Agent Forge – Comprehensive Project Documentation

Version 1.0  
Generated: 27 April 2025, 17:21 UTC

Table of Contents – update fields in Word to refresh.

# 1 Introduction

Agent Forge is a modular, extensible workbench for designing, building, testing and evolving AI agents.

This document provides an end-to-end view of the system—goals, architecture, implementation details, operating procedures, and future roadmap.

# 2 Vision & Goals

The vision is to democratize advanced agent development by supplying a safe, declarative and observable environment that supports rapid experimentation and continual improvement.

Key goals:

• Modularity – plug-and-play capabilities, skills, strategies.

• Declarative first – YAML definitions validated by Pydantic schemas.

• Safety – ethical guard-rails, sandboxing and trace auditing.

• Observability – structured JSON logging and rich UI visualisations.

• Evolution – automated evaluation and genetic / reinforcement based optimisation.

# 3 High-Level Architecture

The project is organised as a Python package (agent\_forge\_project/). Core layers:

1. Definitions – YAML files describing agents, skills, tools, ethics, test-cases.

2. Forge Core – Loader, Builder, Behaviour Tree engine, evaluation harness, logging, guardrails.

3. Capabilities – concrete Skills & Tools.

4. Agents – concrete Agent implementations (simple, coordinator, BT-driven, etc.).

5. Knowledge – RAG components (document processing, vector store).

6. UI & CLI – Streamlit workbench and run\_forge.py runner.

# 4 Directory Layout

```

agent\_forge\_project/

├── definitions/ # YAML declarative configs

├── strategies/ # Behaviour-tree YAML files

├── agents/ # Agent classes

├── capabilities/ # Skills & Tools

├── forge\_core/ # Core engine / registry / evaluation / logging

├── knowledge/ # RAG helpers

├── logs/ # Execution & evaluation traces

├── forge\_ui.py # Streamlit front-end

└── run\_forge.py # CLI entry-point

```

# 5 Declarative Components

Every YAML definition is validated by a matching Pydantic model in forge\_core/schemas.py. Component types:

• ToolDefinition – atomic I/O wrappers (e.g., WebSearch).

• SkillDefinition – higher-level logic that can orchestrate tools.

• AgentDefinition – metadata + allowed skills/tools + optional BT strategy.

• EthicalFrameworkDefinition – list of principles and keyword checks.

• TestCaseDefinition – evaluation specs with checkpoints and metrics.

# 6 Core Modules

• component\_registry.py – loads YAML, constructs strongly-typed registry.

• agent\_builder.py – resolves dependencies and instantiates agents.

• behavior\_tree.py – lightweight BT executor with YAML loader.

• evaluation.py – harness that batch-runs agents against test cases.

• forge\_logging.py – structured JSON logging helpers.

• safety\_guardrails.py – keyword-based content filter.

• llm\_judge.py – optional LLM-as-judge scoring.

# 7 Agent Lifecycle

1. Builder loads AgentDefinition.

2. Skills & Tools are built and injected.

3. Agent receives run(prompt) call.

4. Agent executes Behaviour Tree, skills, and/or tools.

5. All steps are logged with unique run-id / step-id hierarchy.

6. Output returned to caller & optionally judged/evaluated.

# 8 Skills & Tools

Skills wrap business logic; Tools wrap external APIs. Both inherit BaseCapability and implement execute(...). They receive the current run-id so they can emit nested logs for full traceability.

# 9 Behaviour Tree Strategies

Strategies are defined as YAML documents mapping to Sequence, Selector, and Action nodes. Action nodes delegate to agent.run\_action() allowing arbitrary code while preserving declarative control flow.

## 10 Retrieval-Augmented Generation (RAG)

The knowledge/ package supplies:

• document\_processor.py – loads PDF / web pages and chunks text.

• vector\_store.py – ChromaDB wrapper for embeddings.

The rag\_search\_strategy.yaml BT demonstrates how an agent retrieves context before answering.

## 11 Evaluation Harness

• Loads all test\_cases/\*.yaml.

• For each case, constructs the target agent, runs with input\_prompt, calculates pass/fail via keyword matching or custom checkpoints.

• Logs summary to logs/evaluation\_results.jsonl.

This mechanism underpins automated regression testing and future evolutionary optimisation.

## 12 Logging & Observability

All Forge activities emit JSON lines in two files: agent\_execution.jsonl and evaluation\_results.jsonl. Fields: timestamp, run\_id, step\_id, component\_type, event\_type, data. These traces drive debugging, visual replay in UI, and guard-rail enforcement.

## 13 Safety & Ethics

Ethical frameworks are declared in YAML and attached to agents. safety\_guardrails.check\_text() is invoked on generated content. The design allows pluggable policies (LLM moderation, regex, OpenAI filters, etc.).

## 14 UI Workbench

forge\_ui.py (Streamlit) exposes: prompt box, agent selector, run button, real-time display of agent output, expandable trace viewer (future feature).

## 15 Dev-Setup & Deployment

• Python 3.10+ recommended.

• pip install -r requirements.txt.

• Optional: Ollama running locally for Qwen2 model.

• Docker-compose file can be added later for full stack (Ollama + ChromaDB).

## 16 Contribution Guide

1. Fork & create feature branch.

2. Run pytest and python forge\_core/evaluation.py to ensure no regressions.

3. Adhere to PEP-8 and add docstrings.

4. Update documentation and unit tests.

5. Submit PR with concise description.

## 17 Roadmap & Phases

Phase 0 – Baseline (DONE): Manual workbench, BT engine, RAG, evaluation.

Phase 1 – Sandboxing & Plugin API: syscall isolation, tool marketplace.

Phase 2 – Multi-Agent & Coordination protocols.

Phase 3 – Automated Evolution Controller (genetic search & RL-HF).

Phase 4 – Cloud UI, sharing hub, community contributions.

## 18 Glossary

• BT – Behaviour Tree.

• RAG – Retrieval-Augmented Generation.

• LLM – Large Language Model.

• Skill – Re-usable capability triggered by agent.

• Tool – Thin wrapper around an external system/API.