuration with a maximum of 5 iterations.

This command solves the given problem while providing access to the `sample.py` file.

Customizing LLM Models

```
kgot single -p "Solve this complex algorithm problem." --llm-plan "gpt-4" --llm-exec "gp
```

This command uses GPT-4 for planning steps and GPT-3.5 Turbo for executing tools.

Using Different Components

```
kgot single -p "Analyze this dataset." --db_choice "networkx" --controller_choice "alter
```

This command uses the NetworkX database implementation with an alternative controller.

Integration with KGoT Framework

The CLI serves as the primary interface to the KGoT framework, integrating with various components:

The CLI initializes the selected Controller with all necessary parameters:

```
controller = controller_object(
    neo4j_uri=neo4j_uri,
```

Sources: `kgot/__main__.py` | `83-106`

Snapshots

The KGoT system can save snapshots of the knowledge graph during execution. These snapshots are useful for debugging and analyzing the reasoning process. The **`-s`** or **`--snapshots`** option specifies where these snapshots should be stored:

```
kgot single -p "Your problem" -s "my_snapshots_dir"
```

If not specified, snapshots are stored in a directory named "snapshots".

Sources: `kgot/__main__.py` | `119`

Menu

KGoT Framework

Relevant source files

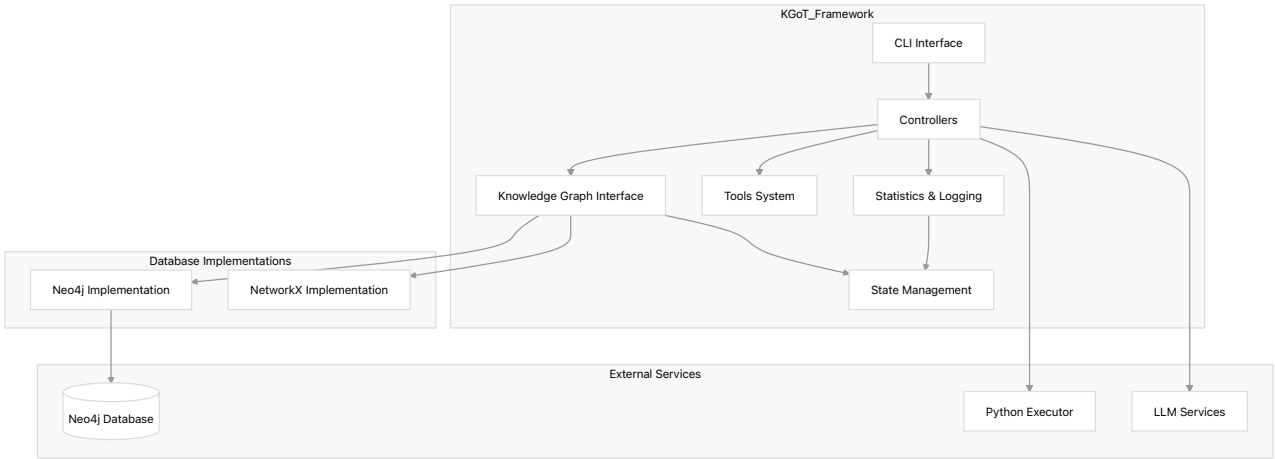
Purpose and Scope

The Knowledge Graph of Thoughts (KGoT) Framework is the core system that enables integration of large language models (LLMs) with dynamically constructed knowledge graphs to solve complex reasoning tasks. This document explains the architecture, components, and capabilities of the framework. For information about the GAIA system that builds upon this framework, see GAIA System.

Sources: `kgot/__main__.py` 1–218

Framework Architecture

The KGoT Framework consists of several integrated components that work together to facilitate reasoning with knowledge graphs:



Ask Devin about spcl/knowledge-graph-of-thoughts

Deep Research

Key Components

Command-Line Interface

The KGoT Framework provides a command-line interface for interacting with the system. The CLI supports commands for solving problems and offers various configuration options:

The CLI dynamically imports and initializes the appropriate controller based on the selected database and controller type. For more detailed information on the CLI, see CLI Interface.

Sources: `kgot/__main__.py` `123–211`

Controllers

Controllers are the heart of the KGoT framework, implementing different reasoning strategies for solving problems. They coordinate:

1. Knowledge graph initialization
2. Planning using LLMs
3. Tool execution
4. Graph updating
5. Solution generation

The framework supports multiple controller implementations, which are selected through the **`--controller_choice`** parameter. The default controller is **`queryRetrieve`**.

Sources: `kgot/__main__.py` `80–106`

1. Neo4j - A graph database (default)
2. NetworkX - A Python package for graph manipulation

The knowledge graph interface is instantiated through the `State` class, which maintains a global reference to the graph instance.

Sources: `kgot/utils/log_and_statistics.py` `122-138`

Tools System

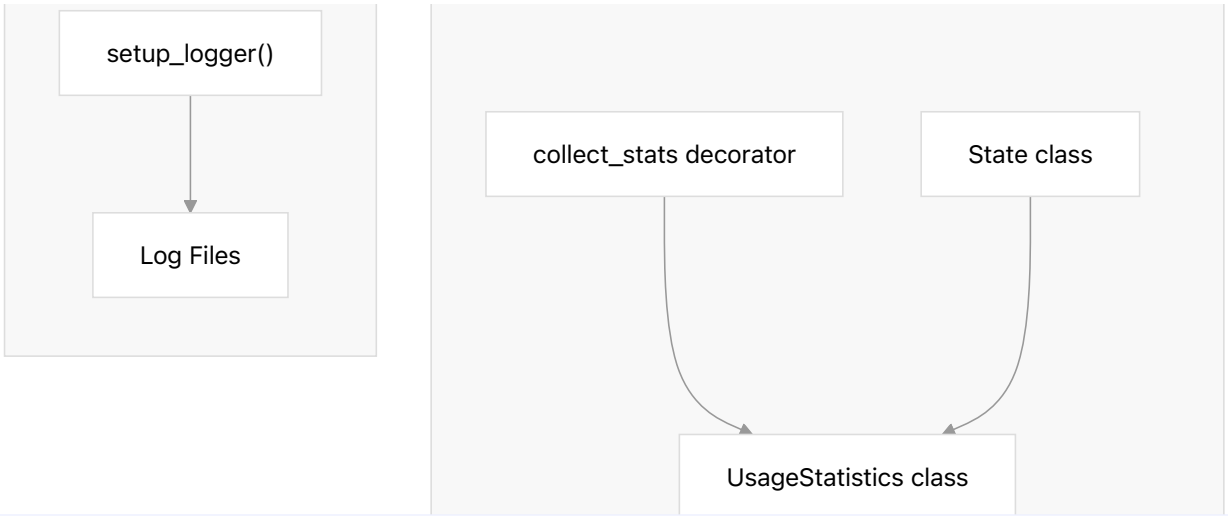
The Tools System provides controllers with functionality to gather information and manipulate the knowledge graph. The framework includes multiple tool versions, selected through the `--tool_choice` parameter. The default is `tools_v2_3`.

For more information on the Tools System, see Tools System.

Sources: `kgot/__main__.py` `187-188`

Statistics and Logging

The framework includes comprehensive capabilities for tracking LLM usage, costs, and performance:



The statistics system tracks LLM usage through the `collect_stats` decorator, which wraps LLM-calling functions and records:

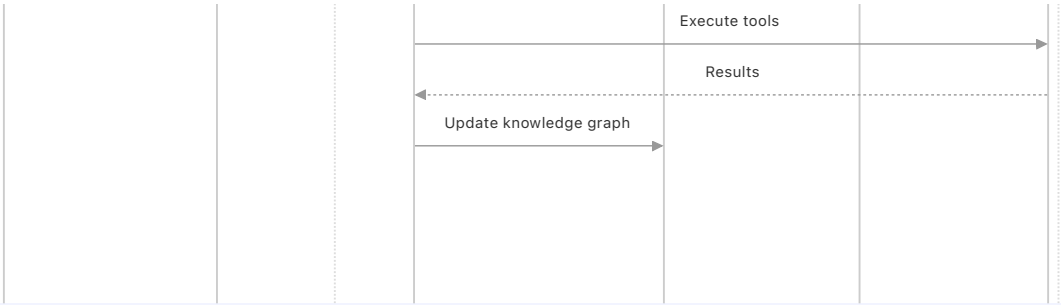
- Function name
- Execution time
- Model used
- Prompt and completion tokens
- Cost

For more detailed information, see Statistics and Logging.

Sources: `kgot/utils/log_and_statistics.py` `26–247`

Execution Flow

The typical execution flow of the KGoT Framework follows this pattern:

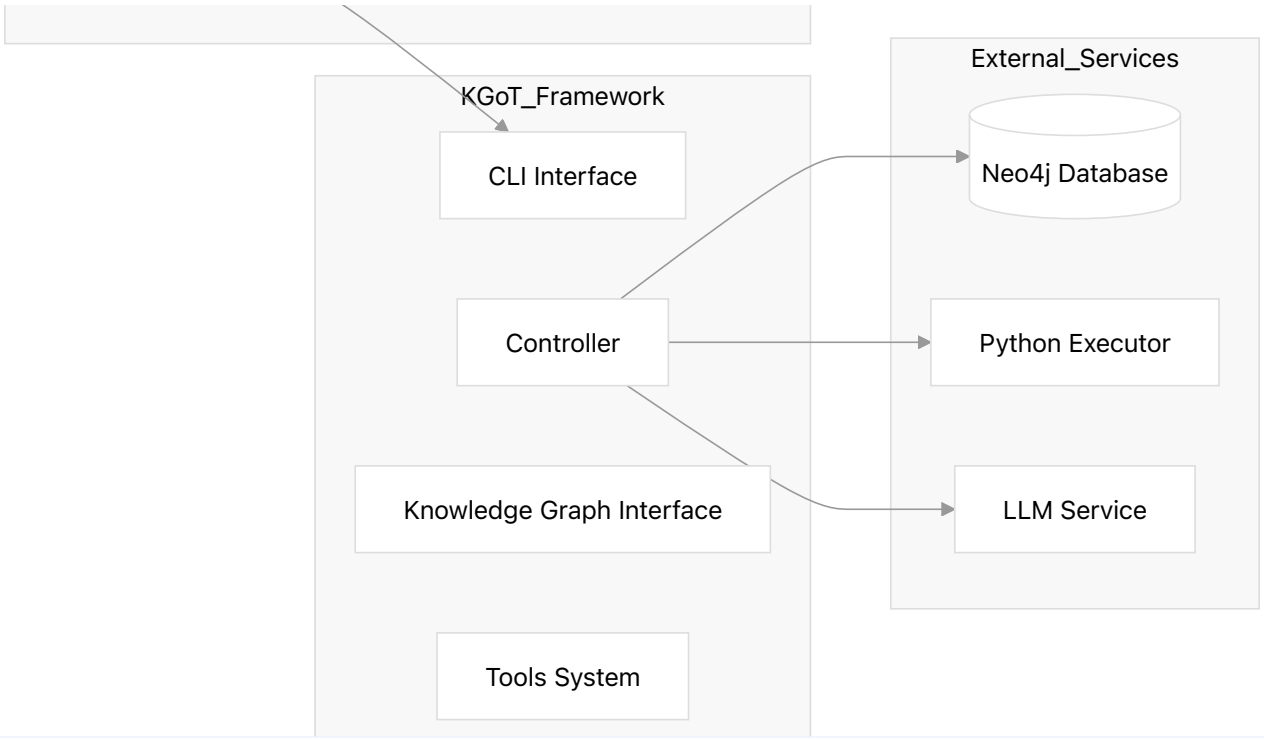


This process can run for up to the maximum number of iterations specified (default: 7), or until the controller determines a solution has been found.

Sources: `kgot/__main__.py` `61–120`

Framework Integration

The KGoT Framework is designed to be integrated with other components and services:



The framework relies on:

1. Neo4j database for storing the knowledge graph
2. Python executor for running code
3. LLM services for planning and reasoning

It's primarily used by the GAIA system (see GAIA System) to solve complex reasoning tasks.

Sources: `kgot/__main__.py` 1–218 `kgot/utils/log_and_statistics.py` 1–247

Configuration

The framework can be configured through command-line arguments, environment variables, and configuration files:

Configuration	Description	Default
`--iterations`	Maximum number of iterations	7
`--llm-plan`	LLM model for planning	gpt-4o-mini
`--llm-plan-temp`	Temperature for planning	0.0
`--llm-exec`	LLM model for execution	gpt-4o-mini
`--llm-exec-temp`	Temperature for execution	0.0
`--controller_choice`	Controller implementation	queryRetrieve
`--db_choice`	Database implementation	neo4j
`--tool_choice`	Tools version	tools_v2_3
`NEO4J_URI`	Neo4j database URI	bolt://localhost:7687
`NEO4J_USER`	Neo4j username	neo4j
`NEO4J_PASSWORD`	Neo4j password	password

Code Structure and Organization

The KGoT Framework is organized into several key modules:

- **`kgot/__main__.py`** - Entry point for CLI
- **`kgot/controller/`** - Controller implementations
- **`kgot/knowledge_graph/`** - Knowledge graph interfaces and implementations
- **`kgot/tools/`** - Tool implementations
- **`kgot/utils/`** - Utility functions including statistics and logging

Key classes include:

- **`Controller`** - Various controller implementations based on **`db_choice`** and **`controller_choice`**
- **`KnowledgeGraph`** - Graph database interfaces
- **`UsageStatistics`** - For tracking LLM usage and costs
- **`State`** - For managing global state

Sources: `kgot/__main__.py` `80–106` `kgot/utils/log_and_statistics.py` `26–151`

Usage Example

Here's a simple example of using the KGoT CLI to solve a problem:

```
python -m kgot single --problem "Calculate the factorial of 5" --iterations 10 --llm-pla
```

This command initializes the framework with the specified LLM models, runs for up to 10 iterations, and attempts to calculate the factorial of 5 using the knowledge graph approach.

Sources: `kgot/__main__.py` `195–204`

DeepWiki spcl/knowledge-graph-of-thoughts

Share

Ask Devin about spcl/knowledge-graph-of-thoughts

Deep Research

Menu

Overview

Relevant source files

This document provides a comprehensive introduction to the Knowledge Graph of Thoughts (KGoT) repository, a framework that integrates large language models (LLMs) with dynamically constructed knowledge graphs to solve complex reasoning tasks. This page covers the system's core components, architecture, and execution models.

Purpose and Scope

The Knowledge Graph of Thoughts (KGoT) framework is designed to enhance LLM reasoning capabilities by building and leveraging knowledge graphs during the problem-solving process. This approach allows for more structured reasoning, better memory management, and improved traceability compared to traditional LLM prompting techniques.

This overview explains the high-level architecture and components of the system. For more detailed information about specific subsystems, refer to:

- KGoT Framework details: KGoT Framework
- GAIA System: GAIA System
- Deployment instructions: Deployment and Execution

Sources: `GAIA/gaia.py` 10–130 `kgot/__main__.py` 9–21

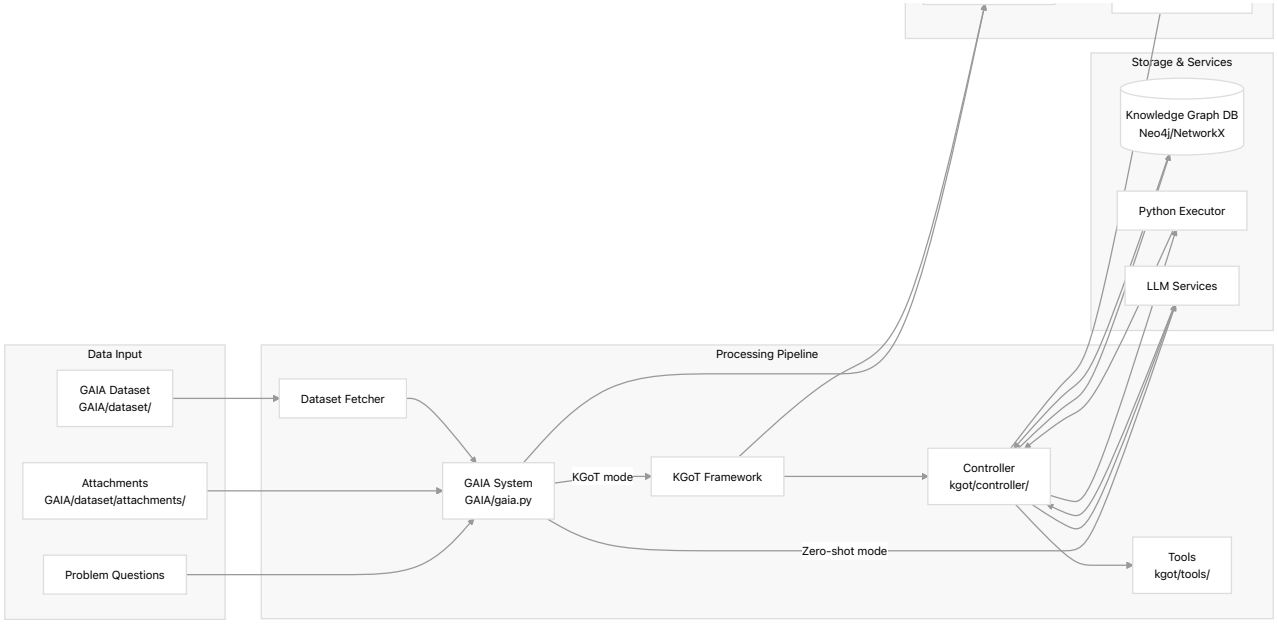
System Architecture

High-Level Component Architecture

The diagram above shows the main components of the KGoT system and their relationships. The core framework provides the reasoning and graph management capabilities, the GAIA system offers a high-level interface for answering questions, and the infrastructure components support the execution environment.

Sources: `GAIA/gaia.py` 1–276 `kgot/__main__.py` 1–217 `docker_instances/kgot_docker/Dockerfile` 1–30 `run_multiple_gaia.sh` 1–252

Data Flow Architecture



This diagram illustrates the data flow through the system, showing how questions and input data are processed through either the full KGoT framework or directly via the zero-shot approach.

Sources: GAIA/gaia.py 35–104 run_multiple_gaia.sh 185–242

KGoT Framework

The KGoT Framework is the core reasoning engine that builds and manages knowledge graphs to solve complex problems. It consists of:

1. **CLI Interface**: Command-line interface for interacting with the KGoT system
2. **Controllers**: Different reasoning strategies for problem-solving
3. **Tools System**: Extensible set of tools for data manipulation and analysis
4. **Knowledge Graph Interface**: Abstraction for working with different graph backends
5. **Statistics and Logging**: Utilities for tracking LLM usage and performance

The framework can be accessed directly via the `kgot` CLI or through the GAIA system.

Sources: `kgot/__main__.py` 60–121 `kgot/__main__.py` 123–188

GAIA System

The General AI Assistant (GAIA) system provides a higher-level interface built on top of the KGoT framework. Key features include:

1. **Multiple Execution Modes**:
   - KGoT mode: Uses the knowledge graph approach for complex reasoning
   - Zero-shot mode: Directly queries the LLM without graph construction
2. **Dataset Processing**: Manages the fetching and processing of problem datasets
3. **Evaluation**: Includes scoring mechanisms to evaluate solution correctness

The GAIA entrypoint script (`GAIA/gaia.py`) orchestrates the execution flow and handles result collection.

Sources: `GAIA/gaia.py` 106–211 `GAIA/gaia.py` 163–177 `GAIA/gaia.py` 180–211

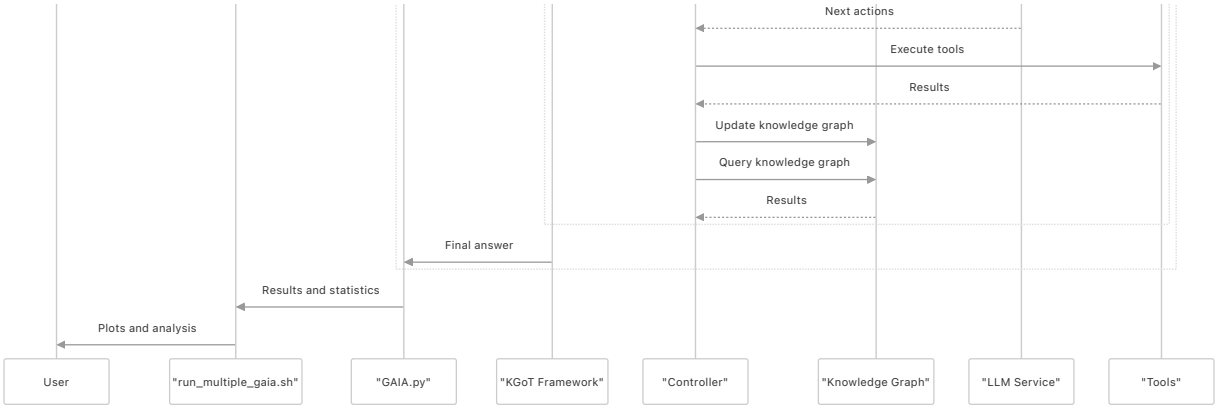
Infrastructure and Deployment

The system includes components for deployment and orchestration:

4. **Python Executor**: Service for safely executing Python code during reasoning

Sources: `docker_instances/kgot_docker/Dockerfile` 1–30 `run_multiple_gaia.sh` 1–252

Execution Model



This sequence diagram shows the execution flow when running the system, highlighting the different paths for zero-shot mode versus KGoT mode. The orchestrator script manages the overall process and collects results for analysis.

Sources: `GAIA/gaia.py` 25–104 `GAIA/gaia.py` 163–177 `GAIA/gaia.py` 180–211 `run_multiple_gaia.sh` 185–243

System Configuration and Options

General	`--log_folder_base`	Directory for storing logs	`logs/[db]_[controller]_[tool]`
	`--attachment_folder`	Path to problem attachments	`GAIA/dataset/attachments/validation`
	`--max_iterations`	Maximum reasoning iterations	7
Database	`--db_choice`	Graph database backend	`neo4j`
	`--neo4j_uri`	Neo4j database URI	`bolt://localhost:7687`
Controller	`--controller_choice`	Controller strategy	`queryRetrieve`
	`--tool_choice`	Tool set to use	`tools_v2_3`
LLM	`--llm_planning_model`	Model for planning	`gpt-4o-mini`
	`--llm_execution_model`	Model for tool execution	`gpt-4o-mini`
	`--zero_shot`	Enable zero-shot mode	`false`

These options can be specified when running the system via the `run_multiple_gaia.sh` script or directly through the Python interfaces.

Sources: `run_multiple_gaia.sh` | `15-145` `GAIA/gaia.py` | `214-276` `kgot/__main__.py` | `123-188`

Tools and Extension Points

The KGoT system provides several extension points to customize its behavior:

1. **Database Backends**: Support for different graph databases (Neo4j, NetworkX)
2. **Controller Strategies**: Different reasoning approaches (queryRetrieve, directRetrieve)
3. **Tools**: Extensible set of tools for data manipulation and analysis

These components can be selected through configuration parameters when running the system.

Sources: `kgot/__main__.py` 80–105 `GAIA/gaia.py` 180–211 `run_multiple_gaia.sh` 41–74

Usage Examples

Running a Single Problem with KGoT

To solve a single problem using the KGoT framework directly:

```
kgot single --problem "Calculate the factorial of 5" --iterations 10
```

Running GAIA with Multiple Configurations

To evaluate a dataset of problems using the GAIA system:

```
./run_multiple_gaia.sh --controller_choice queryRetrieve --max_iterations 7
```

For zero-shot evaluation:

Summary

The Knowledge Graph of Thoughts (KGoT) system combines LLM reasoning with knowledge graphs to solve complex problems. The system consists of the core KGoT framework, the GAIA high-level interface, and infrastructure components for deployment and execution. The modular design allows for different database backends, reasoning strategies, and tool integrations.

By maintaining a structured knowledge graph during reasoning, KGoT improves the ability of LLMs to handle complex problems that require multi-step reasoning, memory management, and integration with external tools and data sources.

Sources: `GAIA/gaia.py` | `1-276` `kgot/__main__.py` | `1-217` `run_multiple_gaia.sh` | `1-252`