Project title	**ACHILLES Human-Centred Machine learning lighter, clearer, safer**		
Project acronym	ACHILLES		
GA number	101189689		
Project start date	01/11/2024	Duration	48 months

D3.3 - MULTI-AGENT ENVIRONMENT REPORT

Due date	30/04/2026	Delivery date	30/04/2026
Work package	WP3		
Responsible Author(s)	Sînică Alboaie (AXL)		
Contributor(s)	Sînică Alboaie (AXL), Daniel Sava (AXL), Mircea Ionut Alupoaie (AXL), Alexandrina Rata (AXL), Nicoleta Mihalache (AXL), Marco Cuomo (CuomoIT)		
Reviewer(s)	Rui Castro (FhAICOS), Joao Fonseca (INESC-ID), André Carreiro (FhAICOS)		
Version	V1.0		
Dissemination level	Public		

Funded by the European Union

This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

VERSION AND AMENDMENTS HISTORY

Version	Date (DD/MM/YYYY)	Created /Amended by	Changes
V0.1	20/03/2026	Sînică Alboaie	Integrated content prepared during M1-M16
V0.2	02/04/2026	Nicoleta Mihalache Sînică Alboaie	Internal WP1 review and fixes
V0.3	10/04/2026	Rui Castro Joao Fonseca	Consortium-level Review
V0.4	23/04/2026	Sînică Alboaie	Improved version
V0.5	28/04/2026	ALL	Final Review and Fixes
V1.0	30/04/2026	André Carreiro	Final revision and clean up

Funded by the European Union

This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

TABLE OF CONTENTS

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

LIST OF FIGURES

LIST OF TABLES

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
API	Application Programming Interface
CLI	Command Line Interface
CNL	Controlled Natural Language
CoRE	Code Representation and Execution
CPU	Central Processing Unit
CRUD	Create, Read, Update, Delete (basic database operations)
DB	Database; used in the context of DBTable skills
DEV	Development environment
DCG	Dynamic Code Generation
DPU	Data Processing Unit
DSPy	Declarative Self-improving Python framework for LLM pipelines and optimisation
DSU	Domain Specific Unit
Dx.y	Deliverable x.y
HERA	Holistic Evaluation and Regulatory Adherence
HTTP	HyperText Transfer Protocol
IDE	Integrated Development Environment
IR	Intermediate Representation
JSON	JavaScript Object Notation
LM	Language Model
LLM	Large Language Model

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

LLM Proxy (Soul Gateway)	Centralized gateway for managing and governing access to LLM providers
LMQL	Language Model Query Language
MCP	Model Context Protocol
ML	Machine Learning
MRP-VM	Meta-Rational Pragmatics Virtual Machine
NSAI	Neuro-Symbolic Artificial Intelligence
PAL	Program-Aided Language
PostgreSQL	Relational database management system used for persistence
PROD	Production environment
R&D	Research and Development
RAG	Retrieval-Augmented Generation
SCRIPTA	Structured Creative Writing Intelligent Platform for Textual Authoring
SDK	Software Development Kit
SMT	Satisfiability Modulo Theories
SOP	Standard Operating Procedure
T3.3	Task 3.3 (project work package WP3 task)
VM	Virtual Machine
VSA	Vector Symbolic Architectures
WPx	Work Package x

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

1 EXECUTIVE SUMMARY

This deliverable reports the preliminary multi-agent execution environment developed in Task T3.3 of ACHILLES at M18. It describes the architectural foundations, implementation status, initial validation practice, and research trajectory of the environment, in line with the project's objective of supporting AI systems that are lighter, clearer, and safer. The document should be read as a technical baseline: it clarifies what is already operational, what remains under active refinement, and which research directions will be further developed in the next project phases.

The central premise of T3.3 is that practical multi-agent capability cannot rely only on prompt engineering or loosely coupled conversational loops. It requires a structured execution environment in which deployment, model access, orchestration, reusable operational units, observability, and intermediate representations are separated into well-defined layers. This separation is important for both efficiency and trustworthiness. Systems that repeatedly rely on large-context inference, implicit state, or uncontrolled recomputation are not only costly, but also harder to inspect, stabilise, audit, and reuse. T3.3 therefore prioritises decomposition into bounded skills, explicit execution sessions, selective recomputation, persistent intermediate artefacts, and controlled access to external capabilities.

At M18, the operational baseline of the ACHILLES multi-agent environment consists of three main components. First, Ploinky provides the deployment substrate for packaging, isolating, and exposing agents and services in a standardised way. It supports the notion of a Ploinky Agent as a deployable unit and provides the foundation for integrating agentic backends into the ACHILLES IDE. Second, the LLM Proxy, also referred to as Soul Gateway, provides a controlled access layer for large language models, adding observability, authentication, configuration management, cost monitoring, and audit logging to raw model-provider access. Third, AchillesAgentLib provides the main agent development framework, including model interaction, session management, and the execution of reusable operational units referred to as skills. Within this architecture, a skill is treated as more than a narrow callable tool. It is a governed execution unit that may encapsulate prompts, tools, procedural logic, model calls, coordination behaviour, and validation mechanisms behind a reusable interface. Skills are executed through agentic sessions, including loop-based sessions inspired by reasoning-and-action patterns. This gives the project a practical way to build specialised agents that are more structured and more inspectable than monolithic LLM interactions, while still preserving the flexibility needed for natural-language-driven workflows.

The work in T3.3 directly supports the ACHILLES IDE and the project use cases. For the IDE, it provides the multi-agent substrate, deployable backend services, skill-oriented execution model, and model-access governance layer needed to move from isolated tools toward an integrated agentic environment. For use cases such as SCRIPTA and HERA, it supports long-running, structured workflows in which tasks can be decomposed into planning, generation, retrieval, validation, compliance

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

checking, reporting, and selective updating of intermediate artefacts. This is particularly relevant where traceability, cost control, repeatability, and human oversight are essential.

The deliverable also presents the language and representation layer centred on SOP Lang. SOP Lang is being developed to make agentic workflows more explicit, inspectable, and dependency-aware. Its role is to support procedural orchestration, intermediate representations, and structured planning in a way that allows parts of a workflow to be reused, validated, or selectively recomputed. At M18, this layer is partially implemented and remains under active refinement. It is therefore presented as an important bridge between the current skill-based execution model and more structured future forms of agent orchestration.

Beyond the current operational stack, the deliverable introduces Meta-Rational Pragmatics as an emerging research direction. MRP Virtual Machine (MRP-VM) explores how natural-language artefacts, structured representations, and interpreter-like execution mechanisms could be combined to reduce reliance on unconstrained LLM-mediated planning and execution. At this stage, it should be understood as exploratory and prototypical rather than as a stabilised production component. Its purpose in this deliverable is to clarify a possible next architectural step: progressively moving recurring control flow, validation, constraint handling, and structured reasoning toward more explicit and computationally predictable substrates, while continuing to use LLMs where they provide the greatest value for interpretation and interaction.

The literature and technology survey included in the deliverable positions this work in relation to current developments in agent frameworks, graph-based execution, reusable skills, and neuro-symbolic AI. The survey is selective rather than exhaustive, identifying the architectural patterns most relevant for ACHILLES: explicit state, modular composition, bounded delegation, observability, reusable capabilities, human oversight, and structured execution. These patterns confirm the relevance of the project's staged approach while also showing that long-term architectural stability cannot depend on any single external framework or product trajectory.

Initial validation at M18 is component-oriented. The project has assessed the operational readiness of core components through internal functional testing, controlled execution scenarios, development use, and preliminary evaluation suites. These activities provide evidence that the current baseline can support agent deployment, model access governance, skill execution, and integration with IDE-oriented workflows. However, full use-case-level benchmarking and systematic performance assessment will require broader integration with WP6 and WP7 activities. This deliverable therefore reports a preliminary evaluation baseline and identifies the need for expanded benchmarking in the next reporting phase.

Overall, D3.3 establishes the first public technical baseline for the ACHILLES multi-agent execution environment. It shows that the project has moved beyond conceptual design toward a deployable and reusable infrastructure for agentic execution. At the same time, it carefully distinguishes operational components from ongoing research. The immediate contribution is a

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

modular stack composed of Ploinky, the LLM Proxy, AchillesAgentLib, skills, and agentic sessions. The medium-term contribution is the progressive refinement of SOP Lang and evaluation practice. The longer-term research direction is to explore MRP-VM and related neuro-symbolic mechanisms. Together, these elements provide ACHILLES with a credible path toward multi-agent systems that are more efficient, inspectable, governable, and suitable for trustworthy AI workflows.

2 INTRODUCTION

2.1 General Approach

In line with the Grant Agreement, D3.3 provides the first public report on the multi-agent execution environment developed under T3.3 at M18. It establishes the preliminary architectural and implementation baseline, reports the current state of component-level validation and evaluation practice, and identifies the work that will be expanded in D3.4, the updated M38 report. Its purpose is to describe not only a set of technical components, but also the broader architectural direction through which ACHILLES aims to support efficient, trustworthy, and reusable agentic workflows across the project use cases and within the ACHILLES IDE. At a practical level, the work focuses on the creation of a deployable multi-agent substrate in which capabilities can be exposed as reusable skills, orchestrated in a controlled manner, and integrated into higher-level tools such as the ACHILLES copilot environment. This includes the deployment environment referred to as Ploinky, the integration logic required for agentic backends, and the progressive organisation of functionality into explicit services and bounded computational units rather than monolithic conversational loops.

A central concern throughout the task is the intersection between efficiency and trustworthiness. In the ACHILLES view, these two dimensions are strongly connected. Systems that over-rely on repeated large-model calls, carry excessive context, or recompute broad reasoning traces from scratch tend to become not only more expensive, but also harder to inspect, stabilise, and justify. For this reason, T3.3 complements the efficiency-oriented work of T3.1 by focusing on execution structure: decomposition into skills, explicit planning, selective recomputation, local context projection, and the preservation of intermediate artefacts that can be reused across steps.

Beyond the investigation of efficiency and the orchestration of agentic skills, a significant part of the research effort has also been directed toward a broader neuro-symbolic direction, which will remain an important focus in the continuation of the project. The goal is to reduce the dependence of agentic systems on LLMs as the universal mechanism for planning and execution, while preserving their major strength in handling the nuances of natural language. In this perspective, LLMs should remain essential for interpretation, interaction, and linguistic flexibility, but a larger part of planning, control flow, structured reasoning, and execution should gradually be transferred to more predictable and computationally efficient mechanisms.

This direction reflects a strategic attempt to build agentic systems that are not only more efficient, but also more stable and easier to deploy. It encourages stronger reuse of methods from

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

classical AI, symbolic AI, structured representations, constraint-based processing, and other forms of computation that can complement modern machine learning. It also supports the ambition of making substantial parts of the environment workable on CPU-oriented infrastructures, reducing exclusive dependence on costly accelerator-heavy execution wherever possible.

Overall, this deliverable should be read as an overview of an emerging architectural approach that combines multi-agent deployment, reusable skills, efficiency-aware execution, and neuro-symbolic research into a common framework. The aim is to create an agentic environment in which LLMs are used where they bring the greatest value, while planning and execution become progressively more structured, economical, and predictable.

Regarding terminology in this deliverable, an agent is an executable component capable of model-mediated or rule-based action. A tool is a narrow callable capability with a stable interface. A skill is a higher-level governed execution unit that may combine prompts, tools, model calls, procedural logic, validation steps, and context-handling conventions behind a reusable interface. An agentic session is the runtime context in which agents and skills are invoked, coordinated, and observed.

2.2 Impact on Use Cases and ACHILLES IDE

Task T3.3 has a direct impact on both the ACHILLES IDE and the project use cases by providing the deployment infrastructure required for a practical multi-agent environment. Its main contribution is the creation of Ploinky, the deployment environment through which reusable backend capabilities, specialised agents, and bounded skills can be exposed and orchestrated in a controlled manner. Connecting with WP6, T3.3 also supports the implementation of the ACHILLES copilot, referred to in implementation terms as ACHILLES CLI. D6.1 (WP6) specifies the ACHILLES IDE requirements and identifies Ploinky, SOP Lang, and agent integration standards as core elements of the IDE architecture. D3.3 complements that specification by reporting the T3.3 contribution: the multi-agent execution substrate, agent runtime abstractions, skill model, model-access gateway, and preliminary evaluation practice required to make those IDE concepts executable. These make the ACHILLES IDE operational as a structured multi-agent environment in which domain-specific capabilities can coexist with core operational functions such as filesystem interaction, secure identity and authentication handling, authorisation, and controlled access to capabilities. At the backend level, these components are deployed as Ploinky Agents, which expose their functionality through APIs aligned with the Model Context Protocol (MCP). This makes them usable not only by the ACHILLES copilot, but also, in principle, by any AI agent able to consume correctly described skills. In this sense, T3.3 provides both a deployment substrate for the IDE and a broader interoperability layer for reusable agentic services.

This architectural direction is especially relevant at the intersection between efficiency and trustworthiness, which is a central concern of ACHILLES. T3.1 addresses efficiency at model and inference level, while T3.3 addresses the execution environment in which such gains can be operationalised through explicit planning, bounded skills, local context projection, and selective

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

recomputation. Their combination is particularly important for the SCRIPTA (Structured Creative Writing Intelligent Platform for Textual Authoring) and HERA use cases (D7.1).

In SCRIPTA, the challenge is not only text generation, but support for a long, iterative, and controllable creative workflow. Writing tasks such as outline generation, scene drafting, revision, consistency correction, style normalisation, or editorial filtering should not all be treated as the same open-ended conversational process. Here T3.3 contributes by enabling a more structured decomposition of work, where generation is used where needed, while more procedural stages can be handled through explicit workflows and reusable intermediate artefacts. Together with the efficiency-oriented work of T3.1, this supports lower latency and lower cost, but also broader creative exploration, better revision discipline, and improved scalability for longer works, larger narrative contexts, and more demanding filtering requirements. **In HERA**, the relevance of this approach is even more explicit because the use case focuses on regulatory compliance and pharmaceutical knowledge management. The main need is not unrestricted generation, but stable and justifiable effort over large and evolving document collections. In this context, T3.3 supports the decomposition of workflows into bounded tasks such as retrieval, summarisation, extraction of obligations and exceptions, document comparison, compliance checking, and reporting. This is important because HERA must avoid repeated large-scale reinterpretation of the entire corpus whenever a regulation, procedure, or guideline changes. Explicit plans, intermediate artefacts, and selective recomputation make it possible to preserve prior work and update only the affected regions, improving both efficiency and traceability.

Task T3.3 functions as a supporting layer for the ACHILLES Copilot and, indirectly, for the use cases developed within the project. The WP7 deliverables constitute the main context in which the specific nuances and innovations developed by our team can be presented more explicitly, with **D7.1** representing a first step in this direction. At the same time, the capabilities provided by Ploinky and the approaches for building agents using the technologies developed in ACHILLES have now reached a greater level of maturity and can begin to be used more concretely.

Overall, the impact of T3.3 is to make the ACHILLES IDE and all its use cases more deployable, more modular, and more operationally credible. In SCRIPTA, this improves controllable creative exploration and revision. In HERA, it improves stable, scalable, and traceable compliance-oriented knowledge work. At the IDE level, D3.3 provides the multi-agent substrate, reusable service interfaces, and deployable skill-oriented backend structures needed to transform these use cases into practical tools rather than isolated demonstrations. Given their nature, these developments require qualitative evaluation based on architectural judgment and practical validation. This deliverable focuses on component-level assessment, while the full impact will be demonstrated through use-case synergy in future T3.3 updates.

Table 1 summarises the main contributions discussed in D3.3, showing the maturity level at M18, and referring the reader to the corresponding sections in the document.

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

Table 1. Summary of main T3.3 contributions and their status at M18

Layer	Component	M18 status	Evidence
Deployment	Ploinky	Operational baseline	Section 5.2
Model access	LLM Proxy / Soul Gateway	Operational baseline	Section 5.3
Agent runtime	AchillesAgentLib	Operational baseline	Section 5.4
Execution model	Agentic sessions / skills	Implemented and evolving	Sections 4.8–4.9; 5.4
Orchestration language	SOP Lang	Partially implemented / under refinement	Section 4.6
Evaluation	Internal suites	Preliminary / component-level	Section 5.5
Research trajectory	MRP-VM	Exploratory	Section 6

3 LITERATURE REVIEW AND TECHNOLOGY SURVEY

3.1 AI Agents

This section summarises the literature review and market-oriented survey on AI agents carried out during the first phase of the project. The review is intentionally selective. Rather than attempting exhaustive coverage of a field evolving too rapidly to permit a stable snapshot, it concentrates on the architectural patterns and engineering directions most relevant for ACHILLES, especially coding agents, deep-research agents, modular tool use, and controllable multi-step execution. During the first phase of the project, this activity constituted an important enabling line of work, because many design choices in the multi-agent environment depended not only on internal experimentation but also on a careful reading of emerging research and industrial practice. For this reason, the review places greater emphasis on industrial systems and technical documentation than on the older academic literature. This reflects a substantive shift in the field: the success of coding agents during 2025 made AI agents a much more concrete and operational research topic, centred on systems that work in practice and can be studied comparatively at the level of architecture, control, and deployment. A more consolidated academic literature is likely to follow these developments, and this deliverable should therefore be read as an early contribution to that emerging body of work, combining theoretical interpretation with implementation-oriented analysis.

A useful starting point is that agency is better understood as a spectrum than as a binary property, as also suggested in Anthropic's (Anthropic, 2024) discussion of agent design patterns. Some LLM-

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

based systems merely transform an input into an output without materially affecting control flow. Others can route between branches, choose tools, iterate through intermediate steps, invoke sub-agents, or generate executable code that reshapes the workflow itself, as increasingly visible in more recent OpenAI product and engineering directions (OpenAI, 2025a). A router, a tool-calling loop, a planner-executor system, and a code agent therefore place different demands on the surrounding runtime and require different control mechanisms.

Within this broader space, the literature repeatedly converges on a relatively stable set of design patterns. Prompt chaining remains one of the most robust patterns when a complex task can be decomposed into an ordered sequence of simpler steps. Routing is used when the first problem is to classify the task and dispatch it toward the most suitable prompt, tool, model, or specialised sub-agent. Parallelisation matters both for throughput and for reliability, either by splitting a task into independent parts or by generating multiple candidate outputs and later selecting or synthesising among them. ReAct-style loops remain central whenever the system must alternate reasoning and action over multiple steps, especially when tools are involved, as established by Yao et al. (2023). Planner-worker or orchestrator-worker architectures become useful when the task is long, heterogeneous, or naturally divisible across specialised competencies. Reflection loops, evaluator-optimiser schemes, tree-based search over candidate reasoning paths, and multi-agent debate patterns occupy a more specialised place, but they are increasingly relevant for constrained generation, verification, and deliberate exploration of alternatives.

These patterns should not be read as competing dogmas. In practice, the most effective systems combine them. A coding agent may begin with routing, continue with planning, execute a ReAct loop over tools, and then pass its output through an evaluator stage (Anthropic, 2024; OpenAI, 2025a). Likewise, a deep-research agent may combine retrieval, browsing, decomposition, parallel evidence gathering, and synthesis (LangChain, 2026). The main lesson from the reviewed literature is therefore methodological pluralism: agentic quality depends less on adherence to one canonical architecture than on the disciplined composition of bounded patterns under explicit constraints.

A second concept that deserves explicit treatment is the distinction between tools and skills. In the narrower engineering sense, a tool is an executable capability exposed to an agent through a stable interface, such as search, file editing, code execution, or API access. A skill is a higher-level reusable capability that packages one or more tools together with prompts, control logic, memory conventions, and evaluation routines, so that the resulting behaviour can be invoked as a relatively coherent agentic function. The transition from tools to skills is important because many useful capabilities in practice are not atomic. They involve several coordinated actions and therefore benefit from packaging as reusable units. This is especially relevant for ACHILLES, where the roadmap moves from infrastructure optimisation to packaged tools and then to interoperable agentic skills deployable across compatible environments (OpenAI, 2026b).

The review must also be read considering an important historical shift. Agent research has existed for decades, but the form of agency most relevant for ACHILLES is not identical to the older academic

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

notion of software agents or multi-agent systems. During 2025, the practical frontier changed substantially. Coding agents, deep-research agents, and other tool-using, long-horizon LLM systems began to appear as concrete products and developer platforms rather than only as experimental ideas. As a result, the meaning of AI agent in current engineering practice became more specific, more operational, and more commercially consequential. This is the main reason why the present review gives substantial weight to industrial systems and documentation. The academic literature remains important, but the contemporary object of interest was first stabilised in practice, while the scientific literature is still consolidating around it.

Among current frameworks, LangChain (LangChain, 2026a) remains a broad experimentation layer, but LangGraph has become particularly important for structured agent execution. Its main contribution lies in representing agent workflows as explicit graphs with shared state, conditional transitions, persistence, checkpointing, and human-in-the-loop intervention (LangChain, 2026b). This does not make LangGraph a universal answer to agent engineering, but it does make it especially relevant where long-running workflows, resumability, inspection, and approval gates matter. A more distinctive research direction is represented by Hugging Face's smolagents (Hugging Face, 2026), whose explicit separation between CodeAgent and ToolCallingAgent makes visible a deeper design choice: whether the model should act through a bounded vocabulary of structured calls, or whether it should express intermediate behaviour as executable code (Hugging Face, 2026).

OpenAI's Agents SDK (OpenAI, 2026c) reflects another important direction in the field: the move toward compact but production-oriented abstractions centred on agents, tools, sessions, handoffs, and tracing. Particularly important are tracing and handoffs. Tracing recognises that observability becomes essential once agent loops are multi-step and tool-rich. Handoffs recognise that delegation is not an exceptional case but a first-class orchestration primitive. Together, these features point toward a more mature view of agents as systems that must be inspectable, instrumented, and decomposable.

The same shift is visible most clearly in coding agents. Claude Code is explicitly presented as an agentic coding tool that can read a codebase, edit files, run commands, and integrate with development tools, while also stressing approval steps, checkpoints, command-line integration, and programmable use through a dedicated SDK (Anthropic, 2026a; Anthropic, 2026b). Deep-research agents form a parallel trajectory. OpenAI's deep research is explicitly described as a multi-step research capability that decomposes a high-level task, browses and analyses multiple sources, and produces citation-rich reports (OpenAI, 2025a). For ACHILLES, these developments matter because they validate an architectural direction in which research assistance, information gathering, synthesis, and action execution begin to converge within a single runtime.

The multi-agent space remains highly active but also unstable. Microsoft AutoGen (Microsoft, 2026) continues to be relevant as a framework for autonomous or human-in-the-loop multi-agent applications, yet its own repository now directs new users toward Microsoft Agent Framework, while AutoGen continues in a more maintenance-oriented mode. This is a useful caution for research

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

projects: current frameworks are valuable sources of patterns, but their trajectories may change quickly. The success of coding agents in 2025 did not merely increase attention around the topic. It also clarified which forms of agency matter most in practice: systems that can operate over long horizons, use tools reliably, remain inspectable, and produce valuable results under human supervision. This practical clarification is now beginning to shape both research agendas and engineering expectations.

Overall, the survey suggests several conclusions directly relevant for ACHILLES. Effective agent systems are increasingly built from composable patterns rather than from a single monolithic architecture. The move from tools to reusable skills is becoming more important, especially where capabilities must be packaged and reused across environments. Coding agents and deep-research agents have emerged as particularly informative practical benchmarks because they stress long-horizon execution, heterogeneous tool use, and user oversight. The first phase of the project has covered only a meaningful subset of this broader space, but that subset has already been sufficient to inform the project's work on specialised planning, controllable execution, modular tool packaging, and the later exposure of higher-level agentic skills. In this sense, the literature review is not only retrospective. It also helps position the project within a newly stabilising area of research in which theoretical clarification and implementation practice are advancing together.

OpenClaw is useful mainly as a secondary case from which practical lessons can be extracted. Its architecture is not especially original at the level of first principles. It combines a gateway-centric runtime, persistent sessions, tool invocation, skills, and multi-agent routing, all of which are now recognisable as convergent patterns across contemporary agent frameworks rather than as radically new inventions (OpenClaw, 2026d). For the present review, its importance is therefore heuristic rather than foundational. It illustrates both the promise and the fragility of open agent ecosystems, especially around skill packaging, persistent assistant surfaces, and extensibility, while also confirming the importance of stronger governance, installation discipline, and explicit trust boundaries when such systems are considered for enterprise or regulated environments (OpenClaw, 2026g).

3.2 Neuro Symbolic AI and Executable Natural Language

Closely related lines of work clarify adjacent parts of the same design space. ReAct (Yao et al., 2023) shows that interleaving reasoning traces and actions can provide an effective local execution loop even in the absence of a sharply formal intermediate language. PAL (Gao et al., 2023) takes a different approach by allowing the language model to translate a problem into code and then delegating execution to Python. This is highly relevant because it makes the division of labour explicit: interpretation remains neural, while execution becomes deterministic (Gao et al., 2023). Toolformer (Schick et al., 2023) adds a further variation by training the model to decide when to invoke external tools and how to integrate their outputs into generation. SayCan (Ahn et al., 2022) is particularly relevant to the discussion of skills because it operationalises natural-language instructions through a discrete skill space filtered by feasibility estimates grounded in the agent's actual capabilities. Taken together, these systems indicate that the central issue is no longer only text-generation quality. It is the design of runtimes in which language can trigger, constrain, or partially define structured execution.

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

This literature suggests that natural language is becoming operational in several distinct ways. In one family, language is translated into logic or code and executed by an external formal substrate, as in controlled natural languages or program-aided approaches (Gao et al., 2023). In a second family, language is interpreted by a runtime that uses memory, branching, and tool calls to execute a procedure incrementally, as in CoRE (Xu et al., 2024) or ReAct. In a third family, language remains relatively unconstrained, but its execution is shaped by external restrictions such as typed tool interfaces, as discussed by Beurer-Kellner et al. (2023), as well as grammars and schemas (Schick et al., 2023), or grounded skill libraries, as in Ahn et al. (2022). This taxonomy is useful for ACHILLES because it clarifies that agent execution is not a single paradigm. Different workflows require different balances between flexibility and formal control. A related issue concerns memory and the evolution of retrieval toward more active knowledge substrates. Standard retrieval-augmented generation remains useful, but once intermediate structures become more explicit, the knowledge base can do more than return passages. It can support typed lookup, local summaries, candidate decompositions, plan fragments, constraint indexes, structured examples, or cached transformations over recurring tasks. This broader perspective is relevant for ACHILLES because repeated agentic reasoning is often expensive precisely when structure must be reconstructed repeatedly from loosely organised context. More explicit intermediate structures create the conditions for turning the knowledge base into a more active reasoning substrate and for optimising recurring skills and workflows more systematically (Xu et al., 2024). This point is closely related to the optimisation of reasoning. Once prompts, tools, intermediate state, and control flow are treated as explicit components, it becomes possible to ask which parts of an agent workflow should remain model-mediated and which parts can be stabilised, learned, cached, or compiled into more efficient mechanisms. DSPy (Khattab et al., 2023) is particularly relevant in this respect because it treats LM-based systems as modular programs that can be optimised against downstream metrics rather than maintained as fragile prompt strings. This is directly relevant for ACHILLES, where one long-term objective is not merely to build agents that work, but to understand how stable patterns of execution may later be made more efficient, more auditable, and easier to maintain. The same perspective also prepares the ground for later discussion of agentic skills. If skills are treated as bounded computational units, then some may remain prompt-heavy and exploratory, while others may gradually migrate toward deterministic transforms, smaller learned routers, cached procedures, or formal validation stages.

Within this broader neuro-symbolic landscape, vector symbolic architectures and hyperdimensional computing are also relevant as possible intermediate substrates (Kleyko et al., 2021). Their importance for the present review lies in the fact that they offer compositional vector representations with algebraic operations such as binding, bundling, and similarity-based retrieval (Kleyko et al., 2021). This makes them potentially useful for compact structured memory, analogical retrieval, theory sketches, or fast pre-reasoning over artefacts before more expensive backends are invoked. In the context of agent research, such methods may support approximate but structured intermediate computations that are more informative than raw embeddings and lighter than full formalisation.

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

Overall, the literature supports a clear conclusion. Neuro-symbolic AI (NSAI) (Wan et al., 2024) is scientifically relevant for agent systems because it addresses the interface between flexible linguistic interpretation and structured computational execution (Colelough & Regli, 2025). This is directly relevant for ACHILLES and for multi-agent research because both require systems that can operate over human-readable artefacts while progressively introducing more explicit structure where the task demands it. The most useful directions for the present project are those that treat natural language as a front end for structured execution, those that separate interpretation from exact execution when stronger guarantees are required, and those that support more explicit memory, reusable skills, constrained runtimes, and selective validation. In this sense, the cited literature contributes complementary components of a broader neuro-symbolic architecture for advanced agent systems.

4 MULTI-AGENT ARCHITECTURE AND DESIGN: INNOVATION AND MAIN INSIGHTS

4.1 Vision

This chapter introduces the main concepts and architectural direction underlying the multi-agent environment developed in Task T3.3. Its role is not to provide low-level implementation detail, but to establish the conceptual and technical framework through which the ACHILLES project approaches deployable, trustworthy, and reusable agentic systems. The focus is therefore placed on the main abstractions, software layers, and execution principles that organise the environment, while more specific implementation aspects are left to dedicated technical documentation and annexes.

The general architectural view adopted in ACHILLES is that of a vertically integrated yet modular stack. At the lower levels, the environment requires a secure and reproducible deployment substrate for agents and services. At the middle levels, it requires a runtime model for agentic execution, including structured interaction with skills, bounded looping, context handling, and interoperability between components. At the higher levels, it must support more advanced research directions, including symbolic and neuro-symbolic mechanisms intended to reduce dependence on unconstrained LLM-based execution and to improve efficiency, predictability, and auditability. Finally, these architectural principles must become operational in the concrete use cases and in the ACHILLES IDE. A central idea of this chapter is that robust multi-agent systems should not be understood merely as collections of conversational agents, but as governed execution environments in which specialised capabilities, services, and skills can be composed under explicit control. In this perspective, the key engineering questions concern decomposition, interoperability, observability, bounded autonomy, and the progressive externalisation of procedural knowledge into reusable execution units. This is also why the chapter places particular emphasis on the distinction between agents, skills, sessions, plans, and

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

deployment components. These are not interchangeable notions, but complementary layers of the same architecture.

Another important point is that the ACHILLES approach goes beyond orchestration alone. While a substantial part of the work concerns multi-agent deployment, skill execution, and runtime governance, the longer-term direction also includes the development of more structured neuro-symbolic approaches. The strategic objective is to reduce excessive reliance on LLMs for planning and execution, while preserving their value for handling the nuances of natural language. This broader direction is reflected in the chapter through the introduction of symbolic execution ideas, intermediate representations, and architectural hypotheses about how classical AI, symbolic AI, and modern machine learning can be combined in a more efficient and predictable environment.

Overall, this chapter should be read as an architectural and conceptual entry point into the rest of the deliverable. Its purpose is to clarify the main building blocks, the terminology, and the design logic of the ACHILLES multi-agent environment before the document moves to more specific mechanisms, research directions, and implementation-oriented results.

4.2 General Architecture

The architectural framework of the T3.3 and the multi-agentic environment proposed by ACHILLES Project could be conceptualised as a vertically integrated stack designed to facilitate complex multi-agent interactions through a modular and scalable approach.

At the foundational deployment level, the environment relies on Ploinky, which introduces the notion of the **Ploinky Agent** as a standardised deployment unit for the sandboxing, containerisation, and operational exposure of AI agents. Although Ploinky is natively optimised for agents implemented through the AchillesAgentLib, it is designed as an engine-agnostic hosting layer and can support heterogeneous agent technologies, provided that they comply with the interaction assumptions and runtime conventions defined in this architecture. In parallel with this deployment layer, the environment includes the **LLM Proxy**, a generic gateway for large language models that exposes an OpenAI-compatible interface. This component is particularly important for production-oriented deployments because it complements raw model access with practical runtime functions such as observability, fine-grained configuration, safety controls, cost monitoring, and audit logging.

On top of this infrastructure, the **AchillesAgentLib** provides the main framework for agent development. Its first important role is to standardise communication with models through the **LLMAgents** module, which supports both completion-style and streaming interaction patterns, either directly against model providers or indirectly through the LLM Proxy using a unified access model. Above this communication layer, higher-level agent execution is organised through **Agentic Sessions**, in particular **Loop Sessions** and **SOP Lang Sessions**, which implement iterative reasoning-and-action cycles closely related to ReAct (Yao et al., 2023) style execution patterns, where intermediate reasoning

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

is interleaved with tool use and observation. This is the main mechanism through which agents can address longer and more complex tasks while preserving explicit intermediate execution structure.

A central architectural distinction in the ACHILLES environment is the replacement of the conventional notion of a tool with the richer notion of a **skill**. In many current agent frameworks, tools are exposed through narrow formal interfaces, typically structured arguments such as JSON schemas or command-line parameters. In the present architecture, by contrast, a skill receives natural language context and is responsible for interpreting its own parameters, selecting the relevant subordinate operations, and coordinating the execution path required to achieve its objective. This design is motivated by the observation, discussed in the broader literature on executable natural language and agent runtimes, that increasingly many useful computational behaviours are specified first in human-readable form and only later stabilised into more formal execution units (Beurer-Kellner et al., 2023; Xu et al., 2024). Within ACHILLES, skills are grouped into several broad categories. **Orchestration Skills** coordinate the behaviour of other modules. **DB Table Skills** focus on structured data access and manipulation. **C-Skills** are implemented through static and validated code. **G-Skills** dynamically generate parts of their own execution logic. The main implementation pattern supporting this model is the **RecursiveSkilledAgent**, which loads skill descriptors and manages the recursive orchestration of more complex behaviours.

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

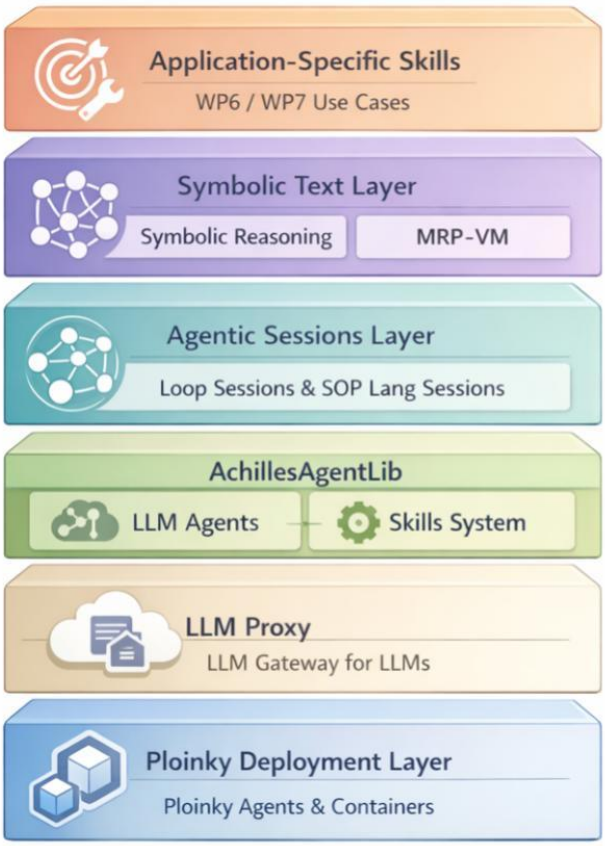

Figure 1 Technologies Stack

The diagram illustrates a vertically layered architecture for a multi-agent system, starting from the deployment layer (Ploinky Deployment Layer and LLM Proxy) up to higher-level orchestration and application layers. At the core, AchillesAgentLib and the agentic sessions manage execution, coordination, and interaction with language models. The upper layers (symbolic reasoning and application-specific skills) enable the development of complex use cases through structured "skills" and advanced decision-making mechanisms.

Above this layer, the stack extends into the **SymbolicText** area, which represents one of the more research-intensive parts of the architecture. The objective here is to strengthen the reliability of probabilistic language-model-based execution by introducing more explicit symbolic or quasi-symbolic structure where this becomes useful. Two experimental components are especially relevant. The first is **Symbolic Reasoning**, which explores algorithms based on **Vector Symbolic Architectures**, also known as **Hyperdimensional Computing**, and related families such as **Holographic Reduced Representations**. These approaches represent information in high-dimensional vectors and support algebraic operations (Kleyko et al., 2021) over compositional structures, making them potentially useful as intermediate substrates between purely neural approximation and rigid symbolic data structures. The second is the **MRP-VM**, which supports the translation of natural-language artefacts into an explicit

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

intermediate representation that can later be manipulated, checked, and executed under more structured computational assumptions. At the current stage of the project, these components remain exploratory and prototypical, but they indicate an important long-term research direction within ACHILLES: the gradual enrichment of agent execution with more explicit internal structure, stronger intermediate representations, and better support for selective validation. At the highest level, the architectural stack culminates in application-specific skills developed for the WP6 and WP7 use cases, which provide the practical and domain-oriented realisation of the broader architectural principles.

Choreography should not be confused with orchestration. Executable choreographies (Alboaie, 2016) should not be confused with orchestration. By choreography, and especially by executable choreography, we refer to the ability to introduce an architectural layer that consolidates interaction contracts in an executable and enforceable form in code, without requiring them to take the form of smart contracts.

This is particularly relevant for multi-agent systems in which privacy and confidentiality are important, and in which larger numbers of AI agents begin to form decentralised systems across legal entities operating under limited trust. In such settings, the presence of an executable choreography layer becomes increasingly important.

Given the speed and precision with which AI agent systems can be attacked, or may themselves organise attacks, executable choreographies represent an important direction for building systems with security and privacy by design. They offer a way to define admissible interaction patterns explicitly, constrain behaviour across organisational boundaries, and make deviations easier to detect, analyse, and contain. In this sense, executable choreographies are not only a coordination mechanism, but also a structured control layer for trustworthy multi-agent environments. At present, AchillesAgentLib provides orchestration, either at the level of agents or at the level of skill planning and execution. A separate question concerns the enforcement of executable interaction contracts in real multi-agent systems. When the number of agents is small and their roles are clear, such a layer is not strictly necessary. In larger and more heterogeneous systems, this becomes a more important architectural concern.

The present document does not discuss in detail a dedicated multi-agent choreography layer. By month 18, the main implementation effort has focused on the parts of the stack that were more urgent to stabilise, validate, and integrate in software. This includes deployment, skill execution, model access, and the first layers of structured symbolic augmentation. This prioritisation reflects a pragmatic sequencing of work rather than a rejection of choreography as a research direction.

Multi-agent choreography remains a relevant direction for the later stages of the project. As the number and heterogeneity of interacting agents increase, explicit executable choreography becomes more justified. It can constrain admissible interaction patterns, clarify responsibilities, define expected sequences of behaviour, and make deviations easier to detect. It can also support analysis at system

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

level by making it easier to reason about emergent behaviour across multiple agents and to localise failures or deviations introduced by malfunctioning, misconfigured, or malicious agents.

Architecturally, this layer should be added explicitly and should be placed close to the Ploinky Deployment Layer, most likely as an extension of Ploinky rather than as a responsibility of AchillesAgentLib. It belongs at the level where agent deployment, interaction boundaries, and execution conditions are controlled. For this reason, although it is not yet a primary implementation focus, it remains an important direction within the broader ACHILLES research agenda on trustworthy and structured multi-agent environments.

4.3 Loops, Skills, and the Path Toward More Structured Execution

This chapter summarises the current architectural positioning of ACHILLES with respect to AI agents and clarifies the trajectory of the project toward more structured execution models. Its main purpose is not to introduce a final architecture, but to state clearly which layer of the system is already relatively mature, which layer remains exploratory, and why this distinction matters for the scientific and technical contribution of Task T3.3. At the current stage, ACHILLES is positioned within the now well-established family of loop-based agentic systems. In this layer, execution is organised around iterative control loops, bounded tool use, explicit traces, and governed interaction with external artefacts and services. This is the part of the architecture that is already sufficiently grounded in current engineering practice and that can support practical integration in ACHILLES IDE, the ACHILLES CLI copilot, and the project use cases. In this sense, loops remain the operational foundation of the present system.

Within this runtime model, skills are the main units of procedural specialisation. They are not treated merely as convenience wrappers around tools, but as bounded execution units that encapsulate local competence, local assumptions, and reusable operational behaviour. This view is central to the ACHILLES contribution. It allows the general runtime to remain relatively compact and governable, while pushing specialised behaviour into explicit, testable, and reusable modules. It also creates a clearer separation between high-level orchestration and local execution, which is important for maintainability, auditability, and later optimisation.

The chapter also marks the transition toward a more ambitious research direction. Once a skill has stabilised functionally, parts of its behaviour may become candidates for more structured execution forms, including deterministic code, validators, parsers, typed retrieval, constrained transforms, or other bounded interpreters. This is the point at which the architecture begins to connect with the broader neuro-symbolic direction of the project. The goal is not to abandon loop-based agents, but to use them as the practical starting point from which recurrent local behaviours can gradually be transformed into cheaper, more auditable, and more structurally explicit execution regimes.

The architectural thesis is therefore straightforward. Loops provide the current execution discipline. Skills provide the current specialisation mechanism. More structured execution is the

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

longer-term direction through which some of these skill-level behaviours may later be stabilised into stronger internal representations and alternative interpreters. The contribution of ACHILLES at this stage is to make this trajectory explicit: to treat the current skill-oriented agent layer not as an endpoint, but as a practical and scientifically meaningful bridge toward a more structured and more governable computational architecture.

4.4 Agents Taxonomy

The concept of an **agent** is unavoidably overloaded. Its meaning has accumulated across the history of AI research, ordinary natural-language usage, older technical traditions such as software and network agents, and, more recently, the rapidly evolving discourse around LLM-based systems. Our own research introduces further nuance once skills, sub-agents, orchestration loops, and interoperable deployment units are considered together. This chapter defines a precise vocabulary to unify the agency models planned or active within WP6, as detailed in D6.1.

In the present document, an agent is understood as a system that transforms input, typically natural language or other structured signals, into utility-producing action while operating inside a bounded execution environment. In practice, such an agent may rely on a language model, on conventional software logic, on symbolic procedures, or on a combination of these. What matters architecturally is not whether the component appears human-like, but whether it occupies a meaningful role in decision, transformation, mediation, validation, or execution. This broader usage is consistent with older technical meanings of the term, such as user agent in Internet architecture, where the concept already referred to a software entity acting on behalf of a user without implying anything like a full human-like intelligence.

This perspective is closely related to the broader architectural direction of ACHILLES. The project advances a shift away from monolithic and weakly inspectable black-box AI applications toward modular systems built from coordinated units of intelligence. In our view, this shift is best understood not only as a move toward multi-agent systems, but more fundamentally as a move toward the orchestration of an ecosystem of agentic skills. The broader ambition is therefore closer to a society of specialised execution units than to the image of one undifferentiated autonomous intelligence (Wang et al., 2023). This matters both scientifically and practically. As agentic systems become more capable, the central challenge is no longer only to obtain convincing outputs from a powerful model, but to organise heterogeneous capabilities so that they remain performant, inspectable, reusable, and easier to control under real operational constraints.

We also introduce the concept of a **choreographic agent**, that is, an agent whose role is defined through participation in a broader choreography. In our earlier research, choreography was used to describe distributed execution models in which system-level behaviour emerges from the structured interaction of multiple participants rather than from one centralised controller (Alboaie et al., 2016). Although that earlier work did not concern contemporary LLM-based agents directly, the underlying intuition remains highly relevant: scalable intelligent systems often benefit from being understood as

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

coordinated ensembles of bounded units rather than as monolithic entities. In ACHILLES, this intuition is extended to AI agents and skill-oriented execution. The project is therefore interested not only in isolated agents, but also in the explicit orchestration and, later, potentially the executable choreography of multiple specialised components, especially where traceability, substitution, security boundaries, and privacy matter.

At the same time, our current position is more precise than a generic preference for "many agents." In practice, the most stable and optimisable unit is often not the persistent agent as such, but the skill. For this reason, while ACHILLES certainly supports agent-based coordination, its deeper direction is toward the **orchestration of skills** inside a multi-agent environment. This preserves many of the advantages associated with agent societies, such as specialisation, decomposition, and distributed responsibility, while reducing some of the fragility associated with loosely defined long-running agent personae.

Against this background, the taxonomy used in the project should be understood primarily as a taxonomy of roles and architectural viewpoints rather than as a fixed inventory of software entities. In implementation, a single entity may play several of these roles at once. Conversely, in more complex systems, some of these roles may need to be realised as distinct software components. The purpose of the taxonomy is therefore explanatory and architectural: it helps clarify how an agent functions, what kind of responsibility it carries, and how it relates to other components in the ecosystem. It is also useful because it offers a common frame for what is planned or already emerging in WP6 and for the terminology already referenced in D6.1.

A first set of distinctions concerns high-level coordination. A **parent agent** is an agent that can create, initialise, supervise, and terminate other agents or bounded execution units. Its defining feature is lifecycle control over subordinate entities. A **dispatcher agent**, by contrast, assigns tasks to other agents or components without necessarily creating or owning them. Its role is routing rather than lifecycle management. A **super-agent** is a broader and less differentiated coordinating entity that may combine several higher-level functions at once, including interpretation of intent, planning, routing, supervision, and synthesis of results. These roles are conceptually distinct even when, in implementation, they may be combined inside the same top-level component.

Within WP6, the **Master Copilot**, usually exposed as **ACHILLES Copilot** or **ACHILLES CLI**, is the main high-level interaction point for the user and the principal coordinator across the environment. Depending on the implementation context, it may simultaneously act as a **parent agent**, because it can initiate and supervise subordinate execution flows; as a **dispatcher agent**, because it can assign subtasks to specialised downstream components; and as a **super-agent**, because it can combine interpretation, orchestration, and synthesis in one broader coordinating role. We also envisage a **moderator agent**, especially relevant in multi-agent and multi-user settings such as the planned WebMeet tool in WP6. Its role is to supervise interaction quality, policy boundaries, turn-taking, and, where needed, approval or intervention points.

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

A second set of distinctions concerns execution-oriented forms of agency. The **LLM Agent** is the component centred on controlled interaction with a language model and on the management of model-mediated reasoning steps. The **operator agent** is an execution-oriented unit that performs concrete actions over tools, APIs, files, services, or other external interfaces. A **sub-agent** is a specialised bounded execution unit that operates under the broader context of a parent or coordinating agent while following its own local execution loop. Sub-agents are useful when local autonomy is needed, but within explicit scope and under supervision.

A **skilled agent** is an agent whose main competence is organised around reusable skills and bounded procedural modules. This is an important architectural family in ACHILLES, but it should not be mistaken for the only possible architecture. The term is useful because much of the current runtime is indeed **skills-centered**. At the same time, future directions such as MRP-VM point toward other possible architectures in which the central abstraction is no longer the skill, but the interpreter or execution regime. For this reason, the skilled agent should be understood as one major family within the current system, not as the universal model of all future agents in the ecosystem.

An **ACHILLES Agent** is the stronger and more domain-oriented form within this space: pragmatic machinery for automating human work in specific domains under explicit architectural control, built around AchillesAgentLib and around the project's stricter interpretation of the skill concept.

A Ploinky Agent is the minimal interoperable deployment unit recognised by the Ploinky environment. It preserves only the minimum interface and conventions needed to participate in the ecosystem. Such an entity may be implemented with many different technologies, not only with the ACHILLES stack. In this sense, it is a thinner and more infrastructure-oriented notion of agent than the ACHILLES Agent, but it remains useful because Ploinky defines a minimal layer of conventions for interoperability, deployment, and coordination (D6.1).

A third family of roles is directly aligned with the trustworthy-AI and research objectives of the project. The **bias agent** focuses on detecting, surfacing, or mitigating bias-related issues in model behaviour or generated artefacts. The **efficiency agent** focuses on runtime, resource consumption, optimisation choices, and execution cost. The **validation and verification agent** is responsible for stronger checking of outputs, procedures, intermediate artefacts, or compliance with explicit constraints. The **privacy agent** has a particularly important role. In D2.3, this role is described as the **DPU Agent**, that is, the Data Processing Unit Agent. Its function is not only to preserve secrets and control access to private or sensitive data, but also to enable code or bounded computations to run inside a protected environment that has access to the data, thereby reducing unnecessary exposure of the data itself. This makes it possible to move computation closer to protected data rather than moving the data toward less controlled execution contexts.

A fourth family comprises infrastructure, tooling, and integration roles. The **Ploinky CLI** acts as the operational command-line interface to the Ploinky environment and its managed components. In a broad sense, it could also be viewed as a separate type of agent, since it exhibits agentic behaviour: it

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

depends on **AchillesAgentLib**, can assist the user through natural-language interaction, and could itself be deployed as a **Ploinky Agent**. At the same time, this should not be overstated. Many powerful command-line tools, such as git, display forms of structured operational agency without usually being described as agents. This is precisely why the distinction between implementation and architectural role matters.

A further distinction is also needed between the role or taxonomic class to which a component belongs and the effective instances that appear in a concrete solution. The situation is somewhat analogous to the distinction between classes and instances in object-oriented programming, although, paradoxically, in a real system the concrete instance name often matters more operationally than the taxonomic label. In WP6, for example, we also developed a **GitHub Agent** and other components that, at implementation level, may still resemble relatively simple tools, but that can evolve into interpreters of natural-language requests and, over time, into more autonomous and more clearly separated software entities. What matters in the present taxonomy is therefore not only the current implementation form, but also the abstraction and the role that a component plays within the overall system.

Figure 2 Taxonomy of Agents

The figure illustrates a structured overview of AI agent types, starting with high-level categories such as ACHILLES Agents, Ploinky Agents, Sub-Agents, and Unified Agent Definitions. It then presents a detailed taxonomy organised into four main groups: core orchestration and interaction, execution and

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

operational agents, trustworthy AI and research agents, and infrastructure and integration agents, each including specific roles that highlight their functions within AI systems.

The present taxonomy should be read as a pragmatic architectural map rather than as a final ontology. Some standardisations of interfaces across these categories may later become important, especially for interoperability, traceability, and controlled orchestration, but we do not want to harden distinctions prematurely. In many cases, natural language or lightweight textual conventions may remain sufficient.

The main point is that ACHILLES is not centred on one monolithic assistant. It is conceived as a layered environment in which different forms of agency coexist under different constraints and responsibilities. The current architecture is largely skill-centred and loop-based, but this should be seen as one part of a broader design space. Later directions, especially those associated with MRP-VM, may lead to more interpreter-centred and regime-oriented architectures.

It is also useful to distinguish between an agent and an agentic loop. An agent may instantiate one or several loops during its lifetime, including while handling a single request. The loop is not itself a separate agent type, but the basic orchestration pattern through which tools, skills, APIs, MCP-accessible resources, validators, and local execution logic are coordinated. For this reason, the taxonomy is included primarily to orient the reader and to clarify the conceptual scope of the research, not to fully specify each agent type in implementation detail. The deeper point is that ACHILLES aims at a transparent and composable ecosystem of agent-like units, organised around bounded autonomy, explicit coordination, interoperability, and traceable execution.

4.5 Agent Harness for Automated Research

The notion of the **agent harness** has become clearer as the field has moved beyond isolated “agents” and toward the engineering question of how such systems can be made reliable, governable, and reusable in practice. In the present deliverable, an agent harness is understood as the execution and control environment that turns a model, or a collection of models, skills, and tools, into an operational system. It is not identical with the intelligence of the underlying model, and it is not reducible to a prompt, a library, or a workflow script. Rather, it is the runtime layer that governs tool use, state, memory, delegation, context shaping, stopping conditions, validation, observability, and policy boundaries. This makes the concept adjacent to, but distinct from, more familiar software notions such as libraries, frameworks, and middleware. A library offers reusable functions. A framework provides an architectural skeleton. Middleware mediates between components and infrastructure. An agent harness shares something with all three but is not exhausted by any of them. Its role is more operational and more governing: it determines how tasks are interpreted, which tools and skills are admissible, how intermediate artefacts are checked, when escalation is required, how continuity is preserved across long tasks, and how the whole process remains inspectable afterward. In this sense, the harness becomes the main engineering locus of agentic systems, while the foundation model becomes one powerful but replaceable component inside a larger controlled architecture.

Funded by the European Union
This project has received funding from the European Union’s Horizon Europe research and innovation programme under Grant Agreement No 101189689.

This distinction is central for ACHILLES because the project does not aim only to build a conversational copilot. The broader objective is to develop a reusable model for constructing **agent harnesses for automated research**. What is being built is therefore not one monolithic agent, but a family of reusable architectural elements from which domain-specific research harnesses can be assembled, governed, and progressively enriched. This is also the level at which exploitation becomes more realistic: reusable harness components can be adapted to different research and industrial settings without rebuilding the whole system from scratch. Seen in this way, ACHILLES IDE is not only a user interface. It is part of a substrate for research automation. Its value lies in making visible and governable the artefacts, skills, sessions, and pipelines through which research work is structured over time. The copilot developed in WP6 is one human-facing entry point into this environment, but the deeper contribution lies in the ability to orchestrate agentic skills, connect specialised tools, preserve explicit artefacts, and maintain continuity across longer research processes.

This architectural direction is closely related to the project's specification-driven intuition. In research settings, valuable work rarely consists of one-shot generation. It evolves through a sequence of artefacts such as requirements, hypotheses, plans, datasets, analyses, simulations, reports, revisions, and conclusions. A useful harness must therefore operate not only over prompts, but over persistent and manageable artefacts connected through pipelines. This is one of the reasons why the custom pipelines available in ACHILLES IDE are important: they allow skills and transformations first used in an intensive research phase to be stabilised into longer-term workflows that keep projects active, inspectable, and capable of adapting to new evidence.

From this perspective, an agent harness can be understood as a governed collection of agentic skills, tools, memory structures, and orchestration rules, together with the custom pipelines that connect them. Some of these skills may remain strongly LLM-mediated, especially in phases of interpretation, generation, summarisation, or exploratory decomposition. Others may gradually move toward more efficient and more formalised execution regimes, including deterministic procedures, structured validators, symbolic reasoning components, and later forms of partially symbolic execution. This is precisely why the harness concept provides a useful bridge between the practical agent layer of ACHILLES and the more structured neuro-symbolic direction developed later in the deliverable.

The practical scope of this vision is broader than trustworthy AI tooling alone, although such tooling remains an important component of the project. The same harness architecture can gradually be extended to support the controlled execution of simulations, the design and scheduling of experiments within those simulations, the management of research documentation and evolving conclusions, and the integration of external scientific pipelines. In such an environment, agents are valuable not only because they generate text, but because they can coordinate work across heterogeneous computational regimes and preserve the continuity of research projects over time. This is particularly relevant in scientific and industrial research, where the main challenge is rarely to produce a single answer in one interaction.

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

Real research work involves literature navigation, hypothesis generation, protocol comparison, multimodal evidence synthesis, experiment design, simulation control, result interpretation, and iterative revision under uncertainty. Recent survey literature (Yao et al., 2023) on scientific agents and agentic AI for scientific discovery emphasises exactly this shift toward multi-step, tool-rich, closed-loop systems rather than raw language generation alone.

For this reason, different research organisations do not merely need different prompts. They need different bounded epistemic machines. A useful harness must therefore be customisable not only at the level of surface behaviour, but at the level of internal discipline. It must encode which tools are admissible, which sources are trusted, how uncertainty is escalated, how provenance is preserved, when symbolic checks override heuristic suggestions, and how humans remain in control. This is where scientific seriousness enters the architecture.

What ACHILLES is therefore constructing, in practical terms, is a reusable substrate for custom research harnesses. Its reusable elements may include skill abstractions, orchestration patterns, handoff mechanisms, memory and context modules, interfaces to ML (Machine Learning) and simulation pipelines, knowledge-base adapters, traceability layers, governance controls, and persistent artefact pipelines within ACHILLES IDE. These elements can then be recombined to accelerate different forms of research while preserving adaptability and control.

The longer-term implication is significant. If this direction matures, agent harnesses may become for AI-native research systems what application frameworks once became for web software: not the intelligence itself, but the reusable discipline that makes intelligence deployable. In the present case, however, the concept goes further, because it concerns not only software structure, but also bounded agency, explicit artefacts, scientific workflow, and the interaction between generative models, institutional memory, symbolic reasoning, and human oversight. For research-intensive domains, this may become the decisive architectural layer.

4.6 SOP Lang

Understanding this subsection requires familiarity with SOP Lang as presented in D6.1, and we therefore do not repeat here a detailed explanation of what SOP Lang is, its syntax, or its general principles. At the same time, our understanding of SOP Lang has evolved, and this deliverable presents three possible extensions. Among them, SOP Lang IR should still be regarded as a partly speculative direction, arising from experiments connected to the neuro-symbolic research line referred to as Meta-Rational Pragmatics, which may still change substantially. In contrast, SOP Lang Pipeline, already presented in D6.1, and SOP Lang Plan, introduced here for the first time, are results that are expected to remain part of the scientific and technical contribution of the ACHILLES project. The discussion below therefore provides a concise summary and problem-oriented framing of the ideas emerging from our research on a domain-specific language for AI agent orchestration.

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

A significant part of our research on the ACHILLES multi-agent environment concerns the design of a family of closely related languages collectively referred to as **SOP Lang**. Within the current architecture, this family is intended to play three distinct roles. **SOP Lang Pipeline** is used for the orchestration of **Standard Operating Procedure** workflows through a build-like execution model inside **ACHILLES IDE**. **SOP Lang Plan** is used to express agentic execution in a compact and explicit form, suitable for compilation into dependency graphs and executable plans. **SOP Lang IR** aims to serve as an intermediate representation between natural language and more formal reasoning methods, preserving enough structure, and traceability to support inspection, controlled natural language generation, and downstream verification. In the present section, our purpose is not yet to define these variants in full detail, but to introduce the family as a whole and position its role within the broader ACHILLES architecture. Although these variants serve different purposes, they are designed as members of the same language family because they share a common syntactic foundation. At the basic level, all of them follow a line-oriented structure of the form *@identifier_or_variable* command parameters. This shared syntax is not merely stylistic. It is meant to reduce cognitive overhead, enable reuse of tooling, and make it possible to move more easily between procedural, agentic, and representational layers without brittle or excessively lossy translations. In some variants, especially in **SOP Lang IR**, the parameters themselves may follow a subject-verb-objects pattern, so that statements can be represented in a way that remains easy to reference, constrain, annotate, and reinterpret by downstream components. At the same time, the family is not intended to embody one single semantics. Our working assumption is that the needs of workflow orchestration, execution planning, and meaning-preserving intermediate representation should not be forced into a single formal model. Instead, **SOP Lang** is designed to support a combination of semantic and pragmatic roles within a unified structural style. Some variants are primarily operational, some are primarily representational, and some occupy an intermediate position between these two poles.

An important aspect of **SOP Lang Pipeline** and **SOP Lang Plan** is that both are intended to work with **skills**, which will later be defined more precisely as controlled operational units through which workflows and agent plans can invoke bounded capabilities. At this stage, it is sufficient to note that skills form the bridge between the abstract language layer and the concrete execution layer. This is one of the reasons why Pipeline and Plan are closely related, even though they operate at different levels of persistence and abstraction. Another important design element is that the operational variants are meant to admit a graph-oriented interpretation. Plans can be compiled into dependency graphs on which standard techniques such as **topological sorting** can be applied to derive safe execution orderings. To support this deterministically and avoid ambiguity, the language also allows an explicit dependency notation of the form *$identifier_or_variable,* used to signal referenced dependencies clearly rather than relying only on positional interpretation or informal conventions. This is especially important in execution-oriented contexts, where selective recomputation, parallelisation, and failure recovery depend on having a precise and machine-readable dependency structure. The three variants also differ in lifecycle. **SOP Lang Pipeline** and **SOP Lang IR** are intended as relatively stable artefacts that can be stored, versioned, and reused over time. **SOP Lang Plan**, by contrast, is intentionally more ephemeral and is designed to be generated and regenerated on demand as part of execution-time

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

orchestration. This distinction reflects a broader architectural principle in ACHILLES: some representations are meant to persist as durable references, while others are meant to remain lightweight and disposable so that the system can react efficiently to change without repeatedly reconsidering everything from scratch.

Taken together, **SOP Lang Pipeline**, **SOP Lang Plan**, and **SOP Lang IR** define a coherent language family aligned by a shared syntactic intuition but optimised for different architectural needs. Pipeline supports durable procedural orchestration; Plan supports explicit and graph-compilable agent execution, and IR supports persistent and traceable intermediate representation between natural language and more formal reasoning layers. Their common syntax is therefore not an accidental resemblance, but a deliberate engineering decision intended to support interoperability, inspectability, and more trustworthy AI behaviour across the ACHILLES environment.

It should also be emphasised that this deliverable presents these specifications as the current outcome of an evolving line of research rather than a frozen final standard. Several concepts introduced only at a high level in this section, including the exact notion of skill, the distinction between semantic and pragmatic layers, the graph interpretation of plans, the role of explicit dependencies, and the treatment of local scope and namespaces in the IR, will be clarified in the sections that follow. This section presents the design space and rationale for the three SOP Lang variants without providing their detailed specifications

4.7 Specification Driven R&D, Mirroring Pattern and Long-Lived Pipelines

An important innovation explored in ACHILLES concerns not only how agents execute tasks, but how AI-assisted research and development should itself be organised. Our working hypothesis is that, in such settings, the primary asset gradually shifts from isolated prompts or code fragments toward high-quality specifications and the pipelines that transform them into progressively more concrete artefacts. In this view, the central challenge is not only generation, but the maintenance of an explicit chain linking goals, constraints, design choices, intermediate results, and final outputs.

This intuition is reflected in what we currently call the **Mirroring Pattern**. At this stage, it should be understood as an evolving design hypothesis rather than as a fixed doctrine. Its core idea is that a successful AI-assisted project benefits from a ladder of specifications that mirrors the structure of the final artefact. In a software-oriented setting, one may begin from a high-level vision, refine it into functional and non-functional specifications, derive design specifications from these, continue toward module- or file-level specifications, and only then generate or modify code. In this process, code is no longer the primary source of intent, but the most concrete artefact in a longer specification chain.

The importance of this pattern extends beyond code generation. It suggests a more general principle for AI-assisted R&D: quality, traceability, and controllability improve when each level of abstraction is represented by an explicit artefact, and when the transformations between levels are visible and reviewable. A vision can be improved before it hardens into design. A design can be checked

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

against requirements before it propagates into implementation. Local specifications can guide generation and verification close to the place where change occurs. This shifts emphasis away from the idea of one large prompt and toward an explicit chain of governed artefact transformations.

This point becomes especially important when LLMs are involved, because many of their strongest practical capabilities are precisely capabilities of interpretation, reformulation, summarisation, translation between registers, and contextual completion. These capabilities are extremely valuable, but they also make the transition from general intent to concrete artefacts inherently risky. Every step from vision to requirements, from requirements to design, and from design to implementation may introduce subtle reinterpretations, omissions, or distortions of emphasis. In other words, the movement from abstract goals toward concrete outputs is not merely a decomposition process. It is also a sequence of potentially fragile semantic transformations.

For this reason, the specification ladder is not only an organisational convenience. It is also a control mechanism. Even when intermediate specifications are generated with the help of AI, their existence reduces ambiguity by forcing interpretations to become explicit earlier and at the appropriate level of abstraction. Instead of allowing the whole burden of interpretation to remain hidden inside one prompt or one large generation step, the process is externalised into a sequence of inspectable artefacts. This improves traceability because later outputs can be related back to the intermediate specifications from which they were derived. It also improves explainability because the system's decisions can be examined at multiple levels rather than inferred retrospectively from the result alone.

This is also where **human-in-the-loop** becomes essential. If AI systems are asked to translate broad intentions into increasingly concrete artefacts, then human supervision is needed not only for final approval, but also for controlling the interpretive drift that can occur along the way. Human review at key specification levels makes it possible to correct misunderstandings before they propagate downstream, to validate whether constraints and priorities were preserved, and to decide when an ambiguity should remain open and when it should be resolved more strongly. In this sense, human involvement is not a sign of weakness in the architecture, but a deliberate requirement for transparency, accountability, and practical trustworthiness.

This direction is also important because many research and engineering processes are long-lived. Initial intentions are often lost across phases, and later modifications must reconstruct rationale from scattered documentation or code history. Long-lived pipelines address this problem by preserving the project as an evolving chain of artefacts rather than as a sequence of disconnected outputs. New evidence, new requirements, or new evaluation results can then trigger localised regeneration of downstream artefacts instead of forcing repeated reconstruction of the full context from scratch.

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

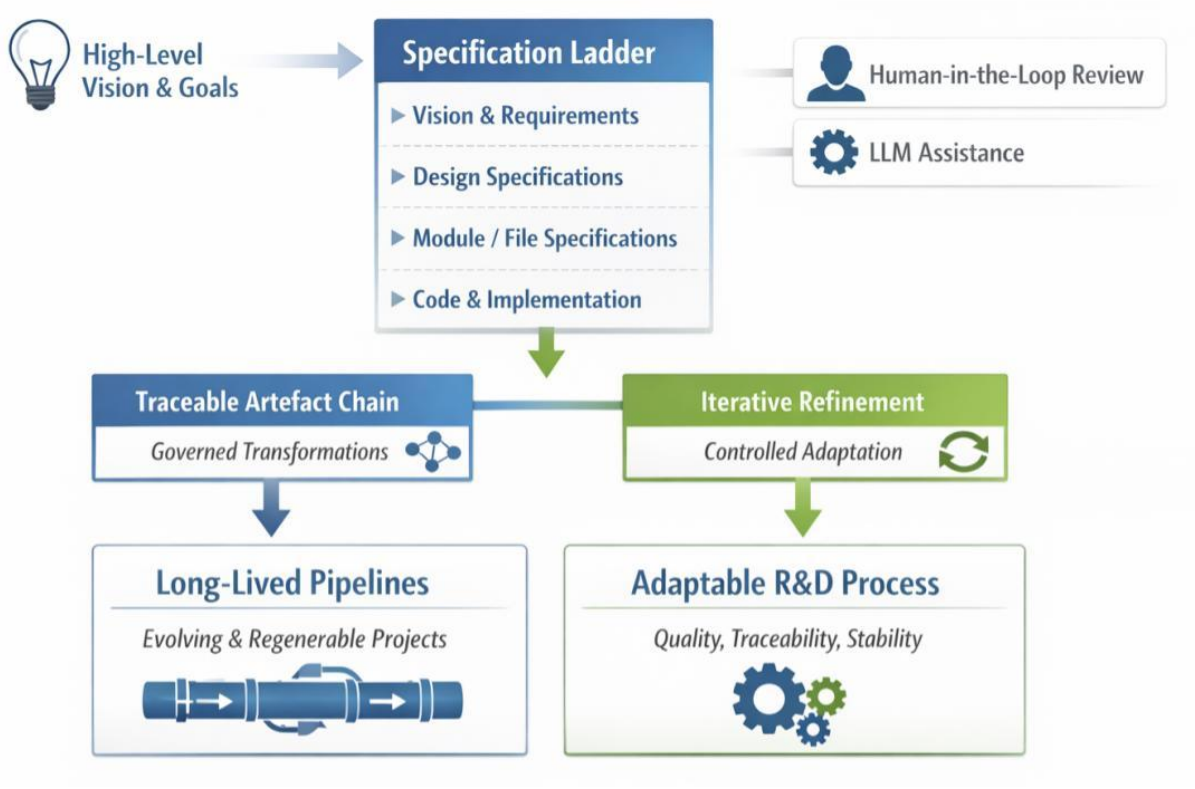

Figure 3 Specification Ladder and Iterative AI Development Process

The diagram shows a structured workflow for developing AI systems, starting from high-level vision and goals and progressing through a **Specification Ladder** that includes requirements, design, modules, and implementation. It emphasises continuous human-in-the-loop review and LLM assistance, alongside traceable artifact chains and iterative refinement. The process ultimately leads to long-lived pipelines and an adaptable R&D approach focused on quality, traceability, and stability.

For ACHILLES, this idea is particularly relevant because the project is not concerned only with short-lived agent interactions, but with environments in which skills, plans, specifications, reviews, and downstream outputs remain visible and governable inside ACHILLES IDE. In such a setting, the IDE and the copilot are not merely execution surfaces. They become environments for maintaining explicit artefact pipelines over time. Skills operate on specific layers of the specification ladder, and the outputs they produce can later become inputs for validation, refinement, or further transformation.

This approach is naturally compatible with context engineering. Rather than repeatedly injecting the full history of a project into a prompt, the runtime can load only the relevant slice of specification for the current task, invoke the appropriate skills, and validate the result locally. High-level intent remains relatively stable and global, while lower-level specifications remain stable and local. Concrete outputs, whether code, plans, analyses, or reports, become regenerable and reviewable artefacts rather than opaque final products.

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

For such a process to work reliably, the specification layer must remain both human-readable and machine-operational. Markdown provides an effective baseline because it is readable, diffable, and compatible with standard review workflows. The approach becomes stronger when at least part of the specification surface can be expressed in a more disciplined form, including controlled natural language or other structured representations. This is one of the reasons why SOP Lang is relevant in the broader ACHILLES architecture: it offers a path for making dependencies, constraints, and execution units more explicit while preserving a surface still close to natural language.

Within this broader picture, SOP Lang Plan provides a useful execution model. Skill invocations become nodes, variable references become explicit dependencies, and the resulting structure can be compiled into a dependency graph. This makes execution more auditable and allows partial reruns when upstream assumptions or artefacts change. Replanning may be triggered not only by execution failure, but also by review or validation steps that detect inconsistency, incompleteness, or deviation from specification. The goal is therefore not to eliminate change, but to absorb it through localised correction rather than uncontrolled global recomputation.

From this perspective, the Mirroring Pattern is one concrete manifestation of a broader innovation in agents planning. Planning should not be understood only as choosing the next action inside a short interaction loop. It should also include the explicit planning of artefact transformations across time. A robust agentic system should therefore be able not only to plan actions, but also to maintain and evolve chains of specifications, reviews, and outputs that keep a project intelligible, adaptable, and verifiable as it develops.

We therefore treat specification-driven R&D and long-lived artefact pipelines as an important research direction within ACHILLES. The Mirroring Pattern is the clearest current example, especially in code-related workflows where its practical value is already visible. More generally, however, it points toward a broader shift in AI-assisted R&D: from prompt-centric production toward specification-centric transformation, where the enduring value of the system lies in preserving quality, interpretability, traceability, adaptability, and controlled regeneration across an evolving chain of artefacts.

The **BMAD Method** (BMAD Method, 2026c) is also relevant in this context because it provides a recent practical example of specification-oriented, agent-assisted software development built around explicit artefact transitions rather than one-shot prompting (BMAD Method, 2026b). Its documented workflow progresses through analysis, planning, solutioning, and implementation, with each phase producing artefacts intended to inform the next, while its agent model organises role-specialised agents that are invoked as skills inside these workflows. For ACHILLES, BMAD is therefore useful less as a template to adopt directly and more as an external signal that the broader field is moving toward structured, context-engineered pipelines in which requirements, architecture, task decomposition, and implementation remain linked through explicit intermediate documents (OpenAI, 2025a). At the same time, it remains an open question how much structure is truly necessary in practice. BMAD shows the value of strong workflow scaffolding, role separation, and specification artefacts, but it also illustrates the overhead that such an organisation can introduce. This is closely related to our own

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

concerns in the present chapter. The Mirroring Pattern similarly assumes that quality, traceability, and controllability improve when the project is maintained as a visible chain of artefacts, but our current position remains more cautious about how far this structure should be formalised. In particular, we are still investigating which parts genuinely benefit from stricter workflow support and interface conventions, and which parts may remain more lightweight, with agents coordinating through natural language and minimally structured skills whenever that is sufficient (BMAD Method, 2026a; BMAD Method, 2026c).

4.8 Agentic Sessions

A significant innovation explored in ACHILLES concerns the explicit structuring of agentic execution around reusable and granular skills (ACHILLES Agent Library, 2026a), lightweight planning artefacts, and, where appropriate, more persistent transformation pipelines between managed artefacts (Khattab et al., 2023). This direction is important because it creates a practical bridge between short-horizon agentic execution and the longer-term vision of ACHILLES IDE, where research and engineering work should become progressively more traceable, reusable, and maintainable. In this broader picture, SOP Lang plays an integrative role by connecting temporary planning, bounded multi-step execution, and more durable pipeline-like orchestration (AssistOS, 2026a). The broader SOP Lang family, especially SOP Lang Pipeline, is described in D6.1. In the present subsection, it is sufficient to focus on SOP Lang Plan as the simpler planning-oriented member of the family.

The main contribution of this line of work is a different way of understanding and constructing AI agents compared with the current mainstream. In many widely adopted agentic systems, orchestration is still driven largely by a single LLM (Anthropic, 2025a) operating over an expanding conversational context, with tools or skills (Anthropic, 2026e) used mainly as auxiliary capabilities (Schick et al., 2023). ACHILLES explores a more structured alternative in which skills are treated as bounded execution units with clearer scope, more explicit responsibilities, and more disciplined input-output behaviour (BMAD Method, 2026a). This difference is important because it creates a more realistic path toward enterprise-grade agentic systems, especially in settings where correctness, reliability, validation, and explainability (Colelough & Regli, 2025) matter more than open-ended flexibility alone.

This approach is motivated both by practical evidence and by first-principles reasoning. On the practical side, the project is already observing efficiency gains, with concrete data discussed in D3.1. On the conceptual side, the use of more granular and more narrowly scoped skills improves the conditions for explainability and trustworthiness (Anthropic, 2026f). A skill that is designed to perform a limited class of actions well is easier to understand, easier to validate, easier to monitor, and easier to compose into larger workflows. In many cases, this reduces the need for heavy external guardrails, because part of the behavioural control (Anthropic, 2025b) is already embedded in the design of the skill itself. For this reason, the ACHILLES approach is especially relevant for enterprise-oriented niches, where generalist agents may remain powerful but are often more difficult to validate under realistic operational constraints.

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

This perspective has led the project to experiment with several forms of agentic sessions and with a broader interpretation of the skill concept itself. In ACHILLES, a skill is more than a simple tool wrapper. It is a bounded operational unit that receives inputs, interprets them under its own local assumptions, performs a coherent piece of work, and returns a result that can be reused downstream. This architecture makes it possible to build one common ecosystem of skills while allowing different orchestration regimes above it. In this sense, the project is not only proposing a set of components, but also a more general execution discipline for AI agents.

At present, ACHILLES supports two complementary kinds of agentic sessions. The first is a loop-based mode, close to ReAct-style execution in which the system repeatedly selects the next action, invokes a skill, observes the result, and continues until a stopping condition is reached. The second is a plan-first mode based on SOP Lang Plan, in which the task is first translated into a compact executable plan and only then executed (Khattab et al., 2023). These two modes share the same skill ecosystem and should be understood as complementary orchestration styles rather than separate architectures. This is an important design choice, because it allows the project to compare and combine different execution strategies without fragmenting the underlying implementation model.

The loop-based mode remains valuable for tasks in which interpretation must evolve progressively, ambiguity must be resolved during execution, or the system must adapt continuously to intermediate findings (Anthropic, 2025a). At the same time, ACHILLES does not implement this mode as a naive conversational loop. Intermediate results are stored under symbolic references (LangChain, 2026b) and can later be reused without repeatedly reinserting all prior outputs into the prompt. This already introduces a more structured and context-efficient execution discipline than the standard pattern in which each step reabsorbs the growing conversational history.

The plan-based mode introduces explicit structure earlier. SOP Lang Plan is intentionally minimal, but still sufficient to represent a sequence of skill invocations with explicit variable dependencies. In simplified form, each line binds an output variable, invokes a skill, and may refer to outputs of previous steps through variable references. This allows the runtime to infer data dependencies and compile the plan into an execution graph (LangChain, 2026a). The result is a more explicit treatment of workflow structure, clearer data flow, more targeted context allocation, and better support for caching, selective recomputation, and auditability (BMAD Method, 2026c). These are precisely the kinds of properties that become increasingly important as agentic workflows move from experimentation toward dependable use in larger systems.

A further relevant point is that the architecture also supports hierarchical orchestration. A top-level session may interpret the user request and delegate part of the work to an orchestration skill, which may itself run an internal loop-based or plan-based session before passing work to downstream execution skills, scripts, or code-based actions. This introduces more structure than mainstream single-loop designs, but it also creates an important advantage: reasoning and context management no longer need to be concentrated in one global prompt. Processing reasoning locally within a skill or session enhances both control and efficiency.

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

This also explains why context optimisation becomes a genuine research topic in this architecture. The issue is not to enforce an unrealistic form of context minimisation. Broader context can still be restored whenever required. The more important question is how far useful context can be reconstructed from explicit artefacts, dependencies, skill descriptions, and intermediate results, instead of being carried forward as an opaque and continuously expanding conversational mass. Early indications are positive and suggest that a substantial part of practically useful context can indeed be recovered in a more disciplined way when decomposition and skill design are done carefully enough.

The same holds for error handling and replanning, which are treated here as first-class parts of the execution model. In this architecture, an error may indicate not only an implementation failure, but also insufficient context, an unmet precondition, an inappropriate skill choice, or a weakness in the current decomposition. For this reason, error information has semantic importance. In loop-based sessions, it feeds into bounded retry and recovery logic (Anthropic, 2026f). In plan-based sessions, failures can be interpreted through the dependency graph (Khattab et al., 2023) and may trigger local repair or the generation of a revised SOP Lang Plan. Replanning is therefore part of the intended architecture and a key aspect of the project's ongoing research.

Overall, the current results already indicate a clear direction. Reusable granular skills, explicit variable dependencies, bounded multi-level sessions, and controlled replanning can significantly improve execution discipline in multi-step agentic workflows (Khattab et al., 2023). Loop-based sessions remain useful for adaptive and context-rich interaction (Anthropic, 2024). Plan-based sessions remain useful for stronger structure, better auditability, lower context overhead, and more localised repair. Their joint use reflects the broader ACHILLES strategy: to retain the flexibility of LLM-based systems while progressively introducing more explicit execution structure wherever this improves efficiency, transparency, trustworthiness, and long-term maintainability (Colelough & Regli, 2025).

This subsection only aims to synthesise the main innovation and its architectural significance. More detailed technical explanations of the session models, the skill abstraction, and the underlying execution mechanisms are provided later in Chapter 6.

4.9 Skills as Natural Language Interpreters

A further innovation emerging from T3.3 concerns the way skills are understood. In many current agent frameworks, a skill is treated mainly as a reusable prompt, a tool wrapper, or a limited orchestration aid within a ReAct-style loop. The work carried out in T3.3 led to a more demanding view. A skill is increasingly understood as a localised interpreter of natural language and task intent, operating under a specific execution discipline. This change is relevant because it affects the architectural role of skills and, with it, the way efficiency, reliability, and maintainability are approached.

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

This conclusion emerged primarily from implementation experience, most clearly in the case of DBTable skills explained in 6.3.9. These skills, focused on offering an abstraction around all database operations, changed our understanding of what is required to build solid enterprise-oriented software in LLM-mediated environments. In both experimental and commercial settings, they showed that robust behaviour does not result simply from better prompting or from attaching more tools to a general agent. What proved necessary was a stronger interpretive layer able to map natural-language requests onto a declarative schema, enforce business constraints, normalise values, preserve consistency, and return results in a controlled form. This led to a broader conclusion: many practically useful skills are better understood as specialised interpreters than as thin prompt-level artefacts.

This observation also clarifies the main difference from mainstream agent design. In the standard ReAct pattern, the central LLM remains the dominant interpreter across the full workflow, while skills or tools mainly support that loop. In T3.3 proposed approach, by contrast, interpretation is distributed into bounded local units. A skill receives a focused natural-language context, derives the relevant parameters and instructions, and then executes under a more disciplined local regime. That regime may involve code, validation logic, symbolic procedures, declarative schemas, database operations, or the orchestration of other skills. The design objective is that the context given to the skill should be sufficiently coherent with the actual needs of that skill, so that interpretation remains local, bounded, and efficient.

A useful way to describe this architecture is as a combination of three layers. First, there is a semantic-intent layer, where the skill interprets the local request and identifies the intended operation. Second, there is a procedural layer, which captures reusable guidance, local rules, and execution constraints for that class of tasks. Third, there is a functional layer, which may include business logic, code, parsing, validation, schema-aware operations, or calls to downstream components. The visible skill description therefore expresses the local entry point, while the subsystem behind it provides the interpretive regime through which that class of work becomes executable. By separating these layers, the architecture already creates the conditions for less generic tools or concrete code-based components to be deployed for the actual execution of tasks, thereby reducing the need for large LLMs that are slow, expensive, and general-purpose, but that still inherently carry a risk of hallucination.

This view also explains why ACHILLES does not treat all skills as instances of one flat abstraction. Different classes of skills correspond to different local interpretive regimes. DBTable skills define one such regime for database interaction under a unified abstraction. Orchestration skills define another, where the role is to coordinate, decompose, and route work between other skills. Code-generating skills are useful where rapid adaptation is more important than prior formalisation, while validated logic skills point toward stricter execution regimes with stronger reliability guarantees. The taxonomy may still evolve, but the main point is already clear: the relevant difference between skills is not only what function they expose, but what kind of interpreter they instantiate.

This perspective also has direct implications for efficiency and user experience. If a skill is treated as a local interpreter whose first role is to extract the right parameters from a focused context,

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

downstream execution can move earlier toward code, symbolic procedures, database primitives, or more deterministic orchestration. This matters because recent thinking-oriented LLMs may introduce substantial latency, and multi-step ReAct loops can accumulate several such calls before producing a usable result. The skill-as-interpreter model therefore supports a practical reduction of latency and cost, while preserving natural-language flexibility where it remains necessary.

At the same time, this approach introduces a clear architectural challenge. It assumes that the main agent or an orchestration layer can provide a downstream skill with context that is sufficiently relevant for its purpose. That judgement may sometimes be incorrect, and this risk should be acknowledged explicitly. For this reason, the ACHILLES approach does not seek to replace mainstream agent architectures in general. Its claim is narrower: for classes of enterprise problems where correctness, consistency, and validation are central, it is often preferable to distribute interpretation into bounded local units rather than leave it concentrated in a single open-ended agentic loop.

This line of work also helps explain why the project gradually opened toward Meta-Rational Pragmatics. Once a skill is understood as a bounded interpreter of executable natural language, it becomes possible to generalise the same idea beyond the current notion of skill. From that perspective, skills are one practical form of a broader architectural insight: natural language can serve not only as conversational input, but also as material for constructing local interpretive regimes with their own execution logic, constraints, and criteria of adequacy.

Although this does not amount to a formal scientific evaluation framework, indirect evidence can still be observed in the way researchers and developers within the team approach the construction of internal agentic systems. In practice, they often prefer to rely on the skills provided by AchillesAgentLib and tend to avoid Anthropic-style skills, or similar mainstream skill formats, even when the latter are easier to integrate. Such approaches may offer a rapid path for prototyping, but they are often more fragile and riskier when systems move closer to production, and they become harder to understand and control as functionality accumulates.

This tendency can be explained both through first-principles reasoning and through practical experience. Given the nature of the problems addressed in the project, many tasks are unique and require custom solutions. In this context, the repeated preference for the more structured ACHILLES approach can be regarded as a useful qualitative indication that the direction being developed is relevant at a more fundamental level, even if its advantages are not always easy to express through simple quantitative metrics.

The broader conclusion is already stable. ACHILLES is moving away from a prompt-centric view of skills and toward a layered architecture of specialised interpreters. The motivation for this shift came first from concrete implementation work, especially with DBTable skills, where the limits of thin prompt-based approaches became visible in practice. From there, the project arrived at a more general insight: many reliable AI capabilities are better implemented as bounded interpretive subsystems that transform focused natural-language input into disciplined execution. This is one of the most important

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

innovations emerging from the project so far and one of the clearest conceptual bridges toward the broader neuro-symbolic direction later articulated as Meta-Rational Pragmatics.

4.10 Symbolic Approach for Generative Use Cases

A further important innovation related to Task T3.3 concerns the way AI agents can be constructed for generative use cases. This subsection introduces the direction through which ACHILLES seeks to make generative workflows more controllable, more reusable, and easier to validate. The central idea is not to replace language models, but to reduce the extent to which generation depends on unconstrained prompt expansion alone. Instead, the objective is to treat generative work as specification-driven assembly under explicit constraints, intermediate representations, validation steps, and regime-sensitive execution. While skills and MRP-VM are treated more fully in the following chapters, they are already relevant here as the main architectural mechanisms through which this direction becomes operational: skills as the current bounded execution units, and MRP-VM as the emerging generalised runtime for governed interpretation, routing, and validation. This is particularly relevant for use cases such as SCRIPTA, where high-quality outputs are rarely obtained in a single undirected pass and where longer artefacts require iterative construction, comparison, revision, and consistency control.

4.10.1 Structured generation as specification-driven assembly

Recent work on structured language-model programming already indicates a shift away from prompt-centric engineering toward explicit intermediate representations, constrained generation, and compilable pipelines. LMQL treats prompting as a programmable and constrained process rather than a raw string operation (Beurer-Kellner et al., 2023). DSPy frames language-model applications as declarative pipelines that can be compiled and optimised against evaluation metrics (Khattab et al., 2023). Guidance (Microsoft Research, 2023) and Outlines (Outlines Documentation, 2026) likewise show that generation can be made more controllable by combining language models with typed constraints, control flow, and grammar-level structure. These directions are directly relevant to ACHILLES. At the same time, they do not by themselves provide a general runtime model for governing heterogeneous execution regimes, including prompts, deterministic code, retrieval, symbolic validators, constrained recombination, and evidence-producing checks, within one coherent operational framework. The proposal developed here addresses that wider problem in a specification-driven setting. The motivating observation, especially visible in the SCRIPTA use case, is that generation becomes difficult to control, reproduce, and align with explicit objectives when it is treated as unconstrained free-form synthesis. Valuable generative artefacts in literature, documentation, and research are rarely produced in a single undirected pass. They typically emerge through iterative assembly, comparison of alternatives, application of constraints, quality assessment, and revision. This supports a stricter operational view in which generation is treated not as one-shot prompt expansion, but as explicit assembly under a governed judgment regime.

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

In this view, the primary unit of generative capability is not the monolithic prompt, but a library of composable components and transformations. Such components may encode literary primitives such as characters, locations, dialogue patterns, scene templates, arcs, pacing controls, rhetorical devices, and chapter-level orchestration strategies. They may equally encode technical and scientific primitives such as specification patterns, experiment templates, evaluation procedures, hypothesis families, parameter schedules, reporting structures, and documentation scaffolds. Each component should be represented in an intermediate form as a typed object with declared inputs, outputs, preconditions, effects, and provenance where relevant. Even when the final realisation is textual, the component is treated as a structured unit rather than as an opaque fragment of prompt text.

The relation with specification-driven development is direct. In such a workflow, the central object is not the prompt but the intermediate specification from which prompts, validations, pipelines, and execution plans can be derived. In the present architecture, that intermediate specification can be written in SOP Lang IR or projected into a controlled natural language surface when a more readable representation is needed for LLM interaction. The IR is the stronger form because it preserves typed commitments, structural relations, constraints, provenance, admissibility conditions, and links to validation or execution hooks. The CNL (Controlled Natural Language) projection is the more operational surface for model conditioning, iterative revision, and human inspection. In both cases, the design principle remains the same: generation and orchestration should be specification-first rather than prompt-first.

A central consequence of this approach is that generation can be treated explicitly as search in a design space. Components define the constructive vocabulary. Constraints define which assemblies are admissible. Value measures define which outcomes are preferred. What differs across use cases is therefore not the existence of structure, but the active admissibility regime. In creative regimes, the system may intentionally permit wider degrees of freedom, including controlled stochasticity, because novelty, surprise, and stylistic variation are part of the objective. In correctness-critical regimes, admissibility is anchored in verifiable constraints, and variation is limited to choices that remain traceably compliant with requirements. This distinction allows one architecture to support both literary generation and more formal technical or scientific synthesis without conflating the standards of judgment that govern them.

The same point clarifies how creativity is preserved. Symbolic structure does not eliminate creativity; it relocates it into structured recombination over composable parts. Novel outcomes can be produced through admissible recombination of scenes, motifs, dialogue patterns, argument structures, experiment templates, or documentation fragments, provided that the resulting assembly remains coherent under the active regime. In practical terms, this means that novelty, coherence, stylistic distance, structural complexity, cost, coverage, and related qualities can be treated not merely as informal preferences, but as explicit optimisation or acceptance criteria. One may specify, for example, that a narrative should maximise novelty subject to continuity constraints, or that a research workflow should prioritise candidates by expected information gain under bounded cost and risk. Such

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

measures become first-class objects in the intermediate representation and can guide both generation and evaluation.

4.10.2 Pipelines, runtime control, and the transition from skills to MRP-VM

For ACHILLES IDE, this implies that generative work should increasingly be expressible as SOP Lang Pipelines, that is, as explicit dependency graphs whose nodes correspond to component selection, local generation, retrieval, validation, transformation, recombination, and quality assessment **(D6.1).** A chapter generator, for example, may be represented as a pipeline that first selects world constraints and character states, then retrieves relevant motifs, generates candidate scene plans, validates continuity, applies editorial constraints, and only then renders final text. A report-generation pipeline may instead assemble template fragments, generate candidate formulations, validate structure and compliance conditions, and emit a traceable artefact. The advantage is not only improved control. Intermediate artefacts become explicit, reusable, comparable, and selectively recomputable.

Within this pipeline perspective, MRP-VM provides the more general runtime logic that governs how such pipelines are interpreted and executed. The control idea remains the same as in the preliminary MRP-VM specification: selective knowledge-base pull, condensation of the current situation into a smaller control nucleus, construction of bounded frames, explicit routing to candidate regimes, local execution, and reintegration into a shared operational form. When embedded under ACHILLES IDE, SOP Lang Pipeline provides the explicit build-like graph structure, while MRP-VM provides the regime-governed execution semantics for nodes whose realisation may differ substantially. Some nodes may remain prompt-oriented. Some may be code-based. Some may invoke retrieval subsystems, symbolic validators, or composition checks. The architectural gain is that one pipeline can orchestrate all of them without collapsing their judgments into a single opaque LLM call.

This also clarifies the implementation path. In the short term, the same general approach can be realised with more classical skills. Skills for literary consistency checking, location continuity validation, editorial policy enforcement, research-template conformance, structural decomposition, citation checking, or quality scoring can already be implemented as callable units and composed into SOP Lang Pipelines. That is feasible immediately and consistent with current ACHILLES engineering practice (D6.1). In intermediate versions, however, those same skills should increasingly be treated as plugins or local operators inside MRP-VM rather than as isolated prompt wrappers. MRP-VM may still rely internally on LLMs for condensation, partial frame construction, or bounded interpretation where needed, but the control model already improves because routing, admissibility conditions, evidence obligations, and reintegration are made explicit. The longer-term objective is that MRP-VM should replace LLM-based judgment wherever stronger symbolic or typed mechanisms are available, while continuing to use LLMs where they remain pragmatically useful as proposers, renderers, or bounded interpreters.

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

The practical meaning of this transition is that current skills can be compiled into a meta-rational execution environment. A skill is no longer merely a callable tool. It becomes a local execution regime with declared assumptions, admissibility conditions, expected evidence, and reintegrable outputs. A literary-quality skill, a standards-alignment validator, a style profiler, a character-consistency checker, or a citation-structure checker may all remain distinct modules, but MRP-VM can determine when each applies, what bounded frame it receives, what supporting artefacts it may inspect, what evidence it must return, and how its judgment should be combined with others. In this sense, skills become orchestrated meta-rationally rather than merely chained procedurally.

4.10.3 Validation, symbolic judgment, and layered execution

At an abstract level, such validation programs are already straightforward to characterise. A literary-quality program may operate over a structured representation of chapters, scenes, entities, transitions, and rhetorical patterns. A character-consistency program may represent characters as typed profiles containing persistent traits, motivations, relation constraints, speech tendencies, and permitted development paths. A location-consistency program may encode spatial relations, environmental facts, object persistence, and transition rules. More generally, generated content can be checked against explicit representations of narrative constraints, fictional world mechanics, editorial requirements, documentation standards, reproducibility criteria, or compliance obligations. The purpose is not to reduce generation to rigid logic, but to make important forms of admissibility explicitly testable, especially in longer artefacts where failures accumulate gradually and are poorly handled by unaided prompt-based memory.

The role of the LLM in this setting is best understood as that of a high-capacity proposer operating within a symbolic execution environment rather than above it. The model may propose candidate components, parameterisations, assemblies, formulations, or local realisations, but it does not define the validity of the result. Validity is determined by the active pragmatic regime, namely the current set of constraints, evaluators, evidence obligations, and admissibility tests. When strictness is required, symbolic backends such as Satisfiability Modulo Theories (SMT) (de Moura & Bjørner, 2008) or Datalog can validate structural properties and consistency. When the regime is softer, scoring models and heuristic evaluators can rank alternatives while preserving traceability, since each candidate remains a named assembly with explicit lineage. This establishes a practical boundary between proposal and judgment. LLMs remain useful for candidate generation, reformulation, rendering, bounded interpretation, and soft synthesis, but evaluative reasoning and admissibility decisions can be progressively relocated into more inspectable mechanisms.

This also clarifies the role of controlled randomness. Rather than injecting randomness directly into unrestricted text generation, randomness can be applied to the selection and parameterisation of components within a bounded search space. A literary generator may sample among dialogue archetypes, scene templates, arc transitions, or motifs while still enforcing character consistency, timeline coherence, and stylistic boundaries. The resulting diversity is therefore variation over different valid assemblies rather than uncontrolled drift. Vector-symbolic retrieval or similar memory

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

mechanisms can further support this process by retrieving a small set of relevant components, motifs, or constraints for the current context, enabling variation that remains contextually grounded.

The same infrastructure supports generative work in research and engineering, where the objective is not surprise but disciplined exploration. In such regimes, the symbolic approach enables agents to execute complex and potentially expensive workflows in a controlled sequence, guided by explicit heuristics and value measures. Instead of a single agent improvising an ad hoc plan, the system can maintain an explicit experiment graph in the IR, where nodes represent candidate experiments, hypotheses, ablations, decision points, and evaluation steps. The orchestrator can then prioritise them according to expected information gain, cost, risk, coverage of the hypothesis space, or other declared criteria, while recording evidence and outcomes in a form that reduces omission, drift, and retrospective bias. The symbolic layer is important here because it preserves memory of what was tried, what assumptions were made, what results were obtained, and which parts of the design space remain underexplored.

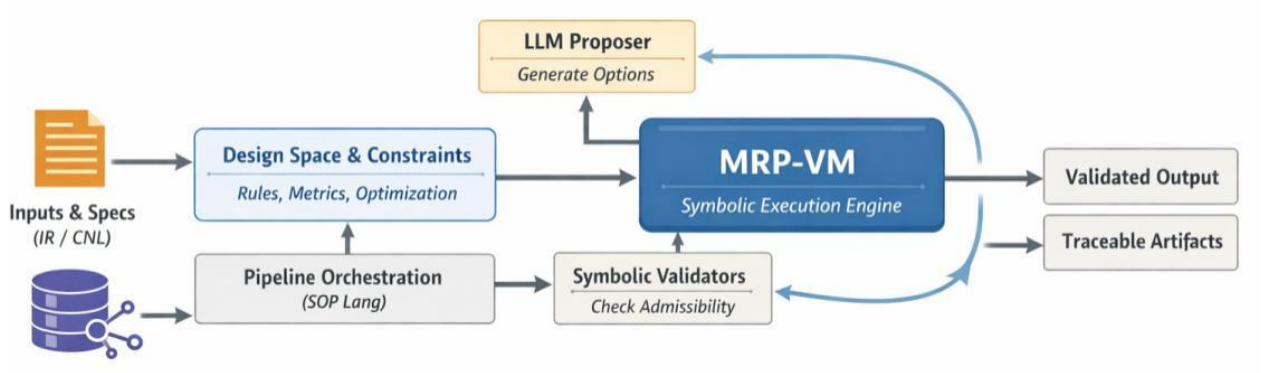

Figure 4 LLM Eval Suite Overview

This also clarifies the relation among SOP Lang IR, SOP Lang Plan, SOP Lang Pipeline, and MRP-VM. SOP Lang IR provides the common representational substrate in which components, constraints, provenance, routes, candidate assemblies, and results can coexist. A CNL projection provides an LLM-usable surface without discarding structure. SOP Lang Plan can express stepwise interactive execution where bounded local tasks must be coordinated. SOP Lang Pipeline can express heavier build-like generative or analytical workflows with explicit dependencies, promoted artefacts, and selective recomputation (D6.1) MRP-VM then acts as the runtime that determines how a given local task should be executed and validated. The layers are therefore complementary rather than redundant: IR provides the representation, Plan provides a compact substrate for interactive coordination, Pipeline provides the graph of work, and MRP-VM provides the regime-governed execution semantics.

It is also useful to state explicitly that the same intermediate representation supports several functions at once. It can act as a planning substrate for interactive agents, where the CNL projection serves as an executable specification for stepwise work (Darif et al. 2026). It can act as a build or deployment substrate aligned with the pipeline model in ACHILLES IDE, where dependencies, checks,

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

and promoted artefacts are compiled into reproducible execution graphs. It can also act as an analysis substrate, where the IR becomes a compact and auditable object for consistency verification, bias and quality analysis, risk surfacing, and evaluation planning. In this sense, the IR is not only a target for solvers. It is a unifying substrate for specification-driven development across interactive episodes, heavy pipelines, and reasoning-intensive analysis.

For ACHILLES IDE, this points toward a family of generative pipelines that are richer than prompt chains. A literary pipeline can combine world retrieval, character-state validation, scene-template selection, arc recombination, local rendering, and post-generation consistency checks. A technical-writing pipeline can combine schema retrieval, deterministic structure generation, terminology normalisation, standards validation, citation checks, and final stylistic revision. A research pipeline can maintain an experiment graph, generate candidate hypotheses, select next steps by explicit heuristics, validate documentation obligations, and record outcomes in auditable intermediate artefacts. In all these cases, MRP-VM can serve as a partial replacement for unconstrained LLM orchestration by moving more of the reasoning burden into symbolic retrieval, typed procedures, validation logic, explicit value measures, and regime-sensitive routing.

The result is a more realistic path toward trustworthy generative systems. The objective is neither to eliminate LLMs nor to force all generative work into fully formal symbolic systems. The objective is to build a layered architecture in which specifications, intermediate representations, dependency graphs, explicit evaluators, and regime-aware runtimes reduce the amount of reasoning left to opaque prompt dynamics. This is compatible with current ACHILLES use cases, compatible with present skill-based implementations, and compatible with an incremental transition toward intermediate versions of MRP-VM. It also aligns naturally with specification-driven development, because the same intermediate specification can function simultaneously as a generation plan, a validation object, a pipeline substrate, and a reasoning artefact. That unification is the main architectural advantage.

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

5 INFRASTRUCTURE AND IMPLEMENTATION

5.1 Main Contributions, Implementation Status

Task T3.3 has produced a set of technical results that should be regarded as contributions, not only as supporting elements for later integration. The main outcome is a coherent runtime and deployment environment for AI agents, structured around three closely connected components: Ploinky as the deployment and interoperability substrate, the LLM Proxy as the governance and observability layer for model access, and AchillesAgentLib as the runtime library through which skills, sessions, orchestration logic, and specialised execution regimes become operational. Together, these developments form the technical core on which ACHILLES IDE, the ACHILLES CLI copilot, and related use-case integrations are being built.

Ploinky is a contribution because it provides a unified deployment model for heterogeneous agentic components, including packaging, configuration, isolation, routing, and MCP-based interoperability. The LLM Proxy is equally important as a reusable infrastructure layer that turns model access into a controlled service with authentication, routing, observability, budget supervision, retry governance, and audit logging. AchillesAgentLib is the main runtime contribution, as it implements controlled LLM mediation, recursive routing, agentic sessions, reusable skill descriptors, and several specialised skill families. A particularly important result is the move toward a more structured view of skills, especially through components such as DBTable skills, which provide a unified abstraction for database operations and illustrate how natural-language interaction can be combined with declarative schemas, validation logic, and disciplined execution.

The session model and the evaluation layer should also be considered part of the contribution. The distinction between loop-based and plan-oriented sessions, together with bounded replanning and reusable skills, defines an execution methodology that goes beyond isolated software components. In parallel, the evaluation mechanisms developed in T3.3 provide a structured way to assess probabilistic agentic behaviour under controlled scenarios.

Most of the developments described in this chapter are already at an advanced stage of implementation, integration, and internal evaluation. They are not purely conceptual proposals, but concrete technical artefacts already used within the current ACHILLES environment. At the same time, further refinements and architectural adjustments remain likely as integration progresses in ACHILLES IDE, in the ACHILLES CLI copilot, and in the project use cases. Technical documentation for the developments produced in T3.3, as well as for the broader ACHILLES IDE environment, is publicly available at **www.assistos.org**.

5.2 Ploinky

5.2.1 Ploinky as the deployment substrate for agents and ACHILLES IDE

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

Ploinky (Ploinky, 2026c) is the deployment and interoperability substrate of the ACHILLES environment. It is the layer through which ACHILLES IDE, the ACHILLES copilot, and compatible agents become executable in a governed, reproducible, and inspectable manner. Its role is therefore broader than process launching. It standardises how agents are packaged, configured, isolated, exposed, and connected across local workspaces, CI, and more controlled environments. Publicly, Ploinky presents itself as a secure AI agent deployment and development platform based on isolated Linux containers, repository-managed agents, web interfaces, and simple declarative configuration. Its architecture is explicitly organised around clear separation of concerns, workspace-local state, runtime neutrality across Docker and Podman, and a routing layer for exposing agent services.

In the diagram below, Ploinky Deployment Platform is presented as the foundational layer for deploying and managing agents within the ACHILLES ecosystem. It highlights how multiple environments—such as IDEs, copilots, custom agents, and CI/production systems—interact through a unified platform that provides isolated containers, per-project workspaces, and standardised interfaces. This approach ensures secure, reproducible, and scalable execution of agent-based systems across heterogeneous environments.

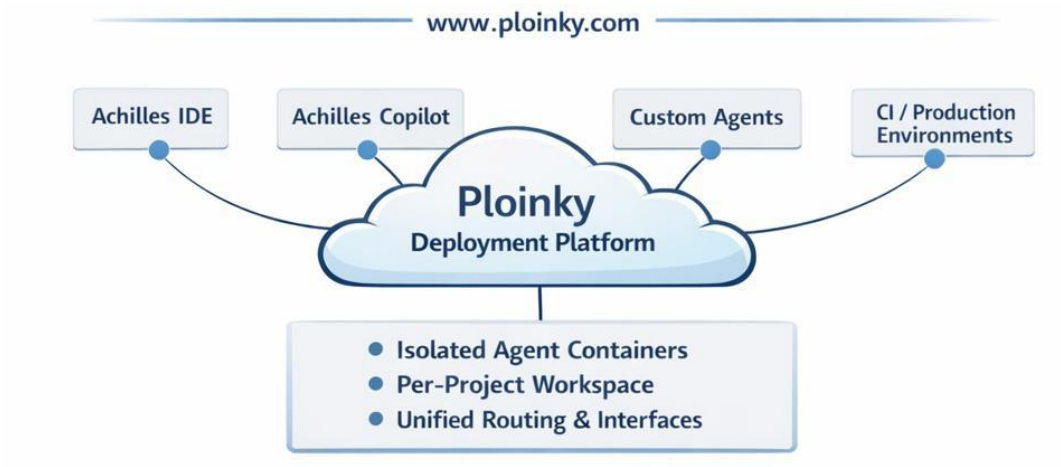

Figure 5 Ploinky Deployment Platform within the ACHILLES Architecture

Within ACHILLES, Ploinky exists because useful agent systems require more than capable models. They also require disciplined packaging, explicit security boundaries, stable operational contracts, and a controlled way to expose capabilities both to humans and to other agents. In this sense, Ploinky plays the role that Kubernetes or Docker Compose play only partially. Like those systems, it standardises deployment and service exposure; unlike them, it is designed specifically around the lifecycle of agents and skills, around workspace-local agent repositories, and around MCP-oriented interoperability. It is therefore better understood as a lightweight, agent-oriented deployment model than as a generic container orchestrator The present document focuses on the qualitative and architectural role of Ploinky. More detailed implementation-level material, including low-level

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

command behavior and internal architecture, is better consulted in the public Ploinky documentation (Ploinky, 2026a) and CLI reference (Ploinky, 2026b).

5.2.2 Standardised packaging, isolation, and the notion of the Ploinky Agent

A major contribution of Ploinky is that it turns heterogeneous services into uniformly deployable operational units. This is captured by the notion of the **Ploinky Agent**. At a practical level, a Ploinky Agent is not defined by one internal framework or one implementation language. It is defined by how it is packaged and exposed. Any service that is encapsulated inside the Ploinky workspace model and exposed through the expected MCP-facing operational contract can become a Ploinky Agent, regardless of the technology with which it was originally built. This is strategically important for ACHILLES because the environment contains components with different technical origins and different operational roles (Ploinky, 2026a).

Figure 5 presents the architectural structure of a Ploinky Agent within the "Isolation & Governance" framework. It is organised into three sequential sections. On the left, the *Agent Container Isolation* block shows an agent encapsulated together with its code and tools inside a container environment (Docker/Podman), emphasising execution isolation. In the centre, the *Local Workspace Scope* illustrates the internal .ploinky directory, which contains logs, secrets, and configuration files, defining the agent's working context. On the right, the *Capability Control* section depicts mechanisms such as resource adapters, secret management, and web access tokens, highlighting how permissions and policies regulate the agent's interactions. Together, these elements depict a controlled and modular setup that ensures secure deployment and operation.

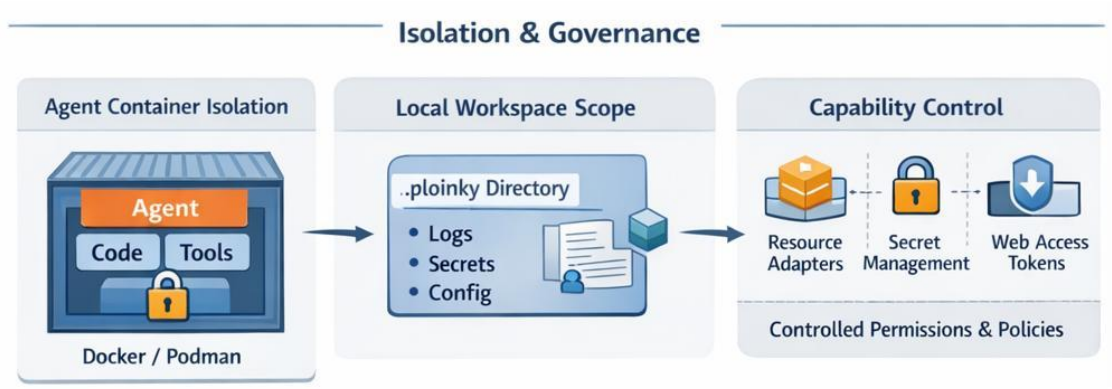

Figure 6 Isolation and Governance Model of Ploinky Agents

This means that a backend for ACHILLES IDE, an authentication and login component built around Keycloak, a Git- or GitHub-facing service, or the DSU Agent described elsewhere in the project can all be treated as Ploinky Agents even if they are not implemented entirely with the same internal libraries. Keycloak is a useful example because it is publicly positioned as an open-source identity and access management system for adding authentication, authorisation, and single sign-on to

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

applications and services (Keycloak, 2026). In ACHILLES terms, such a component need not be rewritten into one proprietary form. It can instead be encapsulated and exposed in a standard way inside the Ploinky environment.

This standardisation is reinforced by workspace discipline and declarative packaging. Publicly, Ploinky emphasises that all configuration remains local to the project directory, with zero global state, and that agents are organised in repositories that can be enabled, disabled, updated, and managed through the CLI. In practice, this gives the system a local, inspectable memory of which repositories are active, which agents are enabled, how they are routed, and which secrets or environment assumptions they depend on. The result is a deployment model that is far easier to reproduce and govern than one based on ad hoc host-side scripts and ambient configuration.

Isolation is equally central. The architecture documents state that every agent runs in its own container, with Ploinky supporting both Docker and Podman transparently (Ploinky, 2026a). In the broader ACHILLES implementation, this strong default posture is complemented by lighter forms of sandboxing for faster iteration. The important point for the deliverable is that Ploinky supports **more than one isolation tier** while keeping the same overall deployment discipline. Full containment is used where stronger separation is required, while lighter modes can support development and rapid experimentation. In all cases, access to code, repositories, secrets, build tools, and external interfaces is mediated explicitly rather than inherited implicitly from the host.

5.2.3 MCP as the interoperability contract

The main reason Ploinky supports true multi-agent environments rather than just multiple containers is that it standardises interoperability through MCP. The public architecture explicitly includes an **Agent MCP Bridge** and describes a routing model in which the router communicates with agent containers through MCP-facing endpoints and can aggregate agent capabilities such as tools and resources under one operational layer. This is crucial in ACHILLES because it means that interoperability is not left to custom pairwise integrations. It is mediated through a common operational contract.

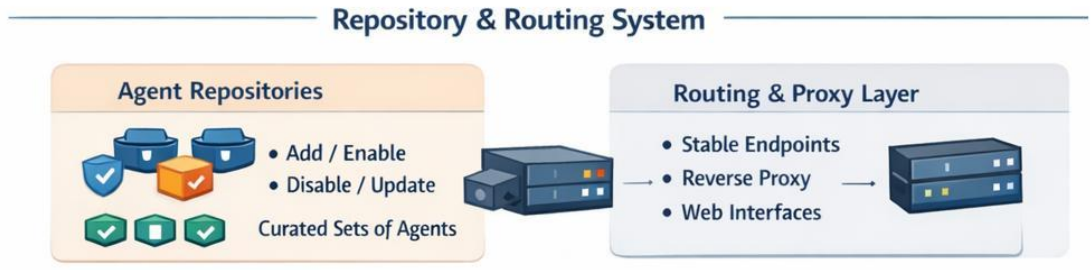

Figure 7 MCP as the Interoperability Contract

The diagram illustrates the interoperability mechanism within the Ploinky architecture based on MCP. It shows how the *Agent MCP Bridge* enables structured communication between agent

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

containers through standardised endpoints, allowing capabilities such as tools and resources to be unified under a common operational layer. In the lower section, the *Repository & Routing System* is presented, consisting of *Agent Repositories* for managing curated agent sets and a *Routing & Proxy Layer* that provides stable endpoints, reverse proxy services, and web interfaces. The overall composition highlights a consistent and scalable approach to integrating and coordinating multiple agents through a shared contract.

This becomes especially valuable when new capabilities are introduced through **declarative MCP tools**. In ACHILLES, a tool is not necessarily hardcoded into the core runtime. Instead, it can be defined structurally and then exposed through the same MCP-oriented orchestration layer used elsewhere. The public architecture shows that agents commit an mcp-config.json file alongside their source code and that tools, resources, and prompts are loaded from that file when the agent starts. The effect is that extensibility becomes a normal architectural property rather than a recurring engineering exception.

A simplified example of such a configuration is the following:

```
{
"name": "summarise_repo",
"description": "Generates a structured summary of the repository",
"command": "./scripts/summarise_repo.sh",
"cwd": "workspace",
"env": { "MODE": "fast" },
"timeoutMs": 60000,
"async": false,
"inputSchema": {
   "path": { "type": "string", "minLength": 1 },
   "depth": { "type": "number", "min": 1, "max": 5, "optional": true }
}
}
```

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

The important point here is not the specific shell command, but the structure of the contract. The tool has a stable identity, a declared purpose, an execution command, an execution context, a timeout policy, an optional asynchronous mode, and a typed input schema. This gives the environment a clear distinction between **implementation** and **orchestration**. The internal logic of the tool may vary freely, but the invocation model remains stable. That stability is what makes tools portable across execution surfaces.

This portability matters in practice. The CLI reference documents commands such as client tool, client list tools, and client list resources, indicating that the CLI does not need prior knowledge of the internal logic of each capability. It addresses tools through the MCP-facing interface aggregated by the router (Ploinky, 2026b). Containerised agents can do the same. As a result, the same configured capability can be invoked from the CLI, resolved through the routing layer, and executed inside an agent container without changing its conceptual contract. The tool is therefore portable not only as a process, but as a protocol-governed capability.

The same logic also extends to asynchronous execution. When a tool cannot return a final result immediately, the interaction does not need to collapse into failure or block indefinitely. Instead, task state can be returned first and the final outcome later retrieved through a polling pattern. This keeps asynchronous execution compatible with the same MCP-oriented operational model that governs synchronous invocation.

5.2.4 Ploinky as the Common Operational Contract

Within ACHILLES, Ploinky functions as the common operational contract for deploying and operating heterogeneous agentic components. Its role is to standardise how agents and related services are packaged, configured, isolated, exposed, and interconnected, so that execution remains reproducible and governable across local development, CI, and more controlled deployment settings. In practical terms, this common contract combines workspace-local state, manifest-driven deployment, repository-based lifecycle management, unified routing, mediated access to sensitive resources, and support for multiple isolation tiers (Ploinky, 2026a). An overview of the configuration and control mechanisms for Ploinky Agents is presented Figure 7. The *Manifest.json* defines the technical aspects of an agent, including the container image, installation commands, CLI/API entry points, and environment variables. In parallel, the *Capability Profile* establishes operational constraints such as filesystem access, network controls, and secret limits. These components work together to enable portable and policy-governed execution across environments, illustrated by the transition from development (DEV) to production (PROD). The figure highlights how standardisation ensures consistent behaviour, security, and manageability.

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

Figure 8 Standardised Manifest and Capability Profiles

This standardisation is important because the ACHILLES environment contains components with different technical origins and different operational roles. A **Ploinky Agent** is therefore not defined by one implementation language or one internal framework, but by the fact that it is encapsulated and exposed in a form compatible with the deployment and interoperability rules of the platform. Under this model, components such as the backend of ACHILLES IDE, identity and access services based on Keycloak (Keycloak, 2026), Git- or GitHub-facing services, and specialised components such as the DSU Agent (D2.3). can coexist within the same managed environment even when they are not implemented with the same internal stack. What unifies them is a common discipline of packaging, isolation, exposure, and operational control.

Ploinky also provides a standardised basis for extensibility and interoperability. Through MCP, capabilities can be exposed and invoked under a shared contract, and new tools can be introduced declaratively through configuration rather than through invasive modification of the runtime. This makes it possible for the same capability to be used coherently from the CLI, from containerised agents, or from other services, while preserving the same operational identity, input schema, and governance model. The same routing and control layer also supports human-facing services such as WebChat, Dashboard, and terminal-oriented interfaces, which allows inspection, debugging, and human-in-the-loop interaction to remain inside the managed operational environment. From the perspective of the ACHILLES architecture, Ploinky provides the operational basis on which agentic capabilities can be treated as deployable and interoperable infrastructure rather than as isolated local integrations. This is relevant not only for current components, but also for the longer-term evolution of ACHILLES IDE toward more persistent, better controlled, and more maintainable AI-assisted workflows.

5.3 LLM Proxy Component

5.3.1 The proxy as the observability and governance layer for model access

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

The LLM Proxy Component, internally named **Soul Gateway**, is the centralised gateway through which agents in the ACHILLES environment access large language model APIs. Its primary role is to decouple agent logic from individual providers and to introduce a common operational layer for authentication, rate limiting, budget supervision, content filtering, retry control, and audit logging. In practice, this turns model access from a fragmented implementation concern into a governable infrastructure service. Annex A includes screenshots illustrating the current implementation of this component at the time of writing this deliverable. We do not describe it here in further detail. More technical documentation concerning installation and use is planned to be published on https://www.assistos.org in the coming period.

In this context, **observability** refers to the capacity to understand system behaviour through emitted telemetry, especially logs, metrics, and traces (OpenTelemetry, 2026). In the ACHILLES setting, this means being able to inspect which agent issued a request, through which session, against which resolved model, with what latency, with what token usage, at what cost, and with what outcome. This is essential in a multi-agent environment. Operational failures are often not caused by the abstract capability of the model itself, but by hidden retry loops, unstable orchestration, weak provider governance, or missing audit trails. A centralised proxy therefore provides a common point from which the behaviour of the agent ecosystem can be monitored, analysed, and corrected. Recent mainstream agent tooling has moved in the same direction, treating tracing, guardrails, and monitoring as first-class elements of production-grade agent systems (Anthropic, 2026f).

Figure 9 LLM Proxy – Soul Gateway

The diagram shows a central proxy connecting agents to LLM providers such as OpenAI, Anthropic, and Google. It provides operational control including authentication, rate limiting, cost monitoring, and audit logging. It also ensures full observability and helps prevent errors or excessive resource usage. Dozens of providers are already supported, and the architecture is designed to allow the addition of middleware layers for virtually any further provider, while maintaining a unified interface. This enables AchillesAgentLib to operate consistently across a wide range of open-source and

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

commercial providers. Models that can be executed locally are also regarded as strategically important.

This layer is also important as protection against ordinary programming and orchestration errors. A badly designed loop, a repeated retry pattern, or a misconfigured agent session may otherwise generate very large volumes of inference calls and consume significant cloud-model budgets in a short time. Soul Gateway reduces this risk by enforcing request limits, concurrency limits, budget thresholds, and loop-detection policies before requests reach upstream providers. At the same time, it gives the project a way to inspect what agents are doing and to reconstruct their behaviour after the fact, which is valuable both for debugging and for reverse engineering unstable workflows.

A further architectural advantage is the separation between **agent behaviour** and **model configuration**. Without an intermediary layer, each agent would need to handle its own provider credentials, retry logic, model identifiers, and cost assumptions. Soul Gateway replaces this with a single OpenAI-compatible access surface, regardless of whether the upstream provider is OpenAI, Anthropic, Google, OpenRouter, Hugging Face, or another compatible service. This allows agents to target logical model capabilities rather than provider-specific details. It also creates the conditions for model abstraction, provider switching, fallback chains, high-availability configurations, and later synthetic model tiers that may implement internal logic such as context compression or provider-aware routing without exposing this complexity directly to every agent.

5.3.2 Architecture, functional scope, and future direction

At implementation level, Soul Gateway is a Node.js HTTP service backed by PostgreSQL for persistent state and deployed as a Ploinky Agent inside a containerised workspace. Its architecture is organised as an ordered processing pipeline. Every inference request passes through a deterministic sequence of stages that may validate, enrich, block, route, transform, or log the request before it reaches an upstream provider. The current ACHILLES implementation includes stages for authentication, request validation, agent and session identification, loop detection, content-policy checks, rate limiting, budget verification, model routing, concurrency management, upstream dispatch, response handling, cost calculation, response validation, and structured logging. The architectural point is not the exact count of stages, but the fact that all requests are subjected to the same ordered control model, independently of which agent or internal code path initiated them.

The component exposes several endpoint families. The primary inference interface is OpenAI-compatible and supports both streaming and non-streaming interaction. Additional passthrough interfaces preserve compatibility with provider-specific request formats where needed. A model-discovery endpoint exposes configured models and logical tiers. A management API supports administration of API keys, model configurations, tier definitions, blacklist rules, and audit log access. In parallel, real-time log streaming and a dashboard provide continuous visibility into cost, latency, failures, retries, and session evolution. This combination is important because it makes the proxy simultaneously an execution gateway, a control point, and an observability surface.

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

A particularly important capability is **logical model routing**. Requests may be resolved not only by exact model name, but also through abstract tiers such as "fast, deep, plan, code-generation, search" (check the screenshots in Annex A) or otherwise policy-defined model classes, together with fallback chains when the preferred model or provider is unavailable. This allows the environment to rebalance traffic, replace providers, introduce new models, and define high-availability strategies without requiring changes in the calling agents. The same layer also centralises API key handling, request-per-minute limits, token budgets, and model-level concurrency control. This is useful both for cost containment and for fair sharing across agents.

The component also includes explicit safeguards against unstable behaviour. It tracks repeated request patterns and request frequency at session level, applies bounded exponential backoff for retriable upstream errors, distinguishes retriable from non-retriable failures, and preserves structured retry history in the audit trail. For non-streaming requests, it can also exploit response caching for repeated calls with the same effective input and resolved model. Agent and session tracking further improve operational visibility by attributing usage, cost, and failure patterns to concrete agent families or workflows without requiring each agent to implement its own monitoring stack.

The persistence layer records structured call logs, including request context, resolved model, response metadata, latency, time-to-first-byte, token usage, retry history, and cost breakdown. This gives the project a durable basis for performance analysis, budget attribution, and evaluation of orchestration patterns. It also provides the infrastructure needed to compare agents, identify unstable workflows, and support later empirical optimisation of prompts, sessions, and skill design.

From the perspective of the broader ACHILLES architecture, Soul Gateway is therefore the layer that turns model access into a monitored, controllable, and extensible service. It creates the conditions for provider independence, cost control, debugging, and system-level analysis. It is also the natural insertion point for future controls that should remain external to individual agents, including stronger guardrails, centralised filtering, recommendation logic, context compression, and later forms of neuro-symbolic pre-processing or post-processing. Keeping such controls at proxy level is architecturally useful because it allows them to be applied consistently across the agent ecosystem rather than reimplemented separately by each agent.

5.4 AchillesAgentLib

The previous sections introduced the main architectural principles of the ACHILLES multi-agent environment at a conceptual level. The present section moves one layer lower and explains how these ideas are implemented concretely in the software stack. In the public AssistOS toolchain, ACHILLES IDE is presented as the development environment, AchillesAgentLib as the cognitive architecture layer, AGISystem2 as the reasoning and validation layer, Ploinky as the deployment substrate, and ACHILLES CLI as the automation surface (AssistOS, 2026a). Within that broader toolchain, AchillesAgentLib is the component that implements, in Node.js, the main runtime techniques discussed so far: controlled LLM mediation, recursive skill-based execution, agentic sessions, reusable skill descriptors, and the

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

separation between orchestration and family-specific execution backends (ACHILLES Agent Library, 2026a).

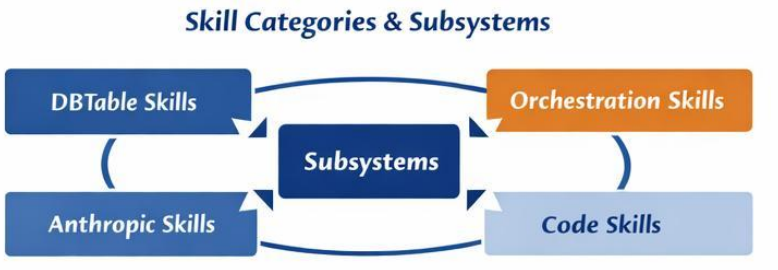

Figure 10 AchillesAgentLib – Main Skills & Subsystems

The diagram shows how AchillesAgentLib organises agent capabilities into skill categories and connects them through subsystems. It highlights orchestration, code, DBTable, and Anthropic-related skills as modular components within the runtime. This structure enables scalable execution, session management, and coordination between different agent functions.

This point is important for the role of the section inside the deliverable. We are no longer only describing a general research direction. We are describing the software layer through which that direction becomes operational. AchillesAgentLib is therefore not merely a utility library for calling models. It is the main runtime through which ACHILLES agents discover skills, interpret requests, construct local execution structures, manage session state, and delegate work into specialised subsystems. In this sense, it is the technical bridge between the higher-level architectural rationale of the present deliverable and the concrete agentic behaviour exposed in ACHILLES IDE and the ACHILLES copilot.

At the same time, the present section should still be read at the right level of abstraction. Its purpose is to explain the runtime model and the main software mechanisms, not to replace the lower-level technical documentation. For the closest public and evolving description of the main components, readers may consult the AssistOS (AssistOS, 2026a). toolchain page and the AchillesAgentLib documentation map, which describe the public role of the library and the major runtime and skill guides (ACHILLES Agent Library, 2026a). Where the public documentation and the present deliverable differ in detail, the present text reflects the current ACHILLES implementation more faithfully.

A useful comparison can be made here with Anthropic's recent Agent Skills model (Anthropic, 2026e) presents skills as modular capabilities packaged through SKILL.md (Anthropic, 2025b), metadata, and optional supporting resources, with the stated purpose of turning general-purpose agents into more specialised agents that can automatically load reusable procedural guidance when relevant. AchillesAgentLib converges with this general direction, but it does not stop there. In our architecture, Anthropic-style skills are only one skill family among several. More broadly, we treat each

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

skill family as a distinct local interpretive regime, with its own descriptor surface, execution assumptions, and operational discipline. This stronger interpretation-oriented view of skills is central to the software architecture and will later connect directly to the more ambitious direction developed in the following chapter.

5.4.1 LLMAgent Class

LLMAgent is the foundational LLM mediation component of AchillesAgentLib (ACHILLES Agent Library, 2026a). Conceptually, it sits between higher-level agent orchestration and the underlying model providers. Its role is to turn raw model access into a controlled runtime service that can be reused consistently across the rest of the system. Public AgentLib documentation similarly describes LLMAgent as the component that centralises communication with language models, exposes helpers for intent classification and parsing, and supports longer-lived agentic sessions beyond one-shot completions. At the bottom of the diagram, LLMAgent is shown handling prompt execution, intent interpretation, and session orchestration. It acts as a central layer connecting high-level and skill-based agents. This ensures controlled and consistent interaction with LLM providers such as OpenAI.

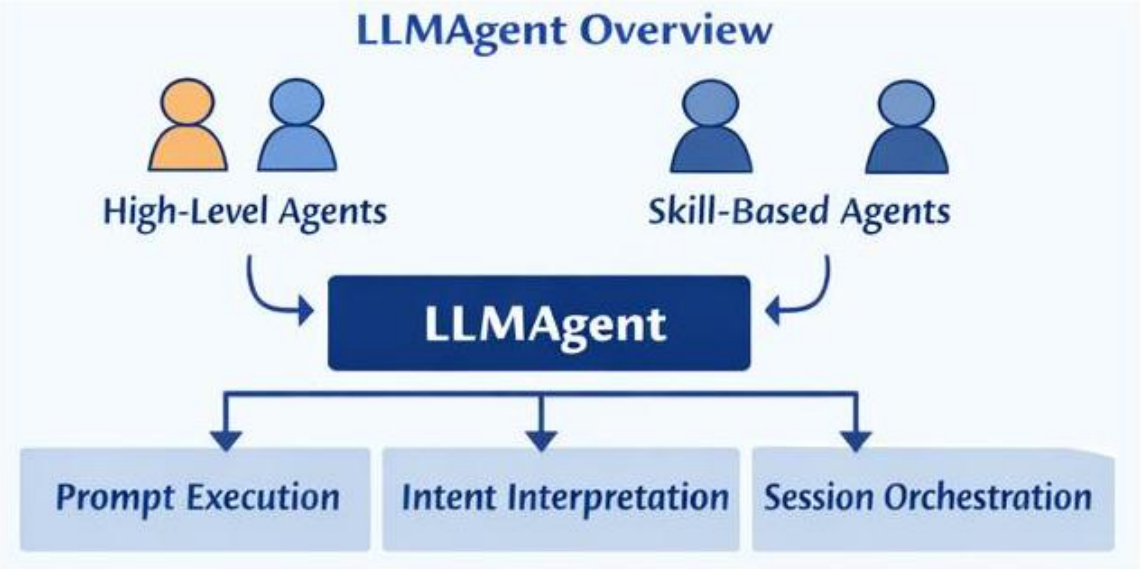

Figure 11 LLMAgent – Overview

The architectural importance of LLMAgent is that it prevents higher-level flows from depending directly on provider-specific inference calls. Instead of letting every runtime component decide independently how to prompt, how to inject memory, how to coerce structured output, or how to interpret confirmation-like language, AchillesAgentLib concentrates these concerns into a single reusable layer. This improves consistency, reduces duplication, and makes the rest of the stack more auditable even though the underlying model remains probabilistic.

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

Its public surface is organised around three main functions. The first is standardised prompt execution with optional memory shaping and output coercion. The second is lightweight interpretation of free-form user language into operationally meaningful decisions such as intent labels, confirmation states, or structured payloads. The third is session creation for longer-running workflows in which the model must repeatedly deliberate, call tools, observe outcomes, and preserve continuity across multiple steps.

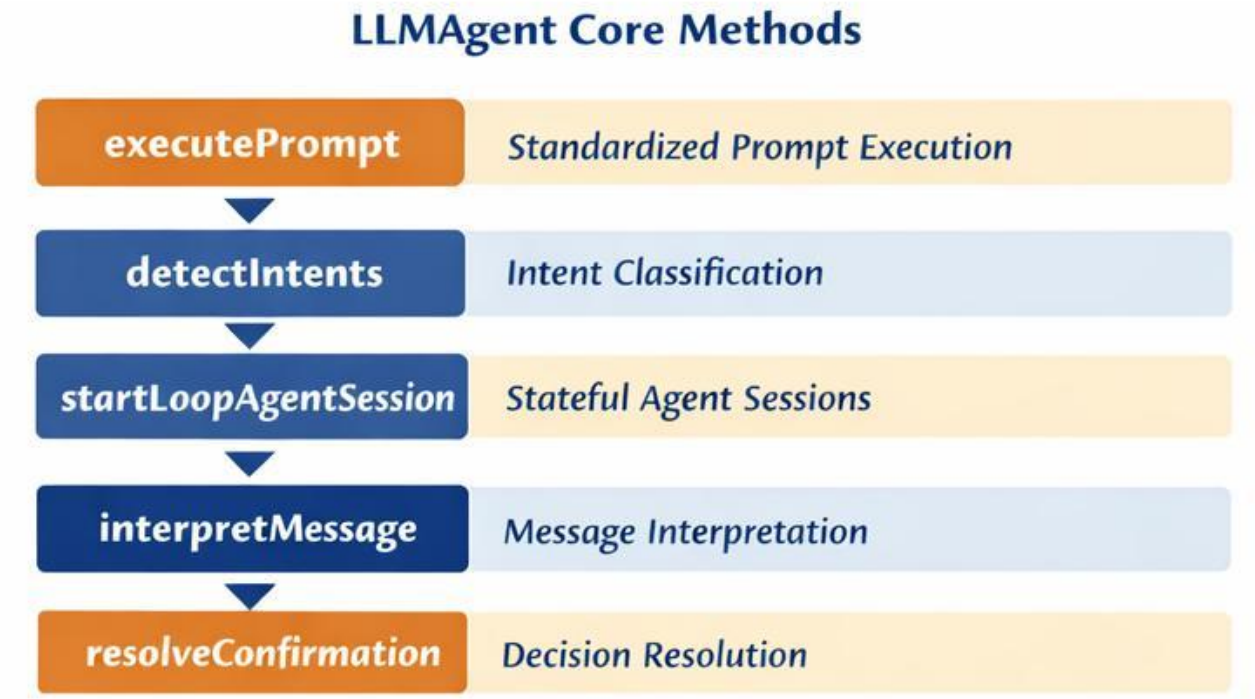

Figure 12 LLMAgent Core Methods

A concise summary of the main methods is given below.

Table 2. Core Methods and Their Functional Roles

Method	Role
executePrompt	Runs a prompt through the standardised runtime pipeline, including optional memory or context injection and post-processing that coerces the model output into a predictable form such as JSON, code, or normalised text.
detectIntents	Classifies a request against a described skill space and returns an explicit machine-readable intent selection, supporting transparent routing and dispatch.

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

startLoopAgentSession	Starts a stateful loop-based session in which the model can iteratively plan, call tools, observe results, and refine its output across multiple steps.
interpretMessage	Converts free-form user replies into constrained operational outcomes such as accept, cancel, update, or idea-like payloads.
resolveConfirmation	Resolves ambiguous yes/no language into a more robust decision state such as yes, no, or unclear, together with a confidence signal.

The importance of executePrompt is not only convenience. It establishes a standard path through which prompt text, optional memory layers, model mode, and output-shaping constraints are combined before the request reaches the provider. This matters because downstream components should not need to know whether output normalisation required heuristics, JSON extraction, or other post-processing. What matters for them is that the result is delivered in a stable and expected form.

detectIntents and interpretMessage are equally important for the runtime. They allow natural-language interaction to be converted into bounded execution signals. In practical terms, this is one of the first places where the architecture already behaves like a controlled interpreter of language rather than like an unconstrained chat interface. The model still handles linguistic variation, but its output is constrained into a small operational vocabulary that the runtime can safely use.

The session methods expose the same idea at a larger time scale. Public AgentLib documentation (ACHILLES Agent Library, 2026a) distinguishes between loop-based sessions and SOP-based sessions, with LLMAgent providing the entry point for both (ACHILLES Agent Library, 2026b). In the current ACHILLES implementation, this makes LLMAgent the natural mediator between local prompting and longer-lived workflows that preserve state, intermediate variables, and controlled tool usage across steps. It is therefore not merely a prompt wrapper. It is the general LLM-facing runtime service on top of which the rest of the library is built.

5.4.2 RecursiveSkilledAgent

If LLMAgent is the general LLM mediation layer, RecursiveSkilledAgent is the main runtime agent through which skill-based execution is initiated. Public AgentLib material describes it as the component that supervises complex multi-step plans, can execute skills directly, and routes execution into the appropriate subsystem depending on the discovered skill type (ACHILLES Agent Library, 2026a). In the current ACHILLES architecture, it is the primary entry point of the library and the top-level coordinator of the skill ecosystem.

Its central responsibility is discovery, routing, and delegation. RecursiveSkilledAgent scans the available skill repositories, registers the discovered descriptors, identifies the relevant skill or subsystem for a request, and starts the corresponding execution flow. This is precisely why it is called recursive. It can invoke skills that themselves orchestrate other skills, launch sessions that generate

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

plans, or forward execution into specialised backends without collapsing all these behaviours into one monolithic implementation.

Figure 13 RecursiveSkilledAgent

This separation is important. RecursiveSkilledAgent owns orchestration, but it does not own every form of execution logic. It does not internally reimplement the semantics of DBTable skills, dynamic code generation skills, MCP skills, orchestration skills, or Anthropic-style skills. Instead, it recognises the skill family, preserves the shared execution context, and forwards the task to the correct subsystem. This keeps the architecture unified at the entry level while preserving the safety rules and execution assumptions of each family-specific backend.

A concise summary of the main methods is given below.

Table 3. Functional Overview of RecursiveSkilledAgent Execution Methods

Method	Role
executePrompt	Main entry point for running a request without an explicit review interruption; may invoke a chosen skill or let the runtime select one.
executeWithReviewMode	Core execution path that applies an explicit review regime and injects session memory and hooks where needed.
executePromptWithReview	Convenience wrapper for LLM-mediated review before final execution or return.

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

executePromptWithHumanReview	Routes the result through a human review step.
executeWithoutExplicitSkill	Forces prompt-only routing so that the runtime decides which skill or subsystem should handle the task.

One of the most important design properties of RecursiveSkilledAgent is that it supports two complementary operational modes. In one mode, it behaves like an open orchestration agent. The user provides only a natural-language request, and the runtime must determine which skill should handle it. In the other mode, it acts as a stable entry point to a known capability by explicitly invoking a named skill. This dual role is valuable because it allows the same runtime to support both assistant-like interaction and deterministic access to a chosen workflow.

Another important responsibility concerns memory and execution context. RecursiveSkilledAgent manages session memory as a shared runtime concern. It creates or retrieves the appropriate execution context, injects shared state when necessary, and applies cleanup policies so that multi-turn interaction remains coherent without leaking state across unrelated runs. This is particularly important in ACHILLES because the same runtime must support CLI-like usage, interactive flows, and more structured orchestrations while preserving bounded execution isolation.

5.4.3 Skills Categories and Subsystems

The role of skills in AchillesAgentLib is to turn repeatable LLM-mediated work into reusable workflows. Instead of reconstructing a task from scratch through prompt-by-prompt improvisation each time, the runtime can rely on a named capability whose descriptor already captures the expected operational pattern. This makes a skill more than a label or a convenience wrapper. It becomes a durable unit of organised procedural behaviour.

As implementation progressed, however, it became clear that not all skills are of the same kind. Some are oriented toward structured data and CRUD (create, read, update, delete) as in DBTable skills. Some prioritise flexible ad hoc execution, as in dynamic code generation skills. Some package procedural guidance according to Anthropic's SKILL.md model. Some act as planners that coordinate other skills. Some connect to external MCP-based tools. These differences are large enough that they should not be hidden behind one undifferentiated execution path.

For this reason, AchillesAgentLib is organised around subsystems. A subsystem is a specialised execution backend for a particular skill family. Discovery and routing remain centralised in RecursiveSkilledAgent, but the actual execution is delegated to the subsystem appropriate for the descriptor type. This architecture is reflected publicly in the AgentLib documentation, which distinguishes explicit guides for orchestration skills, DBTable skills, MCP skills, dynamic code generation skills, code skills, Anthropic skills, and agentic sessions (ACHILLES Agent Library, 2026a).

The result is a layered model. Skills remain the units of reusable workflow. Subsystems provide the family-specific rules through which those workflows become executable. This allows the

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

architecture to preserve one shared runtime entry surface while admitting multiple execution styles without forcing them into one flat abstraction.

The underlying philosophical point is equally important. In ACHILLES, each skill family is treated as a distinct local interpreter of natural-language intent. The descriptor surface defines what counts as relevant input, which operations are available, which constraints apply, and what kind of output should be returned. Optional attached code then further modifies and stabilises that interpretation. In this sense, the skill is not only a prompt artefact. It is a bounded interpretive regime. This interpretive view of skills is one of the main conceptual bridges toward the next chapter, where the architecture is pushed further toward explicit intermediate representations and more structured execution.

5.4.4 Orchestration Skills

Orchestration skills form the declarative coordination layer of the system. Their purpose is not to perform domain work directly, but to map a higher-level goal into a structured sequence of calls to other skills. Public AgentLib documentation describes them as playbooks that map high-level goals to named steps, decide which skills or tools to activate, and use either loop sessions or SOP sessions depending on the descriptor (ACHILLES Agent Library, 2026h).

This makes orchestration skills especially appropriate for repeatable workflows such as investigations, onboarding flows, and multi-step reporting tasks. In such cases, the value does not lie primarily in one local transform, but in deciding which specialised capability should be used next and under what restrictions. An orchestration skill therefore acts as a planner and coordinator rather than as a direct executor.

Its behaviour is defined almost entirely by its descriptor. The key sections are summarised below.

Table 4. Overview of Orchestration Skill Descriptor Sections

Section	Role
Instructions	Defines how the planner should interpret the task, what to prioritise, and how to sequence work.
Allowed Skills	Defines the exact downstream skill set that may be invoked.
Intents or Description	Provides a compact vocabulary for why actions matter and what kinds of actions the planner should consider.
Session Type	Determines whether execution proceeds through a loop-based session or a SOP-style session.

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

Preparation	Supports a pre-context pass that can gather variables or contextual state before the main session begins.

A crucial point is that orchestration skills (ACHILLES Agent Library, 2026h) do not merely expose downstream capabilities. They constrain them. The allowed-skills list becomes the planner's toolbelt, and the session type determines whether the runtime operates in a faster turn-by-turn loop or in a more explicit plan-first mode (ACHILLES Agent Library, 2026b). This improves auditability because the coordination layer is explicit rather than hidden inside an undifferentiated prompt.

From the perspective of the broader architecture, orchestration skills are among the clearest examples of the idea that a skill is a local interpreter of natural language. The descriptor does not merely describe what the skill does. It shapes how the runtime should think about the task, what capabilities it may treat as admissible, and how the resulting workflow should be narrated and justified.

5.4.5 C-Skills

Code skills, or C-Skills, are the family intended for specification-driven execution. Public AgentLib documentation describes them as skills that generate executable JavaScript from detailed specifications, separating code generation from execution and using cskill.md together with a specs/ directory as the authoritative source (ACHILLES Agent Library, 2026d).

This family is appropriate when a capability is too complex for inline logic but still benefits from being generated and maintained from explicit specifications rather than hand-written ad hoc code. Here the descriptor and the specification files carry the real authority, while the generated code remains an implementation artefact. This is a strong fit for situations where repeatability, explicit requirements, and maintainability matter more than short-cycle improvisation.

The cskill.md descriptor defines the public contract of the skill. It focuses on interface and constraints rather than detailed implementation logic. The deeper behaviour is described in the specs/ directory, where each file specifies one module in natural language. At preparation time, the subsystem ensures that the executable code is generated or regenerated from those specs. At runtime, the generated entry point is executed to produce the result (ACHILLES Agent Library, 2026d). The main descriptor sections can be summarised as follows.

Table 5. Functional Structure of C-Skill Descriptors

Section	Role
Summary or Description	Concise statement of the capability used for cataloging and routing.
Input Format	Expected input structure and required fields.

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

Output Format	Expected output type and examples of success or failure.
Constraints	Hard requirements that generated code must satisfy.

The architectural importance of C-Skills is that they make the relation between natural-language specification and executable behaviour explicit. The subsystem reads the specification files, generates coherent code, and then executes that code in a stable manner. In other words, the local interpreter is progressively specialised. This makes C-Skills one of the clearest current bridges toward the next chapter, because they already embody the idea that some local LLM-mediated behaviour can later be stabilised into more explicit and more predictable execution regimes.

5.4.6 Dynamic Code Generation Skills

Dynamic Code Generation skills, or DCG skills, exist for the opposite end of the spectrum. Public AgentLib documentation describes them as skills that allow the agent either to answer directly or to emit JavaScript that is then executed inside a guarded environment, with the main emphasis placed on sandbox policy, prompt guidance, and rapid productivity (ACHILLES Agent Library, 2026f).

Their value lies in flexibility. When requirements are unclear, one-off, or too transient to justify a permanent specification-driven capability, it may be more appropriate to let the LLM decide whether the request should remain textual or become procedural. DCG skills therefore optimise for rapid prototyping, unusual transformations, and proportionality of effort. They allow useful functionality to be delivered without forcing every capability to become a permanent subsystem or code skill.

The DCG descriptor supplies the guidance, expected inputs, and execution mode that shape how the LLM should behave. At runtime, the subsystem decides whether the task should be answered directly or through temporary generated code, and it normalises the output in either case. This yields an execution family optimised for exploration and short-cycle delivery rather than long-term formalisation.

The contrast with the other families is important. DBTable skills optimise for consistency and integrity. C-Skills optimise for specification-driven control and repeatability. DCG skills optimise for adaptability. They therefore occupy a necessary region of the design space rather than a lower-quality one. Their strength is precisely that they let the runtime remain agile where stronger formalisation would be disproportionate.

5.4.7 DBTable Skills

DBTable skills are the most declarative and business-rule-oriented family in the current implementation. Public AgentLib documentation describes them as skills that wrap CRUD-style workflows around structured business tables, using tskill.md as the single source of truth for schema rules, validation hints, lookup strategies, and field behaviour, which are then compiled into executable helpers. This family exists because many enterprise-oriented tasks are not primarily about free-form generation. They are about correctly interpreting requests over entities with a relatively stable schema,

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

applying business constraints, resolving ambiguous field values, deriving additional values when necessary, and presenting records back to the user in a consistent and human-readable form. In such cases, the main difficulty is not storing data alone. It is making sure that the data model is interpreted coherently in an LLM-mediated environment.

The heart of the family is the tskill.md descriptor. This file defines the purpose of the table, the fields, validation logic, alias resolution, derivations, and presentation rules. Public documentation confirms that DBTable skills parse these descriptor sections into a blueprint, generate helper functions such as record preparation, validation, and presentation, and then use LLMAgent to classify the user request into CREATE, UPDATE, SELECT, or DELETE before applying the corresponding flow (ACHILLES Agent Library, 2026e).

What matters architecturally is that tskill.md is not just database schema documentation. It is a broader operational specification for entity interpretation under natural language. The descriptor determines how values should be resolved, what counts as valid, which fields are derived rather than persisted, how aliases are normalised, and how results are presented back to the user. This is why DBTable skills changed our practical understanding of what a skill can be. They showed that a robust enterprise skill is often best understood as a specialised interpreter over a declarative model.

At runtime, the subsystem uses the LLM only where it is genuinely useful, namely for classifying the intended CRUD operation and for local interpretive flexibility over user phrasing. The rest of the flow is deliberately structured. Inputs are normalised, field resolvers and derivators are applied, validators are executed, persistence is handled through the configured database adapter, and returned data is passed through presentation logic (ACHILLES Agent Library, 2026e). The result is a strong reduction in duplicated business logic and a much more coherent way of handling structured data through natural-language interaction.

5.4.8 Anthropic Skills

Anthropic-style skills are the skill family in AchillesAgentLib that aligns most directly with the increasingly visible external standard centred on SKILL.md. We refer to them as *Anthropic Skills* because Anthropic most visibly promoted this now mainstream concept, which has since been adopted more broadly across the industry. In practice, this corresponds closely to what much of the industry currently means by a “skill.” Anthropic’s own documentation describes Agent Skills as modular capabilities that package instructions, metadata, and optional supporting resources such as scripts and templates to provide reusable domain-specific expertise that an agent can load on demand (Anthropic, 2025b). Their engineering material further presents them as organised folders of instructions, scripts, and resources that capture procedural knowledge and transform a general-purpose agent into a more specialised one (Anthropic, 2026e).

AchillesAgentLib supports this style explicitly but also positions it more precisely within a broader architecture. Our public AchillesAgentLib documentation describes Anthropic Skills as

Funded by the European Union
This project has received funding from the European Union’s Horizon Europe research and innovation programme under Grant Agreement No 101189689.

bundles centred on SKILL.md, with optional scripts/, resources/, and assets/, executed through an agentic loop session (ACHILLES Agent Library, 2026c). In the current implementation, the normative centre of such a skill is indeed SKILL.md. The file does not function merely as documentation. It provides the primary behavioural guidance enacted during the session. For this reason, instructional quality is particularly important in this family: the skill body must remain clear, explicit, and operational, because vague prose becomes an execution problem rather than only a style issue.

The structure around SKILL.md extends the skill beyond instruction alone. scripts/ provides executable support logic, resources/ provides auxiliary reference material, and assets/ provides supporting artefacts belonging to the local environment of the skill. Together, these elements form a structured bundle in which instruction, procedure, and supporting material remain distinct but coordinated. This is strongly aligned with Anthropic's external conception of skills as organised folders of instructions, scripts, and resources (Anthropic, 2025b). In AchillesAgentLib, however, this family is placed inside an explicit runtime model. Anthropic-style skills are executed through loop-session behaviour rather than treated as static descriptor lookups. The model may therefore iteratively interpret the instructions, access supporting material when relevant, invoke available tools, and continue execution as a stateful process. The skill is not simply loaded; it is operationalised inside an instrumented runtime.

At the same time, the similarity with other ACHILLES skill families is only partial. Anthropic Skills are superficially close to C-Skills or other structured skill forms, but the underlying philosophy is substantially different. The mainstream industry view, shaped largely around general-purpose agents, treats skills mainly as reusable guidance structures inside a broader agentic loop. The ACHILLES view is more demanding. In our architecture, skills are understood more granularly, as bounded local interpreters of natural language and as mechanisms for making natural language executable, rather than leaving it embedded in a broad and relatively amorphous conversational context. This difference in perspective is significant and explains why Anthropic Skills, although useful for interoperability, do not occupy the same conceptual place in the architecture as the more specialised ACHILLES skill families.

For this reason, current support for Anthropic Skills remains relatively fragile in AchilelsAgentLib. The issue is not only implementation maturity, but also the architectural tension between two different design philosophies. Their importance nevertheless remains clear for two reasons. First, they provide practical interoperability with a visible external skill model that has effectively become a reference point in current agent engineering, especially for coding agents and other general-purpose agent settings (Anthropic, 2026e). Second, they make visible a broader architectural claim of the project: even a seemingly simple file-based skill model already behaves as a local interpreter of natural-language intent. AchillesAgentLib extends this insight further by treating other skill families as more specialised interpretive regimes. A plausible future direction is therefore that Anthropic Skills may be integrated less as fully native skills and more through a dedicated

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

orchestration layer for general-purpose agents, allowing interoperability without obscuring the stronger architectural distinctions on which the ACHILLES approach is based.

5.4.9 MCP Skills

MCP skills are the family through which AchillesAgentLib choreographs external tools that speak the Model Context Protocol. Public AgentLib documentation describes them as skills that decide which MCP tool to call, attach rationale to the step, and return an auditable plan, with descriptor sections for instructions, allowed tools, and optional LightSOPLang blocks (ACHILLES Agent Library, 2026g).

Their role is important because not all external-tool orchestration should be left to unconstrained agent improvisation. In many workflows, especially those involving document hunts, telemetry checks, or compliance-related evidence gathering, the relevant problem is not merely choosing any tool that might work. It is choosing among an approved set of tools under explicit policy and reporting constraints.

For this reason, MCP skills are descriptor-driven constrained interpreters of external-tool usage (ACHILLES Agent Library, 2026g). Their main customisation surface typically includes instructions, an explicit whitelist of allowed tools, optional deterministic LightSOPLang scripts, and further notes or limits that cap plan length or sharpen review expectations. The descriptor therefore does two things at once: it tells the runtime how to approach the task, and it narrows the operational surface to a bounded and auditable toolchain.

This family is especially important in the ACHILLES architecture because it illustrates how a skill may sit between natural-language task interpretation and a protocol-governed external environment. It does not merely wrap tools. It interprets the task into a controlled interaction pattern over those tools. This is again why we treat the skill as an interpreter and not just as a label.

5.5 Evaluation Suites and Validation Practice

Evaluation in the ACHILLES environment should not be understood as a single standalone component, but as a set of evaluation suites and automated tests distributed across components, features, and implementation layers. Their role is practical and tactical: they help guide implementation, detect regressions, compare alternatives where comparison is meaningful, and maintain an acceptable engineering discipline around systems that include probabilistic behaviour.

This role should be distinguished from that of ordinary software testing, even though traditional automated tests remain an absolute and implicit requirement wherever deterministic behaviour is expected. Deterministic code paths, infrastructure logic, interfaces, and bounded execution routines must continue to be covered by ordinary tests. In parallel, where LLM-mediated behaviour or more open-ended agentic execution is involved, dedicated evaluation suites are used to assess recurring quality dimensions such as intent recognition, tool-use correctness, planning behaviour, routing

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

quality, stability across repeated runs, latency tendencies, and other performance aspects where measurement is appropriate.

At the same time, the project does not treat these evaluation suites as the main strategic source of justification for the architectural direction of T3.3. In this area of software development, many of the most important properties are only partially measurable in isolation. Reliability in practice, clarity of interaction, boundedness of behaviour, maintainability, integration quality, and user trust often depend on many small design and implementation decisions whose value becomes visible primarily in real usage rather than in a single benchmark. For this reason, the deeper motivation of the ACHILLES approach remains grounded first in first-principles reasoning, architectural judgement, and practical implementation experience.

Axiologic, as the partner leading the commercialisation direction of these technologies, therefore relies primarily on practical validation signals: feedback from programmers using the environment, feedback from clients and prospective partners, and the extent to which the benefits of the approach can be clearly explained, demonstrated, and defended in concrete operational terms. This is especially important at month 18 of an Innovation Action that is still far from its final implementation state. At this stage, adoption depends not only on technical correctness in a narrow benchmarking sense, but also on whether the project can present a coherent value proposition to the market, show why the architectural choices matter, and demonstrate that the resulting systems can become useful, trustworthy, and commercially credible in real workflows.

The existence of evaluation suites nevertheless remains important and should be stated clearly. They provide disciplined internal feedback during implementation, especially when several models, prompts, planners, skill configurations, or execution variants must be compared under controlled conditions. They are therefore part of the engineering discipline of the project, but they should not be overstated as if they exhausted the meaning of validation for the ACHILLES environment. Their value at this stage is primarily to support implementation decisions, reduce avoidable regressions, and make some aspects of system behaviour more visible during development.

At this stage, evaluation is tactical rather than strategic, as the rapid evolution of agent-optimised LLMs renders benchmarks from even six months ago largely obsolete. Consequently, D3.3 focuses on defining the architectural direction and implementation baseline of the multi-agent environment, rather than serving as an exhaustive report on internal testing and benchmarking.

A later version of the deliverable will provide greater visibility into the evaluation suites, their coverage, and the lessons learned from their use. Likewise, some aspects related to efficiency and performance have already been discussed in the D3.1 context where relevant, but their practical significance ultimately depends on how they contribute to robust systems, usable workflows, and credible adoption rather than on isolated raw metrics alone.

5.6 Implementation Status and Concluding Positioning

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

To make the present maturity boundary explicit and reviewer-safe, Table 6 summarises the implementation status of the main components at month 18, distinguishing between the already operational baseline, the elements under active refinement, and the research directions that remain exploratory at this stage.

Table 6. Implementation Status and System Positioning at M18

Component	Implementation status at M18
Ploinky	Ploinky is implemented and used as the deployment and interoperability substrate of the current ACHILLES environment. At this stage, it provides the practical basis for packaging, configuration, isolation, routing, and MCP-aligned exposure of agentic services. Its role in the project is already operational, while further hardening and broader use-case integration remain part of the next phases.
LLM Proxy (Soul Gateway)	The LLM Proxy is implemented and operational as the governance and observability layer for model access. It already supports controlled access to model providers through authentication, routing, observability, budget supervision, retry governance, and audit logging. At M18, it should be understood as a working infrastructure component on which later optimisation and broader deployment can build.
AchillesAgentLib	AchillesAgentLib is the main operational runtime contribution of T3.3. It already supports controlled model interaction, agentic sessions, reusable skill descriptors, recursive routing, and specialised skill families. It constitutes the current baseline for agent development and integration in the ACHILLES environment, while continued refinement is expected as use-case integration progresses.
Agentic sessions and evaluation layer	The session model and evaluation mechanisms are already implemented to a meaningful degree and are part of the current technical contribution rather than only future work. At the same time, their coverage, benchmarking depth, and use-case-specific evaluation scenarios will continue to expand in later project stages.

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

SOP Lang	SOP Lang should be presented as partially implemented and already useful in operational form for selected orchestration and representation tasks, while some parts remain under active refinement. At M18, its value lies in making plans, dependencies, and intermediate artefacts more explicit, rather than in claiming a fully stabilised language stack.
MRP-VM / Meta-Rational Pragmatics	MRP-VM should be presented as an emerging research direction supported by early experiments, conceptual work, and initial implementation efforts, but not yet as a stabilised production component. At this stage, it is best framed as the most promising next implementation path opened by the current work, rather than as an immediate replacement for the existing runtime.

Taken together, these statuses show that ACHILLES already has a meaningful operational core at M18, while remaining at an appropriate stage of progressive maturation and is expected to strengthen integration, validation visibility, and use-case coverage in the next phases.

6 TOWARDS MRP-VM

6.1 Research direction and motivation

As discussed in the previous sections, one of the central insights emerging from ACHILLES is that many useful agentic capabilities are better understood as bounded local interpreters of natural language than as generic skills attached to a broad ReAct loop. This insight first became visible in practical components such as DBTable skills and then gradually opened a broader research direction. The public AGISystem2 site, available at **www.agisystem2.com**, already gathers a growing series of articles, notes, and experimental perspectives related to this direction (Alboaie S., 2026b). These materials should be read as part of an active process of conceptual and technical maturation. For this reason, the present deliverable keeps this section concise. The direction is promising, but its architecture is still evolving, and substantial refinements remain likely in the next phase of the project.

The central problem is already visible in practical AI systems. Many tasks do not arrive as clean formal objects. They arrive as requests, policies, procedures, notes, or specifications expressed in natural language, often incomplete, ambiguous, and only partly formalised. Such inputs become actionable only relative to a local regime of interpretation: what is relevant, what must be checked, what evidence is admissible, what uncertainties can be tolerated, and what counts as an acceptable result in context (Alboaie S., 2026h). This is the deeper reason why ACHILLES does not assume that one universal semantics, one monolithic model, or one fixed reasoning style is sufficient. The broader AGISystem2 perspective formulates this by treating stable semantics as something often achieved

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

locally and progressively, under constraints, rather than assumed globally in advance (Alboaie S., 2026c; Alboaie S., 2026d).

This is the sense of the term *Meta-Rational Pragmatics*. The term *pragmatics* indicates that language is treated as operational material guiding decomposition, routing, validation, and execution. The term *meta-rational* indicates that the system must reason not only within a chosen local regime, but also about the adequacy of that regime itself: whether it is too weak, too costly, too fragile, or too rigid, and whether stronger validation, a different interpreter, or a revised decomposition is required (Alboaie S., 2026c; Alboaie S., 2026e). In this sense, the direction does not relax rigor. It seeks to make the transition from language to action more explicit, more inspectable, and more governable.

6.2 From skills to interpreters

This direction can be understood as a generalisation of the architectural lesson already learned from ACHILLES skills. In the current runtime, skills are increasingly treated as bounded local interpreters of natural-language intent. Some interpret text through prompting and orchestration, some through enterprise code and validation logic, some through database abstractions, and some through more structured execution regimes. The term *skill* was useful during the first phase of development because it captured a practical and reusable execution unit. However, as the architecture became more ambitious, that term also became more limiting.

One reason is conceptual clarity. In mainstream agent engineering, especially in coding agents and other general-purpose assistants, the word *skill* has come to denote relatively lightweight reusable guidance packaged around broader agentic loops. That meaning is now strongly associated with the external standards promoted most visibly by Anthropic and then adopted more broadly across the industry. The ACHILLES perspective has gradually moved in a different direction. Here the main object is not simply a reusable package of instructions, but a bounded interpretive regime with its own assumptions, execution discipline, validators, and criteria of adequacy. For this reason, the more general term *interpreter* now appears increasingly appropriate.

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

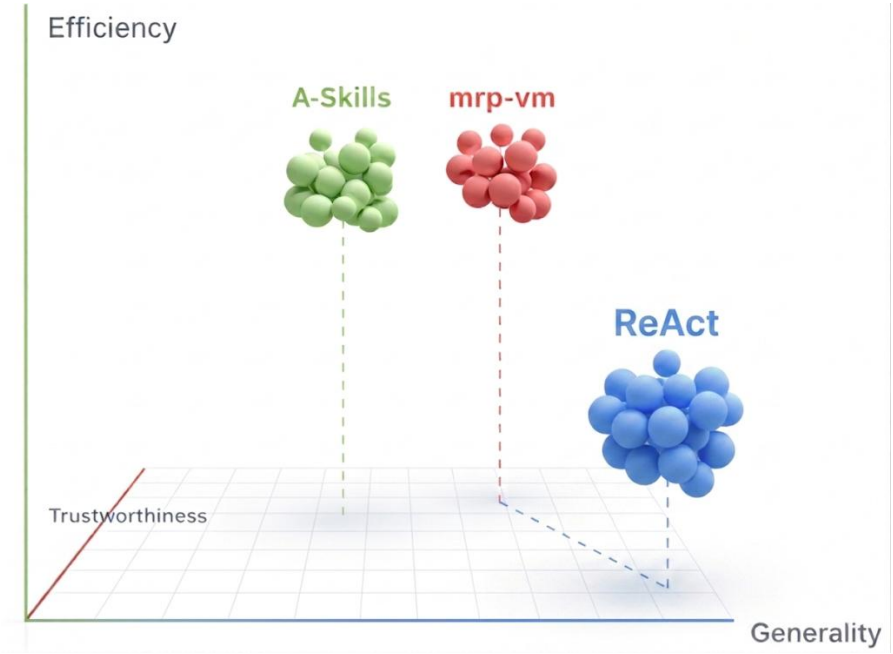

Figure 14 MRP-VM Comparative Positioning

In the accompanying diagram, we illustrate the proposed positioning of **mrp-vm** in contrast to **ReAct** Agents and **A-Skills (AchillesAgentLib Agents)**. Under the A-Skills architecture, skills function as natural language interpreters (sub-agents operating within a coordinator-selected context). Through this model, we aim to preserve inherent **reliability and performance** while bridging the gap toward the **general-purpose flexibility** of ReAct.

This terminological shift is not cosmetic. It reflects a more general architectural ambition. The project seeks a framework in which different local problems can be routed to different modes of interpretation and execution: language-model-based, symbolic, algorithmic, schema-driven, validation-oriented, retrieval-oriented, or hybrid. Under this interpretation, what matters is no longer only whether a capability can be called, but under what regime it should operate, how its outputs should be validated, and how its assumptions should be recorded and revised. This is also what gives the direction its neuro-symbolic significance. It creates a path beyond prompt-centric agent engineering toward a runtime able to combine heterogeneous forms of computation under explicit control.

6.3 MRP-VM as a provisional implementation path

MRP-VM should therefore be understood as a provisional implementation path for this broader research direction rather than as its final form. A classical virtual machine executes over a fixed instruction set. MRP-VM is closer to a pragmatic virtual machine that hosts and coordinates heterogeneous interpreters. A local task may be handled by an LLM-centred interpreter, by symbolic procedures, by constrained retrieval and validation mechanisms, by custom code, or by other specialised computational regimes. The role of the VM is not to erase those differences, but to govern

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

them: to preserve local assumptions, to route tasks toward appropriate interpreters, and to reintegrate their results into a common process (Alboaie S., 2026g).

This also explains why earlier descriptions centred on plugin families should now be read as provisional. They were useful during initial experimentation, but they already appear somewhat too tied to implementation detail. The more stable idea is that the runtime should be organised around interpreters directly. Some of these interpreters may still be deployed or configured in plugin-like ways, but the architectural centre of gravity is shifting from a taxonomy of plugins toward a theory of bounded interpretive regimes. This is both a generalisation and a clarification. It avoids confusion with the mainstream notion of skills, and it gives the project a more suitable conceptual basis for further neuro-symbolic experiments.

At the implementation level, at least three directions remain visible. One is an LLM-centred path, in which language models perform most of the disambiguation and a significant part of the operational interpretation. A second is a more symbolic path, in which as many stages as possible are reduced to explicit rules, deterministic procedures, and specialised engines. The third, and at present the most realistic, is a hybrid path in which language models are used where pragmatic resolution is unavoidable, while symbolic and algorithmic components are used wherever stronger structure and better-bounded semantics can be obtained. The project currently treats this hybrid direction as the most credible one, both technically and scientifically.

The expected value of this direction is twofold. First, it offers a way to obtain better generalisation than is usually available in narrowly hard-coded systems, because it preserves multiple forms of bounded interpretation rather than collapsing everything into one fixed pipeline. Second, it offers a clearer path toward trustworthiness than mainstream general-purpose agent loops, because interpretation, validation, and revision can be assigned more explicitly and more locally. In this sense, MRP-VM is not intended as a replacement for the current ACHILLES runtime in the immediate term. It should be read as the next research horizon opened by the current work on skills as natural language interpreters.

For the purposes of this deliverable, this is the appropriate level of commitment. The direction is already conceptually significant and supported by early experiments, public articles, and implementation efforts. At the same time, it is still in active maturation, and its terminology, internal abstractions, and execution model may evolve substantially as integration advances in ACHILLES IDE, in the ACHILLES CLI copilot, and in the project use cases. The prudent conclusion is therefore clear: Meta-Rational Pragmatics already provides a valuable research frame for the next phase of ACHILLES, while MRP-VM should presently be understood as its most promising, but still evolving, implementation path.

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

7 CONCLUSIONS

This deliverable establishes the first technical baseline for the ACHILLES multi-agent execution environment developed under T3.3 at M18. It reports the architectural rationale, the current implementation status, the preliminary validation practice, and the research directions that will guide the next phase of work. Its role is therefore both descriptive and orienting. It documents what has already been implemented, clarifies how the main components relate to the broader ACHILLES architecture, and identifies the areas where further consolidation and evaluation are required.

The main result of T3.3 at this stage is a modular execution stack for agentic systems. At the deployment level, Ploinky provides a standardised substrate for packaging, isolating, and exposing agents and services. At the model-access level, the LLM Proxy/Soul Gateway provides a governed interface to large language models, supporting observability, configuration management, authentication, cost monitoring, and audit logging. At the agent development level, AchillesAgentLib provides the framework for model interaction, session management, and reusable skill execution. Together, these components form the current operational baseline of the ACHILLES multi-agent environment. A central contribution of this baseline is the distinction between tools, skills, agents, and sessions. The project does not treat agentic capability as a single monolithic conversational loop. Instead, it decomposes execution into bounded and reusable operational units that can be invoked, composed, inspected, and refined. This approach supports the wider ACHILLES objectives of efficiency and trustworthiness by reducing unnecessary recomputation, making intermediate artefacts more explicit, and enabling more controlled access to external capabilities.

The work reported in D3.3 is directly relevant to the ACHILLES IDE and to the project use cases. For WP6, it provides the execution substrate needed to integrate agents, backend services, model access, and skill-oriented workflows into a practical development environment. For WP7, it provides mechanisms that can support structured and traceable workflows in use cases such as SCRIPTA and HERA, where planning, generation, retrieval, validation, compliance checking, reporting, and selective updating of intermediate artefacts are expected to play an important role.

The deliverable also identifies two important forward-looking directions. SOP Lang is an active implementation and research line for making workflows more explicit, dependency-aware, reusable, and inspectable. It provides a bridge between the current skill-based execution model and more structured orchestration mechanisms. MRP-VM, by contrast, is presented as an exploratory research direction rather than as a stabilised production component. Its purpose is to investigate how natural-language artefacts, structured representations, and interpreter-like execution mechanisms could progressively reduce reliance on unconstrained LLM-mediated planning and execution.

The evaluation work at M18 remains preliminary and component-oriented. Current validation has focused on functional testing, controlled execution scenarios, development use, and internal evaluation suites for comparing models, prompts, planners, skill configurations, and execution variants. This provides initial evidence of the operational readiness of the main components, but it does

Funded by the European Union
This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

not yet constitute full use-case-level benchmarking. The next phase will therefore need to expand evaluation coverage, define clearer benchmarking scenarios, and measure reliability, efficiency, observability, execution stability, and integration performance in more systematic ways.

In conclusion, D3.3 shows that ACHILLES has moved beyond conceptual design toward a concrete and reusable multi-agent execution environment. The immediate achievement is the establishment of a deployable operational stack centred on Ploinky, the LLM Proxy/Soul Gateway, AchillesAgentLib, skills, and agentic sessions. The next priority is to consolidate this stack through broader integration with WP6 and WP7, strengthen the benchmarking evidence, refine SOP Lang, and determine which exploratory mechanisms, including MRP-VM, can mature into stable project components by D3.4.

Funded by the European Union

This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

8 ANNEX A

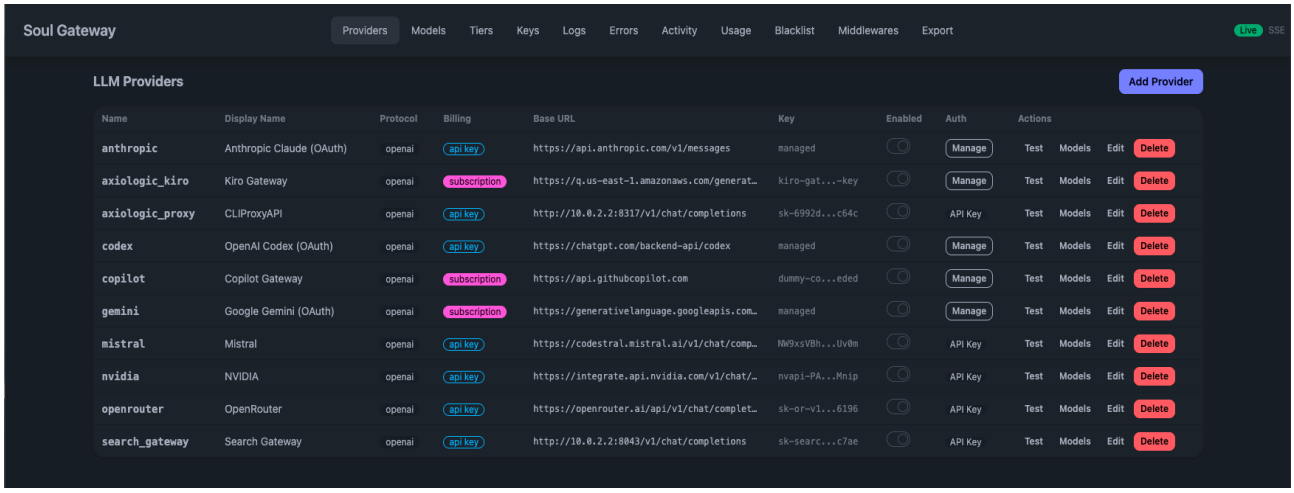

Figure 151 Soul Gateway – LLM Providers

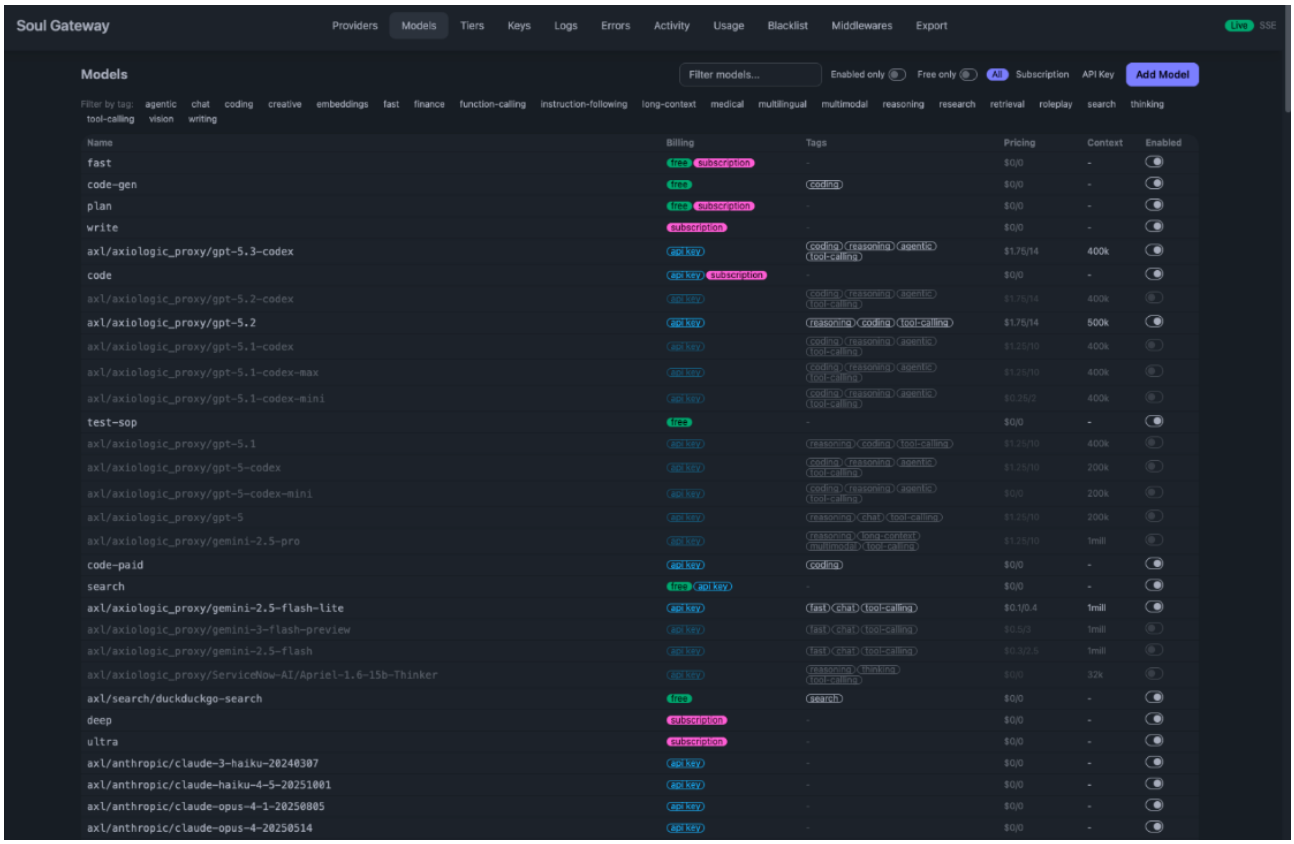

Figure 162 Soul Gateway – Models

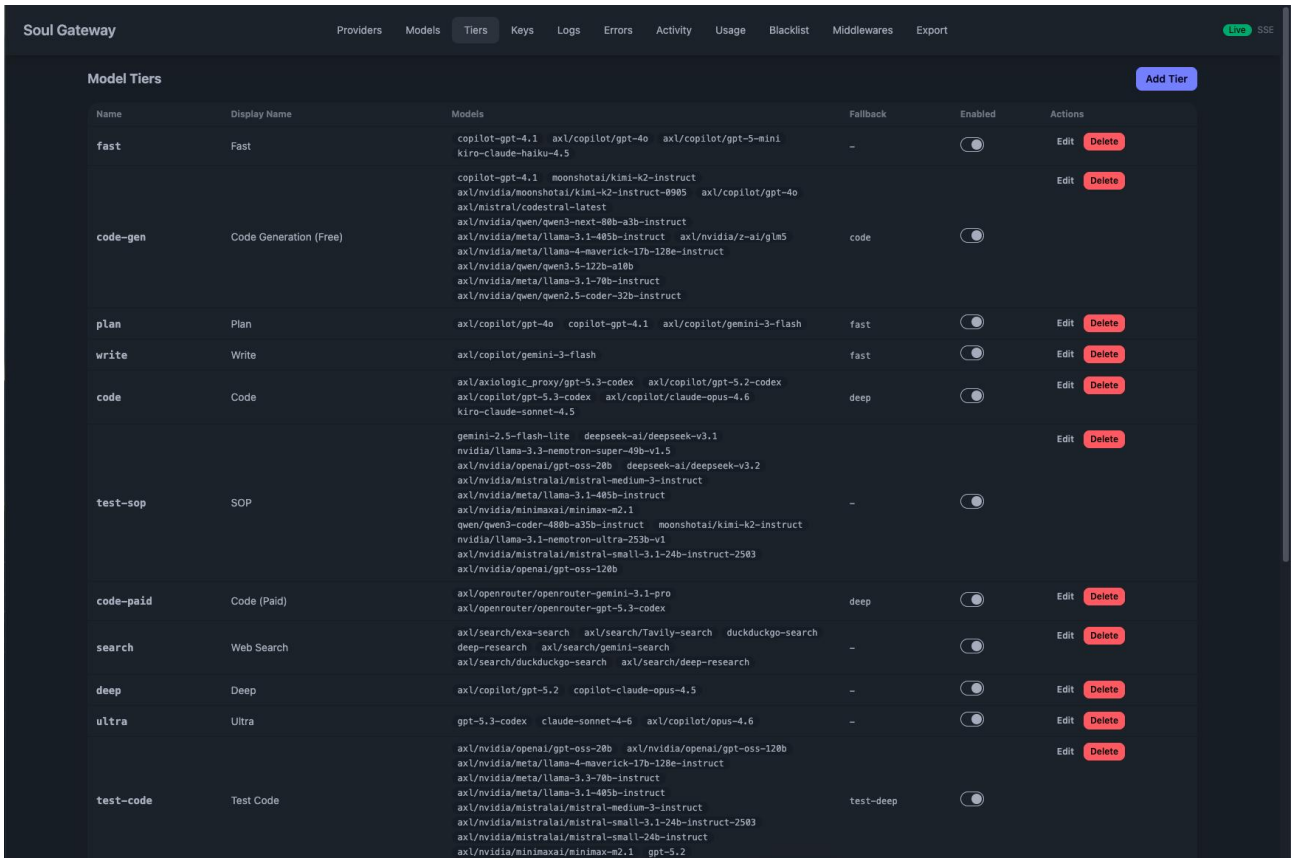

Figure 173 Soul Gateway – Model Tiers

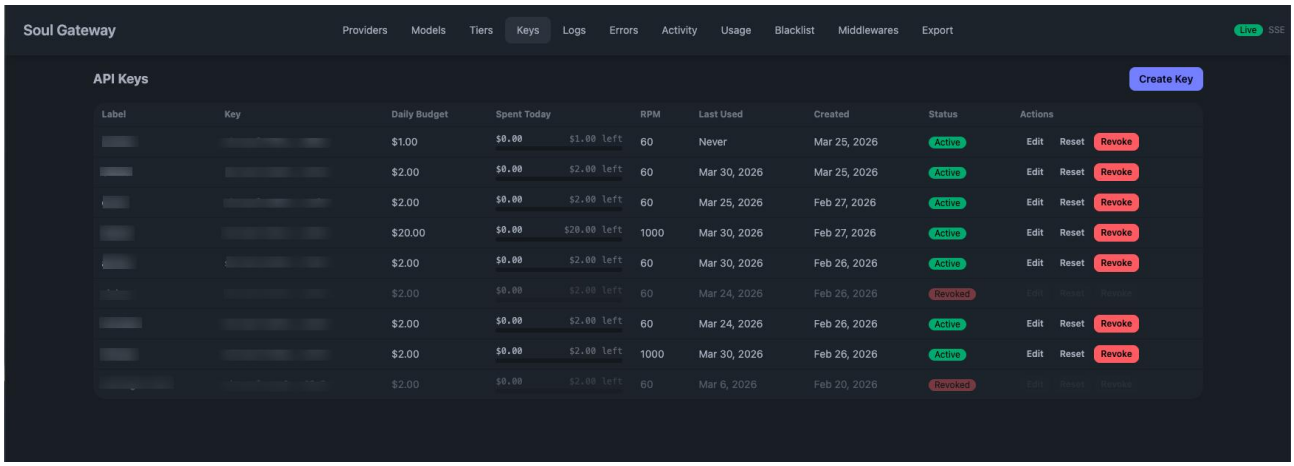

Figure 184 Soul Gateway – API Keys

Funded by the European Union

This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

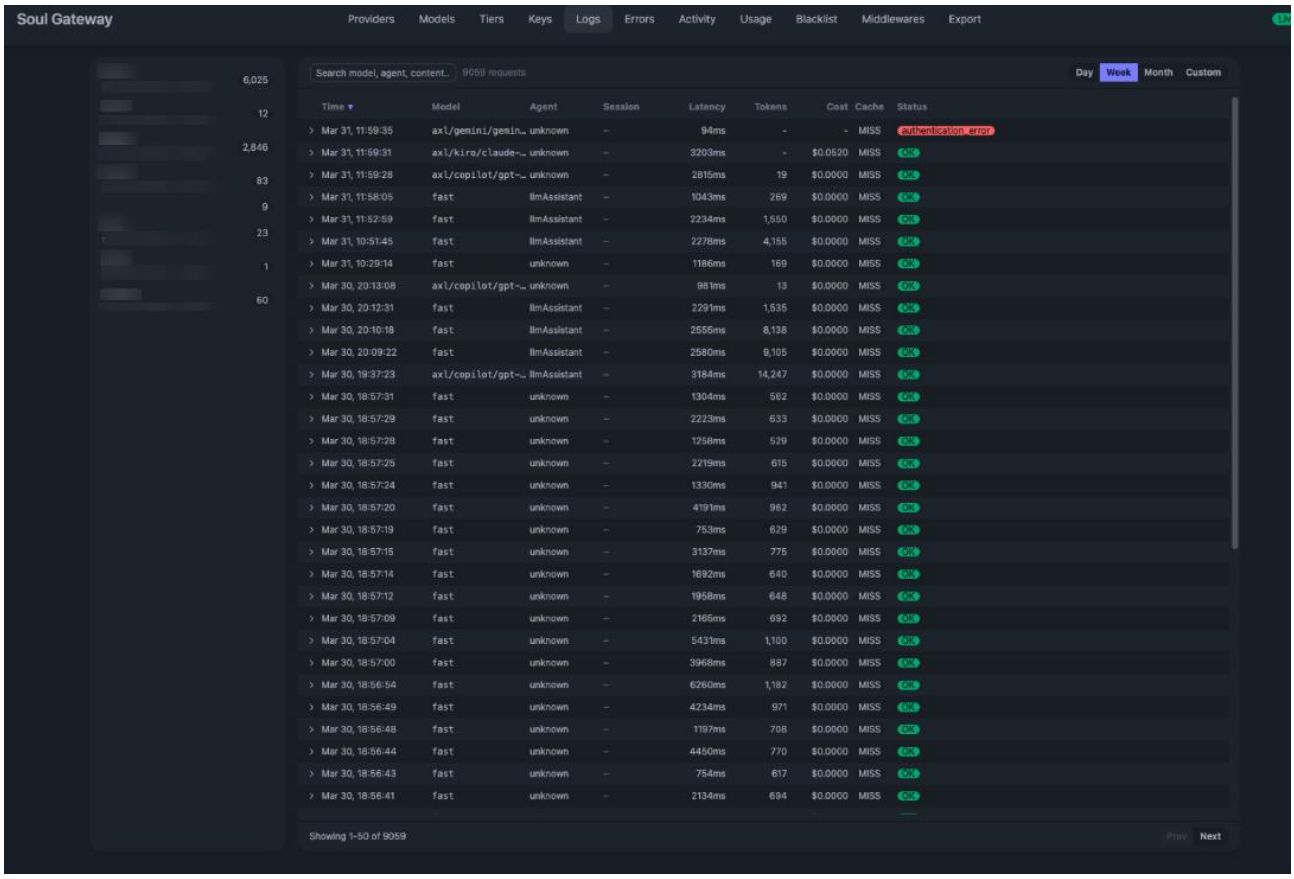

Figure 195 Soul Gateway – Logs

Funded by the European Union

This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

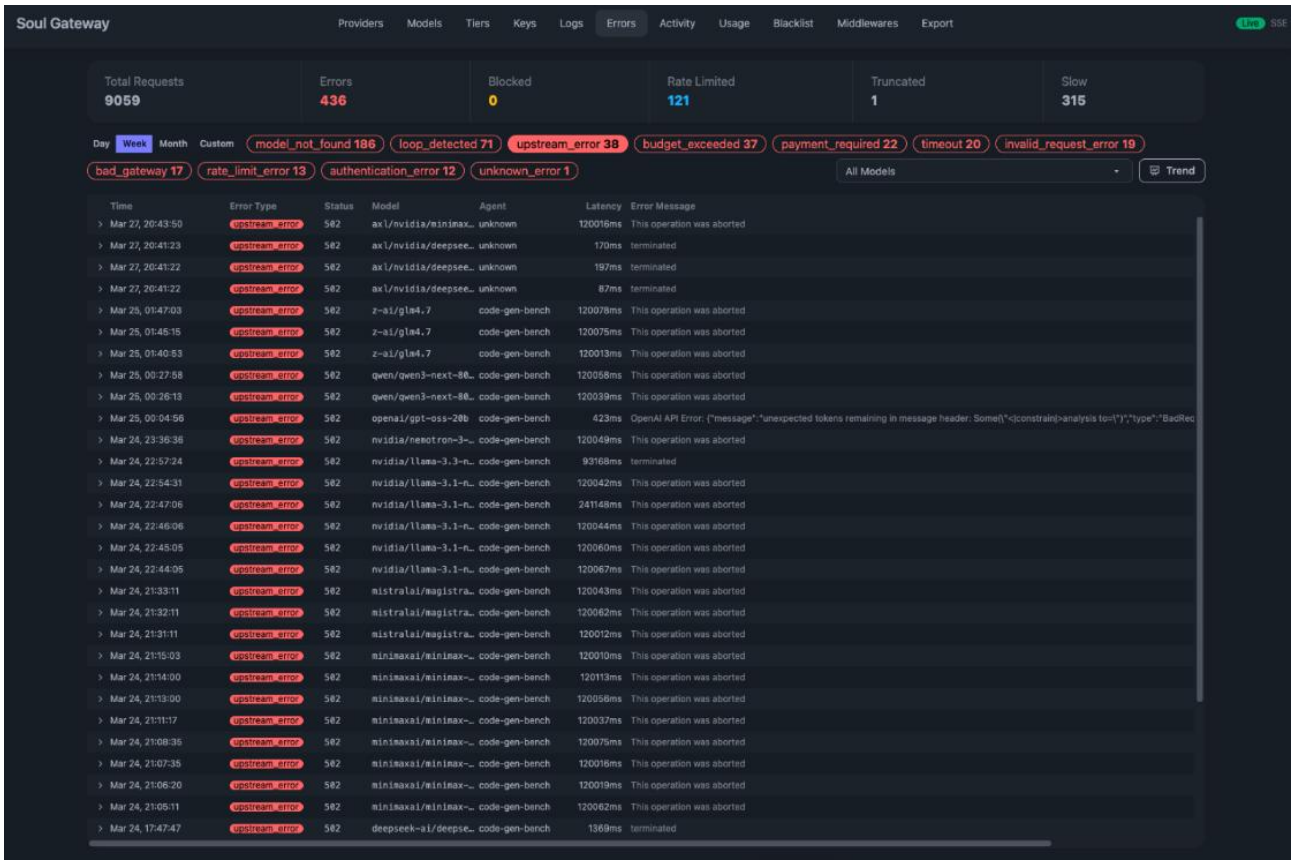

Figure 206 Soul Gateway – Errors

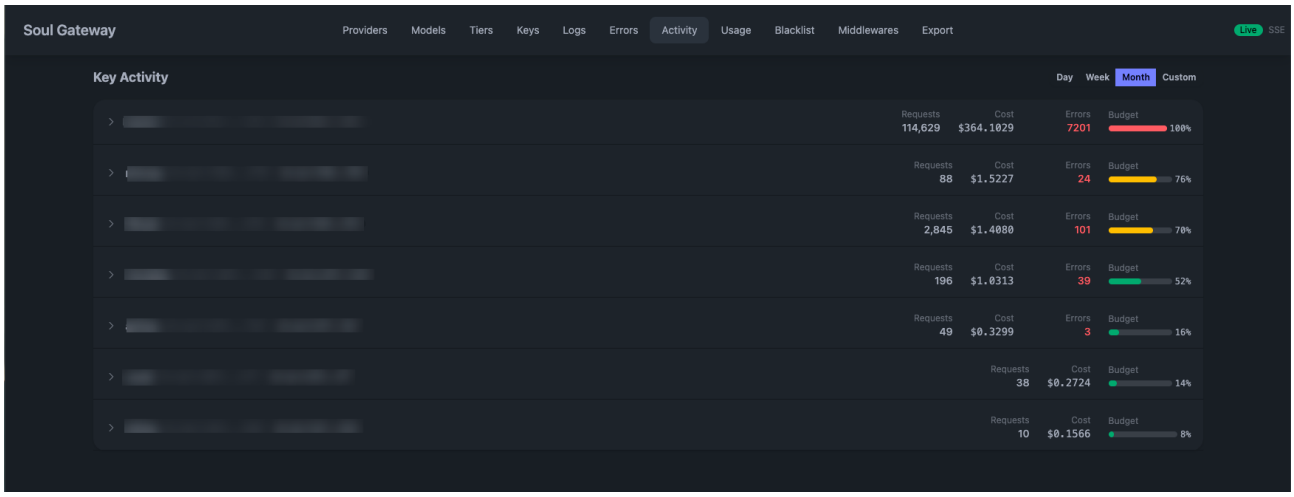

Figure 217 Soul Gateway – Activity

Funded by the European Union

This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

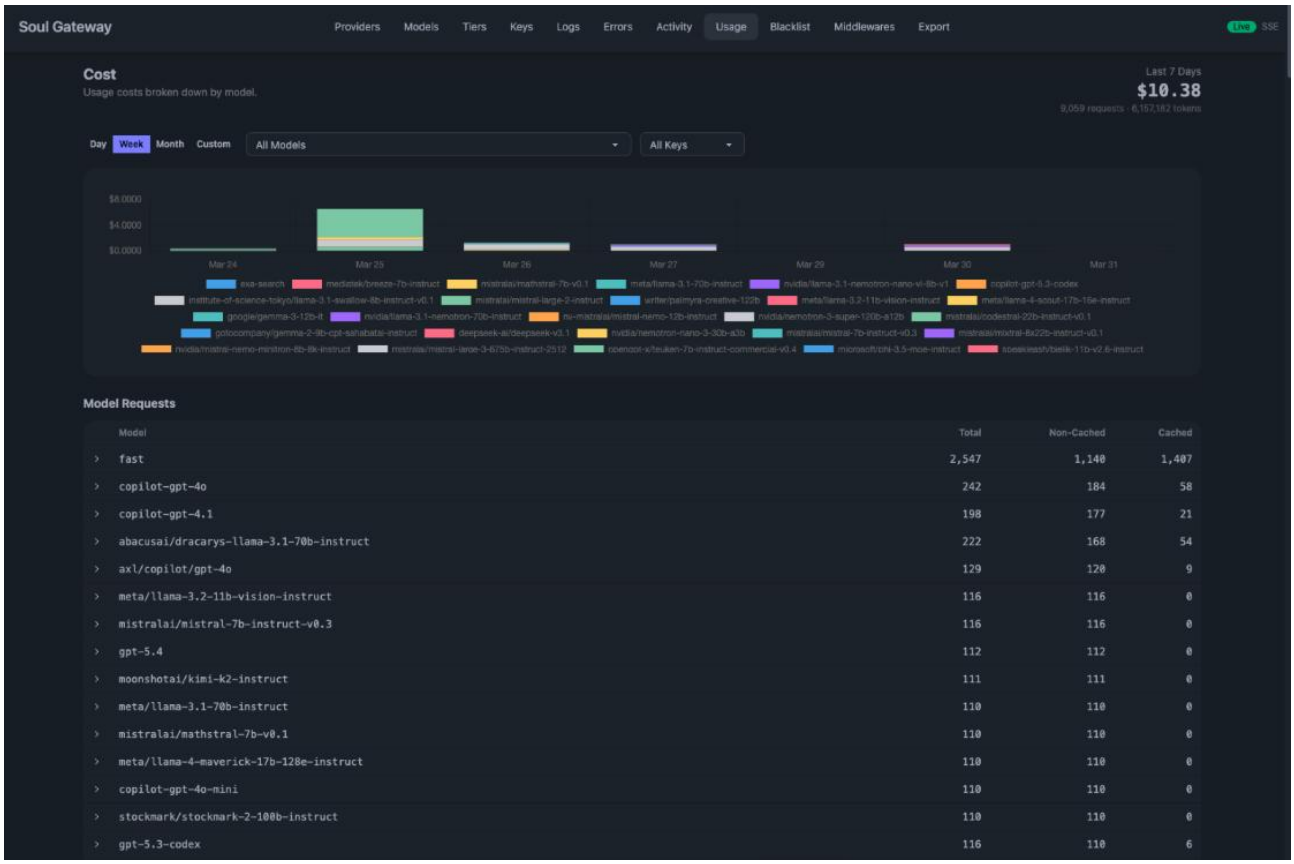

Figure 228 Soul Gateway – Usage

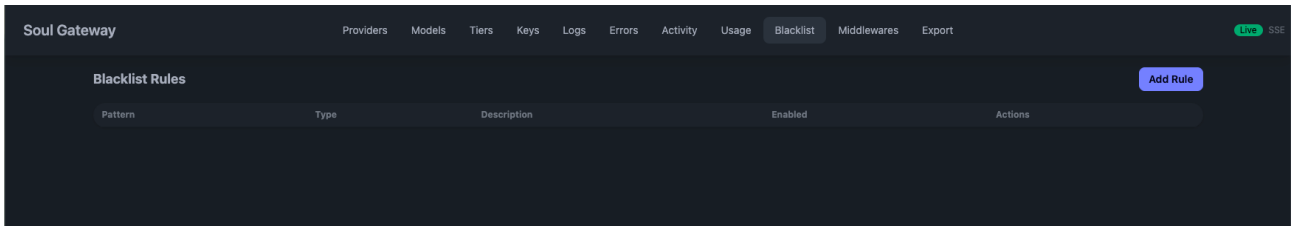

Figure 239 Soul Gateway – Blacklist

Funded by the European Union

This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

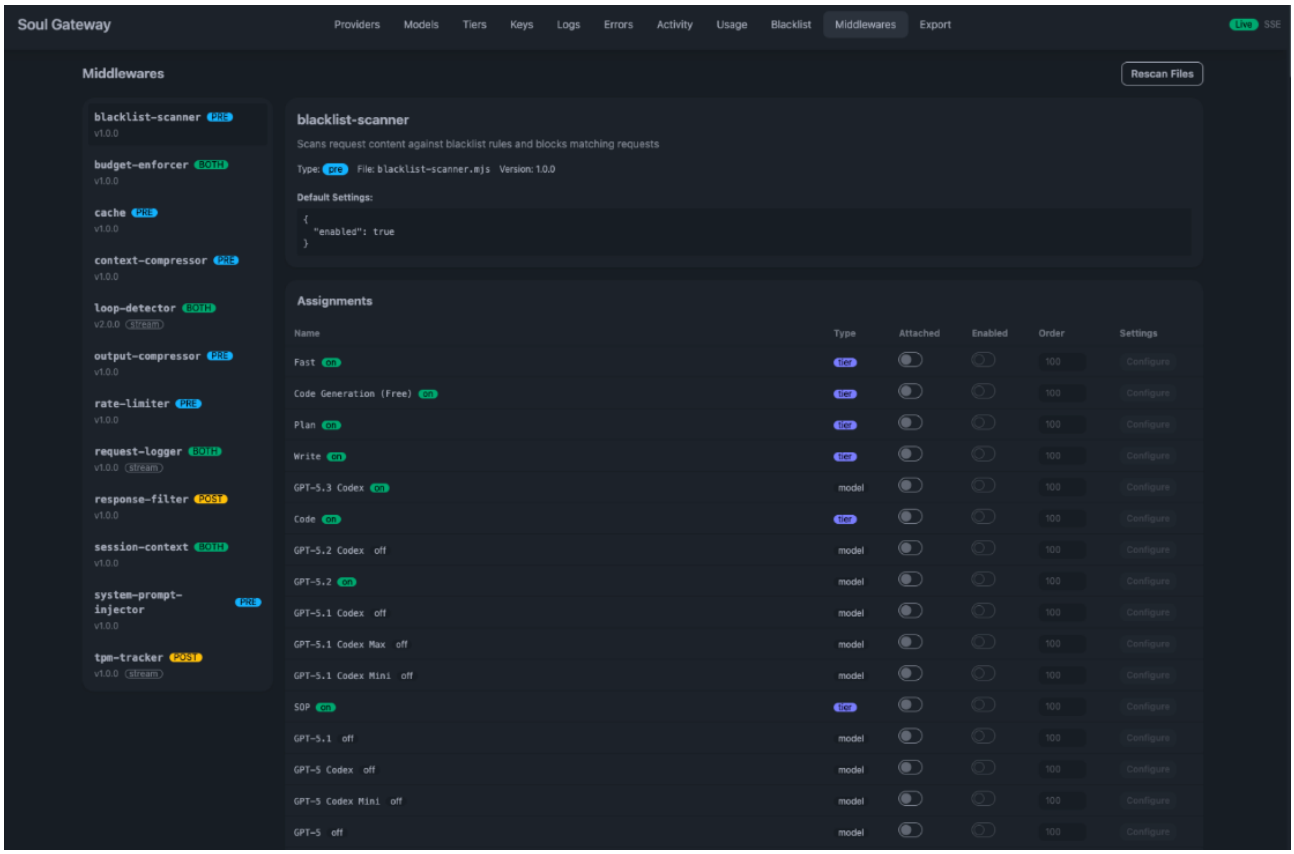

Figure 2410 Soul Gateway – Middlewares

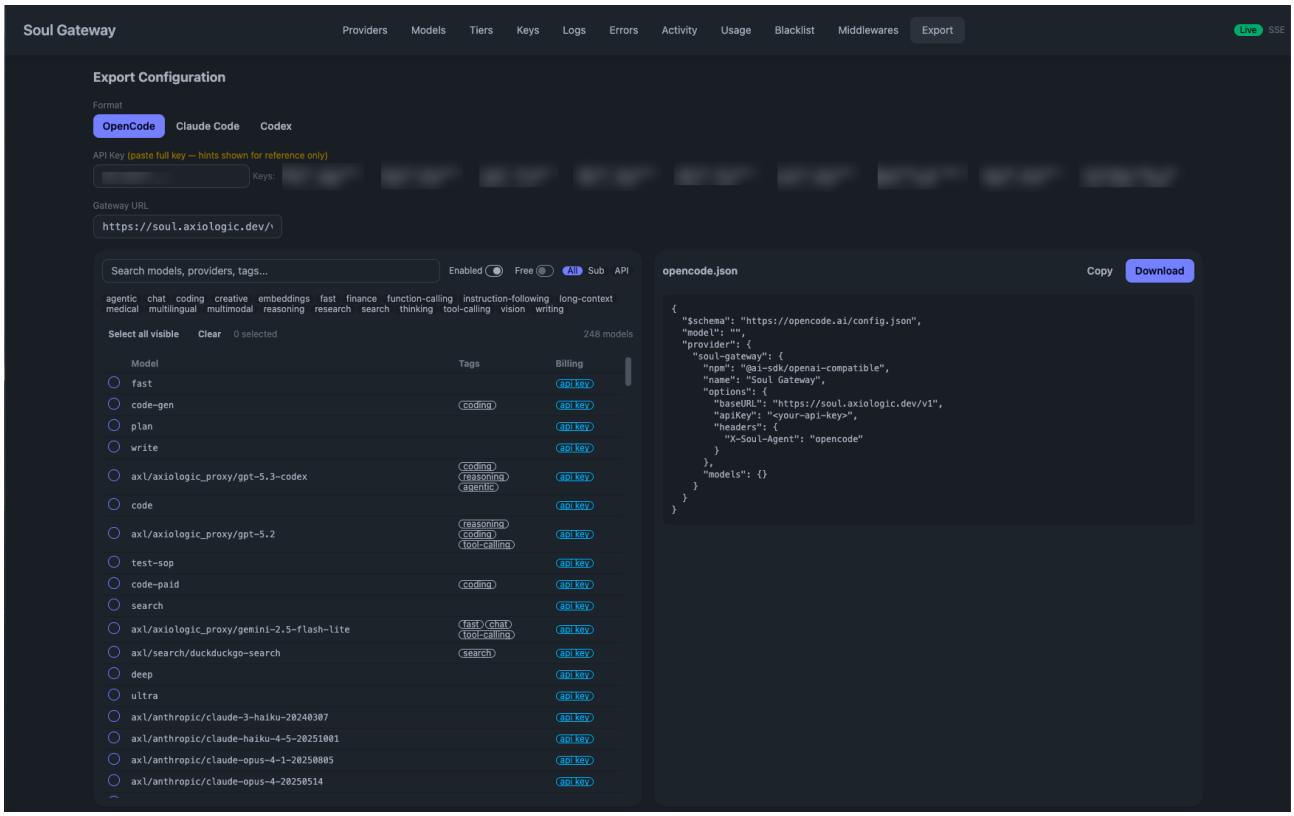

Figure 2511 Soul Gateway – Export Configuration

9 REFERENCES

ACHILLES Agent Library, 2026a, https://outfinityresearch.github.io/AchillesAgentLib/

ACHILLES Agent Library, 2026b. Agentic Sessions, https://outfinityresearch.github.io/AchillesAgentLib/agentic-sessions.html

ACHILLES Agent Library, 2026c. Anthropic Skills, https://outfinityresearch.github.io/AchillesAgentLib/anthropic-skills.html

ACHILLES Agent Library, 2026d. Code Skills, https://outfinityresearch.github.io/AchillesAgentLib/code-skills.html

ACHILLES Agent Library, 2026e. DBTable Skills, https://outfinityresearch.github.io/AchillesAgentLib/dbtable-skills.html

ACHILLES Agent Library, 2026f. Dynamic Code Generation Skills, https://outfinityresearch.github.io/AchillesAgentLib/dynamic-code-generation-skills.html

ACHILLES Agent Library, 2026g. MCP Skills, https://outfinityresearch.github.io/AchillesAgentLib/mcp-skills.html

ACHILLES Agent Library, 2026h. Orchestration Skills, https://outfinityresearch.github.io/AchillesAgentLib/orchestration-skills.html

Ahn, M., Brohan, A., Brown, N., et al., 2022. Do As I Can, Not As I Say: Grounding Language in Robotic Affordances, https://proceedings.mlr.press/v205/ichter23a/ichter23a.pdf

Alboaie S., et al., 2016. Executable choreographies applied in OPERANDO. Computer Science Journal of Moldova, https://www.math.md/files/csjm/v24-n3/v24-n3-(pp417-436).pdf

Alboaie S., 2026a. Executable Natural Language, https://agisystem2.com/MRP/executable-natural-language.html

Alboaie S., 2026b. Meta Rational Pragmatics, https://agisystem2.com/MRP/index.html

Alboaie S., 2026c. Meta-Rational Pragmatics, https://agisystem2.com/MRP/meta-rational-pragmatics.html

Alboaie S., 2026d. Meta-Rational Pragmatics in Context, https://agisystem2.com/MRP/meta-rational-pragmatics-in-context.html

Funded by the European Union

This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

Alboaie S., 2026e. Meta-Rationality: Psychological Roots, Philosophical Lineages, and the Executable Turn, https://agisystem2.com/MRP/meta-rationality-psychology-philosophy-executable-pragmatics.html

Alboaie S., 2026f. Meta-Rational Pragmatics, Semantic Pluralism, and Executable Formalization in the Special Sciences, https://agisystem2.com/MRP/meta-rational-pragmatics-semantic-pluralism-executable-formalization.html

Alboaie S., 2026g. MRP-VM: An Implementation Path, https://agisystem2.com/MRP/mrp-vm-implementation-path.html

Alboaie S., 2026h. Regime Selection and Tractable Computation as Regime Induction, https://agisystem2.com/MRP/regime-selection-tractable-computation-regime-induction.html

Anthropic, 2024. Building effective AI agents, https://www.anthropic.com/research/building-effective-agents

Anthropic, 2025a. Effective Context Engineering for AI Agents. Anthropic Technical Documentation, https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents

Anthropic, 2025b. Equipping agents for the real world with Agent Skills, https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills

Anthropic, 2026a. Claude Code Overview. Anthropic Technical Documentation, https://docs.anthropic.com/en/docs/claude-code

Anthropic, 2026b. Claude Agent SDK Overview. Anthropic Technical Documentation, https://platform.claude.com/docs/en/agent-sdk/overview

Anthropic, 2026c. Claude Code Quickstart. Anthropic Technical Documentation, https://docs.anthropic.com/en/docs/claude-code/quickstart

Anthropic, 2026d. Claude Code CLI Reference. Anthropic Technical Documentation, https://code.claude.com/docs/en/cli-reference

Anthropic, 2026e. Agent Skills Overview. Anthropic Documentation, https://console.anthropic.com/docs/en/agents-and-tools/agent-skills/overview

Anthropic, 2026f. Demystifying evals for AI agents, https://www.anthropic.com/engineering/demystifying-evals-for-ai-agents

AssistOS, 2026a. Integrated Toolchain, https://www.assistos.org/toolchain.html

AssistOS-AI, 2026b. MRP-VM Prototype Documentation, https://assistos-ai.github.io/mrp-vm-v0/

Funded by the European Union

This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

Beurer-Kellner, L., Fischer, M., & Vechev, M., 2023. Prompting Is Programming: A Query Language for Large Language Models, https://arxiv.org/abs/2212.06094

BMAD Method, 2026a. Agents. Documentation. https://docs.bmad-method.org/reference/agents/

BMAD Method, 2026b. Welcome to the BMad Method. https://docs.bmad-method.org/

BMAD Method, 2026c. Workflow Map. https://docs.bmad-method.org/reference/workflow-map/

Colelough, B. C., & Regli, W., 2025. Neuro-Symbolic AI in 2024: A Systematic Review. Link: https://arxiv.org/html/2501.05435v1

Darif, I., El Boussaidi, G., Kpodjedo, S., & Politowski, C., 2025. Controlled Natural Language for Requirements Specification: A Systematic Literature Review, https://dl.acm.org/doi/10.1145/3778169

de Moura, L., & Bjørner, N., 2008. Z3: An Efficient SMT Solver, https://link.springer.com/chapter/10.1007/978-3-540-78800-3_24

D2.3 - Internal project reference. Deliverable D2.3- Privacy-preserving Learning Report v1

D6.1. Internal project reference. Deliverable D6.1 - ACHILLES IDE Specifications

Gao, L., Madaan, A., Zhou, S., et al., 2023. PAL: Program-aided Language Models, https://arxiv.org/pdf/2211.10435

GitHub, 2026. About GitHub Copilot Coding Agent, https://docs.github.com/copilot/concepts/agents/coding-agent/about-coding-agent

Hugging Face, 2026. Smolagents reference: Agents. Hugging Face Documentation, https://huggingface.co/docs/smolagents/reference/agents

Khattab O., Singhvi A., Maheshwari P., Zhang Z., Santhanam K., Vardhamanan S., Haq S., Sharma A., Joshi T.T., Moazam H., Miller H., Zaharia M., Potts C., 2023. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines, https://arxiv.org/abs/2310.03714

Keycloak, 2026. Open-source identity and access management, https://www.keycloak.org/

Kleyko D., Rachkovskij D.A., Osipov E., Rahimi A., 2021. A Survey on Hyperdimensional Computing aka Vector Symbolic Architectures, Part II: Applications, Cognitive Models, and Challenges, https://arxiv.org/abs/2112.15424

LangChain, 2026a. LangGraph Overview. LangChain Documentation, https://docs.langchain.com/oss/javascript/langgraph/overview

Funded by the European Union

This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

LangChain, 2026b. LangGraph Persistence. LangChain Documentation, https://docs.langchain.com/oss/javascript/langgraph/persistence

LangChain, 2026c. LangGraph Memory. LangChain Documentation, https://docs.langchain.com/oss/javascript/langgraph/memory

Microsoft, 2026. AutoGen GitHub Repository. Microsoft Documentation, https://github.com/microsoft/autogen

Microsoft Research, 2023. Guidance: Control LM Output. Microsoft Research, https://www.microsoft.com/en-us/research/project/guidance-control-lm-output/

OpenAI, 2025. A practical guide to building agents, https://openai.com/business/guides-and-resources/a-practical-guide-to-building-ai-agents/

OpenAI, 2025a. Introducing Deep Research. OpenAI Technical Documentation, https://openai.com/index/introducing-deep-research/

OpenAI, 2025b. Introducing ChatGPT Agent: Bridging Research and Action, https://openai.com/index/introducing-chatgpt-agent/

OpenAI, 2025c. Introduction to Deep Research in the OpenAI API. OpenAI Technical Documentation, https://developers.openai.com/cookbook/examples/deep_research_api/introduction_to_deep_research_api/

OpenAI, 2025d. ChatGPT Agent Overview. OpenAI Technical Documentation, https://help.openai.com/en/articles/11752874-chatgpt-agent

OpenAI, 2025e. New Tools for Building Agents. OpenAI Blog, https://openai.com/index/new-tools-for-building-agents/

OpenAI, 2025f Building agents, https://developers.openai.com/tracks/building-agents/

OpenAI, 2026c. OpenAI Agents SDK, https://openai.github.io/openai-agents-python/

OpenAI, 2026d. Using Skills to Accelerate OSS Maintenance. OpenAI Blog, https://developers.openai.com/blog/skills-agents-sdk/

OpenAI, 2026e. Agent Skills and related skills documentation. OpenAI Documentation, https://developers.openai.com/api/docs/guides/tools-skills

OpenClaw, 2026a. ACP Agents. OpenClaw Technical Documentation, https://docs.openclaw.ai/tools/acp-agents

Funded by the European Union

This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

OpenClaw, 2026b. ClawHub. OpenClaw Technical Documentation, https://docs.openclaw.ai/tools/clawhub

OpenClaw, 2026c. Discovery and Transports. OpenClaw Technical Documentation, https://docs.openclaw.ai/gateway/discovery

OpenClaw, 2026d. OpenClaw Documentation Overview. OpenClaw Technical Documentation, https://docs.openclaw.ai

OpenClaw, 2026e. Gateway Runbook. OpenClaw Technical Documentation, https://docs.openclaw.ai/gateway

OpenClaw, 2026f. Hooks. OpenClaw Technical Documentation, https://docs.openclaw.ai/automation/hooks

OpenClaw, 2026g. Multi-Agent Routing. OpenClaw Technical Documentation, https://docs.openclaw.ai/concepts/multi-agent

OpenClaw, 2026h. Plugin Internals. OpenClaw Technical Documentation, https://docs.openclaw.ai/plugins/architecture

OpenClaw, 2026i. Security. OpenClaw Technical Documentation, https://docs.openclaw.ai/gateway/security

OpenClaw, 2026j. Session Management. OpenClaw Technical Documentation, https://docs.openclaw.ai/concepts/session

OpenClaw, 2026k. Skills. OpenClaw Technical Documentation, https://docs.openclaw.ai/tools/skills

OpenClaw, 2026l. Sub-Agents. OpenClaw Technical Documentation, https://docs.openclaw.ai/tools/subagents

OpenClaw, 2026m. OpenClaw Partners with VirusTotal for Skill Security. OpenClaw Technical Documentation, https://openclaw.ai/blog/virustotal-partnership

OpenCode, 2026. Agent Skills, https://opencode.ai/docs/skills/

OpenTelemetry, 2026. Observability Primer, https://opentelemetry.io/docs/concepts/observability-primer/

Outlines Documentation, 2026. Outlines Documentation. Outlines Docs, https://dottxt-ai.github.io/outlines/latest/

Ploinky, 2026a. Architecture, https://ploinky.com/architecture.html

Funded by the European Union

This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.

Ploinky, 2026b. CLI Reference, https://ploinky.com/cli-reference.html

Ploinky, 2026c. Ploinky home and platform overview, https://ploinky.com/index.html

Schick, T., Dwivedi-Yu, J., Dessì, R., et al., 2023. Toolformer: Language Models Can Teach Themselves to Use Tools, https://arxiv.org/abs/2302.04761

Tom's Hardware, 2026. Malicious OpenClaw 'skill' targets crypto users on ClawHub — 14 malicious skills were uploaded to ClawHub last month, https://www.tomshardware.com/tech-industry/cyber-security/malicious-moltbot-skill-targets-crypto-users-on-clawhub

Wan, Z., Liu, C.-K., Yang, H., et al., 2024. Towards Cognitive AI Systems: A Survey and Prospective on Neuro-Symbolic AI, https://arxiv.org/abs/2401.01040

Wang L., Ma C., Feng X., et al., 2023. A Survey on Large Language Model based Autonomous Agents. arXiv preprint, https://arxiv.org/abs/2308.11432

Xu, S., Li, Z., Mei, K., & Zhang, Y., 2024. AIOS Compiler: LLM as Interpreter for Natural Language Programming and Flow Programming of AI Agents, https://arxiv.org/abs/2405.06907

Yao, S., Zhao, J., Yu, D., et al., 2023. ReAct: Synergizing Reasoning and Acting in Language Models, https://arxiv.org/abs/2210.03629

Funded by the European Union

This project has received funding from the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101189689.