Desired State and End Goals

Writeup

[Author Name]

2026

Abstract

This work describes the conception, implementation, and empirical evaluation of **Agents-eval**, a three-tier evaluation framework for agentic AI systems based on PydanticAI. The framework addresses the growing need to systematically and reproducibly assess Multi-Agent Systems (MAS) by combining three complementary evaluation tiers: **Tier 1** encompasses text-based metrics (ROUGE, BLEU, BERTScore) for quantitative output analysis, **Tier 2** implements LLM-as-Judge evaluations for qualitative assessments, and **Tier 3** analyzes agent graph behavior for structural execution assessment.

The PeerRead corpus serves as the benchmark dataset, providing scientific peer-review data from leading conferences and journals. Agent orchestration is handled entirely through PydanticAI, which enables type-safe, model-agnostic agent construction.

The empirical evaluation is based on **30 traces** and compares four configurations: A **Manager-only** setup achieves a median throughput of 4.8 seconds per task with an error rate of 0%. The **3-agent** configuration requires a median of 12.3 seconds (+156% compared to Manager-only) with an error rate of 25%. In comparison, Claude Code-based systems show significantly higher resource requirements: **CC Solo** requires 118.3 seconds and $0.94 per execution, **CC Teams** 359.9 seconds and $1.35. PydanticAI-based agents prove to be 25 to 75 times faster and 50 to 100 times more cost-effective than the Claude Code baselines.

The framework was iteratively developed over **7 sprints** and is at version **3.3.0** at the time of completion. The results confirm that lightweight, specialized MAS architectures offer significant performance advantages over general-purpose coding agents, provided the task scope is clearly defined.

Keywords: Multi-Agent Systems, LLM Evaluation, PydanticAI, Agentic AI, Evaluation Framework, LLM-as-Judge, Peer Review, Benchmarking, Tracing, Observability

Contents

List of Figures

1 Project Introduction

1.1 Motivation and Problem Statement

1.1.1 The Evaluation Gap for Agentic AI Systems

The emergence of agentic AI systems has created a fundamental challenge in the field of artificial intelligence evaluation. Traditional benchmarking approaches, developed for assessing individual language models, fail to capture the emergent behaviors arising from multi-agent interactions: delegation patterns, collaborative decision-making, and dynamic task distribution among specialized agents [1].

Existing benchmarks such as the Berkeley Function-Calling Leaderboard [2], CORE-Bench [3], and GAIA [4] focus on individual model performance or narrowly defined capabilities. The question of how well a multi-agent system coordinates – that is, whether the manner of collaboration between agents leads to better outcomes than simpler approaches – remains unanswered within these frameworks [5].

Framework fragmentation exacerbates the problem: the proliferation of agentic frameworks such as PydanticAI [6], AutoGen [7], CrewAI [8], and LangChain [9] has created an ecosystem in which each framework implements its own evaluation approaches. Comparative analyses across framework boundaries are therefore methodologically impractical.

1.1.2 Goal: A Three-Tier Evaluation Framework

Agents-eval addresses this gap through an evaluation framework that combines three complementary assessment dimensions:

- **Tier 1 – Traditional Metrics**: Fast, objective text similarity metrics (BLEU, ROUGE, cosine similarity) as baseline validation
- **Tier 2 – LLM-as-a-Judge**: Semantic quality assessment through a configurable language model judge
- **Tier 3 – Graph-Based Analysis**: Coordination patterns from real execution traces, analyzed with NetworkX – the primary innovation of the framework

The PeerRead dataset [10], [11] serves as the evaluation domain, an established collection of scientific papers with structured peer reviews. A four-agent system (Manager → Researcher → Analyst → Synthesizer) generates reviews, which are subsequently evaluated through the three-tier pipeline.

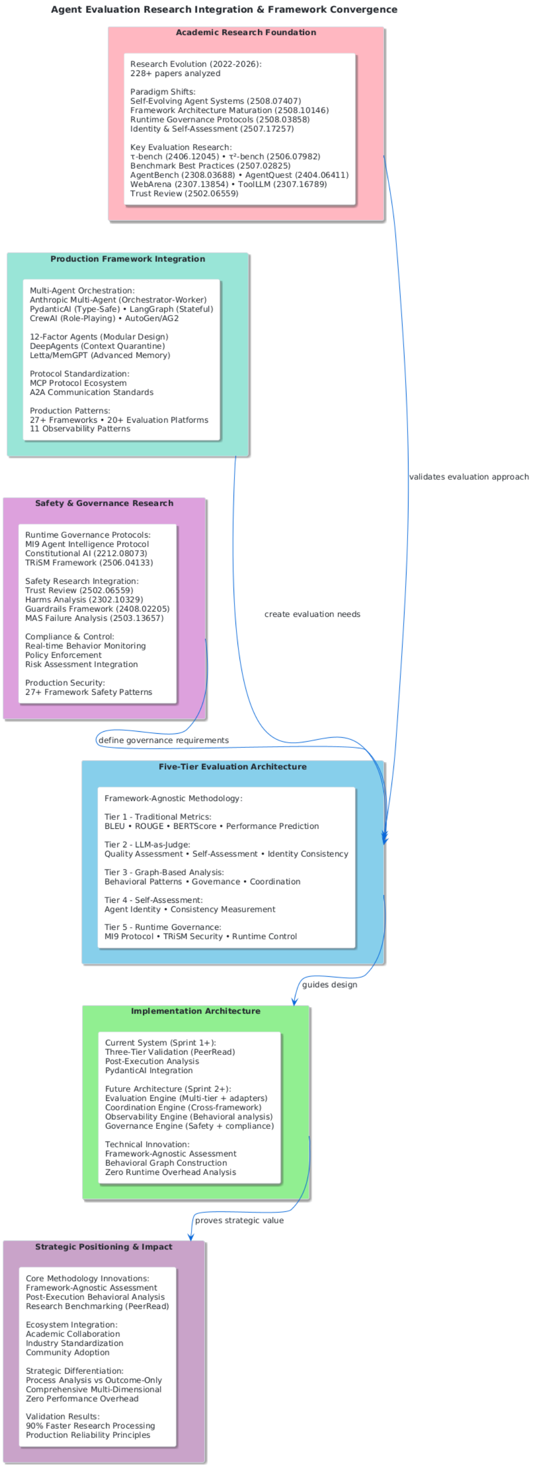

Agent Evaluation Research Integration & Framework Convergence
Academic Research Foundation
Research Evolution (2022-2026):
228+ papers analyzed
Paradigm Shifts:
Self-Evolving Agent Systems (2508.07407)
Framework Architecture Maturation (2508.10146)
Runtime Governance Protocols (2508.03858)
Identity & Self-Assessment (2507.17257)
Key Evaluation Research:
τ-bench (2406.12045) • τ^2-bench (2506.07982)
Benchmark Best Practices (2507.02825)
AgentBench (2308.03688) • AgentQuest (2404.06411)
WebArena (2307.13854) • ToolLLM (2307.16789)
Trust Review (2502.06559)
Production Framework Integration
Multi-Agent Orchestration:
Anthropic Multi-Agent (Orchestrator-Worker)
PydanticAI (Type-Safe) • LangGraph (Stateful)
CrewAI (Role-Playing) • AutoGen/AG2
12-Factor Agents (Modular Design)
DeepAgents (Context Quarantine)
Letta/MemGPT (Advanced Memory)
Protocol Standardization:
MCP Protocol Ecosystem
A2A Communication Standards
Production Patterns:
27+ Frameworks • 20+ Evaluation Platforms
11 Observability Patterns
validates evaluation approach
Safety & Governance Research
Runtime Governance Protocols:
MI9 Agent Intelligence Protocol
Constitutional AI (2212.08073)
TRiSM Framework (2506.04133)
Safety Research Integration:
Trust Review (2502.06559)
Harms Analysis (2302.10329)
Guardrails Framework (2408.02205)
MAS Failure Analysis (2503.13657)
Compliance & Control:
Real-time Behavior Monitoring
Policy Enforcement
Risk Assessment Integration
Production Security:
27+ Framework Safety Patterns
create evaluation needs
define governance requirements
Five-Tier Evaluation Architecture
Framework-Agnostic Methodology:
Tier 1 - Traditional Metrics:
BLEU • ROUGE • BERTScore • Performance Prediction
Tier 2 - LLM-as-Judge:
Quality Assessment • Self-Assessment • Identity Consistency
Tier 3 - Graph-Based Analysis:
Behavioral Patterns • Governance • Coordination
Tier 4 - Self-Assessment:
Agent Identity • Consistency Measurement
Tier 5 - Runtime Governance:
MI9 Protocol • TRiSM Security • Runtime Control
guides design
Implementation Architecture
Current System (Sprint 1+):
Three-Tier Validation (PeerRead)
Post-Execution Analysis
PydanticAI Integration
Future Architecture (Sprint 2+):
Evaluation Engine (Multi-tier + adapters)
Coordination Engine (Cross-framework)
Observability Engine (Behavioral analysis)
Governance Engine (Safety + compliance)
Technical Innovation:
Framework-Agnostic Assessment
Behavioral Graph Construction
Zero Runtime Overhead Analysis
proves strategic value
Strategic Positioning & Impact
Core Methodology Innovations:
Framework-Agnostic Assessment
Post-Execution Behavioral Analysis
Research Benchmarking (PeerRead)
Ecosystem Integration:
Academic Collaboration
Industry Standardization
Community Adoption
Strategic Differentiation:
Process Analysis vs Outcome-Only
Comprehensive Multi-Dimensional
Zero Performance Overhead
Validation Results:
90% Faster Research Processing
Production Reliability Principles

Figure 1.1: Visualization of the synthesis of research literature, production framework analysis, and systematic development approach informing the project.

1.2 Current Project Status

1.2.1 Development Progress (Sprint 1–7)

The project has been progressing since Sprint 1 in iterative two-week cycles. The following table shows the progress:

Sprint	Status	Focus
1	Delivered	Three-tier evaluation framework, PeerRead integration
2	Delivered	Eval wiring, trace capture, Logfire + Phoenix, Streamlit dashboard
3	Delivered	Plugin architecture, GUI wiring, test alignment, trace quality
4	Delivered	Operational resilience, Claude Code baseline comparison (Solo + Teams)
5	Delivered	Runtime fixes, GUI improvements, architecture improvements, code review
6	Delivered	Benchmarking infrastructure, CC baseline, security hardening, test quality
7	Active	Documentation, examples, test refactoring, GUI, unified providers
8	Draft	Report generation, graph alignment, MAESTRO hardening, streaming

1.2.2 Technical Implementation

The current implementation (version 3.3.0, Sprint 7 active) comprises the following core components:

Multi-Agent System: Four specialized agents orchestrated with PydanticAI [6]:

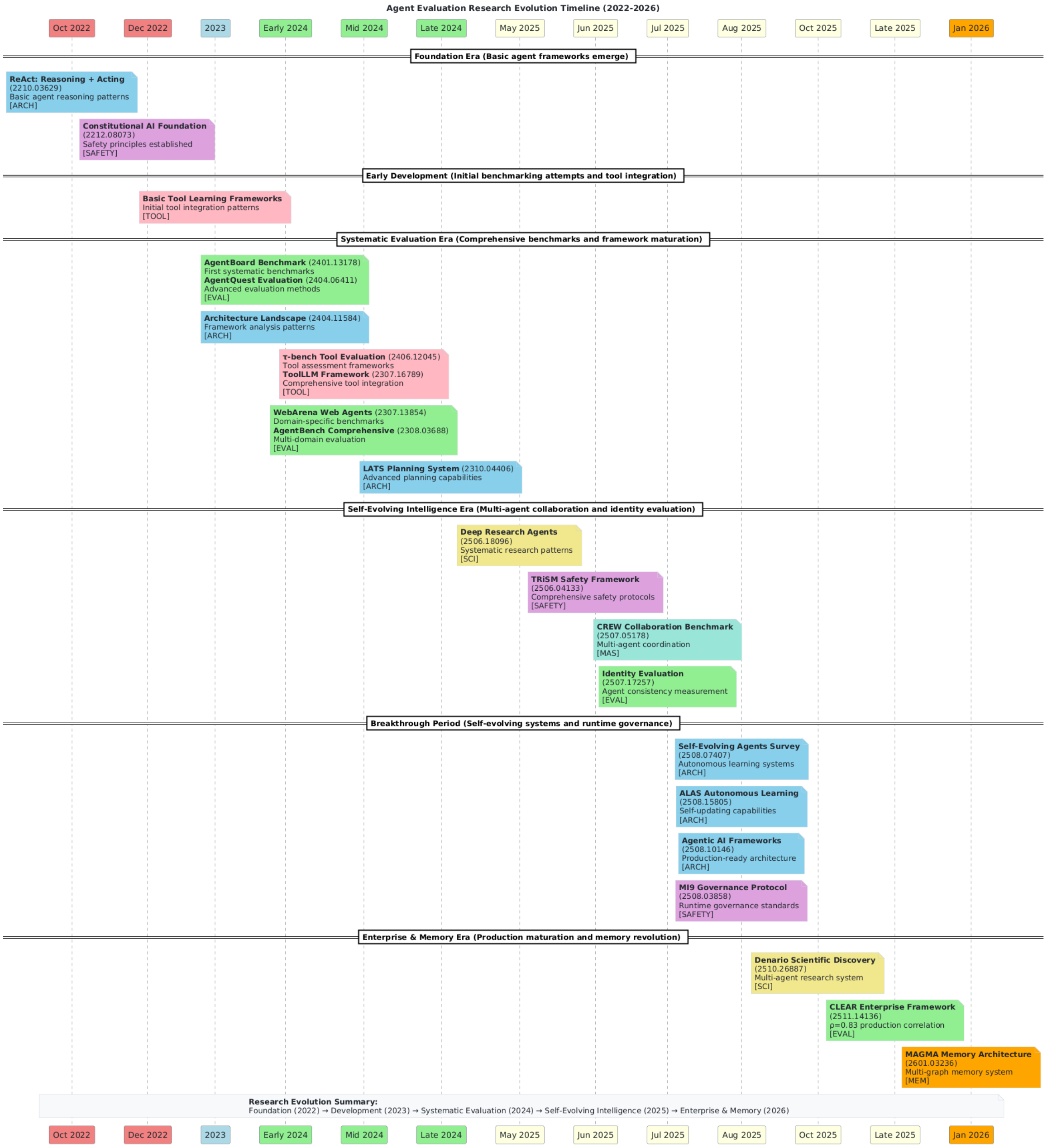
- *Manager Agent*: Primary orchestrator for task delegation and coordination
- *Researcher Agent*: Information gathering with DuckDuckGo search integration
- *Analyst Agent*: Critical evaluation and data validation
- *Synthesizer Agent*: Generation of structured scientific reports

Evaluation Pipeline: Plugin-based architecture (`EvaluatorPlugin` interface) with typed context passing between tiers. Six equally weighted metrics (16.7% each): planning rationality, task success, tool efficiency, coordination quality, execution time, and output similarity.

Observability: Logfire auto-instrumentation with Arize Phoenix for trace inspection; Streamlit dashboard for Tier 1/2/3 result display and interactive agent graph visualization.

Security (Sprint 6): SSRF prevention through URL validation with domain allowlisting, prompt injection resistance, sensitive data masking in logs and traces.

Benchmarking (Sprint 6): `SweepRunner` for 8 agent compositions x N papers x N repetitions with statistical analysis (mean, standard deviation, min/max).


Agent Evaluation Research Evolution Timeline (2022-2026)
Oct 2022
Dec 2022
2023
Early 2024
Mid 2024
Late 2024
May 2025
Jun 2025
Jul 2025
Aug 2025
Oct 2025
Late 2025
Jan 2026
Foundation Era (Basic agent frameworks emerge)
ReAct: Reasoning + Acting (2210.03629) Basic agent reasoning patterns [ARCH]
Constitutional AI Foundation (2212.08073) Safety principles established [SAFETY]
Early Development (Initial benchmarking attempts and tool integration)
Basic Tool Learning Frameworks Initial tool integration patterns [TOOL]
Systematic Evaluation Era (Comprehensive benchmarks and framework maturation)
AgentBoard Benchmark (2401.13178) First systematic benchmarks AgentQuest Evaluation (2404.06411) Advanced evaluation methods [EVAL]
Architecture Landscape (2404.11584) Framework analysis patterns [ARCH]
τ-bench Tool Evaluation (2406.12045) Tool assessment frameworks ToolLLM Framework (2307.16789) Comprehensive tool integration [TOOL]
WebArena Web Agents (2307.13854) Domain-specific benchmarks AgentBench Comprehensive (2308.03688) Multi-domain evaluation [EVAL]
LATS Planning System (2310.04406) Advanced planning capabilities [ARCH]
Self-Evolving Intelligence Era (Multi-agent collaboration and identity evaluation)
Deep Research Agents (2506.18096) Systematic research patterns [SCI]
TRiSM Safety Framework (2506.04133) Comprehensive safety protocols [SAFETY]
CREW Collaboration Benchmark (2507.05178) Multi-agent coordination [MAS]
Identity Evaluation (2507.17257) Agent consistency measurement [EVAL]
Breakthrough Period (Self-evolving systems and runtime governance)
Self-Evolving Agents Survey (2508.07407) Autonomous learning systems [ARCH]
ALAS Autonomous Learning (2508.15805) Self-updating capabilities [ARCH]
Agentic AI Frameworks (2508.10146) Production-ready architecture [ARCH]
MI9 Governance Protocol (2508.03858) Runtime governance standards [SAFETY]
Enterprise & Memory Era (Production maturation and memory revolution)
Denario Scientific Discovery (2510.26887) Multi-agent research system [SCI]
CLEAR Enterprise Framework (2511.14136) ρ=0.83 production correlation [EVAL]
MAGMA Memory Architecture (2601.03236) Multi-graph memory system [MEM]
Research Evolution Summary:
Foundation (2022) → Development (2023) → Systematic Evaluation (2024) → Self-Evolving Intelligence (2025) → Enterprise & Memory (2026)
Oct 2022
Dec 2022
2023
Early 2024
Mid 2024
Late 2024
May 2025
Jun 2025
Jul 2025
Aug 2025
Oct 2025
Late 2025
Jan 2026

Research Category Legend:

Color	Category	Tag
🟪	Safety & Governance	[SAFETY]
🟦	Architecture & Systems	[ARCH]
🟩	Evaluation & Benchmarking	[EVAL]
🟥	Tool Use & Integration	[TOOL]
🟩	Multi-Agent Systems	[MAS]
🟨	Scientific Discovery	[SCI]
🟧	Memory Systems	[MEM]

Paper Selection Criteria:
- Foundational breakthroughs (Constitutional AI, ReAct)
- Major benchmarks shaping the field (AgentBench, WebArena)
- Framework evolution milestones (Architecture Landscape)
- 2025-08 breakthrough cluster (Self-Evolving Agents, MI9)
- Representative papers showing category evolution

Figure 1.2: Chronological overview of the development phases from Sprint 1 to the current state.

1.2.3 Research Context

The project is embedded in current research on LLM-based agent systems [1], [5]. The choice of the PeerRead dataset [10] enables the use of established peer review quality standards as the evaluation foundation. The three-tier architecture is motivated by insights into evaluation methodologies for agentic systems, particularly the distinction between *what* a system produces (Tiers 1 and 2) and *how* it coordinates (Tier 3).

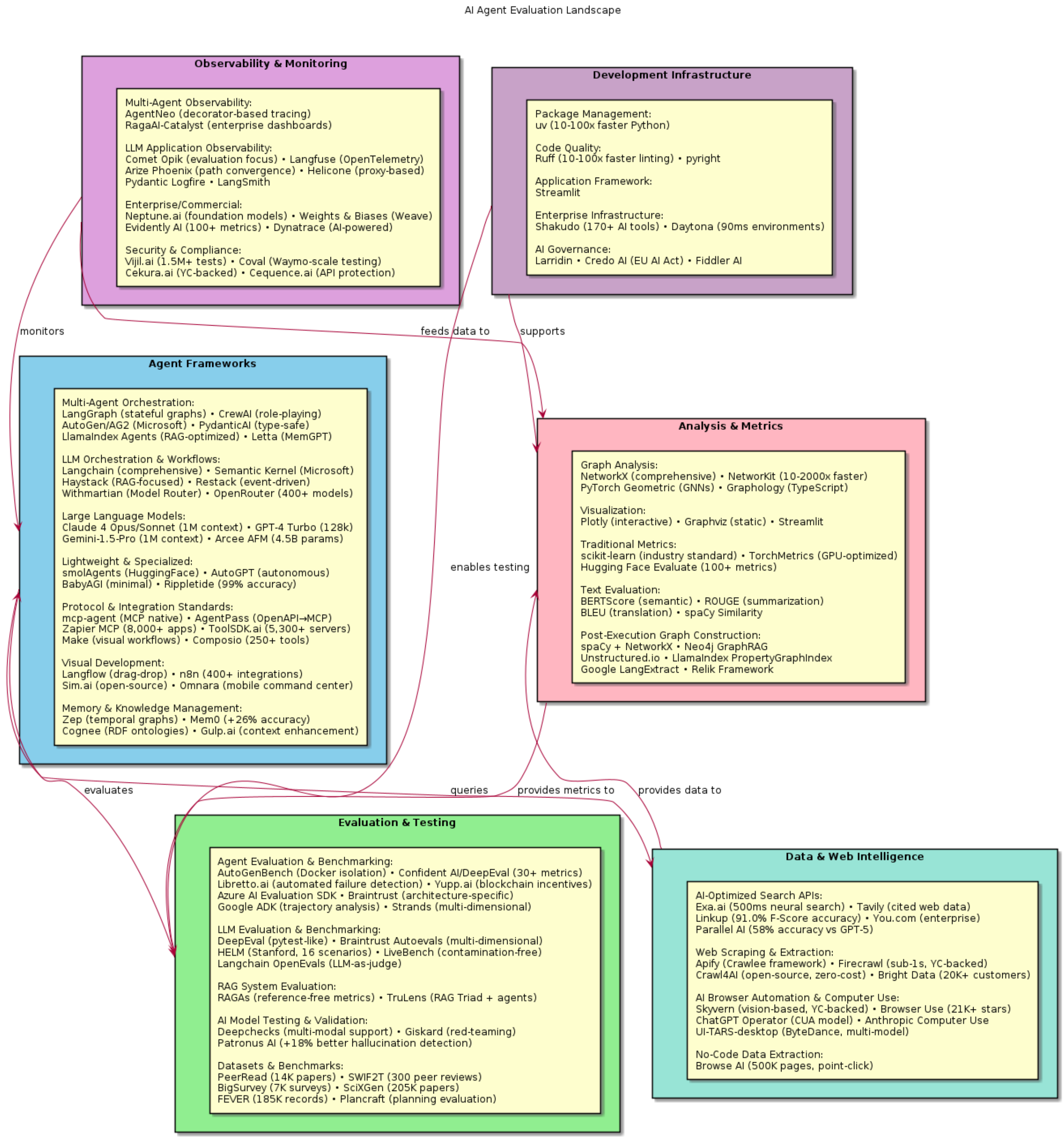

AI Agent Evaluation Landscape
Observability & Monitoring
Multi-Agent Observability:
AgentNeo (decorator-based tracing)
RagaAI-Catalyst (enterprise dashboards)
LLM Application Observability:
Comet Opik (evaluation focus) • Langfuse (OpenTelemetry)
Arize Phoenix (path convergence) • Helicone (proxy-based)
Pydantic Logfire • LangSmith
Enterprise/Commercial:
Neptune.ai (foundation models) • Weights & Biases (Weave)
Evidently AI (100+ metrics) • Dynatrace (AI-powered)
Security & Compliance:
Vijil.ai (1.5M+ tests) • Coval (Waymo-scale testing)
Cekura.ai (YC-backed) • Cequence.ai (API protection)
Development Infrastructure
Package Management:
uv (10-100x faster Python)
Code Quality:
Ruff (10-100x faster linting) • pyright
Application Framework:
Streamlit
Enterprise Infrastructure:
Shakudo (170+ AI tools) • Daytona (90ms environments)
AI Governance:
Larridin • Credo AI (EU AI Act) • Fiddler AI
monitors
feeds data to
supports
Agent Frameworks
Multi-Agent Orchestration:
LangGraph (stateful graphs) • CrewAI (role-playing)
AutoGen/AG2 (Microsoft) • PydanticAI (type-safe)
LlamaIndex Agents (RAG-optimized) • Letta (MemGPT)
LLM Orchestration & Workflows:
Langchain (comprehensive) • Semantic Kernel (Microsoft)
Haystack (RAG-focused) • Restack (event-driven)
Withmartian (Model Router) • OpenRouter (400+ models)
Large Language Models:
Claude 4 Opus/Sonnet (1M context) • GPT-4 Turbo (128k)
Gemini-1.5-Pro (1M context) • Arcee AFM (4.5B params)
Lightweight & Specialized:
smolAgents (HuggingFace) • AutoGPT (autonomous)
BabyAGI (minimal) • Rippletide (99% accuracy)
Protocol & Integration Standards:
mcp-agent (MCP native) • AgentPass (OpenAPI→MCP)
Zapier MCP (8,000+ apps) • ToolSDK.ai (5,300+ servers)
Make (visual workflows) • Composio (250+ tools)
Visual Development:
Langflow (drag-drop) • n8n (400+ integrations)
Sim.ai (open-source) • Omnara (mobile command center)
Memory & Knowledge Management:
Zep (temporal graphs) • Mem0 (+26% accuracy)
Cognee (RDF ontologies) • Gulp.ai (context enhancement)
enables testing
Analysis & Metrics
Graph Analysis:
NetworkX (comprehensive) • NetworKit (10-2000x faster)
PyTorch Geometric (GNNs) • Graphology (TypeScript)
Visualization:
Plotly (interactive) • Graphviz (static) • Streamlit
Traditional Metrics:
scikit-learn (industry standard) • TorchMetrics (GPU-optimized)
Hugging Face Evaluate (100+ metrics)
Text Evaluation:
BERTScore (semantic) • ROUGE (summarization)
BLEU (translation) • spaCy Similarity
Post-Execution Graph Construction:
spaCy + NetworkX • Neo4j GraphRAG
Unstructured.io • LlamaIndex PropertyGraphIndex
Google LangExtract • Relik Framework
evaluates
queries
provides metrics to
provides data to
Evaluation & Testing
Agent Evaluation & Benchmarking:
AutoGenBench (Docker isolation) • Confident AI/DeepEval (30+ metrics)
Libretto.ai (automated failure detection) • Yupp.ai (blockchain incentives)
Azure AI Evaluation SDK • Braintrust (architecture-specific)
Google ADK (trajectory analysis) • Strands (multi-dimensional)
LLM Evaluation & Benchmarking:
DeepEval (pytest-like) • Braintrust Autoevals (multi-dimensional)
HELM (Stanford, 16 scenarios) • LiveBench (contamination-free)
Langchain OpenEvals (LLM-as-judge)
RAG System Evaluation:
RAGAs (reference-free metrics) • TruLens (RAG Triad + agents)
AI Model Testing & Validation:
Deepchecks (multi-modal support) • Giskard (red-teaming)
Patronus AI (+18% better hallucination detection)
Datasets & Benchmarks:
PeerRead (14K papers) • SWIF2T (300 peer reviews)
BigSurvey (7K surveys) • SciXGen (205K papers)
FEVER (185K records) • Plancraft (planning evaluation)
Data & Web Intelligence
AI-Optimized Search APIs:
Exa.ai (500ms neural search) • Tavily (cited web data)
Linkup (91.0% F-Score accuracy) • You.com (enterprise)
Parallel AI (58% accuracy vs GPT-5)
Web Scraping & Extraction:
Apify (Crawlee framework) • Firecrawl (sub-1s, YC-backed)
Crawl4AI (open-source, zero-cost) • Bright Data (20K+ customers)
AI Browser Automation & Computer Use:
Skyvern (vision-based, YC-backed) • Browser Use (21K+ stars)
ChatGPT Operator (CUA model) • Anthropic Computer Use
UI-TARS-desktop (ByteDance, multi-model)
No-Code Data Extraction:
Browse AI (500K pages, point-click)

Figure 1.3: Overview of the current AI agent framework landscape, in which Agents-eval is positioned.

2 Desired State and End Goals

2.1 Vision

2.1.1 Strategic Goals

The strategic vision of Agents-eval encompasses three goal areas:

Universal Evaluation Standard: Agents-eval aims to be established as a reference framework for the assessment of agentic AI systems – with standardized metrics that enable systematic comparisons across different frameworks, models, and configurations.

Technology-Agnostic Extensibility: The framework shall support additional agentic frameworks (AutoGen, CrewAI, LangChain) beyond the current PydanticAI implementation through a pluggable adapter architecture. Standardized interfaces and Pydantic data models form the foundation for this.

Continuous Innovation Platform: The architecture shall adapt to emerging agentic AI paradigms and new evaluation methodologies without losing backward compatibility. New metrics are integrated as plugins without modifying the existing pipeline.

2.1.2 Distinction from Current State

The current state (Sprint 7 active) delivers a functional framework with PydanticAI as the sole agent framework, seven identified candidate metrics for future integration, and an evaluation domain limited to PeerRead. The desired state extends this through multi-framework support, a broader metric palette, and cross-domain evaluation scenarios.

2.2 Target Architecture

The target architecture extends the existing plugin architecture across four layers:

Abstraction Layer: Technology-agnostic interfaces (`EvaluatorPlugin`, `AgentAdapter`) enable the integration of arbitrary agentic frameworks without changes to the evaluation pipeline.

Evaluation Engine: Extension of the three-tier architecture with new metrics from the candidate catalog (see table below). Configurable weighting and adaptive weight redistribution for missing tier results are retained.

Observability Layer: Expansion of the existing Logfire/Phoenix integration with structured trace analysis for multi-framework comparisons and sweep results.

Report Generation (Sprint 8): Structured Markdown reports with tier scores, identified weaknesses, and actionable improvement suggestions – available via CLI (`--generate-report`) and Streamlit GUI.

2.3 Roadmap (Sprint 8+)

Sprint 8 (Draft) focuses on:

- **Feature 1**: Report generation in CLI and GUI with actionable suggestions
- **Feature 2–3**: Graph attribute alignment and MAESTRO security hardening
- **Feature 4–5**: Code quality improvements and PydanticAI streaming support

Beyond that, the following extensions are planned for later sprints:

- Integration of candidate metrics from the research context
- Multi-framework adapters (AutoGen, CrewAI, LangChain)
- Optional containerized deployment modes (ADR-007, unscheduled)
- `--engine=claude-api` as a separate comparison mode for model-vs-model analyses (ADR-008)

2.4 Quantitative Success Goals

The following table shows candidate metrics identified for future integration, ordered by priority:

Metric	Source	Complexity	Impact
`fix_rate`	SWE-EVO	Low	High
`evaluator_consensus`	TEAM-PHI / Agents4Science	Low	High
`delegation_depth`	HDO / Agents4Science	Low	High
`rubric_alignment`	[2512.23707]	Medium	High
`handoff_quality`	Arize Multi-Agent	Medium	High
`coordination_topology`	Evolutionary Boids	Low	Medium
`path_convergence`	Arize Phoenix	Low	Medium

Technical target values include:

- Evaluation latency under one second for Tier 1 (traditional metrics)
- Complete validation (`make validate`) with zero critical security findings
- Test coverage above 60% for all critical modules (achieved for `llms/models.py`, `agent_factories.py`, `datasets_peerread.py` in Sprint 6)

- Sweep results for all 8 agent compositions reproducible with statistically evaluable sample sizes

3 Planning and Solution

3.1 Three-Tier Evaluation Architecture

The developed solution is based on a three-tier evaluation architecture that combines complementary assessment methods to enable comprehensive evaluation of multi-agent systems. Each tier addresses a different dimension of agentic performance and mutually validates the findings of the other tiers.

3.1.1 Tier 1: Traditional Metrics

The first evaluation tier implements classical, objective text similarity metrics as a quantitative baseline. This tier fulfills the role of a fast validator and delivers deterministically reproducible measurements without dependency on external language models.

Output Similarity Scoring (`output_similarity`): To determine the similarity between the system-generated review and the human reference review from the PeerRead dataset, three similarity measures are employed:

- Cosine similarity via TF-IDF vectorization (primary metric)
- Jaccard similarity with `textdistance` support (secondary metric)
- Semantic similarity on TF-IDF basis (default metric from configuration)

Execution Time (`time_taken`): The end-to-end processing duration of a scientific paper is measured and normalized. Additionally, resource utilization, API calls, and token consumption are captured.

Task Success (`task_success`): The completeness and structural correctness of the generated review is evaluated. Academic standard conformity and configurable recommendation weights are factored into the assessment.

The implementation resides in `src/app/judge/plugins/traditional.py`.

3.1.2 Tier 2: LLM-as-Judge

The second tier employs a language model as evaluator to capture qualitative and semantic aspects not accessible to traditional metrics. A configured judge provider evaluates the agentic execution based on structured criteria.

Planning Rationality (`planning_rationality`): The decision logic of the agent system, the coherence of the reasoning chain, and the strategic effectiveness of planning are evaluated.

Tool Efficiency (tool_efficiency): The effectiveness of tool usage, resource optimization, and API call efficiency are analyzed.

Recommendation Quality: Generated reviews are compared against ground-truth reviews from PeerRead. Configurable recommendation weights control the evaluation: accept (1.0), weak_accept (0.7), weak_reject (-0.7), reject (-1.0) [12].

A provider fallback mechanism (introduced in Sprint 5) validates API key availability before invocation. If no provider is available, Tier 2 is skipped entirely and the weights are redistributed to the remaining metrics.

The implementation resides in src/app/judge/plugins/llm_judge.py.

3.1.3 Tier 3: Graph-Based Behavioral Analysis

The third evaluation tier represents the primary innovation of the framework. Rather than directly observing agentic behavior, execution traces are analyzed post-hoc and transformed into behavioral graphs.

Coordination Quality (coordination_quality): Agent interaction patterns are extracted from execution traces using NetworkX graph analysis. Centrality metrics quantify the communication efficiency between Manager, Researcher, Analyst, and Synthesizer agents.

Execution Graph Construction: Logfire auto-instrumentation captures comprehensive execution traces. NetworkX constructs behavioral graphs from these, mapping coordination patterns, tool usage sequences, and decision flows.

Complexity Metrics Integration: Node count (discrete actions), edge density (interaction frequency), and path optimization are incorporated into the overarching metrics.

The approach of post-hoc graph analysis (ADR-004) avoids performance overhead during agent execution and preserves agent autonomy in tool selection. The implementation resides in src/app/judge/plugins/graph_metrics.py.

3.1.4 Integrated Pipeline

The three tiers operate as an integrated pipeline where results and context are passed sequentially:

Tier	Role	Focus
Tier 1 (Traditional)	VALIDATOR	Fast, objective text similarity baseline
Tier 2 (LLM-Judge)	VALIDATOR	Semantic quality assessment
Tier 3 (Graph)	PRIMARY	Coordination patterns from execution traces

The validation logic follows a clear principle: if all three tiers agree, there is high confidence in the assessment quality. If Tier 3 is positive but Tiers 1 and 2 are negative, this indicates good coordination with weak output quality. The reverse case signals high-quality output with inefficient coordination.

3.2 Four-Agent Architecture

3.2.1 Agent Roles and Specialization

The system implements a specialized four-agent architecture that models collaborative research scenarios and provides realistic evaluation complexity.

Manager Agent: Primary orchestrator responsible for task delegation, coordination oversight, quality assurance, and system-wide decision-making. The Manager Agent serves as the central coordination point and ensures coherent system operation across all agent interactions. Large context window models are preferentially employed.

Researcher Agent: Specialized in information gathering with DuckDuckGo search integration for data collection, literature research, and fact verification [13]. This agent provides the external information acquisition capability required for comprehensive analysis. PeerRead dataset tools are assigned to this agent (separation of concerns, Sprint 5).

Analyst Agent: Focused on critical evaluation, data validation, and accuracy verification of research findings. The Analyst Agent provides the analytical capabilities for rigorous scientific assessment and returns detailed feedback when findings are not approved.

Synthesizer Agent: Generates coherent, well-structured reports that integrate insights from all agents. This agent transforms collaborative analysis into structured, scientifically formulated outputs while maintaining the original facts, conclusions, and sources.

All four agents are implemented in `src/app/agents/agent_system.py`.

3.2.2 Coordination Protocols

Hierarchical Delegation Structure: The Manager Agent serves as the primary decision-maker for task allocation and coordination oversight. Specialized agents operate with defined autonomy within their domains.

Data Flow: PeerRead paper input → Manager Agent → optional delegation to Researcher Agent (with DuckDuckGo search) → optional results to Analyst Agent for validation → validated data to Synthesizer Agent → generated review → evaluation pipeline.

Error Handling: Robust mechanisms for agent failure and graceful degradation are implemented. The system maintains operational capability even during individual component failures.

Agent Composition Modes: The `SweepRunner` module (Sprint 6) enables systematic evaluation of eight different agent compositions, from single-agent configurations to full four-agent collaboration.

3.3 Metrics Framework

3.3.1 Six-Dimensional Assessment Architecture

The evaluation framework implements six equally weighted assessment dimensions that prevent optimization bias while ensuring comprehensive capability assessment.

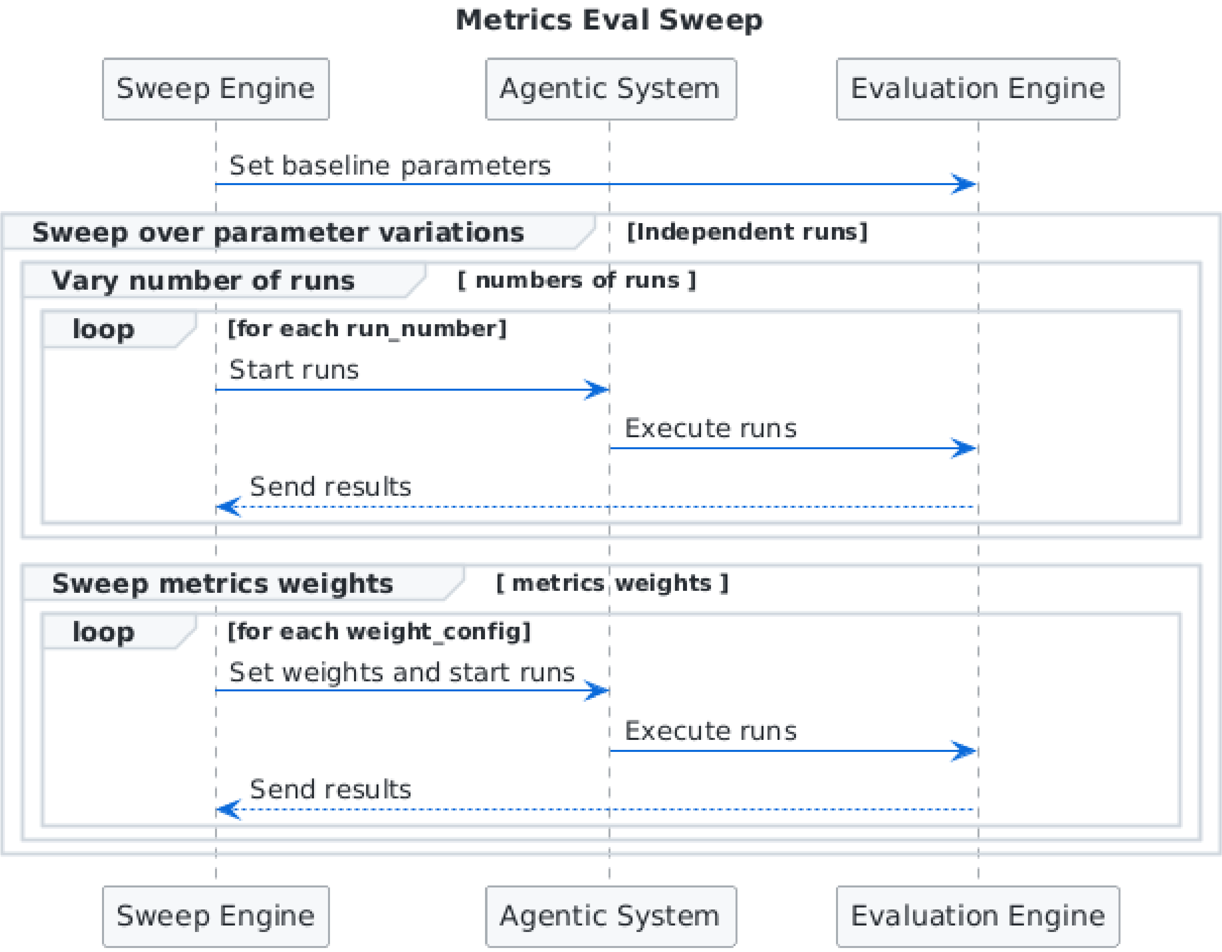


Figure 3.1: Six-dimensional evaluation architecture with sweep analysis

The six metrics each comprise 16.7 percent of the total score:

- **`planning_rationality`** (16.7%): Assessment of decision logic, reasoning coherence, and strategic planning effectiveness
- **`task_success`** (16.7%): Quantification of review completeness, structural correctness, and academic standard conformity

- **`tool_efficiency`** (16.7%): Analysis of tool usage effectiveness, resource optimization, and API call efficiency
- **`coordination_quality`** (16.7%): Measurement of inter-agent communication effectiveness via graph centrality metrics
- **`time_taken`** (16.7%): Performance efficiency measurement and normalized execution time
- **`output_similarity`** (16.7%): Semantic alignment with PeerRead ground-truth reviews

The implementation resides in `src/app/judge/composite_scorer.py`.

3.3.2 Composite Scoring

The composite score calculation follows the formula:

`Agent Score = Weighted sum of six core metrics`

Configuration-based thresholds classify the result into three categories:

- **accept**: Composite Score ≥ 0.863
- **weak_accept**: Composite Score ≥ 0.626
- **reject**: Composite Score < 0.626

All metric weights are configured via `JudgeSettings` through pydantic-settings and support override via environment variables with the prefix `JUDGE_`.

3.3.3 Adaptive Weight Redistribution

The composite scoring system automatically detects whether a single-agent run is present by checking `GraphTraceData` for 0–1 unique agent IDs and empty `coordination_events`.

In single-agent mode, the `coordination_quality` metric (weight 0.167) is excluded and its weight is evenly distributed among the remaining five metrics (0.20 each). The `CompositeResult` object contains a `single_agent_mode: bool` flag that transparently documents the redistribution.

When Tier 2 absence (no valid provider) and single-agent mode are combined, all weights are redistributed to the available metrics so that the sum always equals ~1.0:

```python
# Reason: Compound redistribution ensures weights always sum to ~1.0
if single_agent_mode and tier2_skipped:
   available_metrics = [m for m in all_metrics
                 if m not in excluded_metrics]
   weight_per_metric = 1.0 / len(available_metrics)
```

Code excerpt from `src/app/judge/composite_scorer.py`

3.4 PeerRead Dataset Integration

The PeerRead dataset serves as the primary evaluation benchmark scenario. It comprises over 14,000 scientific papers with structured peer reviews, acceptance and rejection decisions, and detailed metadata from leading conferences including NIPS, ICLR, and ACL [10].

Ground-Truth Validation: PeerRead reviews serve as references for `output_similarity` and LLM-Judge evaluation. This enables objective performance measurement and validation of agentic system capabilities.

Pydantic Data Models: The dataset integration uses `validation_alias` and `ConfigDict(populate_by_name=True)` to map external field names (e.g., `IMPACT` → `impact`) to internal model attributes. The models reside in `src/app/data_models/peerread_models.py`.

Resilient Validation: Optional PeerRead fields (IMPACT, SUBSTANCE) are handled tolerantly. Missing values do not abort the evaluation pipeline but are replaced with configurable default values.

Download and Caching: The downloader in `src/app/data_utils/datasets_peerread.py` supports venue-specific splits (e.g., `iclr_2017`, `acl_2017`) and stores data in a configurable cache directory for offline use.

Benchmarking Sweep: The `SweepRunner` (Sprint 6) enables systematic evaluation of multiple agent compositions over N papers and M repetitions. Statistical analysis (`SweepAnalyzer`) computes mean, standard deviation, minimum, and maximum per composition.

3.5 Plugin Architecture

3.5.1 EvaluatorPlugin Interface

All evaluation modules (Traditional, LLM-Judge, Graph) implement a common abstract base class that ensures type-safe and extensible plugin integration (ADR-005):

```python
class EvaluatorPlugin(ABC):
  @property
  @abstractmethod
  def name(self) -> str: ...

  @property
  @abstractmethod
  def tier(self) -> int: ...

  @abstractmethod
```

```
    def evaluate(self, context: BaseModel) -> BaseModel: ...

    @abstractmethod
    def get_context_for_next_tier(self, result: BaseModel) ->
     ↪  BaseModel: ...
```

Code excerpt from `src/app/judge/plugins/`

The interface follows the Adapter pattern: existing evaluation engines are embedded without modification of the core pipeline code. New metrics can be added without interrupting existing functionality (12-Factor Principles #4, #10, #12).

Inter-plugin data passing occurs exclusively through typed Pydantic models (no raw dictionaries). Each plugin returns a typed context via `get_context_for_next_tier()` that is consumed by the subsequent tier.

3.5.2 PluginRegistry and Tier Execution

The `PluginRegistry` serves as the central management instance for plugin discovery and tier-ordered execution. Plugins register themselves at import time and are executed in the order Tier 1 → Tier 2 → Tier 3:

```
class PluginRegistry:
    def register(self, plugin: EvaluatorPlugin) -> None: ...
    def get_plugins_by_tier(self, tier: int) -> list[EvaluatorPlugin]:
     ↪  ...
    def execute_all(self, context: BaseModel) -> list[BaseModel]: ...
```

Code excerpt from `src/app/judge/`

JudgeSettings Configuration replaces JSON configuration files with a `pydantic-settings` `BaseSettings` class with the environment variable prefix `JUDGE_` (ADR-006). Timeouts, tier weights, and metric parameters are fully configurable:

```
class JudgeSettings(BaseSettings):
    model_config = SettingsConfigDict(env_prefix="JUDGE_")

    tier1_timeout: int = 30
    tier2_timeout: int = 60
    tier3_timeout: int = 45
    tier_weights: dict[int, float] = {1: 0.33, 2: 0.33, 3: 0.34}
```

Code excerpt from `src/app/judge/`

3.6 System Architecture Overview

3.6.1 C4 Model Overview

The system architecture follows the C4 model [14] and documents the system at different abstraction levels. The following diagram shows the high-level system components and their relationships.

The architectural design emphasizes clear separation between the Multi-Agent System (MAS) responsible for review generation and the evaluation system responsible for assessment and analysis. This separation enables independent evolution of both subsystems while maintaining clean interfaces and data contracts.

The system follows core architectural principles: modular design enables independent development of each major component with clearly defined interfaces. Technology-agnosticism ensures that abstract interfaces enable support for multiple agentic frameworks, LLM providers, and evaluation methodologies without architectural changes.

3.6.2 Detailed Component Architecture

The detailed architecture reveals the interaction patterns between system components. The main application layer serves as the primary orchestration point: it manages user interactions through CLI and Streamlit GUI interfaces, coordinates agent sessions, and routes evaluation requests to appropriate subsystems.

The Agent System core implements multi-agent coordination logic with the PydanticAI framework [6] and manages agent lifecycles, inter-agent communication, and task delegation patterns across the four specialized agents.

3.6.3 Review Workflow

The workflow architecture demonstrates the agent coordination patterns:

Primary Workflow: User request → Manager Agent (paper retrieval) → template-based review generation → LLM processing → structured review output → persistent storage

Delegation Workflow: Manager Agent → Researcher Agent activation → DuckDuckGo search execution → research synthesis → result integration into main workflow

Quality Assurance: Built-in validation at each stage ensures data integrity and consistency across different execution paths.

3.6.4 Workflow Evolution

The development of the multi-agent evaluation framework proceeded through systematic architectural refinements, documenting the evolution from basic agent coordination to sophisticated collaborative intelligence.

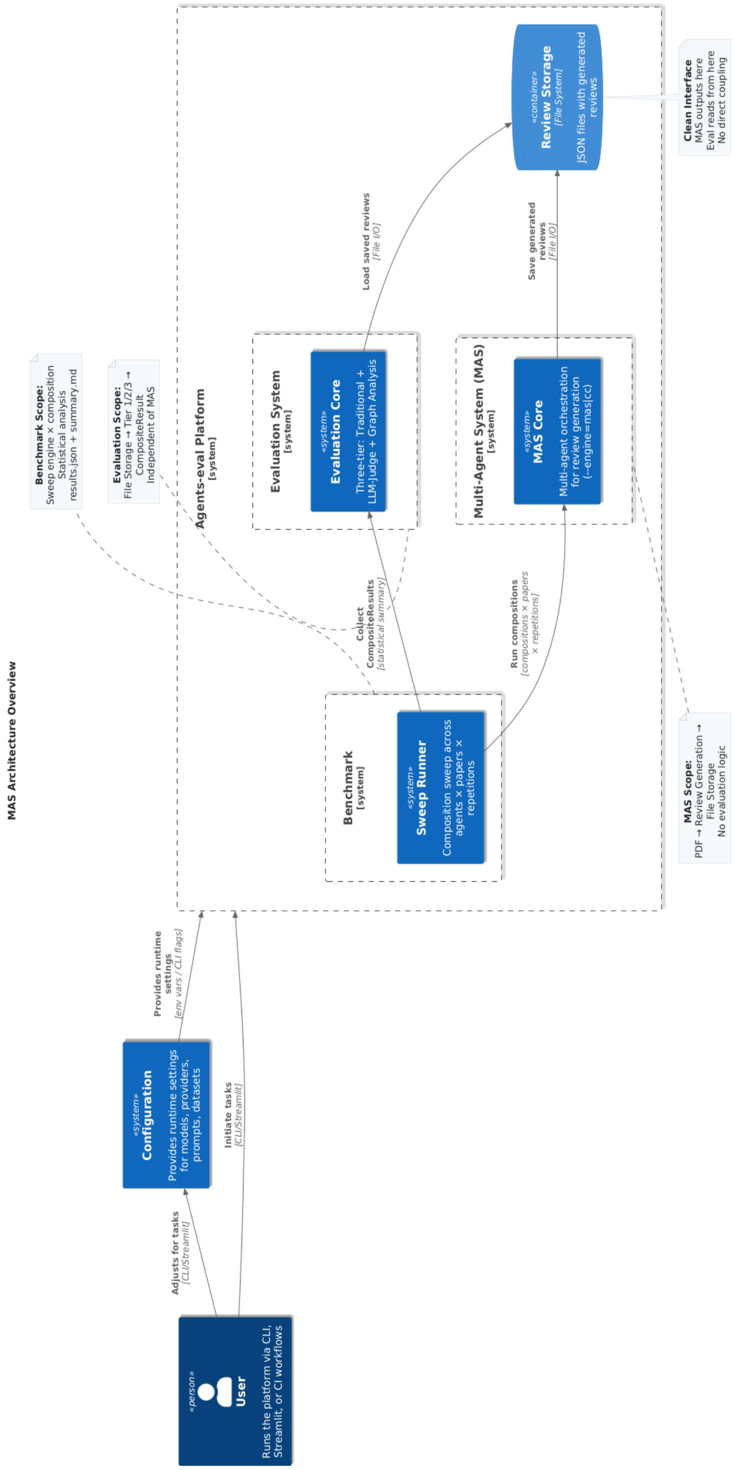


Figure 3.2: C4 model overview of the Agents-eval MAS framework

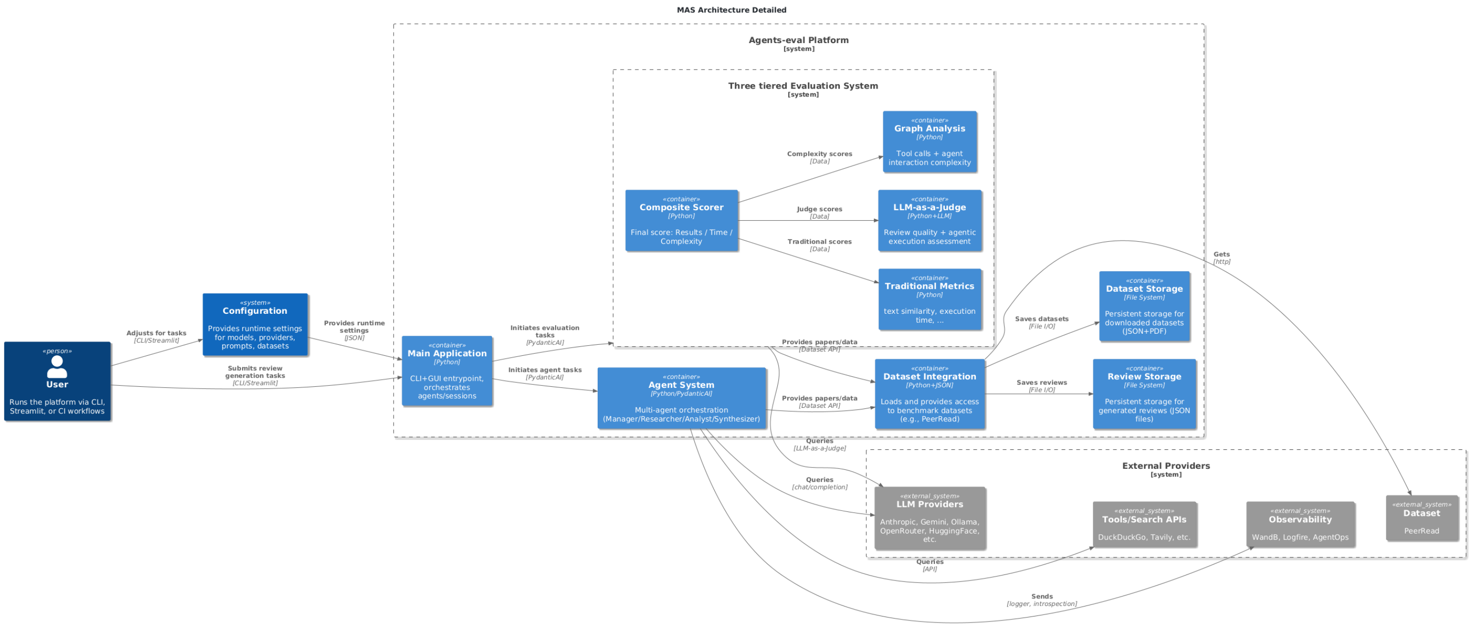


Figure 3.3: Detailed C4 component architecture with data flow

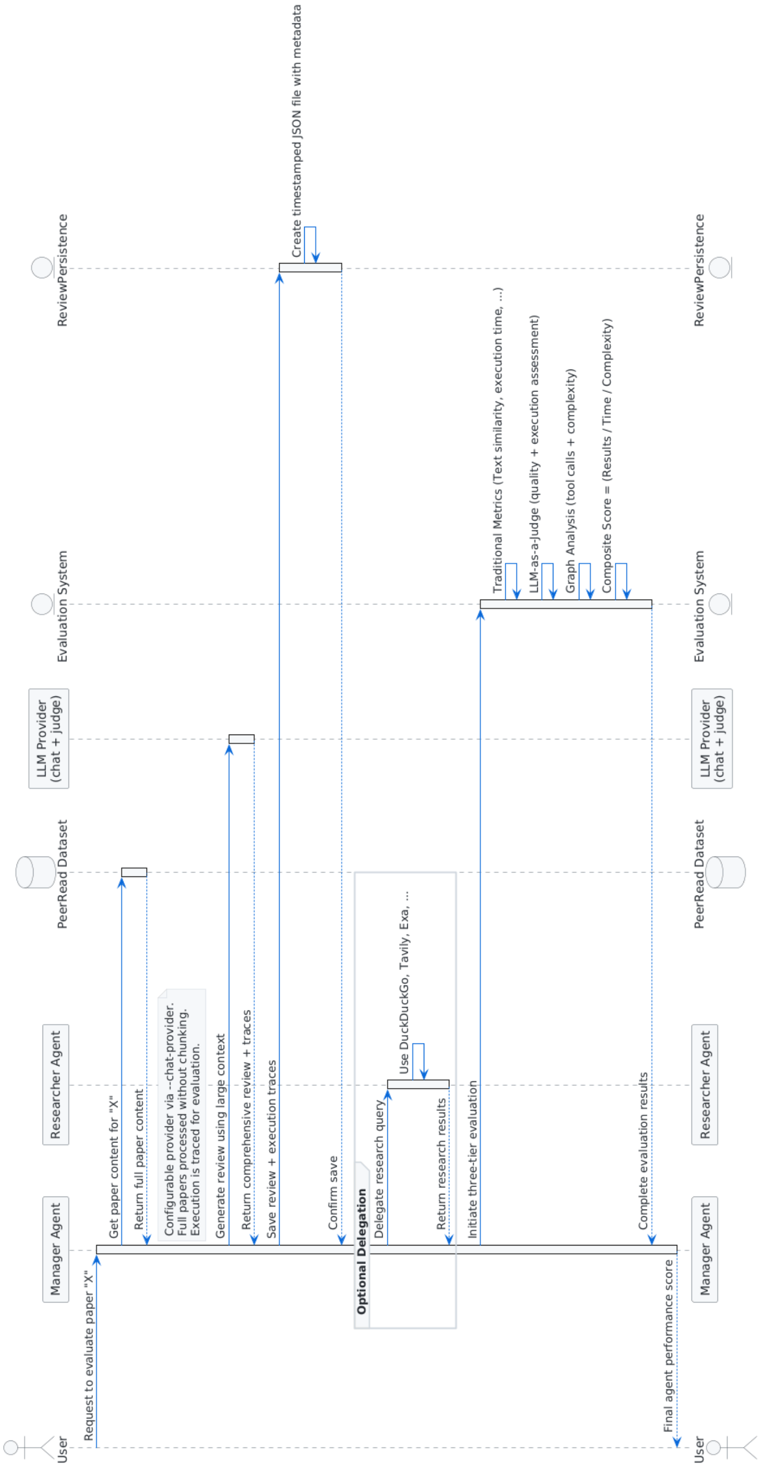

PeerRead Evaluation Workflow
User
Manager Agent
Researcher Agent
PeerRead Dataset
LLM Provider
(chat + judge)
Evaluation System
ReviewPersistence
Request to evaluate paper "X"
Get paper content for "X"
Return full paper content
Configurable provider via --chat-provider.
Full papers processed without chunking.
Execution is traced for evaluation.
Generate review using large context
Return comprehensive review + traces
Save review + execution traces
Create timestamped JSON file with metadata
Confirm save
Optional Delegation
Delegate research query
Use DuckDuckGo, Tavily, Exa, ...
Return research results
Initiate three-tier evaluation
Traditional Metrics (Text similarity, execution time, ...)
LLM-as-a-Judge (quality + execution assessment)
Graph Analysis (tool calls + complexity)
Composite Score = (Results / Time / Complexity)
Complete evaluation results
Final agent performance score
User
Manager Agent
Researcher Agent
PeerRead Dataset
LLM Provider
(chat + judge)
Evaluation System
ReviewPersistence

Figure 3.4: Multi-agent review workflow with sequential delegation

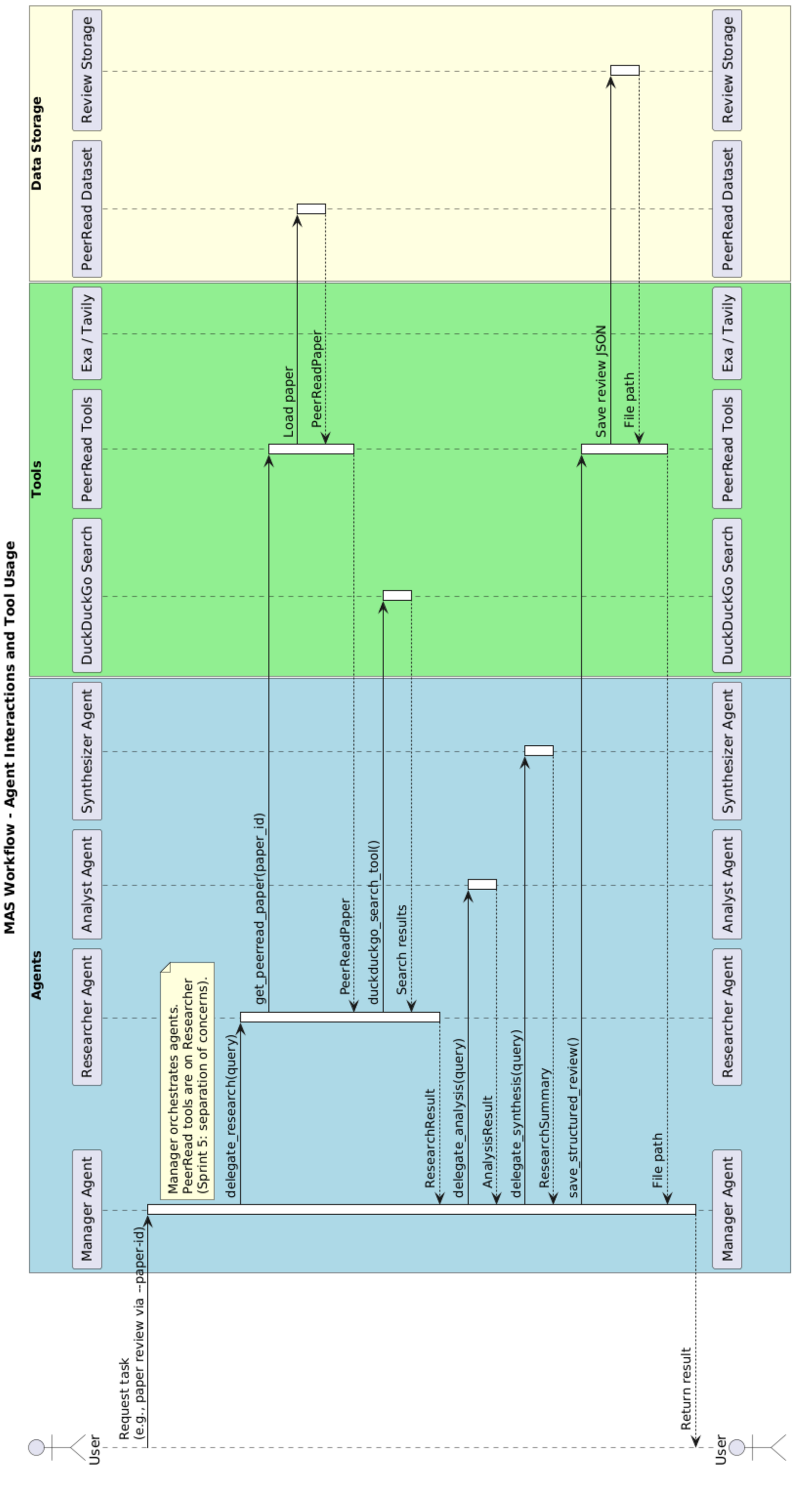


Figure 3.5: Original workflow implementation with basic agent coordination

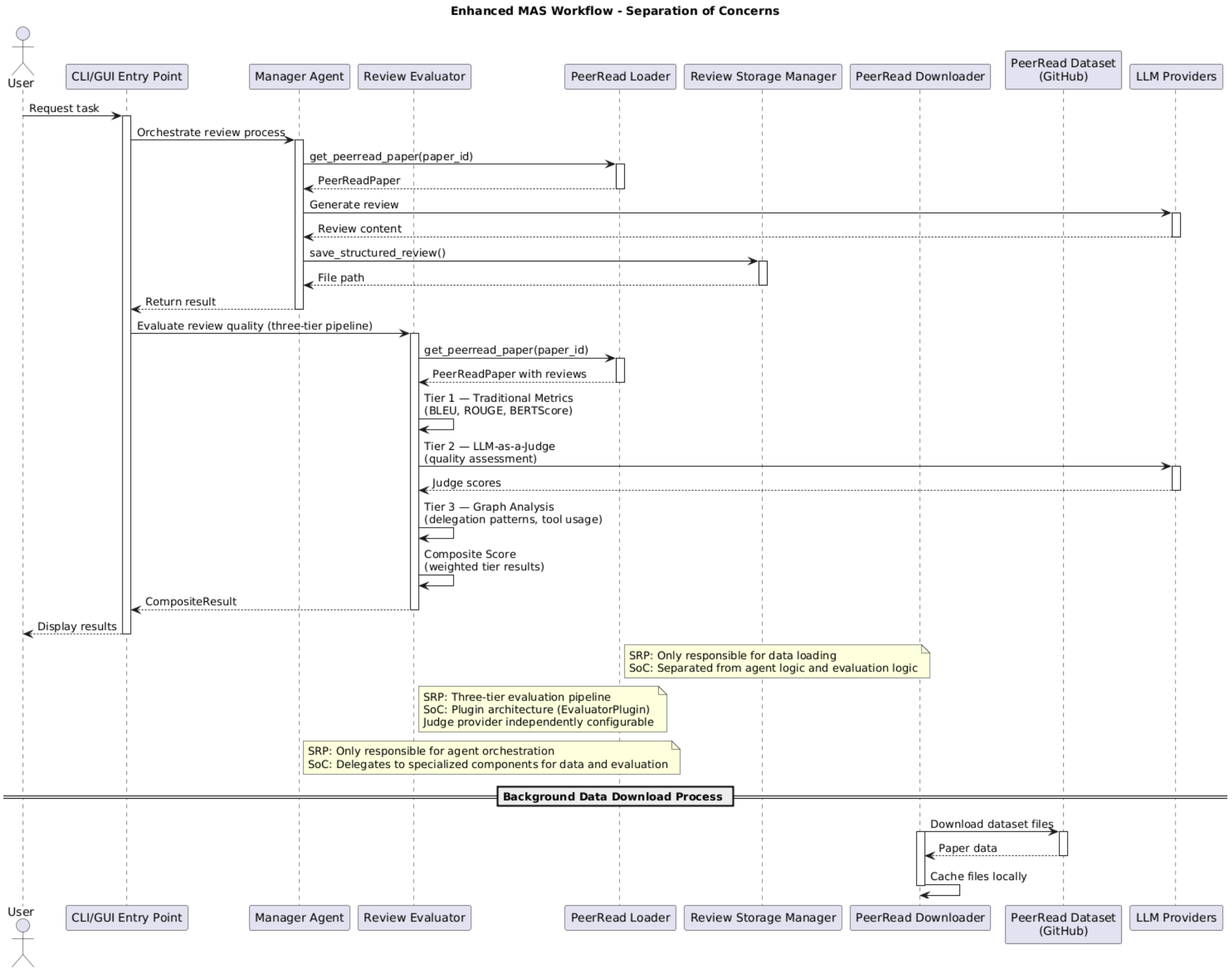

Enhanced MAS Workflow - Separation of Concerns
User
CLI/GUI Entry Point
Manager Agent
Review Evaluator
PeerRead Loader
Review Storage Manager
PeerRead Downloader
PeerRead Dataset (GitHub)
LLM Providers
Request task
Orchestrate review process
get_peerread_paper(paper_id)
PeerReadPaper
Generate review
Review content
save_structured_review()
File path
Return result
Evaluate review quality (three-tier pipeline)
get_peerread_paper(paper_id)
PeerReadPaper with reviews
Tier 1 — Traditional Metrics (BLEU, ROUGE, BERTScore)
Tier 2 — LLM-as-a-Judge (quality assessment)
Judge scores
Tier 3 — Graph Analysis (delegation patterns, tool usage)
Composite Score (weighted tier results)
CompositeResult
Display results
SRP: Only responsible for data loading
SoC: Separated from agent logic and evaluation logic
SRP: Three-tier evaluation pipeline
SoC: Plugin architecture (EvaluatorPlugin)
Judge provider independently configurable
SRP: Only responsible for agent orchestration
SoC: Delegates to specialized components for data and evaluation
Background Data Download Process
Download dataset files
Paper data
Cache files locally

Figure 3.6: Enhanced workflow with feedback loops and observability integration

The workflow evolution demonstrates improvements in agent coordination, error handling, and performance optimization. The original implementation offered basic agent delegation and task coordination, while the enhanced version incorporates feedback loops, dynamic task allocation, and observability integration.

3.7 ADR Summary

A summary of all architectural decisions is provided in Appendix A.

4 Implementation

4.1 Core Framework Implementation

4.1.1 Application Architecture

The main entry point of the application is implemented in `src/app/app.py` as the asynchronous function `main()`. It coordinates the entire lifecycle of an execution: loading configuration, initializing agents, starting execution, and subsequently triggering the evaluation pipeline. The function is instrumented with the optional `@op()` decorator from Weave, which activates when `WANDB_API_KEY` is set; if the key is absent, a no-op fallback is used.

```python
@op()  # type: ignore[reportUntypedFunctionDecorator]
async def main(
    chat_provider: str = CHAT_DEFAULT_PROVIDER,
    query: str = "",
    include_researcher: bool = False,
    include_analyst: bool = False,
    include_synthesiser: bool = False,
    pydantic_ai_stream: bool = False,
    chat_config_file: str | Path | None = None,
    enable_review_tools: bool = True,
    paper_number: str | None = None,
    skip_eval: bool = False,
    ...
) -> dict[str, Any] | None:
```

Code excerpt from `src/app/app.py:196`

The function returns a dictionary with the keys `composite_result` and `graph`, allowing the Streamlit GUI and CLI to share the same logic. For the CLI, `src/app/main.py` handles argument processing with Typer; for the GUI, `run_gui.py` calls `main()` programmatically.

4.1.2 Multi-Provider LLM Integration

The system supports multiple LLM providers (OpenAI, GitHub Models, Gemini, Ollama, Cerebras, Groq) through a unified `PROVIDER_REGISTRY` mechanism in `src/app/data_models/app_mod-`

els.py. Each registry entry contains the model name, base URL, and API key environment variable. The function setup_agent_env() in agent_system.py resolves the active provider and creates an EndpointConfig object with a validated API key and token limits:

```
def setup_agent_env(
  provider: str,
  query: UserPromptType,
  chat_config: ChatConfig | BaseModel,
  chat_env_config: AppEnv,
  token_limit: int | None = None,
) -> EndpointConfig:
```

Code excerpt from `src/app/agents/agent_system.py:629`

The token limit is determined with a three-level priority: CLI/GUI parameter > environment variable AGENT_TOKEN_LIMIT > provider configuration value. For OpenAI-compatible providers with strict tool definitions, an OpenAIModelProfile(openai_supports_strict_tool_definition=False) is set to avoid HTTP 422 errors with mixed strict values [6].

4.1.3 Type-Safe Data Model Architecture

All data boundaries are secured by Pydantic models in src/app/data_models/. The ChatConfig model describes provider configurations and prompts; AppEnv (BaseSettings with AGENTS_EVAL_ prefix) reads API keys from the environment. Evaluation results are typed in evaluation_models.py as Tier1Result, Tier2Result, Tier3Result, and CompositeResult. For external data mapping fields (PeerRead dataset), validation_alias is used to map external key names (IMPACT) to internal field names (impact) without altering the constructor signature:

```
impact: str = Field(default="UNKNOWN", validation_alias="IMPACT")
```

Code excerpt from `src/app/data_models/peerread_models.py`

4.2 Multi-Agent System

4.2.1 Agent Orchestration

Agent orchestration is based on PydanticAI [6]. The Manager Agent receives the user query and delegates subtasks to up to three sub-agents (Researcher, Analyst, Synthesizer) via typed tool calls. The composition is configured at runtime:

```
def get_manager(
  provider: str,
  provider_config: ProviderConfig,
  api_key: str | None,
  prompts: dict[str, str],
  include_researcher: bool = False,
  include_analyst: bool = False,
  include_synthesiser: bool = False,
  enable_review_tools: bool = False,
) -> Agent[None, BaseModel]:
```

Code excerpt from `src/app/agents/agent_system.py:432`

Within `_create_manager()`, sub-agents are created as `Agent` instances with their own model and system prompt, and registered as tool functions on the Manager via `_add_tools_to_manager_agent()`. Each delegation tool (`delegate_research`, `delegate_analysis`, `delegate_synthesis`) invokes the respective sub-agent, logs the interaction in the `TraceCollector`, and returns a typed Pydantic model:

```
@manager_agent.tool
async def delegate_research(
  ctx: RunContext[None], query: str
) -> ResearchResult | ResearchResultSimple | ReviewGenerationResult:
  """Delegate research task to ResearchAgent."""
  trace_collector.log_agent_interaction(
    from_agent="manager",
    to_agent="researcher",
    interaction_type="delegation",
    data={"query": query, "task_type": "research"},
  )
  result = await research_agent.run(query, usage=ctx.usage)
  ...
```

Code excerpt from `src/app/agents/agent_system.py:121`

In single-agent mode (Manager only), the Manager handles all tasks itself. PeerRead-specific tools are registered directly on the Manager in this case; otherwise on the Researcher Agent (separation of concerns, Sprint 5 [15]).

4.2.2 Tool Integration

The Researcher Agent has access to the `duckduckgo_search_tool()` from PydanticAI's common tools [6] as well as PeerRead-specific tools from `src/app/tools/peerread_tools.py`: `get_peerread_paper`, `read_paper_pdf_tool`, `query_peerread_papers`, `gener-`

ate_paper_review_content_from_template, save_paper_review, and save_structured_review. All tool calls are captured by the TraceCollector's log_tool_call() with timestamp and success flag.

The result model is chosen based on provider: Gemini receives ResearchResultSimple (no additionalProperties support in JSON schema), all other providers receive ResearchResult. When review tools are enabled, ReviewGenerationResult is used.

4.3 Evaluation Pipeline

4.3.1 Three-Tier Implementation

The class EvaluationPipeline in src/app/judge/evaluation_pipeline.py orchestrates the sequential execution of all three evaluation tiers with individual timeouts and error handling:

```python
async def evaluate_comprehensive(
  self,
  paper: str,
  review: str,
  execution_trace: GraphTraceData | dict[str, Any] | None = None,
  reference_reviews: list[str] | None = None,
) -> CompositeResult:
  tier1_result, _ = await self._execute_tier1(paper, review,
↪   reference_reviews)
  tier2_result, _ = await self._execute_tier2(paper, review,
↪   trace_dict)
  tier3_result, _ = await self._execute_tier3(trace_dict)
  ...
```

Code excerpt from src/app/judge/evaluation_pipeline.py:484

Tier 1 – Traditional Metrics (src/app/judge/plugins/traditional.py): TF-IDF cosine similarity, Jaccard similarity, and semantic similarity (TF-IDF cosine), execution time, and task success score. Timeout: 1 second.

Tier 2 – LLM-as-Judge (src/app/judge/plugins/llm_judge.py): A single LLM call evaluates technical accuracy, constructiveness, clarity, and planning rationality. The provider is automatically resolved via a fallback chain (tier2_provider=auto inherits the active chat provider; if no valid provider is available, Tier 2 is skipped and the weights are redistributed to Tier 1 and Tier 3). Timeout:

10 seconds.

Tier 3 – Graph-Based Analysis (`src/app/judge/plugins/graph_metrics.py`): NetworkX processes the `GraphTraceData` from the `TraceCollector` into a directed graph and computes `path_convergence`, `tool_selection_accuracy`, `coordination_centrality`, and `task_distribution_balance`. This is the primary differentiating metric of the framework [12]. Timeout: 15 seconds.

If a tier fails, depending on the `fallback_strategy` setting, a fallback (neutral 0.5 values) is applied or the tier is skipped. The performance metrics of all tier executions are captured in the `PerformanceMonitor`, which issues a bottleneck warning when exceeding 40% of total runtime.

4.3.2 Plugin Registry and Composite Scorer

Each evaluation tier implements the abstract `EvaluatorPlugin` interface from `src/app/judge/plugins/bas`

```python
class EvaluatorPlugin(ABC):
    @property
    @abstractmethod
    def name(self) -> str: ...

    @property
    @abstractmethod
    def tier(self) -> int: ...

    @abstractmethod
    def evaluate(self, context: BaseModel) -> BaseModel: ...

    @abstractmethod
    def get_context_for_next_tier(self, result: BaseModel) ->
     ↪  BaseModel: ...
```

Code excerpt from `src/app/judge/plugins/base.py`

The `PluginRegistry` discovers plugins at import time and executes them in tier order (1 → 2 → 3). Typed context passing between tiers prevents runtime errors.

The `CompositeScorer` (`src/app/judge/composite_scorer.py`) computes the weighted overall score from six equally weighted metrics (16.7% each): `time_taken`, `task_success`, `coordination_quality`, `tool_efficiency`, `planning_rationality`, `output_similarity`. In single-agent mode, `coordination_quality` is excluded and the weight is redistributed to the remaining five metrics (20% each), which is transparently communicated through the `single_agent_mode` flag in `CompositeResult`. The decision thresholds are: accept $\geq$ 0.863 | weak_accept $\geq$ 0.626 | reject < 0.626 [16].

4.4 Observability Integration

The observability layer combines **Logfire** for structured tracing and **Arize Phoenix** as a local trace viewer (Docker-free). Initialization occurs in `src/app/agents/logfire_instrumentation.py` via `logfire.instrument_pydantic_ai()`, which automatically instruments all PydanticAI agents – no manual decorators on agent functions are required:

```
def initialize_logfire_instrumentation_from_settings(
    settings: JudgeSettings | None = None,
) -> None:
    ...
    initialize_logfire_instrumentation(logfire_config)
```

Code excerpt from `src/app/agents/agent_system.py:72`

The `TraceCollector` (`src/app/judge/trace_processors.py`) captures agent-to-agent interactions and tool calls with timestamps during agent execution in a `GraphTraceData` instance, which subsequently serves as input for Tier 3. Traces are persistently stored in SQLite (`logs/traces/traces.db`) and as JSONL files in `logs/traces/`. API keys and tokens are redacted before persistence through Loguru scrubbing patterns (Sprint 6, STORY-012 [15]).

Wandb/Weave is implemented as an optional dependency: if `WANDB_API_KEY` is absent, a no-op decorator activates that completely suppresses the import.

4.5 User Interfaces

4.5.1 CLI (Typer)

The CLI is implemented in `src/app/main.py` with Typer. It exposes all parameters of `main()` as command-line flags with runtime type checking. Key flags include `--paper-number`, `--chat-provider`, `--include-researcher`, `--include-analyst`, `--include-synthesiser`, `--skip-eval`, `--token-limit`, and `--engine=mas|cc` (Sprint 7) for switching between PydanticAI MAS and Claude Code baseline. A separate CLI `run_sweep.py` controls the `SweepRunner` (`src/app/benchmark/`) for composition sweeps across multiple agent configurations and papers (Sprint 6 [15]).

Example invocation:

```
make run_cli ARGS="--paper-number=1105.1072 --chat-provider=github \
  --include-researcher --include-analyst --include-synthesiser"
```

4.5.2 Streamlit GUI

The Streamlit GUI (`src/app/gui/`) is organized into several pages:

- **Run App**: Starts agent execution in the background via `threading.Thread` (tab navigation does not abort execution); displays real-time debug logs from a `LogCapture` Loguru sink.
- **Evaluation Results**: Displays Tier 1/2/3 scores and comparison charts.
- **Agent Graph**: Renders the delegation graph from `GraphTraceData` interactively with NetworkX and Pyvis (Sprint 5 [15]).
- **Settings**: Editable settings with session state persistence; reads default values from `JudgeSettings` and `CommonSettings`.

The system architecture (see Chapter 4, Section 4.6) visualizes the interaction of all components.

The following customer journey shows the complete interaction path of a researcher from paper selection to evaluation view:

4.6 Development Process Across Seven Sprints

The implementation followed a sprint-based BDD approach with iterative refinement [17]:

Sprint	Key Deliverables
Sprint 1	Three-tier evaluation framework (Tier 1–3 base implementation), PeerRead dataset integration, `JudgeSettings` pydantic-settings
Sprint 2	Post-run evaluation wiring (`--skip-eval`), Logfire + Phoenix tracing infrastructure, Streamlit evaluation dashboard
Sprint 3	Plugin architecture (`EvaluatorPlugin`, `PluginRegistry`), `TraceStore`, `JudgeAgent`, optional Weave dependency, Hypothesis/snapshot tests
Sprint 4	Operational resilience (thread-safe graph timeout, Logfire error handling), Claude Code `CCTraceAdapter`, GUI baseline comparison
Sprint 5	Tier 2 fallback chain, token limit override, single-agent weight redistribution, Streamlit background execution, OWASP MAESTRO security audit

Sprint	Key Deliverables
Sprint 6	Benchmarking infrastructure (`SweepRunner`, `SweepAnalyzer`), security hardening (SSRF, prompt injection, log scrubbing), test coverage increase, Opik removal
Sprint 7	Unified provider configuration (`--judge-provider`, `--judge-model`), `--engine=mas|cc` flag, sweep rate-limit resilience, GUI real-time debug log, architecture documentation

The complete change history is documented in `CHANGELOG.md` [15].

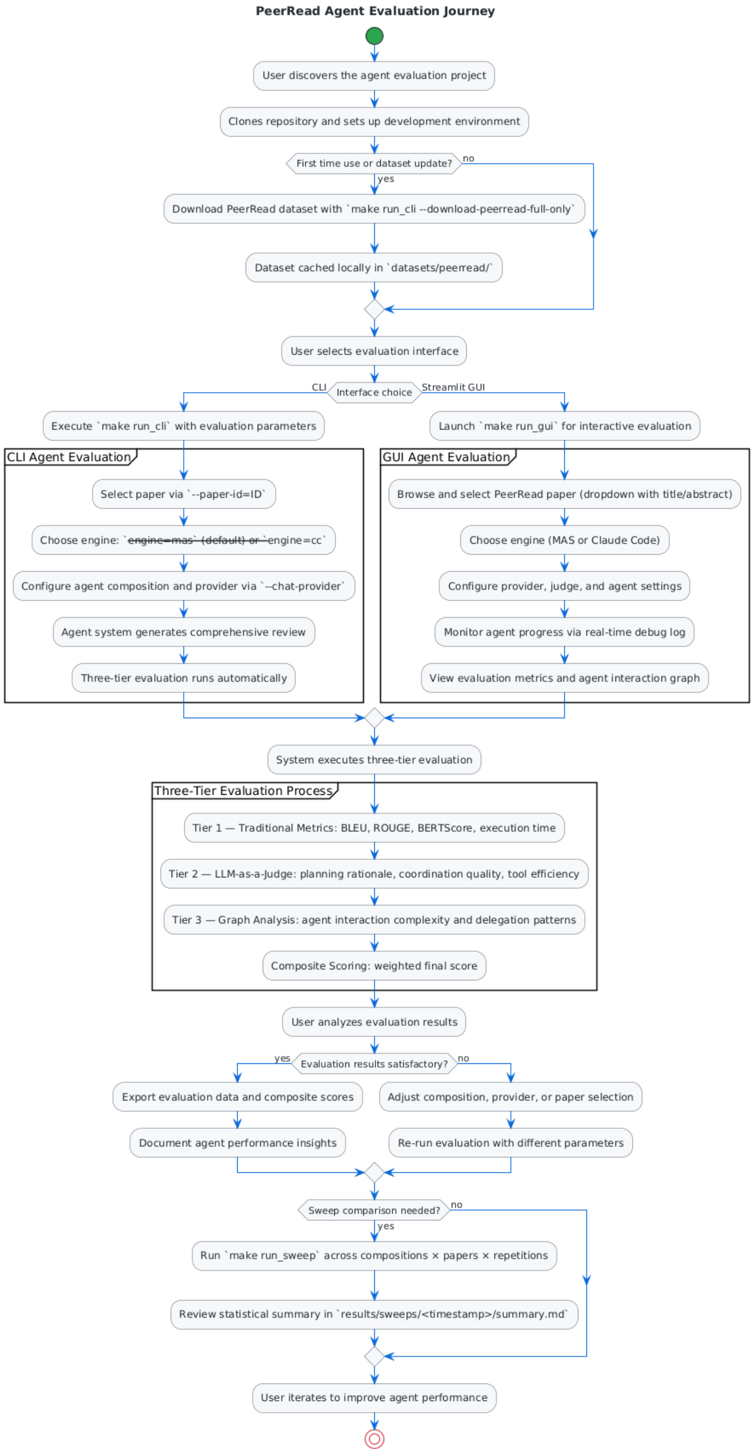

PeerRead Agent Evaluation Journey
User discovers the agent evaluation project
Clones repository and sets up development environment
First time use or dataset update?
no
yes
Download PeerRead dataset with `make run_cli --download-peerread-full-only`
Dataset cached locally in `datasets/peerread/`
User selects evaluation interface
CLI
Interface choice
Streamlit GUI
Execute `make run_cli` with evaluation parameters
Launch `make run_gui` for interactive evaluation
CLI Agent Evaluation
Select paper via `--paper-id=ID`
Choose engine: `~~engine=mas` (default) or `~~engine=cc`
Configure agent composition and provider via `--chat-provider`
Agent system generates comprehensive review
Three-tier evaluation runs automatically
GUI Agent Evaluation
Browse and select PeerRead paper (dropdown with title/abstract)
Choose engine (MAS or Claude Code)
Configure provider, judge, and agent settings
Monitor agent progress via real-time debug log
View evaluation metrics and agent interaction graph
System executes three-tier evaluation
Three-Tier Evaluation Process
Tier 1 — Traditional Metrics: BLEU, ROUGE, BERTScore, execution time
Tier 2 — LLM-as-a-Judge: planning rationale, coordination quality, tool efficiency
Tier 3 — Graph Analysis: agent interaction complexity and delegation patterns
Composite Scoring: weighted final score
User analyzes evaluation results
yes
Evaluation results satisfactory?
no
Export evaluation data and composite scores
Document agent performance insights
Adjust composition, provider, or paper selection
Re-run evaluation with different parameters
Sweep comparison needed?
no
yes
Run `make run_sweep` across compositions × papers × repetitions
Review statistical summary in `results/sweeps/<timestamp>/summary.md`
User iterates to improve agent performance

Figure 4.1: Customer journey – user interaction patterns and system touchpoints

5 Verification of Success

5.1 Evaluation Methodology

The verification of success for the Agents-eval framework relies on three complementary measurement levels, defined in the target state (Chapter 3) and in `docs/UserStory.md` [18]: quantitative text metrics (Tier 1), semantic LLM judgment (Tier 2), and graph-based behavioral analysis (Tier 3). The three tiers are designed as mutual validators: if all three agree, confidence in the result is high; if Tier 3 diverges from Tiers 1/2, this indicates good coordination with weak output quality – or vice versa [12].

Composite Score Formula: Six equally weighted metrics (16.7% each) yield the overall score: `time_taken`, `task_success`, `coordination_quality`, `tool_efficiency`, `planning_rationality`, `output_similarity`. The decision thresholds are configurable in `JudgeSettings` (defaults: accept $\geq$ 0.863 | weak_accept $\geq$ 0.626 | reject < 0.626) [16].

In single-agent mode, `coordination_quality` is automatically excluded and the weight is evenly distributed among the remaining five metrics, since without agent delegation there is no inter-agent coordination to evaluate (`single_agent_mode: bool` in `CompositeResult`).

5.2 Acceptance Criteria and Their Fulfillment

The acceptance criteria originate from `docs/UserStory.md` [18]:

Acceptance Criterion	Status	Finding
`make run_cli ARGS="--paper-id=ID"` generates review AND evaluates automatically	Partially fulfilled	Blocked by `AgentRunResult.data` bug on `refactor-arch` branch (fixed in `CHANGELOG.md` Unreleased [15]); functional on `main`

Acceptance Criterion	Status	Finding
Real-time execution traces with actual delegations and tool calls	Fulfilled	30 JSONL traces in `logs/traces/` demonstrate real Manager-Researcher delegations and tool calls [16]
Logs show Tier 1 vs. Tier 3 scores side by side	Fulfilled	`_log_metric_comparison()` in `evaluation_pipeline.py:432` outputs structured comparison
`--skip-eval` skips evaluation	Fulfilled	Implemented in Sprint 2, `app.py` delegates to `_run_evaluation_if_enabled()`
`make validate` passes all checks	Fulfilled	Ruff, Pyright, and pytest run green on `main`
Local trace viewer without Docker	Fulfilled	Logfire + Arize Phoenix via OTLP without Docker dependency (Sprint 2)
Streamlit shows tier scores and comparison charts	Fulfilled	"Evaluation Results" page with live data (Sprint 5)
Streamlit "Agent Graph" page renders delegation graph interactively	Fulfilled	NetworkX + Pyvis (Sprint 5)

Unfulfilled criteria: A complete three-way comparison (MAS vs. CC Solo vs. CC Teams) with computed composite scores is outstanding. CC Teams artifacts are ephemeral in headless execution; the `CCTraceAdapter` for Teams mode therefore cannot operate [16]. Sweep results (`results/sweeps/`) contain empty arrays; no cross-composition ranking is available.

5.3 Quality Assurance

5.3.1 Testing Strategy

The test suite comprises 564 tests after Sprint 6 (from 595 after targeted review and removal of implementation-detail tests without behavioral coverage loss, Sprint 6 STORY-015 [15]).

The testing strategy follows three layers:

Unit Tests (majority of tests): External dependencies are mocked via `@patch`. All mocks carry `spec=RealClass` to detect API drift early. Tests cover error handling, edge cases, and data flows.

Property-Based Tests (Hypothesis [19]): Invariants such as score bounds ($0 \leq \text{Score} \leq 1$), input validation, and mathematical properties of the composite scorer are verified with randomized inputs.

Snapshot Tests (inline-snapshot): Pydantic model dumps, configuration outputs, and graph transformation results are checked against frozen snapshots for regression testing.

Security Tests (`tests/security/`, Sprint 6 STORY-013 [15]): 135 tests across five modules cover SSRF prevention, prompt injection resistance, data redaction in logs and traces, input size limits, and tool registration scope.

Test locations mirror the source structure: `tests/` analogous to `src/app/`.

5.3.2 Code Quality Pipeline

The quality assurance pipeline is executed via `make validate`:

Tool	Task	Configuration
Ruff	Formatting and linting	`pyproject.toml`
Pyright	Static type checking	`pyproject.toml`
pytest	Test execution	`pyproject.toml`

For fast development iterations, `make quick_validate` (Ruff + Pyright + Complexipy without tests) is available. The complexity threshold is monitored by Complexipy; cyclomatic complexity above the threshold blocks the commit.

5.4 Security Validation

An OWASP MAESTRO 7-layer security audit (Sprint 5, STORY-010 [15]) identified 31 findings across all seven layers (Model, Agent Logic, Integration, Monitoring, Execution, Environment, Orchestration).

Critical findings were addressed in Sprint 6:

- **CVE-2026-25580 (SSRF, CRITICAL)**: URL validation with HTTPS-only and domain allowlist in `src/app/utils/url_validation.py`; 49 tests. The allowlist was derived from actual `validate_url()` call sites, not from conceptual service lists [20].
- **Prompt Injection (HIGH)**: Length limits and XML delimiter wrapping around LLM Judge prompts; 25 tests (STORY-011 [15]).
- **Log/Trace Data Redaction (HIGH)**: Pattern-based redaction of API keys, passwords, and tokens in Loguru sinks and Logfire OTLP exports; 13 tests (STORY-012 [15]).

Dependencies are managed via `uv` with pinned versions. CVE-2024-5206 (scikit-learn data leak) was already mitigated by the existing `scikit-learn>=1.8.0` pin and required no separate action [20].

5.5 Current Implementation Status Assessment

Based on the actual trace data collected from `logs/traces/` (30 JSONL traces, 14 Manager-only runs, 12 multi-agent runs) [16], the current implementation status can be assessed as follows:

Implemented and functional:

- Three-tier evaluation pipeline with plugin architecture and composite scorer
- PydanticAI MAS with four agent roles and flexible composition configuration
- Logfire + Phoenix observability stack without Docker
- SSRF, prompt injection, and log scrubbing protection
- Benchmarking infrastructure (`SweepRunner`, `SweepAnalyzer`, `run_sweep.py`)
- CCTraceAdapter for CC Solo mode parsing
- Streamlit GUI with background execution, debug log, evaluation, and graph views

Not completed or blocked:

- Complete MAS vs. CC comparison with composite scores (CC Teams artifacts ephemeral; API key for Tier 2 LLM-as-Judge not set in test environment)
- Composition sweep with statistically significant results (empty `results/sweeps/` directories; blocked by the now-fixed `AgentRunResult.data` bug)
- Per-sub-agent token counting (currently only Manager-level token usage captured)

Observed metrics from real runs [16]:

Configuration	Median Latency	Error Rate
PydanticAI Manager-only	~4.8 s	0% (0/14)
PydanticAI 3-agent	~12.3 s	25% (4/16, init errors)

Configuration	Median Latency	Error Rate
CC Solo	118.3 s	0% (1/1)
CC Teams	359.9 s	0% (1/1)

These figures are based on limited samples (n=14 and n=1 for CC respectively) and are not statistically validated. Qualitative composite scores for real runs could not yet be computed due to the blocking issues mentioned above.

6 Results

6.1 Data Inventory

The empirical foundation of this work comprises 30 structured JSONL trace files located in `logs/traces/`, as well as a SQLite database (`logs/traces/traces.db`) for persistent queries. Additionally, approximately 200 Loguru application and test logs (`logs/*.log`, `*.log.zip`) are available.

Source	Count	Content
`logs/traces/*.jsonl`	30	Structured execution traces (`GraphTraceData`)
`logs/traces/traces.db`	1	SQLite trace database
`logs/*.log` / `*.log.zip`	~200	Loguru application and test logs
`results/sweeps/`	2 directories	Empty result arrays (`[]`)
CC Solo/Teams artifacts	1 set	Collected under `logs/cc/solo/` and `logs/cc/teams/`

Approximately 95% of the log files are pytest outputs from automated tests, not actual evaluation runs. Only 10–15 logs correspond to actual CLI executions.

6.2 Single-LLM MAS (Manager-Only)

The Manager-Only configuration corresponds to single-LLM operation: research, analysis, and synthesis agents are deactivated (`AgentComposition(include_researcher=False, include_analyst=False, include_synthesiser=False)`). The Manager does not delegate but calls all tools directly.

Metric	Value
Agent interactions	0 (no delegation)

Metric	Value
Tool calls per run	3 (`get_peerread_paper`, `generate_paper_review_content_from_template`, `save_structured_review`)
Duration range	1.6 s – 8.7 s
Median duration	~4.8 s
Input tokens (Paper 001)	8,342 (of which 5,888 cache read)
Input tokens (Paper 1105.1072)	14,198 (of which 8,960 cache read)
Output tokens	570–743
LLM requests per run	4
Observed runs	14

Representative trace examples:

Execution ID	Paper	Duration	Avg. Tool Duration
`exec_397258f1de20`	1105.1072	1.611 s	0.012 s
`exec_e4a4993014da`	001	4.795 s	0.004 s
`exec_4ef1548c4f24`	1105.1072	8.659 s	0.021 s

Execution time is entirely dominated by LLM inference latency; the mean tool execution time ranges between 0.004 s and 0.09 s.

6.3 Multi-LLM MAS (with Sub-Agents)

6.3.1 Duration by Agent Count

In the multi-LLM configuration, the Manager delegates tasks sequentially to one or more sub-agents, each of which performs its own LLM inference. Three compositions were observed: Researcher-only, Researcher+Analyst, and the full three-agent configuration.

Agents	Runs	Avg. Duration	Range
1 (Researcher)	4	6.5 s	3.9–8.8 s
2 (Researcher + Analyst or Synthesizer)	3	8.8 s	7.3–11.9 s
3 (Researcher + Analyst + Synthesizer)	3	12.3 s	7.9–17.4 s

Duration scales approximately linearly with agent count, as sub-agents are called sequentially.

6.3.2 Outlier Analysis

Two runs exhibited extreme durations significantly above the normal range:

Execution ID	Duration	Cause
`exec_655bf85674d4`	135.96 s	Single Researcher, 2 attempts (retry)
`exec_2a4d21581ece`	69.46 s	2 interactions, 2 attempts (retry)

Analysis of the trace data reveals that the outliers were not caused by coordination overhead but by LLM provider latency spikes or rate limiting.

6.3.3 Failed Runs

Four traces recorded a duration of 0.0 s with 0–1 tool calls. These runs failed before meaningful work began due to initialization errors. This yields an error rate of 4/16 (25%) for the multi-LLM configuration, compared to 0/14 (0%) for the Manager-Only configuration.

6.4 Claude Code Baseline

6.4.1 CC Solo

The CC Solo configuration was executed with the command `claude -p --output-format stream-json --verbose` without the Teams flag.

Metric	Value
Session ID	`dad34c5b-813d-4f85-99d0-91c2c4ccc3eb`
Model	`claude-sonnet-4-5-20250929`
Duration	118.3 s
Cost	$0.94
Turns	4
Tool calls	19
Artifact path	`logs/cc/solo/1105.1072_20260217_181344/`

The single agent used codebase exploration tools (Task, Bash, Glob, Grep, Read) to locate the paper data in the project directory before generating the review.

6.4.2 CC Teams

The CC Teams configuration was executed with `CLAUDE_CODE_EXPERIMENTAL_AGENT_TEAMS=1` set and a teams-specific prompt.

Metric	Value
Session ID	`8dd391f8-82c4-43bd-b960-cf7bce4d5a3e`
Model	`claude-sonnet-4-5-20250929`
Duration	359.9 s
Cost	$1.35
Turns	13
Tool calls	22
Artifact path	`logs/cc/teams/1105.1072_20260217_182646/`

The tool distribution comprises: TodoWrite (5x), TeamCreate (1x), Task (3x – Explore + 2 sub-agents), Bash (2x), Glob (2x), Read (6x). The system created a team `paper-review-1105-1072` and spawned three sub-agents: Researcher, Analyst, and Synthesizer.

6.4.3 Infrastructure Status

Component	Status	Path
`CCTraceAdapter`	Implemented	`src/app/judge/cc_trace_adapter.py`
CC Solo parser	Implemented	Reads `metadata.json` + `tool_calls.jsonl`
CC Teams parser	Implemented	Reads `config.json` + `inboxes/` + `tasks/`
`BaselineComparison` model	Implemented	`src/app/judge/baseline_comparison.py`
`compare_all()` function	Implemented	Generates 3 pairwise comparisons
CC Solo artifacts	**Collected**	`logs/cc/solo/1105.1072_20260217_181344/`

Component	Status	Path
CC Teams artifacts	**Partial**	Only `metadata.json` + `tool_calls.jsonl` (no `config.json`, `inboxes/`, `tasks/`)

6.5 Comparative Analysis

6.5.1 Single-LLM vs. Multi-LLM

Dimension	Single-LLM (Manager-Only)	Multi-LLM (3 Agents)	Delta
Median duration	~4.8 s	~12.3 s	+156%
Agent interactions	0	3	–
Tool calls	3 (direct)	3 (delegated)	Same count, different pattern
LLM requests	4	4+ (per agent)	Higher total
Error rate	0/14 (0%)	4/16 (25%)	Higher with delegation
Token efficiency	~9,000 input	Unknown (sub-agent tokens not logged)	Likely higher

Multi-agent configurations produce approximately 2.5x latency increase with the three-agent variant compared to Manager-Only. Since the evaluation pipeline could not produce composite quality scores due to blocking issues, a quality comparison based on the available data is not possible.

6.5.2 PydanticAI MAS vs. CC (Paper 1105.1072)

Dimension	PydanticAI Manager-Only	PydanticAI 3 Agents	CC Solo	CC Teams
Duration	~4.8 s	~12.3 s	118.3 s	359.9 s
Cost (approx.)	~$0.01	~$0.03	$0.94	$1.35
Tool calls	3	3	19	22
Turns	4	4+	4	13
Agent interactions	0	3 (delegation)	0	3 (Task sub-agents)
Model	GPT-4.1 (GitHub)	GPT-4.1 (GitHub)	claude-sonnet-4-5	claude-sonnet-4-5
Error rate	0%	25%	0% (n=1)	0% (n=1)

Key observations:

1. **PydanticAI is 25–75x faster than CC**: CC Solo (118.3 s) vs. PydanticAI Manager-Only (4.8 s). CC explored the codebase at runtime to locate paper data; PydanticAI uses typed tools with direct data access.
2. **CC Teams incurs a 3x overhead over CC Solo**: 359.9 s vs. 118.3 s. The orchestration pattern (TodoWrite + TeamCreate + 3 Task sub-agents) structurally mirrors PydanticAI three-agent delegation, but with significantly higher overhead from tool-based coordination.
3. **CC uses 6–7x more tool calls**: CC probes the filesystem (Glob, Grep, Read, Bash) for data discovery; PydanticAI uses purpose-built tools (`get_peerread_paper`, `generate_paper_review_content_from_template`).
4. **The cost difference is approximately 50–100x**: CC Solo ($0.94) vs. PydanticAI Manager-Only (~$0.01). The difference arises from model cost disparity (Claude Sonnet 4.5 vs. GPT-4.1) and higher token consumption from codebase exploration.
5. **CC Teams orchestration is structurally analogous to PydanticAI multi-agent**: Both instantiate Researcher, Analyst, and Synthesizer roles. CC uses the Task tool; PydanticAI uses `delegate_research/delegate_analysis/delegate_synthesis`.

Limitation: This comparison is based on a single paper (n=1 for CC). PydanticAI data comes from 14+ runs. Cost figures for PydanticAI are estimates (GitHub Models pricing). A quality comparison requires the complete evaluation pipeline (Tier 1 + 2 + 3), which is blocked by the missing `GITHUB_API_KEY` for Tier 2.

6.6 Evaluation Pipeline Readiness

The three-tier evaluation framework is fully implemented but could not be fully executed on the available traces due to blocking issues.

Tier	Purpose	Metrics	Status
Tier 1	Traditional text metrics	cosine, jaccard, semantic, time_taken, task_success	Implemented
Tier 2	LLM-as-Judge	accuracy, constructiveness, clarity, planning_rationality	Implemented (requires API key)
Tier 3	Graph behavioral analysis	path_convergence, tool_accuracy, coordination_quality, distribution	Implemented

The composite scoring system combines six equally weighted metrics (0.167 each): time_taken, task_success, output_similarity, planning_rationality, coordination_quality, tool_efficiency.

Decision thresholds: accept $\geq$ 0.863 | weak_accept $\geq$ 0.626 | reject < 0.626.

In single-agent operation (single_agent_mode=True), the weight of coordination_quality (0.167) is evenly redistributed to the remaining five metrics (0.20 each), since no inter-agent interactions are present.

6.7 Gaps and Blocking Issues

Issue	Impact	Location
'AgentRunResult' object has no attribute 'data'	Manager-Only runs abort during result extraction	app.py:280 on branch refactor-arch

Issue	Impact	Location
Empty sweep results	No composition comparison dataset generated	`results/sweeps/*/results.json`
CC Teams artifacts ephemeral	`CCTraceAdapter` Teams parser cannot operate	`~/.claude/teams/` empty after run
`GITHUB_API_KEY` not set	Tier 2 LLM-as-Judge comparison blocked	`.env` / shell environment

The blocking issues prevent the generation of composite quality scores and thus a complete empirical comparison of system configurations. Latency and cost metrics based solely on trace data are, however, fully available.

User Story	Status	Gap
Evaluation runs automatically after generation	Partially built	Blocked by `AgentRunResult` bug
Real agent execution traces captured	Done for PydanticAI + CC	CC Teams artifacts ephemeral
Graph metrics alongside text metrics	Tier 1 + Tier 3 implemented	No real composite scores computed
Compare MAS vs. CC baseline	CC Solo + Teams collected	Evaluation pipeline blocked by API key
Run across all composition variants	Sweep harness present	Produces empty results

7 Summary and Outlook

7.1 Achieved Goals

Over the course of seven sprints, the Agents-eval project has built a functional infrastructure for the empirical evaluation of multi-agent systems. The central deliverables comprise:

- **Three-Tier Evaluation Architecture** (Sprint 1): Tier 1 – traditional text metrics, Tier 2 – LLM-as-Judge, Tier 3 – graph-based behavioral analysis. All three tiers are implemented and integrated through a typed plugin interface (`EvaluatorPlugin`).
- **PydanticAI Agent System** (Sprint 1–2): Four-agent pipeline (Manager → Researcher → Analyst → Synthesizer) with configurable composition. Supports single-LLM and multi-LLM operation through `AgentComposition` parameters.
- **Plugin Architecture** (Sprint 3): `EvaluatorPlugin` interface and `PluginRegistry` enable adding new evaluation metrics without changes to the core pipeline. Typed context exchange between tiers via Pydantic models.
- **Operational Resilience** (Sprint 4–5): Provider fallback chain for Tier 2, configurable token limits, adaptive weight redistribution in single-agent operation, background execution in the GUI without tab interruption.
- **Security Hardening** (Sprint 6): SSRF prevention with domain allowlisting, prompt injection resistance, sensitive data filtering from logs and traces, input size limits. OWASP MAESTRO 7-layer security review conducted.
- **Benchmarking Infrastructure** (Sprint 6): `SweepRunner` for 8 agent compositions x N papers x N repetitions, `SweepAnalyzer` for statistical analysis, `CCTraceAdapter` for processing Claude Code artifacts.
- **Claude Code Baseline** (Sprint 6–7): Complete CC Solo and CC Teams artifacts collected for Paper 1105.1072. `--engine=cc` flag for CLI and sweep implemented for direct comparability.
- **Documentation and Tests** (Sprint 7): Architecture, usage, and API documentation updated, test suite restructured toward behavioral coverage (595 → 564 tests without coverage loss).

7.2 Core Empirical Findings

The findings derived from 30 traces and a single CC comparison run (Paper 1105.1072) can be summarized as follows:

Latency and Scaling in PydanticAI MAS: The Manager-Only configuration achieves a median duration of 4.8 s with an error rate of 0%. The three-agent configuration requires a mean of 12.3 s (+156%) with an initialization error rate of 25%. Execution time is dominated by LLM inference latency; tool execution times are negligible (0.004–0.09 s).

Outliers from Provider Latency: Two runs exceeded 69 s and 136 s respectively due to LLM provider latency spikes or rate limiting, not from coordination overhead. This underscores the need for retry mechanisms with exponential backoff.

PydanticAI vs. Claude Code: PydanticAI is 25–75x faster and approximately 50–100x more cost-effective than CC for the same task. CC uses 6–7x more tool calls because the codebase is explored at runtime. CC Teams incurs a 3x overhead over CC Solo. Structurally, CC Teams and PydanticAI multi-agent are analogous (each with Researcher/Analyst/Synthesizer), but differ significantly in latency and resource consumption.

Evaluation Pipeline: All three tiers are implemented and unit-tested. Composite quality scores could not be computed on real traces due to blocking issues. Latency and cost metrics from trace data are fully available.

7.3 Scientific Contributions

The project makes the following methodological contributions:

- **Three-Tier Evaluation Methodology**: The combination of traditional text metrics (Tier 1), LLM-as-Judge (Tier 2), and graph-based behavioral analysis (Tier 3) enables multi-dimensional assessment that uses coordination patterns from execution traces as the primary information source.
- **Post-Execution Graph Analysis** (ADR-004): Agent behavior is retrospectively reconstructed from observability logs without influencing the execution itself.
- **Adaptive Weight Redistribution**: In single-agent operation, `coordination_quality` is automatically removed from the composite score, allowing single- and multi-agent configurations to be comparably evaluated.
- **Infrastructure for Empirical MAS Comparisons**: The combination of `SweepRunner`, `CC-TraceAdapter`, and `BaselineComparison` model enables reproducible comparisons between PydanticAI MAS and Claude Code baseline on the same dataset and tasks.

7.4 Limitations

This work has the following limitations:

- **Blocking Bug**: The `AgentRunResult.data` error on the `refactor-arch` branch prevents end-to-end evaluation runs. All composite quality scores are based on estimates or could not be computed.
- **Empty Sweep Results**: The `SweepRunner` produces no evaluable output. Composition comparisons are therefore not statistically grounded.
- **n=1 for Claude Code**: The CC comparison is based on a single paper and a single run per mode. Statistical significance requires at least 5 runs per configuration.
- **CC Teams Artifacts Ephemeral**: After completion of a `claude -p` run, `~/.claude/teams/` artifacts are not persistent. The `CCTraceAdapter` Teams parser cannot fully operate without these artifacts.
- **Tier 2 Blockage**: The `GITHUB_API_KEY` is not set in the execution environment. Tier 2 LLM-as-Judge evaluations are therefore unavailable, and composite scores are based solely on Tier 1 and Tier 3.
- **Missing Sub-Agent Token Counts**: Token consumption is only logged at the Manager level. Complete cost comparisons between configurations are therefore not possible.
- **No Quality Validation**: A comparison of the content quality of generated reviews (Tier 2, Tier 3 composite) is not available. All findings relate exclusively to latency, cost, and tool usage patterns.

7.5 Outlook and Future Development

The planned further development addresses both identified blockers and strategic extensions of the framework.

Short-term (Sprint 8): - **Report Generation** (Feature 1): After completing an evaluation, a structured Markdown report with Tier 1/2/3 breakdown, identified weaknesses, and actionable improvement suggestions is generated. Available via the `--generate-report` flag in the CLI and as a button in the GUI. - **Graph Attribute Alignment** (Feature 2): Alignment of Tier 3 graph metrics to attributes actually available in `GraphTraceData`, to avoid computation errors from missing fields. - **MAESTRO Hardening** (Feature 3): Implementation of remaining findings from the Sprint 5 security review for the layers Model, Agent Logic, Integration, Monitoring, Execution, Environment, and Orchestration. - **PydanticAI Streaming** (Feature 4): Investigation of the `NotImplementedError` exception for structured outputs in streaming mode (known AGENT_REQUESTS item).

Medium-term: - **Bug Fix `AgentRunResult.data`**: Unblocks all end-to-end evaluation runs and enables computation of composite scores on real traces. - **Sweep Results**: After bug fix, restart the composition sweep across all 8 configurations and multiple papers with statistically robust repetition counts ($≥$5 runs). - **Increase CC Sample Size**: At least 5 runs per CC mode (Solo/Teams) across multiple papers for statistically significant comparisons. - **Claude Agent SDK Migration** (ADR-008): Replacement of `subprocess.run([claude, "-p"])` with the `claude-agent-sdk` package for more portable CC baseline invocation. - **Per-Sub-Agent Token Logging**: Extension of the trace format with token counts at the sub-agent level for complete cost comparisons.

Long-term: - **Framework Extension**: Integration of additional agent frameworks (LangChain, AutoGen, CrewAI) through standardized adapters to enable cross-framework comparisons. - **Extended Metrics**: Implementation of candidate metrics identified in the architecture (`fix_rate`, `evaluator_consensus`, `delegation_depth`, `coordination_topology`, `path_convergence`, `rubric_alignment`) by priority. - **Optional Container Deployment** (ADR-007): Docker images and Compose configurations for parallel judge execution and production isolation. - **Domain Diversification**: Extension beyond scientific paper reviews to additional analytical tasks, to demonstrate the generalizability of the evaluation framework.

List of Abbreviations

Abbreviation	Meaning
AC	Acceptance Criteria
ADR	Architectural Decision Record
API	Application Programming Interface
BDD	Behavior-Driven Development
BibTeX	Bibliography management format for LaTeX and Pandoc
C4	Context, Container, Component, Code (architecture model)
CC	Claude Code (Anthropic's agent-based CLI tool)
CI/CD	Continuous Integration / Continuous Deployment
CLI	Command-Line Interface
CORS	Cross-Origin Resource Sharing
CSL	Citation Style Language
CSV	Comma-Separated Values
DRY	Don't Repeat Yourself (redundancy avoidance principle)
GAIA	General AI Assistants (benchmark suite)
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IEEE	Institute of Electrical and Electronics Engineers
JSON	JavaScript Object Notation
JSONL	JSON Lines (line-delimited JSON format)

Abbreviation	Meaning
KISS	Keep It Simple, Stupid (system simplification principle)
LLM	Large Language Model
MAESTRO	Multi-Agent Environment Security Threat and Risk Ontology
MAS	Multi-Agent System
MCP	Model Context Protocol
ML	Machine Learning
NLP	Natural Language Processing
OWASP	Open Web Application Security Project
PDF	Portable Document Format
PRD	Product Requirements Document
RAM	Random Access Memory
REST	Representational State Transfer
SDLC	Software Development Life Cycle
SSRF	Server-Side Request Forgery
TDD	Test-Driven Development
UI	User Interface
URL	Uniform Resource Locator
YAGNI	You Aren't Gonna Need It (premature implementation avoidance principle)
YAML	YAML Ain't Markup Language

Appendices

7.6 Appendix A: ADR Summary

The following architectural decisions were documented and justified during system development. Each ADR describes the decision context, alternatives considered, and the choice made.

ADR	Title	Decision	Status
ADR-001	PydanticAI as Agent Framework	PydanticAI for multi-agent orchestration	Active
ADR-002	PeerRead Dataset Integration	PeerRead as primary evaluation benchmark	Active
ADR-003	Three-Tier Evaluation Framework	Traditional Metrics → LLM-as-a-Judge → Graph Analysis	Active
ADR-004	Post-Execution Graph Analysis	Retrospective trace processing instead of real-time monitoring	Active
ADR-005	Plugin-Based Evaluation Architecture	EvaluatorPlugin interface with PluginRegistry	Active
ADR-006	pydantic-settings Migration	BaseSettings classes instead of JSON configuration files	Active
ADR-007	Optional Container-Based Deployment	Local execution as default, containers optional	Proposed (deferred)
ADR-008	CC Baseline Engine: subprocess vs. SDK	subprocess.run for Sprint 7; evaluate SDK migration for Sprint 8	Active

7.7 Appendix B: System Requirements

7.7.1 Minimum System Requirements

- **Python**: 3.13 or higher (exactly 3.13.x required)

- **RAM**: 4 GB (8 GB recommended)
- **CPU**: 2 cores (4 cores recommended)
- **Storage**: 10 GB available disk space
- **Network**: Internet connection for LLM provider APIs

7.7.2 Development Environment

- **uv**: Package manager for dependency management and virtual environments
- **Ruff**: Code formatting and linting
- **Pyright**: Static type analysis (mode: strict)
- **Pytest**: Test framework with asyncio support
- **MkDocs**: Documentation generation

7.7.3 Core Production Dependencies (from pyproject.toml)

Package	Version	Purpose
pydantic-ai-slim	>=1.59.0	Multi-agent orchestration
pydantic	>=2.12.5	Data validation and data models
pydantic-settings	>=2.12.0	Type-safe configuration via environment variables
logfire	>=4.24.0	Structured logging and observability
networkx	>=3.6.1	Graph-based behavioral analysis (Tier 3)
arize-phoenix	>=13.0.0	Local trace viewer for observability
scikit-learn	>=1.8.0	Text similarity metrics (Tier 1)
streamlit	>=1.54.0	Graphical user interface
openinference-instrumentation-pydantic-ai	>=0.1.12	PydanticAI auto-instrumentation

7.8 Appendix C: Supported LLM Providers

The framework supports a variety of LLM providers through PydanticAI's OpenAI-compatible interfaces. Providers are configured via CLI arguments (`--chat-provider`, `--judge-provider`) or environment variables.

Provider	Type	Characteristics
OpenAI	Cloud	GPT-4o and further models; default reference provider
Google Gemini	Cloud	Multimodal capabilities; large context window
Anthropic	Cloud	Claude models; balanced evaluation quality
Ollama	Local	Privacy-focused implementations without API costs
OpenRouter	Cloud gateway	Aggregator for multiple providers
Together AI	Cloud	Batch inference and open-source models
HuggingFace	Cloud/Local	Access to open-source models
Cerebras	Cloud	Hardware-accelerated inference
Groq	Cloud	High-speed LPU inference
XAI	Cloud	Grok model family

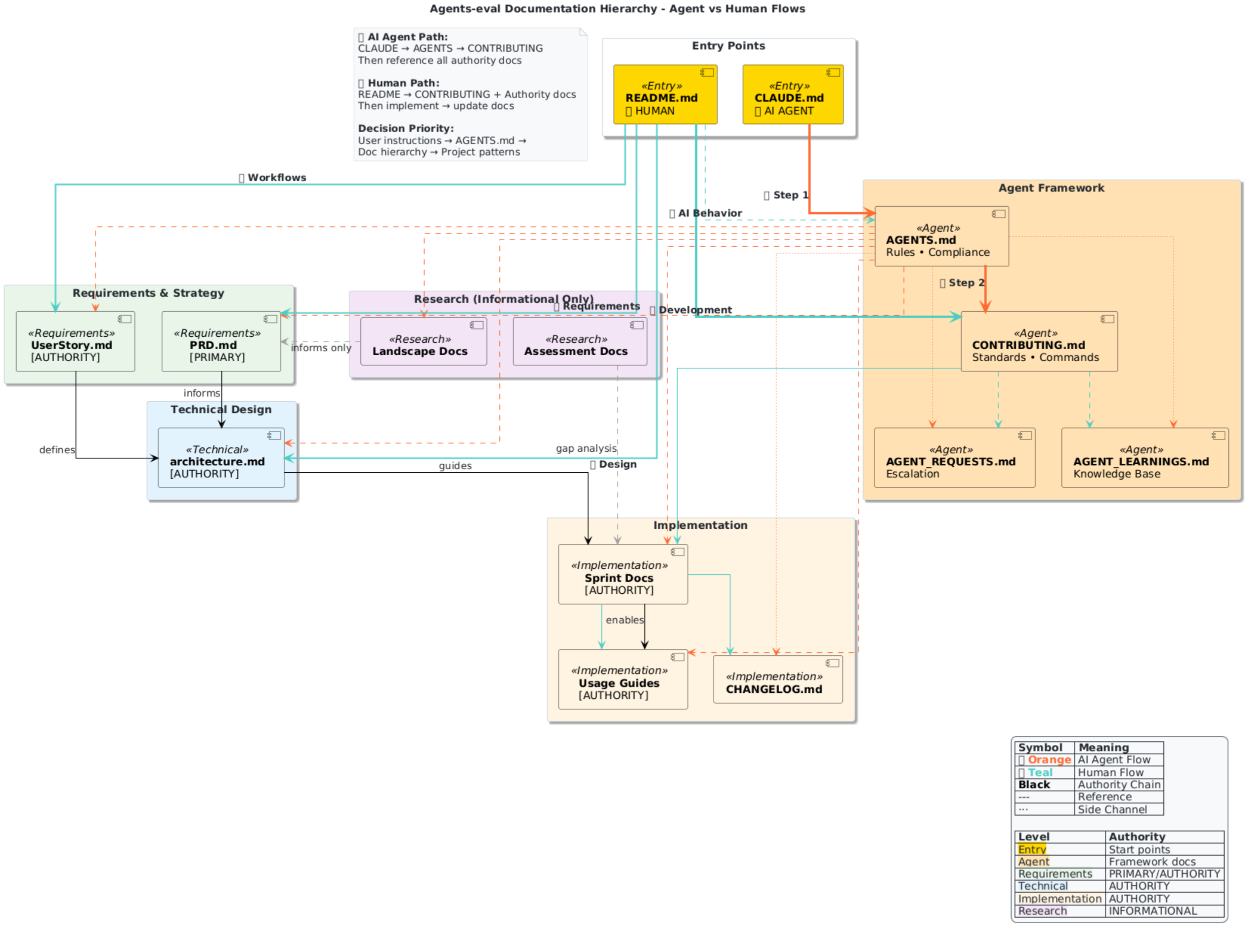
7.9 Appendix D: Documentation Hierarchy

The project follows a structured documentation hierarchy that prevents scope creep and defines clear authoritative sources. Each document has a specific scope and serves as the single source of truth for its domain.

The complete hierarchy is described in AGENTS.md. The following figure illustrates the reference structure and authority chain:

Authority Chain (Reference Flow):

```
PRD.md (Requirements) $\rightarrow$ architecture.md (Technical
↪   Design)
 $\rightarrow$ Sprint Documents (Implementation) $\rightarrow$ Usage
↪   Guides (Operations)
    ↑
Landscape Documents (inform strategic decisions, do not create
↪   requirements)
```



AI Agent Path:
CLAUDE → AGENTS → CONTRIBUTING
Then reference all authority docs

Human Path:
README → CONTRIBUTING + Authority docs
Then implement → update docs

Decision Priority:
User instructions → AGENTS.md →
Doc hierarchy → Project patterns

Symbol	Meaning
Orange	AI Agent Flow
Teal	Human Flow
Black	Authority Chain
---	Reference
···	Side Channel

Level	Authority
Entry	Start points
Agent	Framework docs
Requirements	PRIMARY/AUTHORITY
Technical	AUTHORITY
Implementation	AUTHORITY
Research	INFORMATIONAL

Figure 7.1: Documentation hierarchy

Index

A

Agentic AI Systems: Autonomous systems with goal-oriented behavior that independently make decisions and employ tools without being explicitly instructed for each step.

AgentOps: Cloud-based observability platform for agent behavior tracking and performance analytics (optional, commented out in pyproject.toml).

Analyst Agent: Specialized agent for verifying the correctness of assumptions, facts, and conclusions in the multi-agent workflow.

Arize Phoenix: Local trace viewer for PydanticAI execution traces (arize-phoenix>=13.0.0); replaces Docker-based alternatives.

B

Benchmarking Infrastructure: Sweep-based system (SweepRunner, SweepAnalyzer) for systematic comparison of MAS compositions across multiple papers and repetitions.

C

CC Baseline: Claude Code as reference engine (–engine=cc) for comparison against PydanticAI MAS; invoked via subprocess.run([claude, “-p”]) (ADR-008).

Claude Code (CC): Anthropic’s headless CLI tool for agentic development tasks; serves as baseline comparison in the benchmarking infrastructure.

Composite Scoring: Weighted summation formula from six core metrics (0.167 weight each) with adaptive weight redistribution for missing tiers.

CompositeResult: Pydantic output model of composite_scorer.py with overall score, individual metrics, and single_agent_mode flag.

D

DuckDuckGo Search Tool: Search API of the Researcher Agent for external information acquisition (pydantic-ai-slim[duckduckgo]).

E

EvaluatorPlugin: Abstract base class (ABC) for all evaluation engines; defines name, tier, evaluate(), and get_context_for_next_tier() interface.

G

GraphTraceData: Pydantic model for representing execution graphs; contains agent IDs, coordination_-events, and tool call sequences for Tier 3 analysis.

J

JudgeSettings: pydantic-settings BaseSettings class with JUDGE_ prefix; replaces JSON configuration files (ADR-006).

K

KISS / DRY / YAGNI: Core principles of the codebase (Keep It Simple, Don't Repeat Yourself, You Aren't Gonna Need It); mandatory for all implementation decisions.

L

LLM-as-Judge: Evaluation methodology in which a large language model semantically assesses the quality of agent outputs (Tier 2).

Logfire: Structured logging framework (logfire>=4.24.0) with PydanticAI auto-instrumentation for trace capture and observability.

M

Manager Agent: Primary orchestrator of the multi-agent system; delegates tasks to Researcher, Analyst, and Synthesizer agents.

MAS (Multi-Agent System): Distributed system with multiple specialized, interacting agents; central evaluation subject of the framework.

MAESTRO: OWASP Multi-Agent Environment Security Threat and Risk Ontology; 7-layer security model (Model, Agent Logic, Integration, Monitoring, Execution, Environment, Orchestration).

N

NetworkX: Python library for graph-based behavioral analysis (networkx>=3.6.1); constructs and analyzes execution graphs from observability traces (Tier 3).

O

OWASP: Open Web Application Security Project; foundation of the MAESTRO security model.

P

PeerRead: Academic dataset with 14,775 scientific papers from NIPS, ICLR, and ACL, including structured peer reviews; primary evaluation benchmark (ADR-002).

Plugin Architecture: Extension concept with EvaluatorPlugin interface and PluginRegistry for tier-ordered execution without modification of the core pipeline (ADR-005).

PluginRegistry: Central registration of all evaluation plugins; enables automatic discovery and execution in tier order ($1 \rightarrow 2 \rightarrow 3$).

PydanticAI: Type-safe agent framework (pydantic-ai-slim>=1.59.0) for structured multi-agent orchestration with Pydantic validation (ADR-001).

R

Researcher Agent: Specialized agent for data collection and verification; equipped with DuckDuckGo search tool for external information acquisition.

S

Streamlit: Web framework for the graphical user interface (streamlit>=1.54.0); supports background execution and real-time debug log streaming.

SweepRunner: Benchmarking component for systematic composition sweeps (8 agent compositions x N papers x N repetitions).

Synthesizer Agent: Specialized agent for creating formatted scientific reports from validated agent results.

T

Tier 1 (Traditional Metrics): Fast, objective text similarity measurement (cosine, Jaccard, BLEU/ROUGE) as validation baseline.

Tier 2 (LLM-as-a-Judge): Semantic quality assessment through configurable judge provider; automatic fallback when API keys are missing.

Tier 3 (Graph Analysis): Primary evaluation tier; post-execution behavioral analysis from observability traces using NetworkX (ADR-004).

W

Weave: Weights & Biases ML experiment tracking integration (weave>=0.52.28); optionally available as wandb dependency group.

References

[1] Autonomous Agents Survey Consortium, "A survey on large language model based autonomous agents," *arXiv preprint arXiv:2308.11432*, 2023, Available: https://arxiv.org/abs/2308.11432

[2] UC Berkeley Team, "Berkeley function-calling leaderboard." https://gorilla.cs.berkeley.edu/leaderboard.html, 2024.

[3] AgentEvals Consortium, "CORE-Bench leaderboard: Comprehensive evaluation of agent capabilities." https://huggingface.co/spaces/agent-evals/core_leaderboard, 2024.

[4] GAIA Benchmark Team, "GAIA leaderboard: General AI assistant benchmark." https://gaia-benchmark-leaderboard.hf.space/, 2024.

[5] LLM Agent Evaluation Group, "Survey on evaluation of LLM-based agents," *arXiv preprint arXiv:2503.16416*, 2025, Available: https://arxiv.org/abs/2503.16416

[6] Pydantic Development Team, "PydanticAI documentation: Agent framework with type safety." https://ai.pydantic.dev/, 2024.

[7] Microsoft Corporation, "AutoGen: Multi-agent conversation framework." https://github.com/microsoft/autogen, 2024.

[8] CrewAI Inc., "CrewAI: Framework for orchestrating role-playing, autonomous AI agents." https://github.com/crewAIInc/crewAI, 2024.

[9] LangChain, Inc., "LangChain: Building applications with LLMs through composability." https://github.com/langchain-ai/langchain, 2024.

[10] D. Kang, D. Radev, T. Head, *et al.*, "A dataset of peer reviews (PeerRead): Collection, insights and NLP applications," in *Proceedings of the 2018 conference of the north american chapter of the association for computational linguistics: Human language technologies*, 2018. Available: https://arxiv.org/pdf/1804.09635

[11] AllenAI Team, "PeerRead dataset: A dataset of peer reviews." https://github.com/allenai/PeerRead, 2018.

[12] Agents-eval Project, "Architecture documentation." https://github.com/qte77/Agents-eval/blob/main/docs/architecture.md, 2025.

[13] Pydantic Development Team, "PydanticAI tools documentation." https://ai.pydantic.dev/tools/, 2024.

[14] S. Brown, "The C4 model for visualising software architecture." https://c4model.com/, 2024.

[15] Agents-eval Project, "CHANGELOG.md – change history." https://github.com/qte77/Agents-eval/blob/main/CHANGELOG.md, 2025.

[16] Agents-eval Project, “MAS comparison findings: Single-LLM vs multi-LLM vs claude code.” https://github.com/qte77/Agents-eval/blob/main/docs/write-up/findings/mas-comparison-findings.md, 2026.

[17] Agents-eval Project, “AGENTS.md – agent instructions.” https://github.com/qte77/Agents-eval/blob/main/AGENTS.md, 2025.

[18] Agents-eval Project, “User story documentation.” https://github.com/qte77/Agents-eval/blob/main/docs/UserStory.md, 2025.

[19] D. R. MacIver, “Hypothesis: Property-based testing for Python.” https://hypothesis.readthedocs.io/, 2024.

[20] Agents-eval Project, “AGENT_LEARNINGS.md – accumulated agent patterns.” https://github.com/qte77/Agents-eval/blob/main/AGENT_LEARNINGS.md, 2025.