Project ATLAS — Judyta AI Agent

Document version: 1.0

Document status: Internal

System: Judyta AI Agent

Document type: Technical Documentation

Classification: Internal Company Use

Owner: [To be completed]

Last updated: [To be completed]

---

1. Purpose

This document describes the architecture, functionality, configuration, security model, operation and maintenance of the Judyta AI Agent.

It is intended for internal technical staff responsible for development, administration, troubleshooting and further evolution of the system.

This document describes the implemented system and should be treated as the technical reference for the current project version.

1.1 Security notice

This document intentionally excludes:

API keys

passwords

access tokens

private cryptographic keys

database credentials

user credentials

other authentication secrets

Sensitive configuration must be provided through the approved secret-management or environment-variable mechanism.

---

2. System Overview

Judyta is a modular AI agent designed to interpret user requests, create execution plans, use registered tools, process their results and generate a final response.

The system separates reasoning-related responsibilities into distinct stages:

1. Request handling

2. Planning

3. Plan validation

4. Tool execution

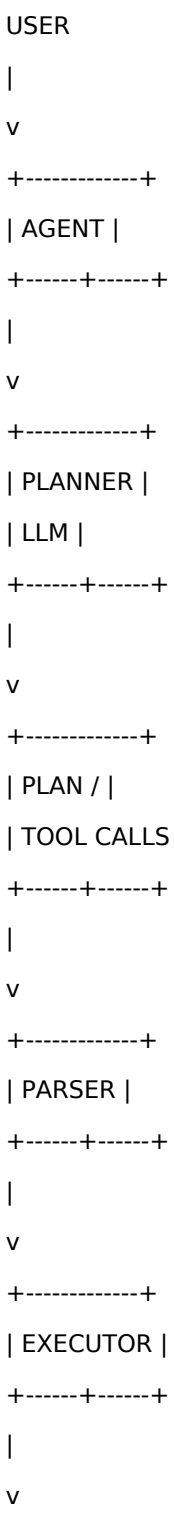
5. Result processing

6. Response generation

The architecture allows Judyta to perform both simple single-tool operations and multi-step tasks where the output of one tool becomes an input to another tool.

---

3. High-Level Architecture



+-------------+

| REGISTRY |

+------+------+

|

+----------------+----------------+

| | |

v v v

SYSTEM FILES WEB

TOOLS TOOLS TOOLS

| | |

+----------------+----------------+

|

v

TOOL RESULTS

|

v

+-------------+

| RESPONDER |

| LLM |

+------+------+

|

v

RESPONSE

The central architectural principle is separation of responsibilities:

the Planner decides **what should be done**;

the Executor performs the planned operations;

tools perform concrete actions;

the Responder decides **how the result should be presented to the user**.

---

4. Project Structure

The principal project structure is:

agent_v2/

  ├── main.py

  ├── app.py

  ├── config.py

  ├── llm.py

  ├── requirements.txt

  |

  ├── core/

```
│   ├── agent.py
│   ├── conversation.py
│   ├── executor.py
│   ├── memory.py
│   ├── plan.py
│   ├── planner.py
│   ├── registry.py
│   ├── responder.py
│   ├── sandbox.py
│   └── tool.py
│
├── planner/
│   ├── base.py
│   ├── llm.py
│   ├── parser.py
│   ├── prompt.py
│   └── rule.py
│
├── memory/
│   └── long_term.py
│
├── tools/
│   ├── cpu.py
│   ├── disk.py
│   ├── fetch_url.py
│   ├── list_directory.py
│   ├── memory.py
│   ├── read_file.py
│   ├── recall.py
│   ├── remember.py
│   ├── system.py
│   ├── terminal.py
│   ├── web_search.py
│   └── write_file.py
│
├── tests/
└── workspace/
```

---

5. Core Components

5.1 Agent

Implementation: `core/agent.py`

The Agent is the main orchestration component.

It connects:

Conversation

LongTermMemory

Planner

Executor

Registry

Responder

The Agent receives the user's request, requests a plan from the Planner, executes the plan and passes the resulting information to the Responder.

The Agent also resolves references to results from previous tool calls.

5.2 Planner

Implementation: `planner/llm.py`

The Planner converts a natural-language request into a structured execution plan.

The Planner does not directly execute operating-system commands or tools.

The expected output is JSON:

{

"calls": [

{

"tool": "tool_name",

"argument": "argument"

}

]

}

A plan may contain multiple calls.

Planner responsibilities

The Planner is responsible for:

interpreting the user's request;

selecting appropriate registered tools;

determining execution order;

creating tool arguments;

deciding when external information is required;

using memory tools when appropriate;

avoiding unnecessary tool calls;

respecting system and security constraints.

5.3 Plan Parser

Implementation: `planner/parser.py`

The parser validates the Planner's JSON response before execution.

It verifies that:

the response is valid JSON;

the root object has the expected structure;

`calls` is an array;

each call is an object;

a tool name is present;

the tool name is a string.

Invalid plans are rejected rather than executed.

5.4 Executor

Implementation: `core/executor.py`

The Executor receives validated tool calls and invokes the corresponding registered tool.

Its responsibilities include:

1. validating the tool name;

2. retrieving the tool from the Registry;

3. executing the tool;

4. returning the tool result.

The Executor does not select tools itself. Tool selection is performed by the Planner.

5.5 Registry

Implementation: `core/registry.py`

The Registry is the central catalogue of available tools.

It provides:

tool registration;

duplicate-name protection;

tool lookup;

tool existence checks;

tool listing;

generation of tool descriptions for the Planner.

The final Agent registers 12 tools.

5.6 Responder

Implementation: `core/responder.py`

The Responder creates the final user-facing response.

It receives:

the original request;

conversation history;

long-term memory;

tool results.

The Responder is instructed to treat tool results as evidence and not claim that an operation succeeded unless the available results support that conclusion.

It is also instructed not to expose internal planning or sensitive information.

---

6. Request Execution Lifecycle

A typical request follows this sequence:

Step 1 — User request

The request enters the Agent.

Step 2 — Conversation update

The user message is added to the current conversation history.

Step 3 — Context construction

The Planner receives relevant:

conversation history;

long-term memory;

available tool descriptions;

current user request.

Step 4 — LLM planning

The Planner LLM produces a JSON execution plan.

Step 5 — Plan validation

The JSON plan is parsed and validated.

Step 6 — Tool execution

The Executor executes the calls in order.

Step 7 — Result chaining

If the plan contains result references, results from previous operations are substituted into later tool arguments.

Step 8 — Response generation

The Responder receives the results and generates the final response.

Step 9 — Conversation update

The assistant response is added to the current conversation history.

---

7. Multi-Step Execution

Judyta supports multi-step execution.

For example:

web_search

|

v

search result

|

v

fetch_url($previous)

|

v

page content

This allows the system to compose several tools into a single workflow.

7.1 Result references

Supported references include:

- $previous
- $result
- $previous.1
- $result.1
- $previous.2
- $result.2

Brace-style equivalents are also supported:

- {{previous}}
- {{result}}
- {{previous.1}}
- {{result.1}}

Unnumbered references point to the most recent result.

Numbered references use one-based indexing.

References can be resolved recursively inside:

strings;

lists;

tuples;

dictionaries.

---

8. Conversation Context

Implementation: `core/conversation.py`

Conversation history is maintained in memory during the process lifetime.

The conversation stores user and assistant messages.

Supported operations include:

adding a user message;

adding an assistant message;

retrieving history;

clearing history.

The current implementation does not provide persistent conversation storage.

A process restart therefore clears the in-memory conversation history.

---

9. Long-Term Memory

Implementation: `memory/long_term.py`

Long-term memory is stored in:

workspace/memory.json

The current format is based on a `facts` collection.

Example:

{

"facts": [

"..."

]

}

The memory implementation supports:

adding facts;

retrieving facts;

retrieving values by key;

clearing memory;

duplicate prevention;

compatibility with an older memory representation.

The implementation currently uses a JSON file rather than a database or vector store.

---

10. Context Limits

The PromptBuilder applies explicit limits when constructing LLM context.

Planner context

The current implementation limits:

conversation history to the last 6 messages;

history to approximately 5000 characters;

long-term memory to up to 30 facts;

memory context to approximately 3000 characters.

Responder context

The current implementation uses:

conversation history;

long-term memory;

tool results;

the current request.

Individual tool results are limited before being incorporated into the response context.

These limits are intended to control prompt size and reduce unnecessary context.

---

11. Tool System

The final Agent registers the following tools:

| Tool | Purpose |

|---|---|

| `uptime` | System uptime information |

| `memory` | RAM usage information |

| `disk` | Disk usage information |

| `cpu` | System load information |

| `terminal` | Shell command execution |

| `list_directory` | Directory listing |

| `read_file` | Read a file within the sandbox |

| `write_file` | Write a file within the sandbox |

| `remember` | Store information in long-term memory |

| `recall` | Retrieve long-term memory |

| `web_search` | Search the Internet |

| `fetch_url` | Retrieve a specific HTTP/HTTPS URL |

---

12. System Tools

12.1 Uptime

The `uptime` tool provides system uptime information using the operating system's `uptime` command.

12.2 Memory

The `memory` tool obtains RAM information using:

free -h

12.3 Disk

The `disk` tool obtains filesystem usage using:

df -h

12.4 CPU

The current `cpu` implementation obtains system load information using `uptime`.

It is therefore not a dedicated CPU telemetry implementation.

This distinction should be preserved in technical documentation and monitoring requirements.

---

13. Terminal Tool

The `terminal` tool executes shell commands through the system process interface.

The current implementation uses:

subprocess.run(..., shell=True, timeout=120)

The command timeout is 120 seconds.

The tool returns command output and the exit status.

13.1 STANDARD mode

The application-level STANDARD mode blocks a set of high-impact commands, including:

sudo

su

apt

apt-get

systemctl

service

iptables

ufw

mount

umount

useradd

userdel

passwd

13.2 ADMIN mode

ADMIN mode permits commands that are blocked by the STANDARD application filter.

ADMIN mode must not be interpreted as a separate operating-system security boundary.

Actual privileges remain determined by the operating-system account running Judyta.

---

14. File Tools

14.1 list_directory

Lists directory contents.

The current implementation is not restricted by the same `safe_path()` mechanism used by `read_file` and `write_file`.

Access is therefore ultimately determined by the operating-system permissions of the Judyta process.

14.2 read_file

Reads text files through the application sandbox.

14.3 write_file

Writes text files through the application sandbox.

---

15. File Sandbox

Implementation: `core/sandbox.py`

The application defines:

workspace/

as the controlled file area.

The `safe_path()` mechanism:

1. resolves the requested path;

2. handles relative paths relative to the workspace;

3. resolves `..` path traversal;

4. resolves symbolic links;

5. verifies that the final path remains inside the workspace.

Attempts to escape the workspace are rejected.

The sandbox is currently applied to `read_file` and `write_file`.

It does not automatically constrain arbitrary shell commands executed through `terminal`.

---

16. Memory Tools

16.1 remember

The `remember` tool stores information in long-term memory.

The expected input format is:

key: value

The tool supports adding new information and updating existing information.

Duplicate facts are not unnecessarily added.

16.2 recall

The `recall` tool retrieves long-term memory.

Without an argument it can return the available stored facts.

With an argument it can retrieve information associated with a key.

---

17. Web Tools

17.1 web_search

The `web_search` tool provides Internet search functionality.

The current implementation uses the `ddgs` library.

The implementation limits:

the number of returned results;

the size of individual result content;

the total returned content.

These limits reduce the amount of external content inserted into the agent context.

17.2 fetch_url

The `fetch_url` tool retrieves a specified HTTP/HTTPS resource.

The implementation handles common compressed HTTP responses and attempts to determine the correct text encoding.

For HTML content it removes non-content elements such as:

`script`;

`style`;

`noscript`.

The extracted text is size-limited before being returned to the agent.

---

18. LLM Integration

Implementation: `llm.py`

The current architecture uses separate LLM roles for:

Planner;

Responder.

The project integrates with Google's Gemini models through the LangChain Google Generative AI integration.

The model is selected through configuration.

Credentials are supplied through environment-based configuration and must not be embedded in source code or documentation.

---

19. Configuration

Configuration is loaded using environment variables.

Sensitive configuration includes credentials such as API keys.

The documentation should describe the variable names and purpose but must not contain their values.

Example:

```
GEMINI_API_KEY=<secret>
MODEL=<model-name>
```

The actual values must be supplied through the approved deployment configuration.

The current project also contains configuration references related to web-search credentials; these should be reviewed against the active implementation before deployment.

---

20. Security Model

Judyta uses several application-level controls.

20.1 Prompt-level controls

The Planner and Responder prompts define operational rules, including:

avoiding disclosure of secrets;

avoiding unnecessary system changes;

using tools according to their intended purpose;

respecting workspace restrictions;

distinguishing evidence from assumptions;

avoiding claims unsupported by tool results.

Prompt rules are behavioral controls and must not be considered a complete security boundary.

20.2 File sandbox

File reads and writes are restricted to the workspace through `safe_path()`.

20.3 Terminal restrictions

STANDARD mode applies an application-level blocklist for selected administrative commands.

20.4 Operating-system security

The effective security boundary remains the operating system.

The recommended deployment model is therefore to run Judyta under a dedicated account with the minimum permissions required for its intended functions.

---

21. Secrets Management

Secrets must never be stored in:

source code;

documentation;

README files;

test fixtures;

example configuration files;

Git history.

The project should use environment variables or an approved secret-management system.

Examples in documentation must use placeholders:

<API_KEY>

<PASSWORD>

<TOKEN>

<PRIVATE_KEY>

If a secret is accidentally committed, removing it from the latest file is not sufficient. The credential must be considered compromised and rotated, and the repository history must be reviewed.

---

22. Error Handling

The system validates plans before execution.

Execution errors are propagated to the orchestration layer and can be presented to the Responder.

The Responder is instructed to distinguish between:

successful operations;

failed operations;

incomplete information;

unsupported operations.

API rate-limit conditions such as HTTP 429 / resource exhaustion are handled explicitly.

---

23. CLI

The main command-line interface is provided by `main.py`.

Standard mode:

```
python main.py
```

Administrative mode:

```
python main.py --admin
```

The interactive session can be terminated using:

exit

or:

quit

---

24. Testing

The project contains a substantial test suite covering the main architectural components.

Test coverage includes:

Agent;

Conversation;

Planner;

plan parser;

Registry;

Executor;

long-term memory;

file operations;

sandbox;

result chaining;

structured arguments;

tool failures;

unknown tools;

Responder;

web search;

Gemini integration;

API rate-limit handling.

Tests requiring the Gemini service are separated using a `gemini` marker.

Testing policy

Before a production release:

1. install the exact project dependencies;

2. run the complete local test suite;

3. run the external-API test subset where credentials and network access are available;

4. verify security-sensitive tools separately;

5. verify the final deployment configuration.

---

25. Dependencies

The project defines Python dependencies in:

requirements.txt

The dependency set includes the LLM integration, configuration handling, HTTP/web functionality and testing framework.

The dependency list must be kept synchronized with actual imports.

In particular, the active web-search implementation imports:

ddgs

and the deployment dependency set should explicitly provide the package required by that import.

---

26. Operational Model

A production deployment should separate:

application code;

configuration;

secrets;

runtime workspace;

logs;

backups.

The workspace should be treated as application data rather than source code.

Operational changes should be documented and versioned.

---

27. Monitoring and Diagnostics

At minimum, operational monitoring should cover:

application availability;

LLM API availability;

API rate limits;

tool execution failures;

unexpected process termination;

disk capacity;

memory usage;

network connectivity for web tools;

integrity of the workspace;

permissions of the runtime account.

When diagnosing a problem, the preferred sequence is:

1. reproduce the issue;

2. inspect application output/logs;

3. identify the failing architectural layer;

4. test the affected component independently;

5. verify configuration;

6. make the smallest required change;

7. repeat the original test;

8. document the root cause and resolution.

---

28. Troubleshooting Principles

Planner problems

Check:

LLM availability;

model configuration;

prompt construction;

returned JSON;

parser errors.

Tool problems

Check:

Registry registration;

tool name;

arguments;

operating-system permissions;

tool-specific errors.

File problems

Check:

workspace path;

`safe_path()` resolution;

permissions;

symlinks;

filesystem availability.

Web problems

Check:

network connectivity;

external service availability;

`ddgs` dependency;

URL accessibility;

content encoding.

Responder problems

Check:

tool results;

context size;

LLM availability;

API rate limits;

responder prompt.

---

29. Known Technical Limitations

The current implementation has the following known limitations:

1. `terminal` uses `shell=True`.

2. STANDARD terminal protection is an application-level blocklist and is not a complete shell security mechanism.

3. `list_directory` is not currently protected by `safe_path()`.

4. The `cpu` tool reports load information through `uptime` rather than dedicated CPU telemetry.

5. Long-term memory is stored in a JSON file rather than a transactional database.

6. Conversation history is process-local and is lost after restart.

7. Planner execution depends on valid structured output from the LLM.

8. Web functionality depends on external services/libraries.

9. Dependency declarations must remain synchronized with actual imports.

10. Full end-to-end validation requires the correct runtime environment and access to external LLM services.

---

30. Development Guidelines

Changes to Judyta should follow these principles:

keep Planner, Executor and Responder responsibilities separated;

add new capabilities as tools where possible;

register tools centrally through the Registry;

validate external/LLM-generated structures before execution;

keep secrets outside source control;

add tests for new tools and execution paths;

document security implications of new capabilities;

avoid unrelated infrastructure changes during troubleshooting;

update this documentation when architecture or behavior changes.

---

31. Feature Status Convention

Documentation and project planning should distinguish:

IMPLEMENTED

The feature exists in the current code and belongs to the active execution path.

PARTIALLY IMPLEMENTED

The feature exists only in part or requires additional work before it can be considered complete.

PLANNED

The feature is a future requirement or roadmap item and is not part of the current implementation.

This convention prevents planned functionality from being mistaken for functionality that already exists.

---

32. Release and Change Management

Each production release should have:

a unique version identifier;

a known Git commit or tag;

validated dependencies;

a tested configuration;

documented changes;

rollback information where applicable.

Architectural changes should be documented before or together with implementation.

Prompt changes should be treated as controlled changes because they can materially affect Planner and Responder behavior without changing application code.

---

33. Security and Repository Handling

The source repository must be treated as a controlled technical asset.

Before sharing the project outside the approved internal environment:

1. remove secret-bearing files;

2. review configuration examples;

3. scan the working tree for credentials;

4. review Git history for previously committed secrets;

5. rotate credentials if exposure is suspected;

6. verify `.gitignore` rules;

7. ensure private keys and authentication material are excluded.

---

34. Future Evolution

Potential future improvements should be tracked separately from the current implementation.

Possible areas include:

stronger process isolation for terminal execution;

extending sandbox coverage to directory listing;

dedicated CPU metrics;

persistent conversation storage;

more robust memory storage;

stronger structured-output validation;

improved tool authorization;

richer observability;

dependency and release automation;

additional integration tests.

These items must not be interpreted as currently implemented functionality unless they are present in the official release.

---

35. Appendix — Key Components

| Component | Location | Responsibility |

|---|---|---|

| Agent | `core/agent.py` | Main orchestration |

| Conversation | `core/conversation.py` | Current conversation history |

| Executor | `core/executor.py` | Tool execution |

| Registry | `core/registry.py` | Tool catalogue |

| Responder | `core/responder.py` | Final response generation |

| Sandbox | `core/sandbox.py` | Workspace path validation |

| LLM Planner | `planner/llm.py` | LLM-based planning |

| Plan Parser | `planner/parser.py` | Plan validation |

| Prompt Builder | `planner/prompt.py` | Planner context/prompt |

| Long-Term Memory | `memory/long_term.py` | Persistent facts |

| LLM Configuration | `llm.py` | LLM clients |

| Application Configuration | `config.py` | Environment configuration |

| CLI | `main.py` | Interactive application entry point |

---

36. Document Maintenance

This document is part of the internal technical documentation for Judyta.

It should be reviewed whenever there is a material change to:

system architecture;

LLM provider/model;

Planner behavior;

Responder behavior;

memory implementation;

tool set;

security model;

deployment model;

configuration;

authentication;

external integrations.

The documentation should describe the released system, not an aspirational roadmap.