

arXiv-MAS

A Provider-Agnostic Multi-Agent System for Faithful Summarization and Trend Analysis of the arXiv Corpus

MSc in Applied & Advanced Artificial Intelligence
Master's Final Project

Author: Victor Ruben Lacort Pellicer

Tuthor: Emilio Soria Olivas

July 10, 2026

Abstract

The volume of research published on arXiv exceeds what any individual can follow, and the two responses it calls for — a faithful summary of each paper and a higher-order analysis of the findings and tendencies across them — are of a different nature and are poorly served by a language model alone. We present ARXIV-MAS, a multi-agent system that performs both over a shared knowledge base and that is organized around a single thesis: that deterministic orchestration and Large Language Model (LLM) reasoning are separate concerns, and that a system which keeps them apart — reserving the model for the interpretation and the production of meaning, and everything else for ordinary code — is more testable, more economical, more reproducible and more honest. The knowledge base is a store of Open Knowledge Format (OKF) documents held to a strict, validated schema. Two subsystems act upon it: a Summarizer of map-reduce shape, which ingests, deduplicates, summarizes and rolls up; and an Analyzer that combines a deterministic coverage planner, an antagonistic Mixture-of-Agents deliberation and a forecast-calibration loop. The model and the orchestration are each reached behind a provider-agnostic abstraction, so that neither is load-bearing. We describe the architecture, its resilience and operational surface, and a prototype whose deterministic core is complete and tested and whose reasoning layer runs against a locally served model, chaining the whole process from a raw submission to a calibrated tendency.

Contents

I Context and Foundations **1**

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Objectives and scope . . . 1
- 1.3 Contributions . . . 2
- 1.4 Structure of the report . . . 2

2 Technical Context **2**
- 2.1 From language model to agent . . . 3
- 2.2 Multi-agent systems . . . 3
- 2.3 Protocols for tools and for agents . . . 5
- 2.4 Memory in multi-agent systems . . . 5
- 2.5 Orchestration patterns and the calibration of forecasts . . . 5

3 Design Decisions **6**
- 3.1 Separation of deterministic orchestration from LLM reasoning . . . 6
- 3.2 A single source of truth . . . 6
- 3.3 Provider independence . . . 7
- 3.4 Fidelity, provenance and auditability . . . 7
- 3.5 Honesty about coverage . . . 7
- 3.6 From principles to concrete decisions . . . 8

II Architecture **9**

4 Architecture Overview **9**
- 4.1 The blackboard: the Knowledge Base . . . 9
- 4.2 The knowledge sources: MAS_i and MAS_o . . . 10
- 4.3 The life cycle of a piece of knowledge . . . 10
- 4.4 Control paths . . . 10

5 The Knowledge Substrate **11**
- 5.1 OKF documents and an orthogonal frontmatter . . . 11
- 5.2 Naming, layout and Concept IDs . . . 12
- 5.3 The concept graph and the attention signal . . . 13
- 5.4 Provenance and lineage . . . 13

III The Subsystems **15**

6 The Summarizer (MAS_i) 15
6.1 Ingestion and selection . . . 15
6.2 The map: per-article summaries . . . 15
6.3 The reduce: daily digests and the ledger . . . 16
6.4 Rollups: completeness, modes and empty days . . . 16

7 The Analyzer (MAS_o) 17
7.1 The coverage planner . . . 17
7.2 Assembly and conditional compression . . . 18
7.3 Findings . . . 18
7.4 The Mixture-of-Agents . . . 18
7.5 The calibration loop . . . 19
7.6 The scope of economic claims . . . 19

IV Cross-Cutting Layers 20

8 The Agent Layer 20
8.1 The registry . . . 20
8.2 The model abstraction . . . 20
8.3 Prompts and the house style . . . 20
8.4 Output contracts and validation . . . 21
8.5 Tool-using agents . . . 21
8.6 Retry and fall-back . . . 21

9 Orchestration 22
9.1 The pipeline as a provider-agnostic DAG . . . 22
9.2 The execution engine . . . 23
9.3 Providers and the scheduling seam . . . 23
9.4 Execution versus scheduling . . . 23

10 Resilience, Operations and Observability 23
10.1 The failure model . . . 24
10.2 Rate limiting and retries . . . 24
10.3 Exit codes and monitoring . . . 24
10.4 Verbosity and logging . . . 25
10.5 Configuration and secrets . . . 25

V Production Concerns 26

11 Security 26
11.1 Why the attack surface widens . . . 26
11.2 Threats particular to agentic systems . . . 26
11.3 Defence in depth . . . 26

12 Observability **27**
12.1 Traces and spans . . . 27
12.2 What to monitor . . . 27
12.3 In this system . . . 27

13 Evaluation **28**
13.1 Three layers of evaluation . . . 28
13.2 Methods and metrics . . . 28
13.3 In this system . . . 29

14 Production Design Patterns **29**
14.1 Workflow, capability and reliability patterns . . . 29
14.2 How they combine, and how they fail . . . 30

VI Implementation and Evaluation **31**

15 Implementation **31**
15.1 The stack . . . 31
15.2 Package structure . . . 32
15.3 Deployment . . . 32

16 Testing and Validation **32**
16.1 Strategy . . . 32
16.2 What is covered . . . 33
16.3 Reproducibility and boundaries . . . 33

17 Discussion **33**
17.1 Trade-offs revisited . . . 34
17.2 Limitations . . . 34
17.3 Threats to validity . . . 34
17.4 Generality . . . 34

18 Related Work **35**
18.1 Blackboard architectures and multi-agent systems . . . 35
18.2 Tool-using and retrieval-augmented agents . . . 35
18.3 Multi-agent orchestration frameworks . . . 35
18.4 Aggregation and deliberation . . . 35
18.5 Summarization and forecast calibration . . . 36

19 Future Work **36**
19.1 The reasoning layer . . . 36
19.2 The knowledge graph . . . 36
19.3 External signals . . . 36
19.4 Scale and evaluation . . . 37

19.5 Provider integrations 37

20 Conclusion **37**

Appendices **38**

A The Agent Registry **38**

B OKF Schema Reference **39**

C CLI and Runbook **39**

D Knowledge-Base Layout **40**

E arXiv Category Taxonomy **40**

F Glossary **45**

References **47**

Part I
Context and Foundations

1 Introduction

1.1 Motivation

The volume of research published on arXiv has, for some time now, exceeded what any individual can follow by reading. In an active category the daily submissions are counted by the hundreds, and even a specialist confined to a narrow subfield cannot keep pace with the primary literature, let alone with the relations and tendencies that emerge across it. The natural response — to summarize — is necessary but, on its own, insufficient: a summary written for a human reader optimizes for engagement and brevity, whereas what a downstream analysis requires is fidelity and density, so that no technical signal is lost before the analysis begins. The problem we address is therefore twofold. On the one hand, we need faithful, machine-consumable summaries of the individual papers; on the other, we need a higher-order layer that reads those summaries and reasons over them, extracting the concrete findings that the literature establishes and the tendencies — with their expected and bounded economic or industrial impact — that it suggests.

These two needs are of a different nature, and it is worth stating the difference at the outset, because it shapes the whole system. Summarization is, in essence, a reduction that must not distort; analysis is a construction that must remain honest about its evidence. A system that serves both must keep the summaries faithful enough to be trusted as input, and must make the analyses auditable enough to be believed as output. Neither is achieved by a language model alone: the model is indispensable where meaning is involved, but a great part of what such a system does — deciding which material to read, avoiding duplicated work, resolving what has already been produced, keeping the store clean and traceable — is not a matter of meaning at all, and is done better, more cheaply and more reliably by ordinary code.

1.2 Objectives and scope

The objective of ARXIV-MAS is to realize this two-layered process as a multi-agent system built around one organizing idea: that deterministic orchestration and Large Language Model (LLM) reasoning are separate concerns and should be kept apart, a thesis we develop in section 3. Concretely, the system ingests the arXiv submissions of a category and date, summarizes each paper, and rolls the summaries up into a hierarchy of temporal digests; and, over a requested range, it analyses the accumulated knowledge into findings and tendencies, deliberated by a Mixture-of-Agents and tempered by a calibration loop that scores its own past predictions. All of this is deposited in a shared knowledge base whose format is deliberately plain and whose schema is deliberately strict.

We are explicit about what the system is and is not. It is a working prototype: its deterministic core is complete and tested, its reasoning layer is functional against a locally served model,

and its orchestration chains the whole process from end to end. It is not, at this stage, a production deployment: the prompts encode a defined style but have not been tuned against output at scale, the economic-impact analysis is bounded on purpose to research-internal signals, and several components are marked as first-version simplifications, which we discuss in section 17. The orientation of the report, nonetheless, is towards real production: the architecture, the resilience model and the operational surface are designed for it, and the report describes them as such.

1.3 Contributions

The contributions of this work are the following. *(i)* A design in which the deterministic and the semantic are separated cleanly, and in which the reasoning model is confined to the tasks that only it can perform; we argue that this separation is not merely an implementation convenience but the property from which testability, cost-efficiency and reproducibility follow. *(ii)* A shared knowledge base in the Open Knowledge Format, with an orthogonal and strongly-typed schema that keeps the store at once human-readable and machine-consumable, together with a concept graph whose accumulated attention over time serves, in itself, as a signal of trend. *(iii)* Two subsystems — a Summarizer of map-reduce shape, and an Analyzer that combines a deterministic coverage planner, a Mixture-of-Agents deliberation and a forecast-calibration loop. *(iv)* Provider-agnostic model and orchestration layers, such that neither the language model nor the orchestrator is load-bearing and either may be substituted through a single point of configuration. *(v)* A resilience and operational model — per-item isolation, non-blocking auxiliaries, retries with back-off, meaningful exit codes and tiered verbosity — appropriate to unattended, scheduled execution.

1.4 Structure of the report

The report is organized in five parts. Part I establishes the context and the design philosophy: the present introduction, the technical background in section 2, and the principles in section 3. Part II describes the architecture as a whole (section 4) and the knowledge substrate on which it rests (section 5). Part III is devoted to the two subsystems, the Summarizer (section 6) and the Analyzer (section 7). Part IV treats the cross-cutting layers: the agent layer (section 8), the orchestration (section 9) and the resilience and operational concerns (section 10). Part V covers the implementation (section 15), the testing strategy (section 16), a discussion of the decisions and their limits (section 17), the related work (section 18), the lines of future work (section 19) and the conclusion (section 20). A set of appendices collects the full agent registry, the schema reference, an operator's runbook, the knowledge-base layout and a glossary.

2 Technical Context

This section situates the system within the technology it is built upon and the field to which it belongs. We proceed from the single agent to the multi-agent system, then to the protocols by which agents reach their tools and one another, to the mechanisms by which such systems

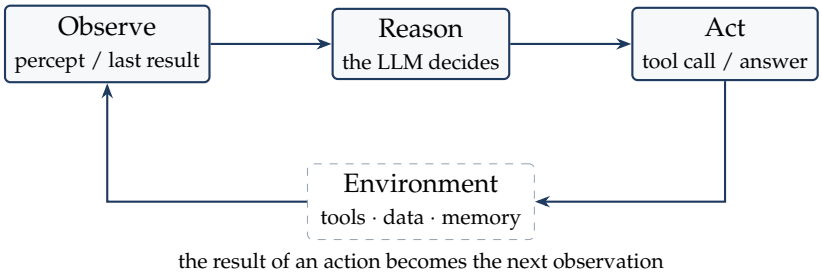


Figure 1: The observe–reason–act (ReAct) loop of an agent. The model reasons; a tool acts on the environment; the result is observed and feeds the next cycle. The loop runs until the goal is met or a bound is reached.

remember, and finally to the orchestration patterns and the evaluation of forecasts. The treatment is deliberately general; the choices ARXIV-MAS makes among these options are the subject of section 3.

2.1 From language model to agent

A large language model, on its own, reasons but does not act: it maps text to text, and cannot of itself consult a source, run a computation or write a file. An *agent* is what results when such a model is given tools with which to act, the means to observe the result of its actions, a memory of what it has done, and a measure of autonomy to decide its own next step. What separates an agent from a chatbot or a fixed automation is the *dynamic decision loop*: a chatbot answers each turn and waits for the human to choose the next, an automation follows a script fixed in advance, whereas an agent pursues a goal and chooses its intermediate steps itself, re-evaluating after each observation.

This loop is usually described, after the pattern that named it, as *ReAct* — reasoning and acting interleaved. The agent observes (the environment, or the result of its last action), reasons (the model decides what to do, and why), and acts (it invokes a tool, or emits an answer); and the result of the action becomes the next observation, as fig. 1 depicts. The cycle repeats until the goal is met or a bound — of steps, of time, of budget — is reached. Its essential components are five: the *model*, which reasons; the *tools*, which act; the *memory*, which persists what has been done; a *planner*, which decomposes a goal into steps; and a *control loop*, which governs how many times the cycle runs and when it stops. Autonomy is best seen as a spectrum rather than a property, from a model that answers once without tools, through an agent that chains tool calls within a bounded task, to long-running systems that manage their own memory, correct themselves, and ask a human for help when the action is critical or irreversible.

2.2 Multi-agent systems

A single capable agent can, in principle, address a broad task, but it does so with one prompt, one set of tools and one context, and its reasoning degrades as that context fills with heterogeneous material. A *multi-agent system* (MAS) instead distributes the task among several agents, each with a smaller and more specialized prompt, toolset and context, that cooperate towards a common end. The gain is not only speed but clarity: each agent reasons over less, and less

noisy, material.

How the agents are arranged — the *topology* — is the most consequential design choice in a MAS, for it determines how information flows, who decides what, and where the bottlenecks lie. Table 1 summarizes the common topologies, from the centralized hierarchy of an orchestrator over specialized workers, through the decentralized network of peers that debate, the fixed sequence of a pipeline and the explicit control-transfer of hand-offs, to the recursive nesting of holonic teams within teams.

Topology	**Coordination**	**Predictability**	**Typical use**
Hierarchical	centralized	high	research then report
Network / P2P	decentralized	low	debate, brainstorming
Pipeline	fixed sequence	very high	staged data processing
Hand-offs	control transfer	medium–high	customer support
Holonic	mixed / recursive	medium	large simulated organizations

Table 1: Commonly used multi-agent topologies.

Beyond who talks to whom, a MAS must decide *how* its agents communicate: by direction (point-to-point, broadcast, or publish–subscribe), by synchrony (synchronous, easier to reason about, versus asynchronous, more efficient but prone to races), and by the channel itself. One channel deserves particular mention: rather than passing messages directly, the agents may share a common data space — a *blackboard* — that all of them read and, in some cases, write. This avoids the quadratic explosion of point-to-point messages when many agents must see the same context, and it is, as section 4 will show, the mechanism this system adopts.

The hardest engineering problem in a MAS is the management of *state* — the history, the intermediate results, the decisions taken — because a model is stateless between calls and each agent's context is finite. Passing every agent the whole history saturates its context and degrades its reasoning; passing too little loses information. The strategies are correspondingly familiar from distributed systems: state local to each agent versus a shared blackboard; passing state by value (copying it) versus by reference (an identifier that the agent resolves only when needed); *checkpointing* the state after each significant step, so that a failure resumes rather than restarts; and the separation of short-term memory (the current task) from long-term memory (knowledge that must outlive a session). With shared or parallel state comes the problem of consistency — two agents disagreeing about what has been done — addressed, as in classical concurrency, by locks, by versioning, or by a single owner per piece of state.

Such systems inherit the difficulties of distributed systems and of human organizations alike: cascading errors, in which an early mistake is amplified downstream; context drift, in which meaning is lost as material is relayed from agent to agent; the combinatorial growth of communication, quadratic in the number of agents; loops and deadlocks in undisciplined debate; and the accumulation of cost and latency, since every hand-off is at least one further model call. A well-designed MAS is, in large part, a set of defences against these failure modes — a theme to which section 10 and the production concerns of Part V return.

2.3 Protocols for tools and for agents

As agents and their tools proliferate across frameworks and organizations, open protocols have emerged to let them interoperate. The one central to this system is the Model Context Protocol (MCP) [Anthropic 2024], which standardizes how an agent discovers and uses external tools and data sources behind a uniform interface; we use it to expose the Knowledge Base (section 5). A complementary protocol, Agent-to-Agent (A2A), standardizes how an agent discovers and communicates with *other* agents, even across frameworks, with interoperability between organizations as its aim rather than any single workflow; related efforts — ACP, and the more decentralized Agent Network Protocol — extend the same idea to open networks of agents. Where MCP connects an agent to its tools, A2A and its kin connect agents to one another; this system relies on the former and not, at present, on the latter, since its agents cooperate through the shared blackboard rather than by direct negotiation.

2.4 Memory in multi-agent systems

Because a model does not remember, memory in an agentic system is an explicit construction, and several mechanisms are in common use, distinguished by what they store and how they retrieve it. The simplest, and the most widely adopted in practice, is *file-based* memory: the system reads and writes ordinary documents, durable, inspectable and independent of any particular database — a mechanism on whose virtues this system leans heavily. *Vector stores* keep embeddings of past material and retrieve it by semantic similarity, which is the basis of retrieval-augmented generation [Lewis et al. 2020]. *Shared* memory realizes the blackboard already described, a common space through which agents coordinate. *Knowledge graphs* store entities and their relations and, in their temporal variants, track how those relations change over time — something neither files nor vectors capture well. And *operating-system-style* memory pages information in and out of the finite context as a virtual memory would, to sustain very long tasks. These mechanisms are not exclusive: a system may, and this one does, combine several — files as the durable substrate, a blackboard for coordination, and a concept graph for the relations among ideas (section 5).

2.5 Orchestration patterns and the calibration of forecasts

Finally, three orchestration patterns and one evaluation technique recur in what follows. The first pattern is *map-reduce* [Dean and Ghemawat 2008]: an independent *map* over many items followed by a *reduce* that combines the partial results — the shape of summarizing many papers and then synthesizing a digest. The second is the *Mixture-of-Agents* (MoA) [Wang et al. 2024]: several agents address the same question from deliberately different stances, and an aggregator merges their outputs, surfacing considerations a single pass would miss. The third is the *pipeline with a feedback loop*, in which a later stage produces information that conditions subsequent runs. As for evaluation, a forecast is of little worth unless one intends to check it: a probabilistic prediction is scored by the *Brier score* [Brier 1950], the mean squared difference between the stated probability and the realized outcome (zero is perfect, 0.25 the score of an uninformed one-half guess), and a qualitative prediction by a weighted hit-rate over categorical

outcomes. With this vocabulary in place we turn, in section 3, to the decisions by which the system selects among these options and composes them into a coherent whole.

3 Design Decisions

The architecture of ARXIV-MAS does not emerge from the accumulation of features, but from a small number of principles that we have applied consistently across every layer. We make these principles explicit before describing the components, because each later decision — from what a coverage planner is allowed to compute, to the way in which an agent is retried after a failure — is a consequence of them. The reader may therefore use this section as a lens: whenever a mechanism appears more elaborate than strictly necessary, its justification is almost always one of the principles that follow.

3.1 Separation of deterministic orchestration from LLM reasoning

Design Principle 1 (Deterministic orchestration versus LLM reasoning)**.** Everything that can be decided by an algorithm is decided by an algorithm; the Large Language Model (LLM) is reserved for the single class of tasks that only it can perform, namely to compress and to reason about meaning.

This is the thesis of the whole system, and the line against which every component is placed. On the one hand, tasks such as selecting which files tile a date range, deduplicating a cross-listed paper, checkpointing what has already been produced, deriving a canonical file name, or validating a document against its schema, are deterministic: they admit an exact and testable definition, and delegating them to a probabilistic model would only introduce cost, latency and non-reproducibility, without any benefit. On the other hand, tasks such as writing a faithful summary of a paper, or weighing the arguments for and against the adoption of a technique, are irreducibly semantic; no amount of control logic can replace them.

The consequences of drawing this line clearly are pervasive. The deterministic layer becomes unit-testable in isolation and offline, without a model server and without a network — which is, in fact, how the majority of the system is validated (see section 16). The reasoning layer, in turn, is confined to well-delimited nodes with typed inputs and outputs, so that a low-quality or failed response is contained rather than propagated. Cost and reproducibility improve for the same reason: the expensive, stochastic component is invoked only where it is indispensable. Throughout the report we will return repeatedly to this criterion — *on which side of the line does this fall?* — as the first question asked of every mechanism.

3.2 A single source of truth

Design Principle 2 (A single source of truth)**.** The behaviour of the multi-agent system is described in one declarative artifact, the agent registry, rather than dispersed across the code.

The models and their tiering, the deterministic toggles (rollup mode, prediction mode, economic scope, concurrency, retries), and the full roster of agents — each with its prompt,

its tools and its output contract — are declared together in a single YAML registry. This concentration has concrete consequences: swapping the model of one agent, or of all of them at once, is a one-line change; the operator reasons about the system from one file; and the code contains no hidden configuration that could contradict the declared intent. The registry is, in a sense, the interface between the engineer who operates the system and the machinery that executes it.

3.3 Provider independence

Design Principle 3 (Provider independence)**.** Neither the choice of language model nor the choice of orchestrator is allowed to permeate the system; both are accessed behind an abstraction and remain interchangeable.

Substitution is admitted along two independent axis. For models, a uniform chat-model interface allows to bind an agent to a *capability* rather than to a vendor: the development configuration uses a local model served by Ollama, and a production configuration may point the same agent to an external API without any change to the code. For orchestration, the pipeline is expressed as a provider-agnostic graph, executed by a shared local engine, while each provider — Hermes by default, Openclaw as an alternative — contributes only the translation to its native workflow and the scheduling. The reason for which we insist on this independence is not merely portability: the value of the system resides in its architecture and in its knowledge base, and neither of the two should be held hostage by a particular vendor.

3.4 Fidelity, provenance and auditability

Design Principle 4 (Fidelity, provenance and auditability)**.** Every derived document is validated against an explicit schema before it is written, and records who produced it, with which model, and from which sources.

The Knowledge Base (KB) is meant to be consumed by other programs and to accumulate over time; it must therefore remain clean and traceable. Two mechanisms serve this end. First, a typed output contract validates each document before it reaches the disk, so that malformed content never enters the KB. Second, a provenance block — agent, model, prompt version and timestamp — together with an explicit lineage — the identifiers of the sources from which the document derives — accompany every generated file. Auditability is thus not an afterthought but a property of the format itself, and it is precisely what makes the analytical claims of the system falsifiable: a tendency, a prediction or a concept node can always be traced back to the material that produced it.

3.5 Honesty about coverage

Design Principle 5 (Honesty about coverage)**.** The system never fabricates coverage; an incomplete period is reported as incomplete, and a rollup is synthesized only when all of its constituents are present.

A summary that silently omits part of its material is worse than one that declares the omission, because a downstream analyzer cannot distinguish the two. For this reason the rollup machinery builds a monthly — or quarterly, or yearly — digest only when every one of its children is present, recursively down to the daily level; an incomplete period is instead decomposed into the individual days that do exist, with the gaps left explicit. A day that legitimately produced no papers is still recorded as covered, so that "incomplete" means "not yet processed" rather than "empty". This honesty about what the KB does and does not contain is what allows the analytical layer to reason without being misled.

3.6 From principles to concrete decisions

The principles above are abstract; it is worth naming, in one place, the concrete decisions they produce, each of which the later sections develop in full. The shared store is realized as a blackboard (section 2) in the Open Knowledge Format, so that the subsystems cooperate through persistent, auditable state rather than by direct messaging (section 4). The Summarizer is given a map-reduce shape, and the Analyzer a pipeline with a Mixture-of-Agents deliberation and a calibration loop (sections 6 and 7), each pattern chosen for the task it fits. Memory is not a single mechanism but a combination — durable files, the blackboard, and a concept graph (section 5). Tools reach the store through the Model Context Protocol, and only for reading. And both the model and the orchestrator are placed behind provider-agnostic abstractions (sections 8 and 9). Each of these is a point at which a general option surveyed in section 2 was resolved into a specific commitment, always on the same side of the line that separates the deterministic from the semantic.

Summary

Taken together, these five principles describe a system in which the deterministic and the semantic are kept apart and each is done well; in which the configuration is declarative and centralized; in which no vendor is load-bearing; and in which every claim is traceable and every gap is visible. The sections that follow may be read as the elaboration of these commitments into concrete mechanisms.

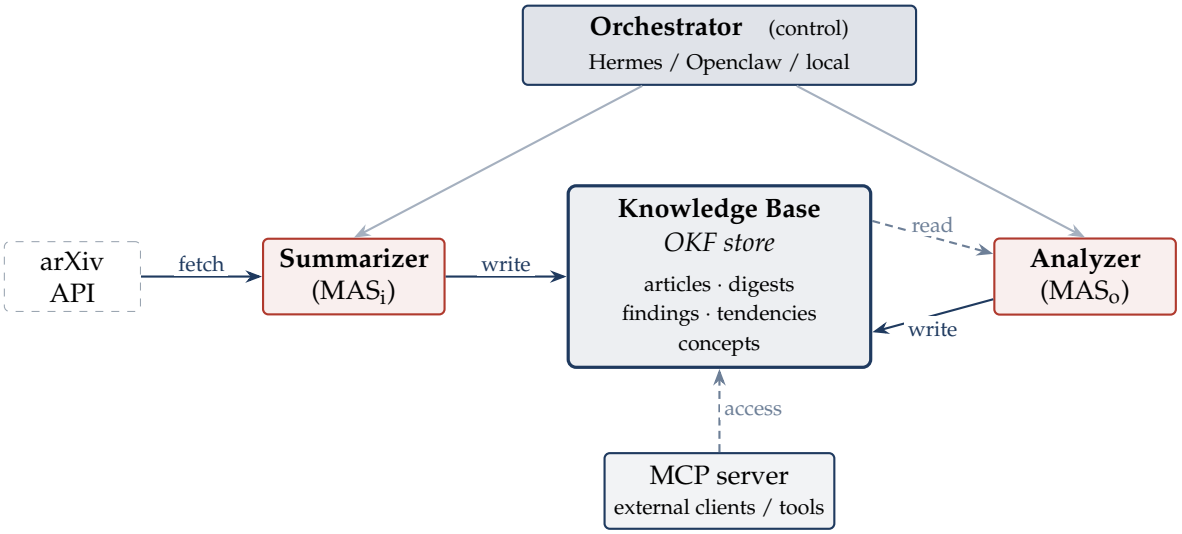


Figure 2: The blackboard architecture. The OKF Knowledge Base is the shared blackboard; the Summarizer (MAS$_i$) and the Analyzer (MAS$_o$) act as knowledge sources that write into it and read from it; the orchestrator provides control; and the MCP server exposes the KB to external clients and to tool-using agents. Solid edges denote deterministic control and input/output; the LLM-mediated steps live inside the two subsystems.

Part II

Architecture

4 Architecture Overview

With the principles established, we now describe the system as a whole, deferring the internal detail of each component to its corresponding section. ARXIV-MAS is organized as a *blackboard* system: a shared and persistent knowledge store around which several independent producers of knowledge cooperate, coordinated by a control component. Figure 2 depicts this organization, which we introduce in the present section from four complementary points of view — the blackboard itself, the knowledge sources that act upon it, the life cycle of a piece of knowledge, and the control paths that drive the machinery.

4.1 The blackboard: the Knowledge Base

At the centre of the system sits the Knowledge Base, a directory of Open Knowledge Format (OKF) documents that we will describe in detail in section 5. For the purposes of this overview, what matters is its role rather than its format: the KB is not a passive store but the *medium* through which the subsystems communicate. The Summarizer and the Analyzer never call each other directly; they read and write the shared KB, and it is through the state left in it that one becomes the input of the other. This indirection is the defining feature of the blackboard style, and it buys two properties that recur throughout the report. Because the KB persists across executions, operation is incremental and resumable: a run contributes what it can and a later run continues from the accumulated state. And because access is mediated, the same

documents can be reached by an in-process subsystem and by an external client alike — the latter through the Model Context Protocol (MCP) server, which exposes the KB behind a small, uniform interface.

4.2 The knowledge sources: $\mathbf{MAS_i}$ and $\mathbf{MAS_o}$

Two subsystems act as knowledge sources. The *Summarizer* (MAS_i) consumes the external arXiv corpus for a given category and date and produces, in a map-reduce fashion, one faithful summary per article and then a hierarchy of temporal digests (daily, and by rollup monthly, quarterly and yearly). The *Analyzer* (MAS_o) consumes the KB over a requested date range and produces higher-order analyses — the concrete *findings* that the literature establishes, and the *tendencies*, with their falsifiable predictions, that it suggests — by means of a pipeline that includes a Mixture-of-Agents deliberation and a calibration feedback loop. Sections 6 and 7 are dedicated to each in turn.

The two sources are deliberately asymmetric. The Summarizer writes far more than it reads, since its input is external; the Analyzer reads far more than it writes, since it derives its output from what the KB already contains. Yet both are knowledge sources in the blackboard sense: each acts on the shared state without depending on the other, and neither is aware of the internal workings of its counterpart. A third and lighter activity, concept curation, completes the picture: it maintains the shared concept graph from the cross-references that the documents accumulate over time, and it is the point at which the knowledge in the KB begins to describe not only individual papers but the relations among the ideas they contain.

4.3 The life cycle of a piece of knowledge

The system may be read as a single, long transformation of raw submissions into structured and interconnected knowledge. An arXiv paper enters through the Summarizer and becomes an article summary; the summaries of a day are reduced into a daily digest; complete periods are rolled up, on demand or by a scheduled sweep, into monthly, quarterly and yearly digests. When the Analyzer is invoked over a range, the coverage planner selects the coarsest digests that tile it, the corpus so assembled is analysed into findings, and the findings are deliberated by the Mixture-of-Agents into tendencies that carry explicit predictions. Those predictions are, in a later cycle, resolved against the accumulated KB and scored, and the resulting calibration tempers subsequent analyses. Meanwhile, every document deposits links into the concept graph, whose nodes the curator enriches and whose accumulated attention over time constitutes, in itself, a signal of trend. This life cycle is the reason for which the KB is central: it is the thread that connects each stage to the next.

4.4 Control paths

Finally, the machinery can be driven in three ways, which differ in their level of abstraction but share the same underlying operations. The lowest level is *direct programmatic invocation*: a subsystem is built as a graph and invoked on an explicit state. Above it, the *command-line interface* exposes one subcommand per operation — summarize, analyze, rollup, curate, count

— and is the natural entry point for manual runs and for scripting. At the highest level, the *orchestrator* chains the operations into a single flow and, when required, schedules it; this is the subject of section 9. It is worth emphasizing that the control component performs no reasoning of its own: it decides only *when* and *in what order* the knowledge sources act, which is exactly the division of labour that section 3 prescribes.

5 The Knowledge Substrate

Everything in ARXIV-MAS rests on the Knowledge Base, and the properties we require of the system as a whole — fidelity, auditability and incremental operation — are properties we must first secure in the substrate. This section makes precise how knowledge is represented, how it is addressed, how the individual documents become an interconnected graph, and how each document remains traceable to what produced it.

5.1 OKF documents and an orthogonal frontmatter

The KB is a directory of documents in the Open Knowledge Format (OKF): each document is a markdown file preceded by a block of YAML frontmatter. OKF prescribes very little — only a `type` field is mandatory, and any additional keys are preserved rather than rejected — and this minimalism is precisely what allows to layer our own schema on top without leaving the format.

The schema we layer is deliberately *orthogonal*. Alongside a human-facing `type` — a composite label such as "Quarterly analysis", derived automatically and never parsed by the code — each document carries a small set of machine fields that vary independently of one another: the `granularity` (article, daily, monthly, quarterly, yearly), the `kind` (summary or analysis), the `analysis_class` (findings or tendencies, for analyses) and, for the temporal documents, an explicit `interval`. Because these fields are orthogonal, the deterministic layer selects and filters documents by reading structured values, and never by interpreting prose. Listing 1 shows the shape for an article summary and for a temporal analysis. Every document is validated against a typed contract before it is written (Principle 4 in section 3), so that a nonconforming document never reaches the disk.

```yaml
# --- an article summary ---------------------------------------
type: Article summary
granularity: article
kind: summary
category: cs.AI
arxiv_id: 2401.12345v2
resource: https://arxiv.org/abs/2401.12345
salience: 0.7
tags: [mixture-of-experts, sparse-attention]
timestamp: 2026-01-16T02:14:00Z
provenance: {agent: article_summarizer, model: qwen3.6:35b,
             prompt_version: "1.0", generated_at: 2026-01-16T02:14:00Z}
source_ids: [https://arxiv.org/abs/2401.12345]
```

```yaml
concept_links: [/concepts/mixture-of-experts]

# --- a quarterly tendencies analysis (resource omitted) -------
type: Quarterly analysis
granularity: quarterly
kind: analysis
analysis_class: tendencies
category: cs.AI
interval: {start: 2026-01-01, end: 2026-03-31}
coverage: {expected_children: 3, included: 3}
prediction_mode: qualitative
source_ids: [cs.AI/findings/quarterly/2026-Q1, cs.AI/digests/monthly/2026-01]
```

Listing 1: Orthogonal OKF frontmatter: an article summary (with a `resource`) and a derived tendencies analysis (`resource` deliberately omitted).

5.2 Naming, layout and Concept IDs

The identity of a document is its location. Following OKF, the *Concept ID* of a document is simply its path within the store with the `.md` suffix removed; the directory structure is therefore not an organizational convenience but the identity namespace itself. From this decision several conventions follow, all oriented towards names that are stable, free of collisions and lexicographically sortable. Dates are written in ISO-8601 with a four-digit year, so that a lexical ordering coincides with the chronological one; the category is encoded by the top-level folder rather than repeated in each file name; the kind and the analysis class are likewise encoded by folders; and the temporal token that names a digest or an analysis normalizes the interval (a day as `YYYY-MM-DD`, a month as `YYYY-MM`, a quarter as `YYYY-Qn`, a year as `YYYY`, and an arbitrary range as `start_end`). Listing 2 shows a representative layout.

```
kb/
  concepts/                              shared concept-graph nodes (cross-category)
    mixture-of-experts.md
  cs.AI/
    articles/2026/01/15/2401.12345.md  article summary (version-less file name)
    digests/
      daily/2026-01-15.md
      monthly/2026-01.md
      quarterly/2026-Q1.md
    findings/quarterly/2026-Q1.md
    tendencies/quarterly/2026-Q1.md
  cross_categories/                      multi-category analyses
    tendencies/quarterly/2026-Q1.md
```

Listing 2: A representative knowledge-base layout. The path is the identity; the category, kind and interval are encoded by folders and by the file-name token.

Two details deserve mention, as both follow from treating the path as identity. The arXiv identifier keeps its dot but drops its version in the file name — the version is retained in the

frontmatter — so that the node denotes the *paper* and stays stable across revisions. And the concept graph lives at the root, in a shared `concepts/` subtree, rather than under any single category: a method used in several categories must be one node to which all of them link, not several fragments.

5.3 The concept graph and the attention signal

Documents do not only describe their own subject; they link into a shared graph of concepts. Each generated document may emit `concept_links` — bundle-relative paths of the form `/concepts/<slug>` — towards the methods, benchmarks or ideas it involves. Here another affordance of OKF becomes essential: a broken link is valid. A summary may therefore link a concept before any node exists for it, and the graph is allowed to grow lazily, references first and nodes later, which is the natural order in which knowledge accumulates.

The value of these links is realized deterministically. A concept index scans the KB and inverts the links into *backlinks*: for each concept, the set of documents that reference it, each with its date. The temporal distribution of those backlinks is, in itself, a signal of attention — how much, and when, a concept has been referenced — and it is a trend indicator obtained without any recourse to a model, purely by counting. The concept curator, an agent, is invoked only afterwards and only for the semantic part: to read the referencing documents and write a concise node. The counting is deterministic; the prose is the LLM's. This is, once more, the separation of section 3 applied to a concrete mechanism.

5.4 Provenance and lineage

Finally, the substrate keeps two notions of traceability distinct, because they answer two different questions. *Provenance* answers "how was this produced?": every generated document carries a block with the agent, the model, the prompt version and the timestamp of its generation. *Lineage* answers "from what was this derived?": a separate list of `source_ids` records the documents — or, for an article, the arXiv paper — from which the content descends. Keeping the two apart is not pedantry; an auditor asking why a tendency was asserted needs the lineage, while an operator debugging a regression needs the provenance, and conflating them would serve neither.

A related decision concerns the OKF `resource` field, whose purpose is to point at the external asset a document describes. For an article summary the choice is natural — it is the paper's arXiv abstract URL. For a derived, temporal document it is, deliberately, absent: a monthly digest or a tendencies analysis describes no external asset but internal, derived knowledge, and to invent a URL for it would be a small dishonesty of exactly the kind section 3 forbids. What such documents carry instead is their `interval` and their `source_ids`, which locate them in time and in the derivation graph without pretending to an external referent they do not possess.

Summary

The substrate is, on purpose, plain — ordinary markdown files in ordinary directories — and this plainness buys longevity and independence from any particular tool. Yet at its boundaries it is strongly typed: nothing is written that a contract has not validated, and every document declares both how it was made and from what. Plainness and typing are not in tension here; together they are what make the KB a store that other programs, and the system's own analytical layer, can trust.

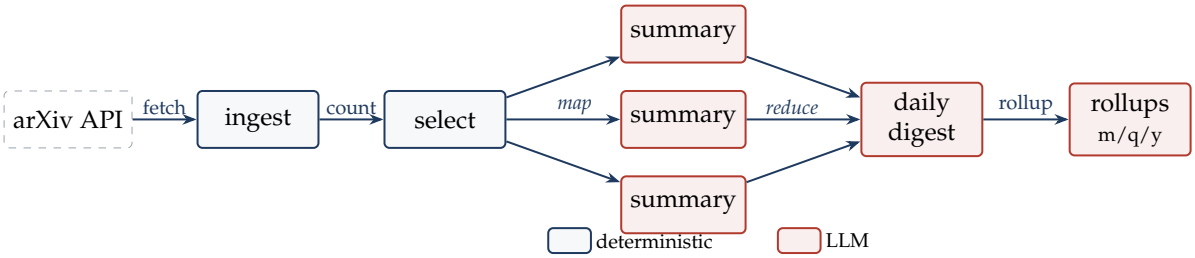


Figure 3: The Summarizer as a map-reduce. Deterministic ingestion and selection (blue) precede the map of per-article summaries; these reduce into a daily digest, above which the rollups (monthly, quarterly, yearly) are built. The LLM-mediated steps are drawn in the accent colour.

Part III

The Subsystems

6 The Summarizer (MAS_i)

The Summarizer is the subsystem that turns raw arXiv submissions into the first two strata of the Knowledge Base: one faithful summary per paper, and the temporal digests that aggregate them. Its shape is that of a map-reduce (section 2): an independent summarization of each paper — the *map* — followed by a synthesis into a daily digest — the *reduce* — with the deterministic work of ingestion and selection before, and the hierarchical rollups after. Figure 3 shows the flow. Throughout it, the division of labour of section 3 is plainly visible: everything that surrounds the two LLM steps — fetching, deduplicating, resolving what already exists, tiling, naming — is ordinary code.

6.1 Ingestion and selection

The subsystem begins by fetching, from the arXiv API, the submissions of the requested category and date. Since an active category may return many hundreds of papers, and since the interface reports the total of a query without materializing it, the system can also be asked only for the count — useful for planning and for bounding a run before it starts, complemented by a limit that caps the number of papers actually summarized. The fetched papers are then deduplicated: because a paper may be cross-listed under several categories, it is summarized once, under its primary category, and reached from the others by reference rather than copied. Finally, selection filters out what the Knowledge Base already contains — the incremental checkpoint that makes a run idempotent and resumable: a crash midway does not re-summarize what has already been written, and a re-run contributes only the papers still missing. None of this involves meaning; all of it is done by code, and it is precisely this that keeps the expensive map as small as it can be.

6.2 The map: per-article summaries

For each new paper an agent produces a faithful and dense summary: the core contribution, the method, the key results with their numbers, the limitations, and what is genuinely novel.

The agent also assigns a *salience* in $[0, 1]$ — a coarse estimate of novelty and importance that later governs how aggressively a paper is compressed in a rollup — and emits the concept links that connect the paper into the shared graph of section 5. The output is not free text but a typed contract, validated before it is written, so that the reduce receives well-formed input.

The map is the costly part of the subsystem, and it is embarrassingly parallel: each paper is independent of the others. It is therefore executed by a bounded thread pool, whose degree of concurrency is a configuration value that should be aligned with what the model server can actually serve in parallel — a client that asks for more than the server admits gains nothing but a queue. Failures are isolated per paper: a paper that cannot be summarized is logged and skipped, and the day's coverage records how many were attempted against how many were produced, so that the omission is visible rather than silent.

6.3 The reduce: daily digests and the ledger

The summaries of a day are then synthesized into a daily digest: related work is clustered into themes, what is novel is surfaced, and the papers are connected. Because the consumer of a digest is the Analyzer and not a lay reader, the digest is written for density and fidelity rather than for engagement.

One mechanism deserves emphasis. A naive hierarchy of summaries-of-summaries loses signal at every level, as each synthesis paraphrases the one below it; to counter this, each digest carries a structured *key-findings ledger* — for every salient paper, its contribution, its novelty and its practical import — which is carried forward across the levels by appending and de-duplicating rather than by re-summarizing. The prose is recompressed at each level; the ledger is not. This is a deliberate defence of fidelity across the hierarchy, and the reason for which a quarterly digest can still name the specific result of a paper submitted three months earlier.

6.4 Rollups: completeness, modes and empty days

Above the daily digest sit the monthly, quarterly and yearly rollups, each built from the next-finer digests: a month from its days, a quarter from its months, a year from its quarters. Two decisions govern their construction, both instances of the honesty of section 3. First, a rollup is synthesized only when every one of its constituents is present — checked recursively down to the existing daily digests — so that a monthly digest is never a partial thing masquerading as a whole; an incomplete period is instead left to be read as the individual days that do exist. Second, so that a day which legitimately produced no papers still counts as covered, the reduce writes a deterministic empty marker for such a day — a daily digest with zero included children and no LLM call — which lets "incomplete" mean "not yet processed" rather than "empty". The distinction matters most in the low-volume categories, where paperless days are common and a strict criterion, without such a marker, would prevent a rollup from ever completing.

When rollups are built is a matter of policy, governed by a single setting with three values. In the *lazy* mode the Analyzer builds a missing rollup on demand, the first time it needs it,

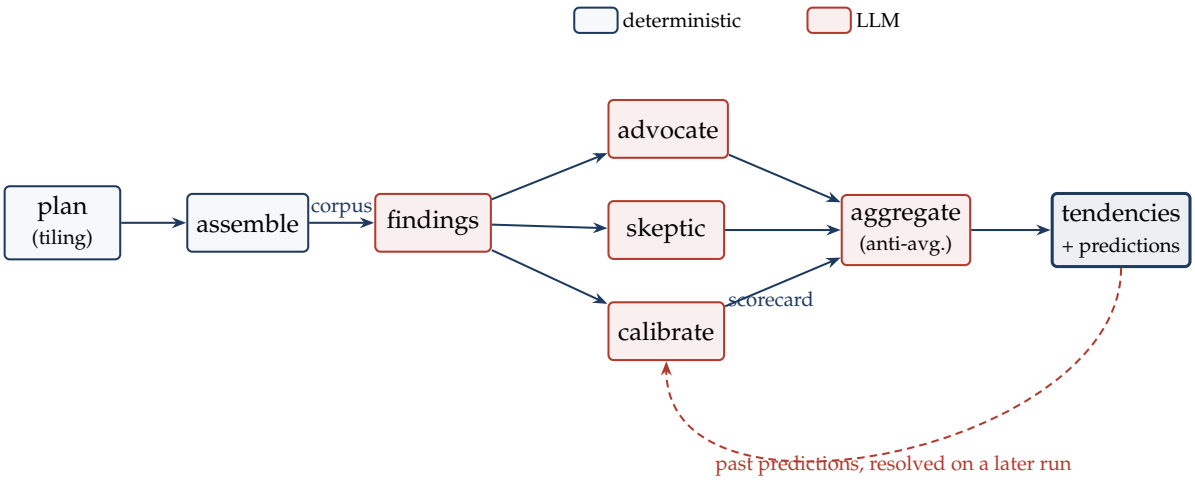


Figure 4: The Analyzer as a pipeline with a Mixture-of-Agents fan-out/merge and a calibration feedback loop. Deterministic planning and assembly (blue) precede the LLM steps (accent); the advocate and skeptic deliberate in parallel with the calibration, and the aggregator merges them under an anti-averaging policy. The dashed edge is the loop: the predictions emitted now are resolved and scored on a later run, and the resulting scorecard tempers the aggregation.

and caches it. In the *eager* mode a scheduled sweep builds every complete-but-missing rollup ahead of time, and the Analyzer only reads. The *mix* mode, which is the default, combines the two: the sweep pre-builds, and the Analyzer still builds on demand whatever the sweep has not yet reached — so that with no scheduler at all the system behaves as in the lazy mode, and with one it gains both the pre-computation and a fall-back. The sweep itself is idempotent and self-healing: a missed run is caught up by the next, and the current, still-open period is simply skipped until it completes.

7 The Analyzer (MAS_o)

If the Summarizer reduces papers to knowledge, the Analyzer reasons over that knowledge. It reads the Knowledge Base over a requested range and produces two kinds of analysis: the *findings* that the literature establishes, and the *tendencies*, with their predictions, that it suggests. Its shape is richer than a plain pipeline: a deterministic planning-and-assembly stage feeds a findings extraction, which fans out to an antagonistic Mixture-of-Agents deliberation and to a forecast calibration, and these merge in a final aggregation. Figure 4 shows the structure, including the feedback loop by which the system's own past predictions inform its present confidence.

7.1 The coverage planner

The Analyzer does not read its range indiscriminately; it first decides, deterministically, which documents tile it most economically. This is the *coverage planner*, and it is a clear example of a task that belongs to code rather than to a model: given a range, it selects the coarsest available digest at each position, descending to finer granularities only where a coarse one does not

align or is not complete. A full calendar month collapses to its monthly digest; the unaligned days at the edges of the range remain as individual days; and a gap — a period for which no digest exists or can be faithfully built — is left explicit rather than papered over. The planner works against the same completeness notion as the rollups of section 6: it will schedule a rollup that must still be generated only if all of its constituents are present, which is what keeps the assembled corpus honest about what it contains.

7.2 Assembly and conditional compression

The planned tiles are then read and concatenated into a single corpus. Most of the time this corpus fits comfortably within the model's context, and it is passed on as it is; the deterministic layer does not compress what does not need compressing. Only when the corpus exceeds a configured budget is a holistic compression invoked — an LLM step that reduces the corpus while preserving every salient result — and even then it enters as a conditional exception, not as a routine stage. This is a small but characteristic decision: the costly and lossy operation is reserved for the case that actually requires it.

7.3 Findings

Over the assembled corpus an agent extracts the concrete findings: the results the literature establishes and what they enable in practice, separated deliberately from speculation. The findings are the factual substrate on which the subsequent deliberation rests, and they are written to the Knowledge Base as an analysis in their own right, with their lineage back to the digests from which they derive.

7.4 The Mixture-of-Agents

The step from findings to tendencies is where a single pass is least trustworthy, because it is here that judgement enters — about feasibility, about adoption, about impact — and a lone judgement tends towards the bland. We therefore adopt a Mixture-of-Agents (section 2) with an antagonistic design. Two agents address the same findings from opposed and clearly assigned tasks: one builds the strongest evidence-based case for rapid adoption and positive impact, the other the strongest case for the barriers, risks and failure modes that would prevent it. Both are required to *steelman* their side — to argue it at its best, grounded in the evidence, rather than to caricature it — and both argue across three lenses: the technical, the market or economic, and the regulatory or adoption-related.

Their briefs are then merged by an aggregator whose policy is explicitly one of *anti-averaging*: it is instructed not to split the difference between the two sides but to weigh the evidence point by point and to decide, and to surface a low-probability yet high-impact possibility as such, rather than to dilute it into a bland consensus. The aggregator's output is the tendencies analysis, and it carries falsifiable predictions — each with a horizon and a criterion of falsification — which are the hooks on which the calibration loop will later pull.

7.5 The calibration loop

A system that only ever asserts, and never checks, has no way to earn or to lose credibility. The Analyzer therefore closes a loop over its own forecasts. When it runs, it gathers the predictions of its past tendencies analyses whose horizon has now matured, and an agent — with read access to the current Knowledge Base through its tools — resolves each of them against what the KB now contains, judging whether it came true, in part, or not at all, and citing the evidence. The resolutions are then scored deterministically: by a Brier score if the predictions were probabilistic, or by a weighted hit-rate if they were qualitative (section 2). The resulting scorecard is fed to the aggregator, so that a class of forecasts found to have been poorly calibrated is, in the present analysis, trusted less.

The calibration is auxiliary by design. Should it fail — because a model call errs, or because a resolution comes back malformed — it degrades to an empty scorecard and the analysis proceeds without it, since a system that cannot grade its past should still be able to reason about its present. Here, too, the two sides of section 3 are kept apart: the *judgement* of whether a prediction came true is the LLM's, for it requires reading and interpreting the evidence, while the *scoring* of that judgement is arithmetic, and is done by code.

7.6 The scope of economic claims

A final word on the economic and industrial impact that the tendencies claim. To reason credibly about market impact requires signals that the Knowledge Base does not, by itself, contain, and to invent them would violate the honesty of section 3. The scope of such claims is therefore a configuration value: by default it is bounded to research-internal momentum — what the literature itself reveals about the direction and the speed of a line of work — and it is left open to be widened, through an external signal source or a search tool, only when such a source is actually available. The system prefers a bounded claim that it can support to an ambitious one that it cannot.

Part IV
Cross-Cutting Layers

8 The Agent Layer

The two subsystems described in Part III are, at bottom, orchestrations of agents; the present section describes the layer they share — how an agent is declared, how it is bound to a model, how it is prompted, how its output is constrained, how it may reach the Knowledge Base, and how it is made resilient. The layer is common to MAS_i and MAS_o, and it is where the principles of section 3 concerning configuration, provider independence and fidelity take their concrete form.

8.1 The registry

Everything the system needs to know about its agents lives in a single declarative file, the *registry* (Principle 2 of section 3). It has three parts. A block of *models* declares the available model tiers — in the development configuration, a single local model, referenced by a YAML anchor so that it is written once and pointed to many times. A block of *settings* holds the deterministic toggles that govern the system's behaviour: the rollup mode, the prediction mode, the economic scope, the degrees of concurrency and of retry, and the orchestration provider. And a block of *agents* declares, for each agent, its subsystem, its profile, the model to which it is bound, the prompt it uses, the tools it may call, its output contract, and any parameters particular to it. The consequence, already anticipated, is that the operator reasons about the whole system from this one file, and that to change the model of an agent — or of all of them at once — is to edit a single line. The full registry is reproduced and annotated in appendix A.

8.2 The model abstraction

An agent is bound to a model not by naming a vendor but by naming a tier in the registry, and the tier resolves, through a uniform chat-model interface, to a concrete client. In development that client speaks to a model served locally by Ollama; in production the same agent may be pointed to an external, hosted API, with no change beyond the registry. This is the first of the two axis of substitution promised by section 3, and it is what allows to develop the system at no cost against a local model and to deploy it against a stronger one — or to tier its agents, reserving the more capable model for the steps whose stakes or difficulty justify it, without touching the code.

8.3 Prompts and the house style

The prompts are kept outside the code, as templates, so that they may be edited by hand without a rebuild and versioned on their own. They are, moreover, factored: a role prompt states only the task — what to read, what to produce, which fields to return — while a small

set of shared fragments carries the writing style, so that the style is defined once and applied consistently. Three fragments suffice. A *base*, common to every agent, fixes the register — structural markdown, verbs calibrated to the evidence, preservation of the specific figures and names, no meta-commentary — and the convention for linking concepts. A *summary* fragment, for the agents whose reader is the machine, asks for maximal density and for the separation of result from speculation. And an *analysis* fragment, for the reasoning agents, asks that evidence be kept distinct from inference, and that claims be made falsifiable and bounded to their declared scope. This factoring is a small instance of the discipline that governs the code as well: a concern that recurs is stated in one place.

8.4 Output contracts and validation

An agent does not return prose that the surrounding code must then parse; it returns a value conforming to a typed contract, declared with Pydantic, and the model is constrained — through its structured-output capability — to produce exactly that shape. The contract is validated before anything is written, so that a malformed or incomplete response is caught at the boundary of the reasoning layer and never reaches the Knowledge Base. This is the mechanism by which the fidelity of section 3 is enforced in practice: the store stays clean not because the model is trusted to be well-behaved, but because nothing enters it that a contract has not admitted. The graph node that receives the contract is then free to translate it into the appropriate document, attaching the provenance and the lineage that make the document auditable.

8.5 Tool-using agents

Most agents are pure functions of their input: they are given the material they need and asked to return a contract. A few, however, are better served by reaching into the Knowledge Base themselves — the agent that resolves past forecasts must gather the evidence that bears on each, and the agent that curates a concept must read the documents that reference it. For these, the agent layer binds a set of tools, drawn from the very operations the MCP server exposes, so that a tool-using agent reads the KB through exactly the contract an external client would use. The binding is restricted, on purpose, to reading: the tools offered are to read a document, to list documents and to test for existence, and never to write. Writing remains the province of the deterministic graph, which validates the contract and computes the canonical path before it persists anything; to hand a model the ability to write arbitrary content to arbitrary paths would be to surrender precisely the guarantees that the contract and the naming discipline exist to provide. The reading agent runs a short tool-calling loop and, at its end, is coerced back to its structured contract.

8.6 Retry and fall-back

Finally, a language model served over a network is an unreliable component, and the agent layer accounts for it. Every agent call is retried, a small and configurable number of times, with a back-off between attempts, so that a transient failure — a momentary loss of the connection,

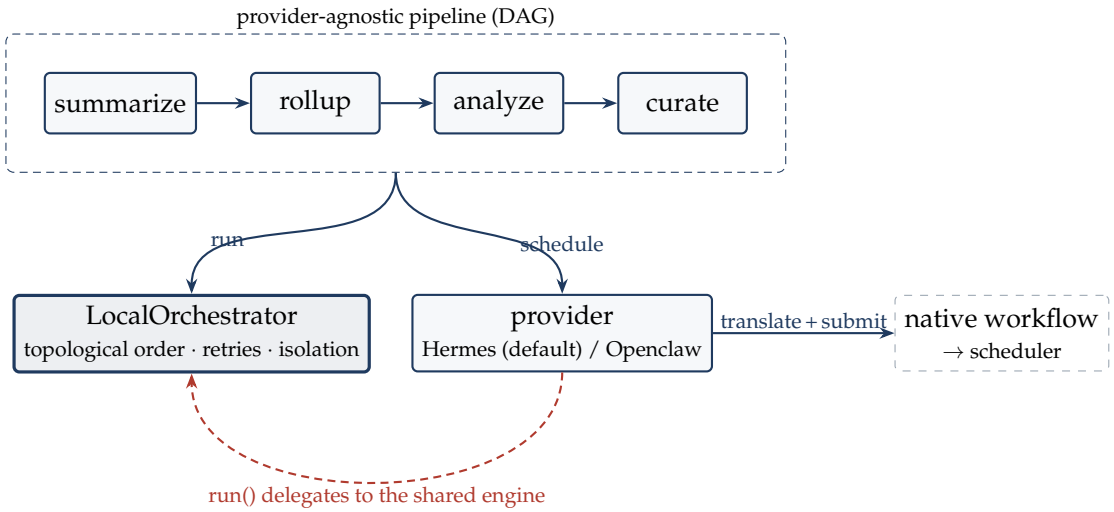


Figure 5: Orchestration. The pipeline is a provider-agnostic directed acyclic graph of four stages. Running it once is provider-agnostic and is performed by the shared `LocalOrchestrator` engine; scheduling it is the only provider-specific act, in which a provider translates the graph into its native workflow and submits it. Providers delegate their one-shot `run()` back to the same engine (dashed).

a timeout, an occasional malformed response — does not propagate. Only when the retries are exhausted does the failure surface, and even then the caller may have supplied a *fall-back* value to be returned in place of a hard error, for those agents whose contribution is not indispensable. The division is deliberate: the *retrying* of a transient failure belongs to the agent layer, which is close to the call, whereas the *isolation or degradation* of a permanent one belongs to the caller, which alone knows whether the agent was essential — a division we shall meet again, at the level of whole stages, in sections 9 and 10.

9 Orchestration

The subsystems and the agent layer suffice to perform each operation in isolation; orchestration is what chains them into a single process and, when required, schedules it. The requirement we place on this layer is the second axis of substitution of section 3: as no model is load-bearing, no orchestrator must be either. The design that meets this requirement rests on a single separation — between *executing* the pipeline and *scheduling* it — which we develop in what follows and which fig. 5 depicts.

9.1 The pipeline as a provider-agnostic DAG

The flow that chains the system is a directed acyclic graph (*DAG*) of four stages — summarize, then rollup, then analyze, then curate — each depending on the previous. Crucially, this graph is described as data, independently of any orchestrator: a stage is a named unit of work together with its dependencies, and the pipeline is the set of such stages. Nothing in the description commits to how or where the graph runs; that is left to whatever executes it. This is the artifact on which both execution and scheduling operate, and keeping it agnostic is precisely what makes the two interchangeable.

9.2 The execution engine

To run the pipeline once, here and now, is a provider-agnostic act, and it is performed by a shared engine, the `LocalOrchestrator`. The engine sorts the stages topologically and runs them in dependency order, and it does three things that a naive sequence would not. It *retries* a stage that fails, a configurable number of times. It *isolates* a failure, so that a stage which fails blocks only the stages that depend on it, and leaves any independent branch to proceed. And it *reports*, for each stage, whether it succeeded, failed or was skipped, which is the information on which the operational concerns of section 10 — exit codes, monitoring — are built. Because this engine is entirely provider-agnostic, it is not only the workhorse of development and of manual runs; it is also the execution back-end to which the providers themselves delegate.

9.3 Providers and the scheduling seam

What, then, distinguishes one provider from another? Not execution, but scheduling. A provider is an adapter that contributes two things: it delegates the one-shot `run()` to the shared engine, and it implements `schedule()` by translating the provider-agnostic DAG into its own native workflow and submitting it to its scheduler. The default provider is Hermes; an alternative, Openclaw, is included precisely to demonstrate that the choice is free — the two present an identical interface and differ only in the translation and the submission —, and substituting one for the other is a single line in the registry. The point at which a real provider is wired is a single, well-delimited seam: a client for the provider's scheduler is injected, and the `schedule()` method hands it the translated workflow. Until such a client is present, scheduling is simply unavailable, while execution — being agnostic — continues to work through the local engine, or through the system cron, for which the local provider can emit the appropriate line.

9.4 Execution versus scheduling

It is worth making the separation explicit, for it is the whole of the design. To *run* the pipeline is provider-agnostic and always available: any provider, asked to run, defers to the same engine. To *schedule* the pipeline is provider-specific, and it is the only thing a provider genuinely adds. The consequence is that the value of the system — its pipeline, its subsystems, its Knowledge Base — is never entangled with a scheduler: one may develop and operate the whole process with no external orchestrator at all, adopt Hermes to schedule it in production, and later replace Hermes with another provider, and in none of these transitions does anything but the scheduling change. This is provider independence made concrete, and it is the orchestration counterpart of the model abstraction of section 8.

10 Resilience, Operations and Observability

The preceding sections have introduced, each in its place, the mechanisms that make the system robust; the present section gathers them into a single operational account, and completes it with the concerns that appear only when the system is actually run — how it fails, how it is rate-limited, how its outcome is signalled, how it is observed and how it is configured. The

orientation, as announced in section 1, is towards unattended and scheduled execution: the system is meant to run with no human watching, and everything in this section is in service of that.

10.1 The failure model

The governing rule is that a failure should be as local as it can be, and it appears at three scales. At the level of an *item*, the loops that process many things independently — the map of article summaries, the curation of concepts, the sweep that pre-builds rollups — isolate each one: an item that fails is logged and skipped, and the batch proceeds, the omission recorded (in the case of the daily map, as a coverage count of produced against attempted). At the level of an *auxiliary* step, a component that is not essential to the result must not be able to sink it: the calibration of past forecasts, in particular, degrades to an empty scorecard when it fails, and the analysis continues without it. And at the level of a *write*, every document is persisted atomically — written to a temporary file and then moved into place — so that a crash cannot leave a half-written document in the Knowledge Base, and so that the parallel map cannot corrupt the store. Nothing that fails is permitted to fail more widely than its own scope.

10.2 Rate limiting and retries

Two of the system's dependencies are unreliable in the ordinary course of operation — the arXiv service, which is rate-limited, and the language model, which is served over a network — and both are handled by pacing and retrying rather than by presuming success. Requests to arXiv are spaced by a minimum interval and retried under an exponential back-off that honours the service's own `Retry-After` signal when it asks the client to wait; a transient rate-limit response therefore slows the client rather than failing it. Calls to the model are retried at two levels which do not duplicate but compose: the agent layer retries an individual call a few times, close to the point of failure (section 8), while the orchestrator retries a whole stage, further out, should a failure escape the finer level (section 9). The effect is that a brief disturbance is absorbed low in the stack, and only a persistent one climbs to where it becomes visible.

10.3 Exit codes and monitoring

For a system that runs unattended, the outcome of a run must be legible to whatever scheduled it, and the natural channel for this is the process exit code. The pipeline is the monitored path: run as a whole, it reports success only if every stage succeeded, and it exits with a non-zero code if any stage failed or was skipped, so that a scheduler — cron, or a provider — can detect the failure without parsing logs. The individual subcommands, invoked directly, surface a hard failure as a non-zero exit in the ordinary way; and the per-item isolation of the map means that a partial success — some papers summarized, some skipped — is a success at the level of the run, its omissions recorded in the coverage rather than in the exit code. The distinction is deliberate: the exit code answers "did the run complete?", while the coverage answers "how completely?".

10.4 Verbosity and logging

Observability is provided through ordinary logging, exposed by a verbosity flag that maps, in three steps, to the level of detail. By default the system is quiet, emitting only warnings and the final result. A first level raises the log to the phase boundaries — ingestion, selection, the map, the reduce, the planning, the analysis, the stages of the pipeline — so that an operator sees the shape of a run as it proceeds. A second level descends to the individual agents and items. A third adds the logs of the underlying libraries, for the occasions on which the fault lies below the system itself. Because the mechanism is the standard logging of the language, it composes with whatever an operator already uses to collect and inspect logs, and it required no logging framework of its own: the phase messages are present in the code, and the flag only chooses how much of them to show.

10.5 Configuration and secrets

Finally, the system is configured from the registry (section 8), which is the single place an operator edits. Two provisions make it suitable for more than one environment. First, the deterministic behaviour is governed by explicit settings — the rollup mode, the prediction mode, the economic scope, the degrees of concurrency and of retry, the orchestration provider — so that the same code runs differently in development and in production by configuration alone. Second, no secret is ever written into the registry: where a credential is required, the configuration carries a placeholder that is resolved at load time from the environment or from a secret manager, and never the secret itself. The term *vault* is reserved, in the system, for that secret store, and is kept distinct from the Knowledge Base precisely so that the two are never confused — a knowledge base is meant to be read widely, a secret store exactly not. In this way the configuration may be committed and shared, while the secrets to which it refers remain outside it.

Part V
Production Concerns

11 Security

A multi-agent system is not merely a larger single agent; it is a distributed system whose components reason, act and trust one another, and its security must be considered as such. This section sets out why the attack surface widens, the threats that are particular to agentic systems, and the layered defences that answer them.

11.1 Why the attack surface widens

Each capability that makes an agent useful is also an entry point. Tools give the system reach into the world, and therefore a way for the world to reach in; every inter-agent message is a channel along which a corrupted instruction may travel; and a shared memory read by many agents is a single point through which all of them can be influenced. The trust chains are longer, too: an agent commonly acts on content it did not author — a retrieved document, the output of another agent, the result of a tool — and a model that cannot reliably distinguish data from instruction becomes a confused deputy, executing as a command what was only meant to be read.

11.2 Threats particular to agentic systems

The characteristic threat is *prompt injection*, and especially its indirect form, in which the malicious instruction arrives inside content the agent consumes rather than from the user, and then propagates from agent to agent. To it are added *tool misuse* and *excessive agency*, where an over-permissioned tool lets a manipulated agent do more harm than the task ever required; *memory or context poisoning*, where tainted material persists and re-influences later runs; and *data exfiltration*, where a tool with outbound reach is turned into a channel for leaking what the system holds. Industry frames such as the OWASP catalogue for agentic and LLM applications enumerate these systematically, and are a useful checklist against which to audit a design.

11.3 Defence in depth

No single control suffices; the discipline is one of layers, each assuming the others may fail (fig. 6). Input is validated and, where possible, the boundary between instruction and data is made explicit; tools are granted under least privilege, scoped to exactly what the task needs and no more; execution is sandboxed and isolated; output is validated before it is allowed to act; an irreversible action passes through a human gate; and the whole is monitored, so that an anomaly is seen.

Several of these controls are structural in this system rather than added after the fact. The tools bound to its agents are *read-only* (section 8): an agent may read, list and test for a document, but it has no tool with which to write, so a manipulated agent cannot corrupt the

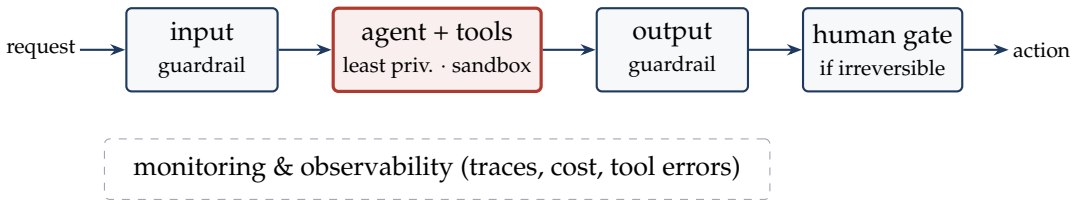


Figure 6: Defence in depth: a request passes through successive controls, each independent of the others, with monitoring underneath.

store. Writing is the exclusive province of the deterministic graph, which validates a typed contract and computes a canonical path before it persists anything — a strict tool contract and a least-privilege boundary in one. And no secret is ever written into the configuration (section 10). The system is not thereby immune — the fidelity of what a model writes is still the model's — but the blast radius of a compromised agent is deliberately small.

12 Observability

If a classical program fails, it leaves a stack trace; if an agent fails, it leaves a *trajectory* — a chain of reasonings, tool calls and hand-offs, any of which may have been the point at which things went wrong. Because the control flow is chosen at run time by the model itself, it cannot be read off the source code, and observability is therefore not a convenience but the only way to know what the system actually did.

12.1 Traces and spans

The unit of observability is the *span*: a single step — a model call, a tool invocation, a hand-off — recorded with its inputs, its outputs, its duration and its cost. A *trace* is the tree of spans that constitutes one run, and it is the trajectory made inspectable. With traces in hand, a failure is no longer a mystery but a place: one can see which agent, at which step, on which input, produced the output that derailed the rest.

12.2 What to monitor

Beyond individual traces, a production system watches a handful of aggregate signals. Latency and *cost* — the tokens consumed — matter because an agentic run multiplies model calls, and an unbounded one can multiply them without limit. The *tool-error rate* reveals a failing integration before it fails the user. Step and loop counts expose an agent that is not converging. And a completion or *coverage* signal distinguishes a run that finished from one that finished only in part. None of this is free: recording and retaining traces has a cost of its own, and at scale it is sampled rather than captured whole.

12.3 In this system

Observability here is deliberately plain, and it reuses the standard logging of the language rather than a bespoke framework (section 10). A single verbosity control selects, in three steps,

how much of the run is shown: the phase boundaries, then the individual agents and items, then the underlying libraries. The completeness of a run is carried by two distinct signals, matching the two questions an operator asks: the process *exit code* answers whether the run completed — the pipeline exits non-zero if any stage failed — while the `coverage` recorded in each document answers how completely, distinguishing a day that produced nothing from one not yet processed. The system does not emit full span trees; it is a batch pipeline rather than an interactive agent, and its shape is fixed and inspectable in the code, so the lighter instrument suffices. Were its agents to gain more autonomy, span-level tracing would be the natural next instrument to add.

13 Evaluation

The most common error in evaluating an agentic system is to judge only its final answer. A system may reach the right answer by luck, or by a reasoning that will not generalize, or having consumed ten times the tools it needed; and it may reach a wrong answer through a process that was almost right and easily fixed. To evaluate the answer alone is to see none of this.

13.1 Three layers of evaluation

A fuller account judges the system at three layers, summarized in table 2. At the level of the *component*, each agent or tool is checked in isolation: does it, given a known input, produce a correct and well-formed output? At the level of the *trajectory*, the path itself is judged: were the steps sensible, was the right tool chosen, were there redundant calls or loops? And at the level of the *outcome*, the end state is checked against the goal. The three are complementary: component tests catch a broken part, trajectory evaluation catches a part that works but is used badly, and outcome evaluation confirms that the whole achieved its purpose.

Layer	Question	Example
Component	is each part correct in isolation?	a tool returns the right value
Trajectory	were the steps sensible?	no loops; the right tool chosen
Outcome	was the goal achieved?	the final state is correct

Table 2: The three layers at which an agentic system is evaluated.

13.2 Methods and metrics

The methods are three, and they trade cost against fidelity: an LLM acting as judge is cheap and scalable but imperfect; a reference-based comparison is exact where a ground truth exists; and human judgement is the most faithful and the least scalable. To the familiar metric of accuracy a multi-agent system must add others that its distributed nature makes salient — cost and latency, the efficiency of its tool use, the quality of coordination between agents, and robustness under perturbation.

13.3 In this system

The separation of concerns pays a dividend here, as section 16 describes: the deterministic core is evaluated exhaustively at the component level, offline. The reasoning is evaluated in one respect already — the Analyzer scores its own past forecasts through the calibration loop (section 7), which is precisely a trajectory-and-outcome evaluation of its predictions, folded back into the system. What is not yet claimed is an evaluation of the *fidelity* of a summary or the *soundness* of an analysis against a reference; that is the evaluation harness left to future work (section 19), and it belongs squarely at the outcome layer.

14 Production Design Patterns

The literature of agentic engineering has, over a short time, converged on a vocabulary of recurring patterns. It is useful to catalogue them, both because they name design choices precisely and because a system can be characterized by which of them it adopts. We group them, following the source material, into workflow, capability and reliability patterns, and note in each case the system's own use (table 3).

14.1 Workflow, capability and reliability patterns

The *workflow* patterns concern the shape of the computation: prompt chaining (a fixed sequence of steps), ReAct (interleaved reasoning and acting), planning (decompose first, then execute), reflection (a critic revises a draft), the evaluator–optimizer loop, the orchestrator–worker division, parallelization, and multi-agent debate. The *capability* patterns concern what an agent can draw upon: tool use, layered memory, retrieval-augmented generation, and the deliberate engineering of what enters the context. The *reliability* patterns concern behaviour under stress: bounded execution (a hard cap on steps, time or cost), the circuit breaker (stop calling a failing dependency), retry with back-off, guardrail layering, the human-in-the-loop as a structural gate, strict tool contracts, graceful degradation, and routing with a fall-back.

Group	Patterns	In this system
Workflow	prompt chaining, ReAct, planning, reflection, evaluator–optimizer, orchestrator–worker, parallelization, debate	orchestrator–worker (the pipeline), parallelization (the map), debate (the Mixture-of-Agents)
Capability	tool use, layered memory, RAG, context engineering	read-only tool use, RAG, layered memory (files + concept graph)
Reliability	bounded execution, circuit breaker, retry with back-off, guardrail layering, human-in-the-loop, strict tool contracts, graceful degradation, routing fall-back	bounded execution, retry with back-off, strict (typed) tool contracts, graceful degradation

Table 3: Common production patterns and the ones this system adopts.

14.2 How they combine, and how they fail

A real system layers these patterns rather than choosing one. This one is, in these terms, an orchestrator–worker pipeline (section 9) whose costly stage is a parallel map (section 6) and whose analytical stage is a debate merged by an aggregator — an evaluator over an advocate and a skeptic (section 7); it uses read-only tools under strict, typed contracts (section 8); and it is made reliable by bounded execution, retry with back-off and graceful degradation (section 10). The patterns also come with their antipatterns, against which the same sources warn: reaching for many agents where one would do, leaving a loop unbounded, over-communicating out of caution until cost dominates, and — the failure the Analyzer's anti-averaging policy is designed to avoid — blending opposed views into a bland consensus rather than deciding between them.

Part VI
Implementation and Evaluation

15 Implementation

The architecture described so far admits, in principle, many realizations; this section records the one we chose and — more usefully — the reasons for the choices, since a technical decision is instructive only together with its justification. We describe first the stack and why each piece is in it, then the structure of the package, and finally what a deployment looks like, from a single development machine to a scheduled production run.

15.1 The stack

The system is written in Python and rests on a small number of libraries, each selected to serve one of the principles of section 3 rather than for its own sake. The control flow of the two subsystems is built on LangGraph [LangChain 2024], which represents a computation as an explicit graph of nodes with a typed state passing between them; this is what lets the deterministic orchestration remain visible and inspectable in the code, with the language model confined to the interior of particular nodes — Principle 1 made mechanical. The model itself is reached through the chat-model interface of LangChain [LangChain 2023], which abstracts over the provider, so that the same node speaks to a local model or to a hosted one according only to what the registry names — the model half of the provider independence of section 8. The output contracts, and the OKF documents themselves, are Pydantic [Pydantic 2023] models, so that the validation-before-write on which fidelity depends is simply the construction of a typed object. The Knowledge Base is exposed to external clients and to tool-using agents through a server built with FastMCP, a small implementation of the Model Context Protocol [Anthropic 2024]. And in development the model is served locally by Ollama [Ollama 2023], which lets the whole system be built and exercised at no cost against a capable local model, and replaced by a stronger hosted one in production with no change to the code. The environment is pinned with conda for the system-level dependencies and resolved with uv for speed.

One choice is as much about what was left out as about what was kept. Frameworks such as CrewAI or AutoGen offer, out of the box, their own paradigms of agent collaboration and their own orchestration; we deliberately did not adopt them. To have done so would have introduced a second locus of control, competing with the explicit LangGraph graphs, and would have blurred exactly the line — between the deterministic control that we own and the reasoning that we delegate — that the whole design is organized to keep sharp. This is not a judgement on those frameworks, which are capable; it is a consequence of our first principle, which asks that the orchestration be ours and legible.

15.2 Package structure

The code follows a `src` layout, and its modules are grouped, unsurprisingly, along the same seam that organizes this report. One group is purely deterministic and has no knowledge of a model: the arXiv client, the coverage planner, the selection and deduplication logic, the concept index. A second group is the knowledge substrate: the OKF models, the document read and write, the naming, and the operations through which the store is reached. A third is the reasoning layer: the agent registry and factory, the prompt library, the output contracts, and the runner that invokes an agent and validates its result. Over these sit the graphs — the two subsystems, the rollup logic and the concept-graph curation — and, above them, the orchestration, with its provider-agnostic pipeline and its providers. The command-line interface and the MCP server are the two entry points, and the configuration — the registry and the prompt templates — lives outside the code entirely, as data. The full module map and the annotated registry are given in appendix A.

15.3 Deployment

In development the system needs only a Python environment and a running Ollama; the Knowledge Base is a directory on disk, and the operations are invoked either one at a time, through the subcommands, or as a whole, through the pipeline. Moving to production changes three things, and by design only three. The agents are pointed, in the registry, at a hosted model — or tiered, with the more capable model reserved for the steps whose difficulty warrants it. The pipeline is scheduled, from the system cron or through an orchestration provider, to run unattended on its cadence. And the secrets that a hosted model requires are supplied from the environment or from a secret manager rather than from the configuration (section 10). Nothing in the architecture, the subsystems or the knowledge base is touched in this transition, which is the practical dividend of the independence that the design has maintained throughout.

16 Testing and Validation

A system that embeds a non-deterministic component cannot be validated in the way a purely deterministic one can, and it would be dishonest to claim otherwise. What can be done — and what the separation of section 3 makes possible — is to test exhaustively the part of the system that is deterministic, to test the control logic of the rest against substitutes for the model, and to be explicit about the boundary beyond which only integration against live services will do. This section describes that strategy and what it covers.

16.1 Strategy

The strategy follows directly from the first principle. Because the deterministic core is kept apart from the reasoning, it can be tested in isolation, offline, with neither a model server nor a network: the coverage planner, the naming and identity logic, the serialization of OKF documents, the selection and deduplication, the concept index — all of these are ordinary functions with exact specifications, and they are tested as such. The control logic that surrounds

the model — the wiring of the map-reduce, the completeness of the rollups, the execution of the orchestration graph, the retry behaviour — is tested by substituting for the agent runner a *stub*: an object that presents the runner's interface but returns a canned contract instead of calling a model. With the stub in place, the graphs run deterministically, and their structure can be asserted — that the map preserves order and isolates a failing item, that a rollup is built only when complete, that a failed stage blocks its dependents but not an independent branch — with no dependence on what a model would actually say. Beyond this boundary lie the two genuinely external things, the model call and the arXiv fetch; these are integration points, and they are exercised against the live services rather than mocked into a false certainty.

16.2 What is covered

The offline suite covers, concretely, the following. The coverage planner is tested for its tiling — coarsest-first, correct at the edges, gaps left explicit — and for its rejection of an inverted range. The naming is tested for the round-trip between a document and its Concept ID. The OKF layer is tested for the round-trip between a model and its on-disk form, frontmatter included. The rollups are tested for strict completeness, for the empty-day markers, for the three build modes, and for the isolation of a single failing rollup within a sweep. The parallel map is tested for the preservation of order and for per-item failure isolation. The concept index is tested for its backlinks and its attention signal, and the curation for its per-concept isolation. The orchestration is tested for topological ordering, for stage-level failure isolation, for retries, and for the provider registry. And the retry helper is tested for its success-after-failure, its exhaustion, and its fall-back. All of this runs without a model and without a network.

16.3 Reproducibility and boundaries

Two properties of this arrangement are worth stating plainly. It is *reproducible*: the offline tests contain no randomness and reach no network, so they yield the same result on every run, which is what makes them useful as a regression guard. And it is *honest about its boundary*: the tests do not pretend to validate the quality of a summary or the soundness of an analysis, which are properties of the model and of the prompts and which no stub can certify; they validate that the machinery around the model is correct, so that when the model is good the system is faithful to it, and when the model errs the failure is contained. The evaluation of the reasoning itself — whether a summary is faithful, whether a prediction proves calibrated — is left to the calibration loop of section 7 and to the evaluation harness proposed in section 19, and it is not claimed here.

17 Discussion

Having described the system, we step back to weigh the decisions that shaped it, to be candid about its limitations, and to consider how far its design might carry beyond the particular problem for which it was built.

17.1 Trade-offs revisited

Several of the decisions taken in the preceding sections are genuine trade-offs, not free choices, and it is worth naming what each one costs. The strict completeness of the rollups (section 6) buys an honest hierarchy at the price of latency in the coarse digests: a monthly digest cannot exist until the month is complete, and in a sparsely populated category this may be a real wait; the empty-day markers mitigate the tension but do not abolish it. The three rollup modes exist precisely because no single policy suits every deployment — lazy minimizes work but defers it to the reader's request, eager front-loads it but needs a scheduler, and mix trades a little duplicated effort for the robustness of having both —, and making mix the default is a judgement that robustness matters more than the occasional redundant build. The anti-averaging aggregation (section 7) buys sharper conclusions at the cost of asking more of the aggregator, which must now decide rather than blend, and is therefore more sensitive to the quality of the model. None of these is regretted, but none is free.

17.2 Limitations

The prototype has limitations that we state plainly. The prompts encode a defined house style but have not been tuned against output at scale, and the quality of the summaries and analyses is, in the end, the quality of the model behind them. The economic and industrial reasoning is deliberately bounded to research-internal signals (section 7), and a genuinely market-aware analysis awaits an external source that the system does not yet integrate. The concept graph is curated but not yet reasoned over — its attention signal is computed, but the relations among the concepts are not themselves analysed. And the provider integrations for scheduling are seams, wired to their clients but not exercised against a live Hermes or Openclaw within this work.

17.3 Threats to validity

Three threats deserve explicit mention. The first is the fidelity of the model: the system is faithful to its input only insofar as the model is, and while the contracts guarantee well-formed output they cannot guarantee true output — a confidently wrong summary passes validation as readily as a correct one. The second concerns calibration: the feedback loop can only score predictions whose horizon has matured, so its corrective effect is delayed by exactly the horizon of the forecasts it grades, and a system early in its life has little history against which to calibrate. The third is the corpus: a category with few submissions yields thin daily digests and slow rollups, and the arXiv interface, though stable, imposes its own rate limits and occasional irregularities that the ingestion must absorb. We have designed against each of these, but none is eliminated.

17.4 Generality

Finally, it is worth asking how much of the design is particular to arXiv and how much would transfer. Very little is particular. The arXiv client is one deterministic ingestion module behind

a clear boundary; replacing it with another source of dated, categorized documents — a patent feed, an internal report repository, a news corpus — would leave the substrate, the two subsystems, the agent layer and the orchestration untouched. What generalizes is not the corpus but the shape: the separation of deterministic orchestration from model reasoning, the OKF blackboard with its orthogonal schema, the map-reduce of summarization and the pipeline-with-feedback of analysis, and the provider-agnostic layers. The system is, in this sense, an instance of a pattern, and the pattern is the contribution.

18 Related Work

The system draws on several established lines of work, which we situate here without attempting a survey of any.

18.1 Blackboard architectures and multi-agent systems

The organizing metaphor of the system is the blackboard architecture [Nii 1986], in which independent knowledge sources cooperate through a shared store under a control component. Our contribution is not to that model but in its application: we use it to separate deterministic control from model reasoning, and we realize the shared store as an OKF Knowledge Base rather than an in-memory structure, which lends the blackboard a persistence and an auditability that the classical formulation did not require.

18.2 Tool-using and retrieval-augmented agents

The reasoning agents are, in the terms of the recent literature, retrieval-augmented [Lewis et al. 2020]: they reason over material drawn from an external store rather than from their parameters alone. We reach that store through the Model Context Protocol [Anthropic 2024], and we restrict the agents' access to it to reading, keeping the writes deterministic — a choice that departs from the more permissive tool-using agents common elsewhere, and that follows from our insistence on a clean and validated store.

18.3 Multi-agent orchestration frameworks

A number of frameworks — among them CrewAI and AutoGen — offer ready-made paradigms for agent collaboration and orchestration. We discussed in section 15 why we did not adopt them: their orchestration would compete with the explicit control we wish to own. Our orchestration layer is closer in spirit to a workflow engine than to an agent-conversation framework, and it is deliberately thin, deterministic and provider-agnostic.

18.4 Aggregation and deliberation

The antagonistic deliberation of the Analyzer is a Mixture-of-Agents [Wang et al. 2024], specialized to an adversarial pair — an advocate and a skeptic — and merged under an anti-averaging policy. Where the general Mixture-of-Agents seeks to combine diverse capable agents, our use

is narrower and more structured: the diversity is prescribed, in the two assigned stances and the three lenses, and the aggregation is instructed to decide rather than to average.

18.5 Summarization and forecast calibration

The map-reduce shape of the Summarizer is the classical pattern [Dean and Ghemawat 2008] applied to text, with a key-findings ledger added to resist the loss of signal across levels. And the calibration of the Analyzer's forecasts rests on proper scoring [Brier 1950]: the Brier score for the probabilistic predictions, a weighted hit-rate for the qualitative ones. What is perhaps less usual is the closing of the loop — that the system scores its own past forecasts and lets the result temper its present confidence —, which we regard as a small step towards analyses accountable to their own track record.

19 Future Work

The prototype leaves several directions open, which we group by the part of the system they extend.

19.1 The reasoning layer

The most immediate work concerns the quality and the robustness of the reasoning. The prompts, which today encode a defined style, would benefit from tuning against output at scale; and the tool-using agents from a finer resilience than the whole-call retry of section 8 — a transient failure of a single paper in the map, for instance, is currently absorbed by skipping it, where a light per-item retry might instead recover it. A tiering of models, sketched but not exercised, would let the more capable model be reserved for the steps that most reward it.

19.2 The knowledge graph

The concept graph is, at present, curated and measured but not reasoned over: its attention signal is computed deterministically, yet the relations among the concepts — which methods supersede which, which benchmarks travel together — are not themselves analysed. A layer that reasons over the graph, rather than merely maintaining it, is a natural extension, and one for which the substrate is already prepared.

19.3 External signals

The economic and industrial reasoning is bounded, on purpose, to research-internal momentum (section 7). Widening it responsibly requires a curated external source — a market-data feed, or a search tool with provenance — integrated behind the same discipline as the arXiv client, so that the claims it enables remain as auditable as the rest. The scope setting already anticipates this; what is missing is the source.

19.4 Scale and evaluation

Two engineering directions remain. The first is scale: the map is already parallel, but a large historical backfill would benefit from batching, and from the count-without-download primitive used as a planner of the work ahead. The second, and the more important, is evaluation. The system grades its own forecasts through the calibration loop (section 7), but it has no harness that measures the fidelity of a summary or the soundness of an analysis against a reference; such a harness — which section 16 deliberately did not claim — would turn a subjective quality into a tracked metric.

19.5 Provider integrations

Finally, the scheduling seams of section 9 are wired but not exercised against a live orchestrator. Completing the Hermes and Openclaw clients, and running the pipeline under each, would close the last gap between the provider independence of the design and its demonstration.

20 Conclusion

We have presented ARXIV-MAS, a multi-agent system that summarizes and analyses the arXiv literature over a shared Knowledge Base, and we have organized both the system and its description around a single thesis: that deterministic orchestration and model reasoning are separate concerns, and that a system which keeps them apart — delegating to the language model only the interpretation and the production of meaning, and reserving everything else for ordinary code — is more testable, more economical, more reproducible and more honest than one that does not.

The thesis is visible at every level of the system. It is in the substrate, where a plain and durable format is held to a strict, validated schema. It is in the two subsystems, where the map-reduce of summarization and the pipeline-with-feedback of analysis surround their model calls with deterministic planning, selection and completeness. It is in the agent layer, where the model is bound behind a uniform interface and constrained to a typed contract. And it is in the orchestration, where a provider-agnostic pipeline is executed by a shared engine and merely scheduled by an interchangeable provider. What the prototype demonstrates is that this separation is not only sound in principle but practical in the whole: the deterministic core is complete and tested, the reasoning layer runs against a local model, and the two compose into a process that chains, from a raw submission to a calibrated tendency, from end to end.

If there is a single lesson we would draw from the exercise, it is that the separation is itself the design. The corpus is replaceable, the model is replaceable, the orchestrator is replaceable; what remains — and what we offer as the contribution of this work — is the shape into which they are arranged.

Appendices

A The Agent Registry

The registry (`config/registry.yaml`) is the single source of truth of section 8: three blocks — `models`, `settings` and `agents`. A per-agent model swap is a repointing of the agent's `model:` key from the development anchor to a production block; a secret is never written in line, but referenced as `${secret:NAME}` and resolved at load time. Listing 3 reproduces the models and settings blocks and three representative agents; the remaining agents follow the same shape.

```yaml
models:
  dev: &dev
    provider: ollama
    model: qwen3.6:35b
    base_url: http://localhost:11434
    options: { temperature: 0.2, num_ctx: 32768 }

settings:
  default_category: cs.AI
  rollup_mode: mix               # lazy | eager | mix
  prediction_mode: qualitative   # qualitative | brier
  economic_scope: research_internal
  map_concurrency: 4             # parallel article summaries
  agent_retries: 2               # per-agent retries on transient failures
  orchestrator: hermes           # hermes | openclaw | local
  concept_graph: { enabled: true, root: /concepts }

agents:
  article_summarizer:            # a pure structured call (no tools)
    subsystem: masi
    model: *dev
    prompt: masi/article_summary.md
    tools: []
    output_contract: OKFArticleSummary

  forecast_evaluator:            # tool-using: read-only KB tools to gather
    evidence
    subsystem: maso
    model: *dev
    prompt: maso/forecast_resolve.md
    tools: [mcp_kb]
    output_contract: ResolutionSet

  trend_aggregator:              # parameterized by an aggregation policy
    subsystem: maso
    model: *dev
    prompt: maso/aggregate.md
```

```
    tools: []
    output_contract: OKFTendenciesAnalysis
    params: { policy: anti_averaging }
```

Listing 3: Registry excerpt (models, settings, and three representative agents).

Every agent declares its `subsystem`, the `model` it is bound to, its `prompt`, the `tools` it may call (`[mcp_kb]` for a read-only tool-calling loop, `[]` for a single structured call), its `output_contract`, and optional `params` (a stance, a granularity list, an aggregation policy). The full roster comprises ten agents: three in the Summarizer (`article_summarizer`, `daily_synthesizer`, `rollup_synthesizer`), six in the Analyzer (`holistic_synthesizer`, `findings_analyzer`, `trend_advocate`, `trend_skeptic`, `forecast_evaluator`, `trend_aggregator`) and the shared `concept_curator`.

B OKF Schema Reference

Every document validates against a Pydantic contract before it is written (section 8). All documents share a common core — `type`, `granularity`, `kind`, `category`, `tags`, `timestamp`, `provenance` (agent, model, prompt version, generated-at) and `source_ids` (lineage) — and each class adds its own fields. Table 4 summarizes the classes.

Class	Contract	Class-specific fields
Article summary	`OKFArticleSummary`	`arxiv_id`, `resource`, `salience`, `concept_links`
Temporal digest	`OKFTemporalSummary`	`interval`, `coverage`, `key_findings_ledger`
Findings analysis	`OKFFindingsAnalysis`	`interval`, `findings[]`
Tendencies analysis	`OKFTendenciesAnalysis`	`interval`, `prediction_mode`, `predictions[]`
Concept node	`OKFConcept`	`related`, `attention`, `source_ids`

Table 4: OKF document classes and their distinguishing fields.

The machine fields are orthogonal: `granularity` $\in$ {article, daily, monthly, quarterly, yearly}, `kind` $\in$ {summary, analysis}, and, for analyses, `analysis_class` $\in$ {findings, tendencies}. The `resource` field is present only for article summaries; derived documents omit it and carry an `interval` and `source_ids` instead (section 5).

C CLI and Runbook

Every subcommand accepts the shared options after the subcommand: `-kb` (the store directory), `-registry`, `-prompts`, and `-v`/`-vv`/`-vvv` (phase / per-agent / library verbosity). Table 5 lists the subcommands.

A hard failure exits non-zero; `run-pipeline` exits non-zero if any stage failed or was skipped, so a scheduler can detect it (section 10). A partial success within the map (some papers skipped) is a success at the level of the run, its omissions recorded in the document's `coverage`. A daily cron entry has the form:

```
@daily arxiv-mas run-pipeline --category cs.AI --kb /srv/arxiv-mas/kb -v
```

Subcommand	Action
`summarize`	run the Summarizer for a category and date
`analyze`	run the Analyzer over a date range
`rollup`	eagerly build all due (complete) rollups (cron)
`curate-concepts`	enrich referenced concept nodes
`run-pipeline`	the full flow: summarize → rollup → analyze → curate
`count`	number of papers for a category/date, without downloading
`mcp-serve`	expose the KB over the MCP server

Table 5: Command-line subcommands.

D Knowledge-Base Layout

The store is a directory of OKF documents whose paths are their identities (section 5). The category is the top-level folder; kind and interval are encoded by sub-folders and by the file-name token. Interval tokens normalize the period: a day as `YYYY-MM-DD`, a month as `YYYY-MM`, a quarter as `YYYY-Qn`, a year as `YYYY`, and an arbitrary range as `start_end`.

```
kb/
  concepts/                              shared concept-graph nodes (cross-category
    )
    mixture-of-experts.md
  cs.AI/
    articles/2026/01/15/2401.12345.md    article summary (version-less file name)
    digests/
      daily/2026-01-15.md
      monthly/2026-01.md
      quarterly/2026-Q1.md
    findings/quarterly/2026-Q1.md
    tendencies/quarterly/2026-Q1.md
  cross_categories/                      multi-category analyses
    tendencies/quarterly/2026-Q1.md
```

Listing 4: The on-disk store layout.

E arXiv Category Taxonomy

arXiv organizes its e-prints into a three-level taxonomy: a handful of top-level *groups*, each containing one or more *archives*, each containing the *categories* (the leaf tags, of the form `archive.XX`) to which a paper is assigned. This system operates on one category at a time — the `-category` flag of appendix C takes a category identifier such as `cs.AI` or `cs.OH` — and the concept graph is the one structure shared across them.

Table 6 lists the full taxonomy as of this writing. The taxonomy evolves (archives and categories are occasionally added, renamed or retired), and a number of categories are aliases cross-listed under more than one archive (for example `math.MP` and `math-ph`, or `cs.SY` and `eess.SY`); the authoritative, current list is maintained at https://arxiv.org/category_taxonomy.

Table 6: The arXiv category taxonomy, grouped by top-level group and archive.

Identifier	Name
Computer Science (cs)	
cs.AI	Artificial Intelligence
cs.AR	Hardware Architecture
cs.CC	Computational Complexity
cs.CE	Computational Engineering, Finance, and Science
cs.CG	Computational Geometry
cs.CL	Computation and Language
cs.CR	Cryptography and Security
cs.CV	Computer Vision and Pattern Recognition
cs.CY	Computers and Society
cs.DB	Databases
cs.DC	Distributed, Parallel, and Cluster Computing
cs.DL	Digital Libraries
cs.DM	Discrete Mathematics
cs.DS	Data Structures and Algorithms
cs.ET	Emerging Technologies
cs.FL	Formal Languages and Automata Theory
cs.GL	General Literature
cs.GR	Graphics
cs.GT	Computer Science and Game Theory
cs.HC	Human-Computer Interaction
cs.IR	Information Retrieval
cs.IT	Information Theory
cs.LG	Machine Learning
cs.LO	Logic in Computer Science
cs.MA	Multiagent Systems
cs.MM	Multimedia
cs.MS	Mathematical Software
cs.NA	Numerical Analysis
cs.NE	Neural and Evolutionary Computing
cs.NI	Networking and Internet Architecture
cs.OH	Other Computer Science
cs.OS	Operating Systems
cs.PF	Performance
cs.PL	Programming Languages
cs.RO	Robotics
cs.SC	Symbolic Computation
cs.SD	Sound
cs.SE	Software Engineering
cs.SI	Social and Information Networks
cs.SY	Systems and Control
Economics (econ)	
econ.EM	Econometrics

(continued)

Identifier	Name
`econ.GN`	General Economics
`econ.TH`	Theoretical Economics
Electrical Engineering and Systems Science (`eess`)	
`eess.AS`	Audio and Speech Processing
`eess.IV`	Image and Video Processing
`eess.SP`	Signal Processing
`eess.SY`	Systems and Control
Mathematics (`math`)	
`math.AC`	Commutative Algebra
`math.AG`	Algebraic Geometry
`math.AP`	Analysis of PDEs
`math.AT`	Algebraic Topology
`math.CA`	Classical Analysis and ODEs
`math.CO`	Combinatorics
`math.CT`	Category Theory
`math.CV`	Complex Variables
`math.DG`	Differential Geometry
`math.DS`	Dynamical Systems
`math.FA`	Functional Analysis
`math.GM`	General Mathematics
`math.GN`	General Topology
`math.GR`	Group Theory
`math.GT`	Geometric Topology
`math.HO`	History and Overview
`math.IT`	Information Theory
`math.KT`	K-Theory and Homology
`math.LO`	Logic
`math.MG`	Metric Geometry
`math.MP`	Mathematical Physics
`math.NA`	Numerical Analysis
`math.NT`	Number Theory
`math.OA`	Operator Algebras
`math.OC`	Optimization and Control
`math.PR`	Probability
`math.QA`	Quantum Algebra
`math.RA`	Rings and Algebras
`math.RT`	Representation Theory
`math.SG`	Symplectic Geometry
`math.SP`	Spectral Theory
`math.ST`	Statistics Theory
Physics — Astrophysics (`astro-ph`)	
`astro-ph.CO`	Cosmology and Nongalactic Astrophysics
`astro-ph.EP`	Earth and Planetary Astrophysics

(continued)

Identifier	Name
astro-ph.GA	Astrophysics of Galaxies
astro-ph.HE	High Energy Astrophysical Phenomena
astro-ph.IM	Instrumentation and Methods for Astrophysics
astro-ph.SR	Solar and Stellar Astrophysics
Physics — Condensed Matter (cond-mat)	
cond-mat.dis-nn	Disordered Systems and Neural Networks
cond-mat.mes-hall	Mesoscale and Nanoscale Physics
cond-mat.mtrl-sci	Materials Science
cond-mat.other	Other Condensed Matter
cond-mat.quant-gas	Quantum Gases
cond-mat.soft	Soft Condensed Matter
cond-mat.stat-mech	Statistical Mechanics
cond-mat.str-el	Strongly Correlated Electrons
cond-mat.supr-con	Superconductivity
Physics — Nonlinear Sciences (nlin)	
nlin.AO	Adaptation and Self-Organizing Systems
nlin.CD	Chaotic Dynamics
nlin.CG	Cellular Automata and Lattice Gases
nlin.PS	Pattern Formation and Solitons
nlin.SI	Exactly Solvable and Integrable Systems
Physics — General Physics (physics)	
physics.acc-ph	Accelerator Physics
physics.ao-ph	Atmospheric and Oceanic Physics
physics.app-ph	Applied Physics
physics.atm-clus	Atomic and Molecular Clusters
physics.atom-ph	Atomic Physics
physics.bio-ph	Biological Physics
physics.chem-ph	Chemical Physics
physics.class-ph	Classical Physics
physics.comp-ph	Computational Physics
physics.data-an	Data Analysis, Statistics and Probability
physics.ed-ph	Physics Education
physics.flu-dyn	Fluid Dynamics
physics.gen-ph	General Physics
physics.geo-ph	Geophysics
physics.hist-ph	History and Philosophy of Physics
physics.ins-det	Instrumentation and Detectors
physics.med-ph	Medical Physics
physics.optics	Optics
physics.plasm-ph	Plasma Physics
physics.pop-ph	Popular Physics
physics.soc-ph	Physics and Society
physics.space-ph	Space Physics

(continued)

Identifier	Name
Physics — single-category archives	
gr-qc	General Relativity and Quantum Cosmology
hep-ex	High Energy Physics – Experiment
hep-lat	High Energy Physics – Lattice
hep-ph	High Energy Physics – Phenomenology
hep-th	High Energy Physics – Theory
math-ph	Mathematical Physics
nucl-ex	Nuclear Experiment
nucl-th	Nuclear Theory
quant-ph	Quantum Physics
Quantitative Biology (q-bio)	
q-bio.BM	Biomolecules
q-bio.CB	Cell Behavior
q-bio.GN	Genomics
q-bio.MN	Molecular Networks
q-bio.NC	Neurons and Cognition
q-bio.OT	Other Quantitative Biology
q-bio.PE	Populations and Evolution
q-bio.QM	Quantitative Methods
q-bio.SC	Subcellular Processes
q-bio.TO	Tissues and Organs
Quantitative Finance (q-fin)	
q-fin.CP	Computational Finance
q-fin.EC	Economics
q-fin.GN	General Finance
q-fin.MF	Mathematical Finance
q-fin.PM	Portfolio Management
q-fin.PR	Pricing of Securities
q-fin.RM	Risk Management
q-fin.ST	Statistical Finance
q-fin.TR	Trading and Market Microstructure
Statistics (stat)	
stat.AP	Applications
stat.CO	Computation
stat.ME	Methodology
stat.ML	Machine Learning
stat.OT	Other Statistics
stat.TH	Statistics Theory

F Glossary

Acronyms and domain terms used throughout the report. Acronyms are expanded on first use in the body; this list is the consolidated reference.

Acronyms

API
Application Programming Interface.

CLI
Command-Line Interface.

DAG
Directed Acyclic Graph — the shape of a pipeline of stages.

HTTP
HyperText Transfer Protocol.

ID
Identifier (e.g. a *Concept ID*).

JSON
JavaScript Object Notation.

KB
Knowledge Base — the shared OKF document store (the blackboard).

LLM
Large Language Model.

MAS
Multi-Agent System.

$\mathbf{MAS_i}$ / $\mathbf{MAS_o}$
the Summarizer and the Analyzer subsystems.

MCP
Model Context Protocol — the access protocol to the KB.

MoA
Mixture-of-Agents.

OKF
Open Knowledge Format — markdown documents with YAML frontmatter.

RAG
Retrieval-Augmented Generation.

YAML
YAML Ain't Markup Language — the configuration and frontmatter syntax.

Terms

Anti-averaging
: the aggregation policy that weighs evidence and flags tail risks instead of splitting the difference between opposing briefs.

Attention (signal)
: the temporal trend derived from a concept's backlinks (first/last reference and per-month counts).

Backlink
: a reference from a generated document to a concept node, via the document's `concept_links`.

Blackboard
: the architectural style in which independent knowledge sources cooperate through a shared knowledge store under a control component.

Brier score
: a proper scoring rule for probabilistic forecasts; lower is better, 0.25 is the chance baseline.

Concept graph
: the shared subtree of persistent concept nodes into which documents link.

Concept ID
: a document's identity: its path in the store, without the `.md` suffix.

Coverage
: the record of expected versus included constituents of a derived document.

Digest
: a temporal summary (daily, monthly, quarterly, yearly).

Findings
: a MAS_o analysis of concrete results a corpus establishes.

Key-findings ledger
: a structured carry-forward that preserves salient findings across rollup levels with minimal loss.

Lineage
: the source identifiers from which a document derives (`source_ids`), distinct from execution provenance.

Provenance
: execution metadata for a document: agent, model, prompt version and timestamp.

Rollup
: a higher-granularity digest built from finer digests.

Salience
a novelty/importance score in $[0, 1]$ assigned to an article.

Steelman
the strongest good-faith version of an argument, as required of the advocate and skeptic agents.

Structured output
an LLM response coerced to a typed schema (a Pydantic contract) before use.

Tendencies
a MAS_o analysis of trends and impact, carrying falsifiable predictions.

Tool use
an agent's ability to call functions (here, read-only KB tools) during its reasoning.

Store
the on-disk directory that holds the KB (its documents in OKF). The term *vault* is reserved, in the system, for the separate secret store.

References

Anthropic (2024). *Model Context Protocol (MCP) Specification*. URL: https://modelcontextprotocol.io/ (visited on 07/06/2026).

Brier, Glenn W. (1950). "Verification of Forecasts Expressed in Terms of Probability". In: *Monthly Weather Review* 78.1, pp. 1–3.

Dean, Jeffrey and Sanjay Ghemawat (2008). "MapReduce: Simplified Data Processing on Large Clusters". In: *Communications of the ACM* 51.1, pp. 107–113.

LangChain (2023). *LangChain Documentation*. URL: https://python.langchain.com/ (visited on 07/06/2026).

— (2024). *LangGraph Documentation*. URL: https://langchain-ai.github.io/langgraph/ (visited on 07/06/2026).

Lewis, Patrick et al. (2020). "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks". In: *Advances in Neural Information Processing Systems (NeurIPS)*. arXiv: 2005.11401.

Nii, H. Penny (1986). "Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures". In: *AI Magazine* 7.2, pp. 38–53.

Ollama (2023). *Ollama*. URL: https://ollama.com/ (visited on 07/06/2026).

Pydantic (2023). *Pydantic Documentation*. URL: https://docs.pydantic.dev/ (visited on 07/06/2026).

Wang, Junlin et al. (2024). *Mixture-of-Agents Enhances Large Language Model Capabilities*. arXiv: 2406.04692.