Adam Dybwad Sioud

P2Engine: A Multi-Agent System Framework

Master's thesis in Computer Science
Supervisor: Surya Kathayat
July 2025

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU

ABSTRACT

Agents demonstrate remarkable capabilities, yet multi-agent systems frequently struggle to achieve collective effectiveness due to missing architectural foundations. This thesis introduces and validates P2Engine, a multi-agent system (MAS) framework addressing this gap by integrating four foundational pillars: orchestration, observability, adaptation, and auditability. Leveraging finite-state machine orchestration, real-time system observability, adaptive performance loops, and distributed-ledger auditability, P2Engine enables agentic behavior at the collective system-wide level. Through detailed evaluation and validation, this thesis demonstrates how these integrated pillars support dynamic, transparent, adaptive, and accountable interactions among agents. This work advances MAS architecture and provides an extensible foundation explicitly designed to facilitate future research and development in multi-agent systems.

SAMMENDRAG

Agenter har imponerende egenskaper, men systemer med flere agenter opplever ofte utfordringer med å oppnå kollektiv effektivitet på grunn av manglende fundament i arkitekturen. Denne masteroppgaven introduserer og validerer P2Engine, et multi-agent system (MAS) rammeverk som adresserer dette gapet ved å integrere fire grunnleggende pilarer: orkestrering, observabilitet, tilpasning og reviderbarhet. Ved å utnytte orkestrering basert på endelige tilstandsmaskiner, sanntids observabilitet av systemet, tilpasningsdyktige ytelsessløyfer og reviderbarhet gjennom distribuerte registre, muliggjør P2Engine agentbasert atferd på kollektivt systemnivå. Gjennom detaljert evaluering og validering viser denne studien hvordan disse integrerte pilarene støtter dynamiske, transparente, tilpasningsdyktige og ansvarlige interaksjoner mellom agenter. Dette arbeidet styrker utviklingen av MAS-rammeverk og gir et utvidbart grunnlag, spesifikt designet for å legge til rette for fremtidig forskning og utvikling innen systemer med flere agenter.

PREFACE

This master's thesis marks the culmination of my studies at the Norwegian University of Science and Technology and extends the work I began in my TDT4501 specialization project. I am deeply grateful to my supervisor, Surya Kathayat, for his support and the creative freedom he afforded me over these past ten months, which made the development of the P2Engine framework possible.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Abbreviations

Abbreviation	Definition
AI	Artificial Intelligence
DLT	Distributed Ledger Technology
FSM	Finite-State Machine
LLM	Large Language Model
MAS	Multi-Agent System

CHAPTER

ONE

INTRODUCTION

Individual artificial intelligence (AI) Agents[1] have reached remarkable capabilities. They write code at a high level, execute complex tasks, and engage in nuanced reasoning across diverse domains (Luo et al. 2025). Yet when these same powerful agents are assembled into multi-agent systems, they consistently fail to deliver on their collective promise (Han et al. 2024). A group of agents that should amplify each other's strengths instead produce brittle, and become unpredictable systems that break down. This fundamental disconnect between individual agent capability and a collective system performance represents one of the most pressing challenges in multi-agent systems today (Cemri et al. 2025) (Yan 2025).

The root cause lies not in the agents themselves, but in the absence of architectural foundations that enable truly agentic behavior at the collective system level. Current multi-agent frameworks force practitioners to choose between rigid, predetermined workflow frameworks that cannot adapt to emergent needs, or chaotic free-form interactions that provide no operational guarantees (Cemri et al. 2025). These systems falter at the collective level whenever the surrounding architecture lacks coherent orchestration, observability, adaptation, and auditability. Motivated by this fundamental gap, this research embarks on a quest to discover and validate the optimal architectural approach for enabling multi-agent systems that can decide while running how many agents to deploy, how to divide labor, and how to readjust their work as conditions evolve, all at the same time as remaining transparent, continuously learnable, and verifiably trustworthy.

[1] In this thesis, an "agent" is defined as autonomous actor powered by a large language model (Yehudai et al. 2025) .

1.1 Foundational Pillars

This thesis identifies four foundational pillars: **orchestration**, **observability**, **adaptation**, and **auditability** as essential for enabling agentic multi-agent systems. These pillars are not arbitrary: they emerge from core principles in systems theory, which are discussed in Chapter 2 to establish the conceptual grounding to achieve the objectives of this thesis.

Orchestration enables dynamic coordination where the system itself decides task distribution and agent deployment, moving beyond rigid workflows to emergent, configurable behavior while maintaining complete conversation traceability.

Observability provides complete transparency enabling real-time monitoring and retrospective analysis of all agent actions and system states, capturing the causal relationships that drive collective behavior.

Adaptation converts experience into improved performance through feedback loops operating at both system-wide and individual agent levels, enabling continuous optimization.

Auditability ensures immutable record-keeping with monetary incentives through distributed ledger technology, enabling agent reward, alignment, and accountability.

1.2 The Integration Challenge

No existing framework provides integrated infrastructure for all four pillars end-to-end (see Section 3.2 in Chapter 3). This forces practitioners to either compromise on capabilities or invest significant effort in infrastructure development rather than advancing agentic behaviors. This infrastructure gap represents a fundamental bottleneck in multi-agent systems research.

1.3 Research Objective and Questions

Research Objective (RO): Design, implement, and validate a multi-agent system framework that integrates orchestration, observability, adaptation, and auditability capabilities required for truly agentic multi-agent systems.

This objective is addressed through one primary research question and four supporting questions:

RQ1: Can a multi-agent system framework combining finite-state orchestration, comprehensive observability, adaptation infrastructure, and distributed-ledger auditability deliver the foundations necessary to enable truly agentic multi-agent systems?

To answer RQ1, investigate the four targeted questions, each corresponding to one of the foundational pillars (see Table 1.3.1):

Table 1.3.1: Supporting Research Questions

Question	Focus
RQ1.1 Orchestration	How can FSM-driven orchestration infrastructure support dynamic agent coordination patterns while maintaining system coherence and extensibility for emergent behaviors?
RQ1.2 Observability	What infrastructure components provide complete system transparency and enable real-time monitoring without compromising performance or agent autonomy?
RQ1.3 Adaptation	How can evaluation and rollout infrastructure provide the foundation for systematic improvement through automated feedback loops at both system and agent levels?
RQ1.4 Auditability	How can distributed ledger infrastructure provide the foundation for tamper-evident audit trails and monetary incentive mechanisms required for agent accountability?

1.4 Research Contributions

Through extensive analysis of existing multi-agent system frameworks (see Chapter 3) the research identifies finite-state machine orchestration (see Chapter 5) as the optimal approach for integrating the four foundational pillars.

Building on this architectural discovery, this thesis presents **P2Engine**, A robust and foundational framework that provides integrated orchestration, observability, adaptation, and auditability infrastructure, demonstrating the first end-to-end implementation of all four pillars required for truly agentic multi-agent systems:

Orchestrate dynamically through finite-state machine orchestration that handles non-deterministic agent outputs while maintaining complete conversation flows

Operate transparently with real-time observability of all system activities and causal relationships

Enable systematic improvement through evaluation-driven adaptation loops that modify behavior based on performance outcomes

Act accountably with distributed ledger audit trails and monetary incentive mechanisms for alignment

This thesis advances multi-agent systems research through four principal contributions that collectively establish a foundational framework for observable, adaptive, and accountable operation, as detailed in Table 1.4.1:

Table 1.4.1: Research Contributions

Contribution	Description and Impact
C1: Architectural Analysis	Comprehensive evaluation of multi-agent system orchestration paradigms leading to the identification of finite-state machine approaches as optimal for four-pillar integration.
C2: MAS Framework	A novel multi-agent system framework (P2Engine) that demonstrates practical integration of finite-state orchestration, comprehensive observability, adaptation infrastructure, and DLT.
C3: DLT Integration	DLT infrastructure providing tamper-evident financial transaction capabilities and extensible foundation for performance-based agent incentive mechanisms
C4: Adaptation Infrastructure	Comprehensive evaluation infrastructure with rollout capabilities providing the foundation for systematic adaptation methods and continuous improvement mechanisms

Together, these contributions establish both a working framework and transferable design principles for building multi-agent systems.

1.5 Scope and Limitations

This thesis covers the end-to-end design, implementation, and evaluation of core MAS components, with a focus on integrating a finite-state orchestrator, distributed-ledger technology, and the adaptation infrastructure. To keep the initial scope manageable, some of the complex integrations such as adaptaion methods have been omitted, treating them as future extensions built on the p2engine foundation (see Table 1.5.1).

Table 1.5.1: Scope Boundaries

Boundary	**Rationale and Future Work**
Orchestration Paradigms	This thesis focuses on finite-state machine orchestration identified as optimal through systematic analysis. Future work could explore integration approaches using alternative paradigms identified in the architectural survey.
Adaptation Methods	While the architecture provides comprehensive infrastructure for adaptation, implementation of complete adaptation methods remains future work, with current focus on infrastructure validation.
Adaptation Depth	Full reinforcement learning or weight fine-tuning loops are stubbed; future work can implement complete adaptation methods within the validated infrastructure.
Auditability Scope	Implementation demonstrates distributed ledger integration through financial transaction verification. Full audit trail of all agent actions constitutes future extension of the established foundation.
Performance and Scale	Thesis prioritizes architectural correctness and functional integration over performance optimization, establishing foundation for future scalability research.
Data Privacy	Only synthetic or public datasets are used; no personal data is processed. Real privacy scenarios remain future work building on the privacy-preserving foundation.

1.6 Thesis Structure

The thesis Structure follows Design Science Research methodology (Kotzé, Merwe and Gerber 2015), progressing from problem identification through systematic architectural analysis to artifact development and continous evaluation, as outlined in Table 1.6.1:

Table 1.6.1: Thesis Structure

Chapters	**Content**
Chapters 2–4	Theoretical foundations, comprehensive related work survey and architectural analysis, and a research methodology establishing the proper approach to framework discovery and validation.
Chapters 5–6	Conceptual system design based on architectural analysis findings, and detailed P2Engine implementation demonstrating integration of the four foundational pillars.
Chapters 7–9	Experimental validation against established criteria, future work directions building on validated foundations, and conclusions synthesizing contributions to multi-agent systems research.

CHAPTER

TWO

FOUNDATIONAL PILLARS

This chapter provides the foundations thst is assumed in later chapters. An exploration of *Systems Theory* (Bertalanffy 1968) and examination on how the research arrived at the four foundational pillars introduced in Chapter 1 (Section 1.1): **Orchestration**, **Observability**, **Adaptation**, and **Auditability**.

2.1 Systems

A system can be defined as a group of interacting components that act together for some purpose and co-evolve with their environment (Bertalanffy 1968). This broad definition spans examples from algae blooms to stock exchanges to swarms of AI agents. Importantly, a system's behavior arises from interactions among parts rather than any single part alone.

Not all systems are equally complicated. Following common taxonomies in systems theory, systems can range from simple (just a few elements with clear cause and effect) through more complicated (many elements but still with predictable interactions) to complex (many interacting parts with emergent and hard to predict behavior) and ultimately to complex adaptive systems (CAS) (complex systems whose components actively change their behavior or rules) (Baldwin et al. 2011). Autonomy becomes compelling only in the upper end of this spectrum, interest point is CAS. In CAS components can modify their own policies. CAS often has a large number of components, some of them are agents, these interact, they adapt, and they learn (Baldwin et al. 2011). In other words, when parts of a system can adapt in response to feedback, top-down control is insufficient and autonomous orchestration is needed (Jessop 2003).

2.2 From Principles to Pillars

Four core systems-theoretic principles motivate the foundational pillars: *Holism*, *Emergence*, *Interdependence*, and *Feedback* (Skyttner 2005). Holism means the system's overall behavior is more than the sum of individual parts; relatedly, emergent properties may appear at the system level that one cannot predict from examining components in isolation (Goldstein 1999). Interdependence implies that a change in one part can propagate in nonlinear ways throughout the system (Bertalanffy 1968). Feedback refers to cyclic cause-and-effect: positive feedback loops can amplify changes, while negative loops counteract changes to stabilize the system (Wiener 1948). In the context of agentic multi-agent systems, the thesis consider incentive alignment (ensuring each component's objectives don't conflict with the system's objectives) as an important design principle. These ideas map directly to the four foundational pillars shown in Table 2.2.1.

Table 2.2.1: Principles Map to Foundational Pillars

Principle	**Resulting Pillar**
Holism & Emergence	Orchestration
Interdependence	Observability
Feedback	Adaptation
Incentive Alignment	Auditability

2.3 The Four Pillars

2.3.1 Orchestration

No single rigid plan can dictate a complex adaptive system; instead, dynamic orchestration infrastructure must support patterns of interaction that can self-organize rather than forcing rigid predetermined flows (Heylighen 2007). Autonomous orchestration leverages emergent teamwork rather than micromanaging each agent. This pillar emerges from the principles of holism and emergence, where system-wide coordination arises naturally from component interactions.

2.3.2 Observability

When everything is interconnected, full observability is critical. One must monitor system-wide state and causal pathways, since a small perturbation can have far-reaching effects. This pillar stems from interdependence, requiring comprehensive visibility into how changes propagate throughout the system.

2.3.3 Adaptation

Feedback loops enable adaptation, agents improving performance through experience (Holland 1995). A structured way to incorporate feedback (rewards, error signals, etc.) allows the system to learn and adjust its policies over time (Xie and Yuan 2025). This pillar is rooted in feedback mechanisms that enable continuous learning and improvement.

2.3.4 Auditability

Properly aligning incentives and maintaining accountability motivates auditability. If components are self-interested, it is a need for tamper-evident records and reward mechanisms to ensure local actions serve collective goals. This pillar derives from incentive alignment principles, ensuring transparency and accountability in autonomous systems (Dafoe et al. 2020).

2.4 From Pillars to Principles

The relationship between principles and pillars is bidirectional, demonstrating how each pillar directly realizes its corresponding theoretical foundation. Orchestration ← holism & emergence enables dynamic coordination without rigid workflows, allowing system-wide patterns to emerge from component interactions. Observability ← interdependence provides full visibility into state transitions and causal pathways, ensuring that interconnected relationships remain transparent. Adaptation ← feedback creates structured loops that convert experience into improved performance, enabling systematic learning from system behavior. Auditability ← alignment ensures tamper-evident records and monetary incentives so that local actions serve collective objectives, maintaining accountability across distributed components.

The pillars are mutually reinforcing: orchestration without observability creates opacity; adaptation without auditability forfeits accountability; and observability without adaptation yields static monitoring.

2.5 Synthesis

These four pillars are not arbitrary choices but emerge necessarily from the fundamental characteristics of CAS. When individual AI agents are assembled into multi-agent systems, the resulting collective behavior exhibits the hallmarks of CAS: emergent properties that cannot be predicted from individual components

alone, intricate interdependencies where small changes propagate throughout the system, and the need for continuous adaptation based on feedback from dynamic environments (Mitchell 2009).

The pillars reinforce one another in critical ways. Monitoring without adaptation is wasted effort, while adaptation without audit trails risks uncontrollable drift. Similarly, trying to orchestrate without observability leads to opaque behavior and makes it difficult for humans to reason about the system's actions and decisions. Most fundamentally, orchestration without observability creates opacity; adaptation without auditability forfeits accountability; and observability without adaptation yields static monitoring.

A well-designed multi-agent system should seek to be agentic therefore needs to respect all four pillars in concert. Only by integrating orchestration, observability, adaptation, and auditability can bridge the gap between individual agent capability and collective system performance, thus enabling capabilities of agentic behavior at the collective system-wide level.

CHAPTER

THREE

RELATED WORK & RESEARCH GAP

3.1 Multi-Agent Systems Frameworks

This section surveys frameworks that help us build Multi-Agent systems (MAS). They are grouped by their *primary orchestration abstraction* and comment on how each family maps to the four foundational pillars **orchestration**, **observability**, **adaptation**, and **auditability**.

3.1.1 Orchestration Abstractions: A Taxonomy

Table 3.1.1 highlights six recurring orchestration styles. The list is illustrative, not exhaustive; some frameworks are general, not MAS specific, but often used in MAS.

3.1.2 Graph Orchestrators

Frameworks such as **LangGraph**, **CrewAI**, and **Orra** model MAS as directed acyclic graphs of tasks, where each node in the graph represents an LLM call, tool invocation, or a interaction from a human, and the edges carry the data and are in control of signals.

LangGraph has low-level primitives which can be used to construct stateful and templated workflows. These support conditional edges, human-in-the-loop, and enables more long-term memory. It can be used to create both hierarchical orchestration and sequential workflows. Simply LangGraph lets you guide, moderate and control agents in MAS (*LangGraph* 2025).

CrewAI similar to LangGraph, CrewAI enables assembly of agents with built

Table 3.1.1: Frameworks

Framework Family	**Representative Projects**	**Core Orchestration Abstraction**	**Typical Use-Cases**
Graph Orchestrators	LangGraph, CrewAI Flows, Orra	Static or conditional DAG of tasks	RAG pipelines; human-in-loop automations
Role-Play Orchestrators	CAMEL, MetaGPT	Role agents with personna	Emergent reasoning; cooperative behaviour studies
Prompt Composition Frameworks	DSPy, BAML, PydanticAI	Typed prompt components compiled and optimised into chains	Rapid prototyping; schema-driven prompt optimisation
Finite-State-Machine	MetaAgent, AG2, State Machines	Explicit FSM/state-chart with guard conditions	Long-horizon, replayable workflows; audit trails
RL / Trajectory-Centric Frameworks	verl, OpenPipe ART, Atropos, PRIME-RL	Standardised trajectory APIs with rollout workers	Continual RLHF; large-scale policy refinement
POMDP Planners	TensorZero	Belief-state search over POMDPs	Robotics and other uncertain real-world control

in templates and primitives. In addition it offers a drag-and-drop no-code interface making it easier to construct MAS (*The Leading Multi-Agent Platform* 2025).

Orra is built on the idea that agents fail often, therefore it has built in internal ability to recover, adapt and progress if so happens. It has an engine that dynamically synthesizes execution plans based on schemas and model reasoning, with automated health monitoring and error recovery, promises to be a resilient orchestration of agents solution (*Make Your AI Agents Unstoppable* 2025).

3.1.3 Role-Play Orchestrators

Frameworks in this family model agent coordination as free-form, role-based dialogue. Control flow emerges from the chat log itself, with agents exchanging natural-language messages under predefined role prompts.

CAMEL uses prompting to assign clear roles to different AI agents, for example the TaskSpecifier, which decides what needs to be done, and the AI Assistant, which carries out the work. The agents then talk back and forth until they finish the task. CAMEL is especially good at exploring how AI agents cooperate and work together.(*CAMEL: The first and the best multi-agent framework. Finding the Scaling Law of Agents* 2025).

MetaGPT turns Standard Operating Procedures (SOPs) into a series of prompts and assigns AI agents to roles like product manager, architect, and engineer, creating a simulated software company. Each agent reviews the work of the others, which boosts the overall quality of the results in collaborative software engineering tasks.(*MetaGPT: The Multi-Agent Framework* 2025)

3.1.4 Prompt Composition Frameworks

Frameworks in this category elevate prompts to be composable, typed constructs and provide tooling for validation, schema enforcement, and search-based optimization of prompts. They offer a more open ended orchestration where the user itself decides how to orchestrate.

DSPy is a declarative framework for building modular AI software: it represents prompts and their associated code as structured components, then compiles them into optimized sequences of tokens and model weights suitable for many MAS tasks (*DSPy* 2025).

BAML is an expressive language for structured text generation, transforming prompt engineering into schema engineering: each prompt is defined as a function with typed inputs and outputs, enabling type-saftey.(*The AI framework that adds the engineering to prompt engineering)* 2025).

PydanticAI takes inspiration from the Pydantic module. It defines agents as type-annotated classes; prompts, function calls, and response parsing all follow the same declarative schema. This simplifies the construction of agents, and makes it easier to work with MAS where there are multiple interactions between agents. (*PydanticAI: Agent Framework* 2025).

3.1.5 Finite-State Machines

More and more researchers now use finite-state machines (FSMs) to guide how multiple agents work together. A clear state chart makes the system predictable, lets you run it again in exactly the same way, and makes it easy to check what happened.

MetaAgent is a system that automatically creates a finite state machine (FSM) from a task description written in plain language. It then repeatedly removes extra states and tests the chart before it is used. At run time the FSM controls when agents take turns speaking, when it calls external tools, and what each role says. Every change of state is recorded so the entire process can be replayed exactly as it happened. (Zhang, Liu and Xiao 2025).

AG2 offers a notebook API where you upload a transition graph to control which agent speaks next. This ensures clear, predictable turn-taking among agents. (*FSM - User can input speaker transition constraints* 2025).

Agentic State Machines is a open-source library thay provides a catalogue of ready-made FSM patterns (tool use, reflection, human-in-the-loop) expressed in TypeScript; each pattern can be embedded into larger agent graphs or executed standalone (Terlson 2025). The repository illustrates how formal state definitions enable static analysis, fine-grained observability, and modular reuse.

Erik's State Machines is a four-part blog research esque series that describes a system where a central clock makes every agent take one transition per tick through a fixed cycle of interaction states:messages, tool calls, agent calls, user input, then finish. Later entries add conditional branching, an append-only artifact log, and Monte Carlo rollouts to turn the state machine into a search tree (Erik 2025a; Erik 2025b; Erik 2025c; Erik 2025d)

3.1.6 RL / Trajectory-Centric Frameworks

These frameworks integrate a reinforcement learning loop directly into the system and provide a standardised `trajectory` API that separates gathering experience from updating policies.

Verl was released by ByteDance's Volcano Engine. It uses asynchronous

workers to stream `(state, action, reward)` tuples to a trainer that applies PPO or GRPO updates and broadcasts new weights to the workers (*verl: Volcano Engine Reinforcement Learning for LLMs* 2025).

OpenPipe ART offers a plug and play harness that wraps any Python agent loop. It logs trajectories to experiment tracking tools and replays them through a GRPO optimiser before deploying updated models (*ART Docs: Getting Started* 2025).

Atropos provides an environment layer that streams trajectories over gRPC. Its architecture separates environment workers and trainers into independent microservices, enabling elastic scaling and continuous updates (*Introducing Atropos* 2025).

PRIME-RL is a decentralised asynchronous framework designed for large-scale training across heterogeneous networks. It decouples rollout generation, policy optimisation, and model distribution through specialised components (*Scalable RL solution for advanced reasoning of language models* 2025).

Together, these libraries illustrate a shift from static prompt engineering to adaption: trajectories become the lingua franca that decouples experience generation, reward computation, and weight optimisation.

3.1.7 POMDP Planners

POMDP planner frameworks models a MAS application as *partially observable Markov decision processes* (POMDPs), driving reasoning via belief-state search to handle uncertainty in observations and outcomes.

TensorZero implements a planning engine based on partially observable Markov decision processes. It maintains probability distributions over hidden states and uses Monte Carlo tree search to select high value actions for language models (*TensorZero: An Open-Source Stack for Industrial-Grade LLM Applications* 2025).

3.2 Research Gap

The comprehensive survey reveals substantial progress across individual components of multi-agent systems, yet a critical architectural gap remains: **no existing framework provides integrated infrastructure for all four foundational pillars required for truly agentic behavior**.

Building on the integration challenge (Section 1.2) and the systematic analysis above, four fundamental limitations reveals itself, that prevent current frameworks

from achieving more agentic capabilites, where agents can dynamically decide task distribution, adapt continuously, and maintain accountability while remaining fully observable.

3.2.1 The Integration Problem

Current multi-agent frameworks excel in specific domains but fail to provide unified infrastructure spanning all four pillars:

- **Graph Orchestrators** (LangGraph, CrewAI) excel at structured workflows but lack adaptation infrastructure and auditability
- **Role-Play Orchestrators** (CAMEL, MetaGPT) enable emergent behavior but provide no systematic evaluation or financial accountability
- **FSM Frameworks** (MetaAgent, AG2) offer deterministic orchestration but minimal adaptation infrastructure
- **RL Frameworks** (verl, PRIME-RL) provide sophisticated adaptation but operate outside production orchestration systems

This fragmentation forces practitioners to either compromise on capabilities or invest significant effort in infrastructure development rather than advancing agentic behaviors, creating a oppurtunity for multi-agent systems research.

3.2.2 Critical Gaps Analysis

Table 3.2.1 distills the specific limitations that prevent existing frameworks from enabling truly agentic behavior:

These limitations are not independent issues but stem from a fundamental architectural failure: existing frameworks treat orchestration, observability, adaptation, and auditability as separate concerns rather than interdependent pillars of an integrated system.

3.2.3 Architectural Hypothesis

Multi-agent systems that are agentic require a **unified architecture** that integrates all four pillars through:

1) Finite-State Orchestration Enables deterministic yet flexible coordination where the system decides agent deployment and task routing at runtime while maintaining complete traceability.

Table 3.2.1: Architectural Limitations

Pillar	**Outstanding Gap**
Orchestration	Existing frameworks rely on either (i) static DAGs that cannot adapt to non-deterministic agent outputs, or (ii) free-form interactions with no execution guarantees. Missing are *deterministic yet flexible* coordination mechanisms that enable runtime decisions about agent deployment and task routing while maintaining complete traceability.
Observability	Current logging is tool-chain specific and fragmented across components; cross-agent causality must be manually reconstructed. A unified, low-overhead event streaming infrastructure that captures *every* state transition with preserved causal relationships remains unavailable, impeding both real-time monitoring and post-hoc analysis.
Adaptation	Except for model-training frameworks, no systems that are deployed in the wild provide integrated evaluation infrastructure with rollout capabilities to support systematic adaptation methods.
Auditability	None of the frameworks showcase end-to-end audit layer spanning *both* economic events and cognitive steps, preventing comprehensive accountability.

2) Universal Event Streaming Captures all state transitions, tool executions, and agent interactions in a unified, queryable timeline with preserved causal relationships.

3) Adaptation Infrastructure Provides comprehensive evaluation and rollout infrastructure that enables systematic adaptation methods and performance optimization.

4) Distributed Ledger Integration Records both economic transactions and agent decision semantics in tamper-evident audit trails supporting comprehensive accountability.

If these components are architecturally integrated, each exposing minimal interfaces while maintaining orthogonality. The result will be multi-agent systems that can exhibit truly agentic behavior: autonomous task distribution, continuous adaptation, and verifiable accountability while remaining fully observable and debuggable.

3.2.4 Research Contribution Positioning

This thesis addresses the integration gap through **P2Engine**, a proof-of-concept framework demonstrating that finite-state orchestration can serve as the unifying abstraction enabling seamless integration of all four pillars:

- **Finite-state orchestration** via InteractionStack enforcing deterministic transitions while supporting dynamic agent coordination
- **Complete observability** through ArtifactBus capturing every state change and tool execution with causal correlation
- **Adaptation infrastructure** enabling systematic A/B testing through rollouts with automated evaluation and configuration updates
- **Distributed ledger integration** providing tamper-evident audit trails with monetary incentives for agent accountability

By empirically validating this integrated approach against the gaps identified in Table 3.2.1, this research demonstrates a concrete path toward truly agentic multi-agent systems while establishing transferable design principles for future MAS architectures.

The architectural design in Chapter 5, concrete implemenation Chapter 6, and experimental results in Chapter 7 provide evidence that this unified approach successfully addresses the fundamental limitations preventing current frameworks from becoming more agentic.

CHAPTER

FOUR

RESEARCH METHODOLOGY

4.1 Methodology

This research employs **Design Science Research (DSR)** (Kotzé, Merwe and Gerber 2015) methodology to design, implement, and validate P2Engine, a multi-agent system framework that integrates orchestration, observability, adaptation, and auditability in a single coherent architecture. It's proof-of-concept validation, a approach prioritizes practical validation through live system demonstrations, directly addressing the integration challenges identified in existing MAS frameworks.

4.1.1 Research Approach

Given that no existing framework provides integrated infrastructure for all four foundational pillars (Section 3.2), this research adopted a **design-and-build strategy** (Kotzé, Merwe and Gerber 2015). P2Engine was architected from the ground up to demonstrate that true integration is both possible and creates emergent capabilities.

4.1.1.1 Iterative Development and Validation

The development process employed continuous integration testing through P2Engine's CLI application, enabling real-time validation of cross-pillar interactions:

Live Testing Commands executed to verify agent behaviors, tool executions, and state transitions

Real-time Observability Immediate inspection of event logs to trace if sys-

tem works as wanted

Dynamic Configuration Runtime modification of agent behaviors to validate adaptability

Ledger Verification Canton Network validated by logs and CLI operations

This approach enabled rapid iteration and refinement of P2Engine's architecture while maintaining coherent operation across all four pillars.

4.1.2 Validation Framework

P2Engine's validation employs a **comprehensive demonstration strategy** (see Table 1.3.1) that tests both individual components and their integrated operation against the research questions:

E1 – Orchestration Integration (addresses RQ1.1): Emergent agent coordination through FSM orchestration.

E2 – Complete Observability (addresses RQ1.2): Real-time system transparency with full event synergizes across the distributed components.

E3 – Adaptation Infrastructure (addresses RQ1.3): Systematic experimentation through automated rollout execution and evaluation.

E4 – Auditability Integration (addresses RQ1.4): Audit trails via DAML operations on Canton Network ledger and expose monetary incentives.

4.1.3 Testing Methods

Validation evidence is provided through comprehensive video demonstrations that showcase P2Engine's integrated functionality:

CLI Demonstrations: Video recordings of live agent interactions, agent delegation, and system responses using `p2engine chat`, `rollout`, and `ledger` commands.

Complete Reproducibility: All demonstrations use standard P2Engine configuration with documented setup procedures, enabling independent verification of framework.

Open Source Framework: The complete codebase is available for immediate testing and validation of all demonstrated capabilities.

4.1.4 Research Quality

Methodological Strengths: The research provides direct visual evidence through demonstration videos and an open source framework enabling reproducibility. Each research question is systematically addressed through practical demonstration rather than theoretical analysis.

Scope and Focus: The study operates at proof-of-concept scale with comprehensive demonstration scenarios, focusing on functional validation and architectural integration rather than performance optimization.

These choices are appropriate for establishing foundational evidence for a proof-of-concept multi-agent system while demonstrating practical viability of the four foundational pillar approach.

CHAPTER

FIVE

SYSTEM DESIGN

This section presents a conceptual system design that weaves together the four foundational pillars to enable agentic capabilities in multi-agent systems. After extensive research, testing, and iteration, the settled design was in large part inspired by Erik's state-machine architecture (see Subsection 3.1.5) Erik 2025a; Erik 2025b; Erik 2025c; Erik 2025d as it best realizes the pillars within the broader MAS framework and the vision for agentic behavior. This chapter, translates the four foundational pillars into concrete system requirements, outlines the logical architecture, and articulates the core design principles that can drive genuinely agentic behavior.

5.1 Functional Requirements

Based on the four foundational pillars identified in Chapter 1, the system must provide the capabilities outlined in Table 5.1.1:

5.2 Architectural Overview

The system architecture comprises four logical layers that directly correspond to the foundational pillars, with a goal to support agentic behavior as illustrated in Figure 5.2.1:

Table 5.1.1: Functional Requirements via Foundational Pillars

Requirement	Description
Emergent Orchestration	The system shall enable dynamic agent coordination where the system itself decides task distribution and agent deployment, moving beyond rigid workflows to emergent, configurable behavior while maintaining complete conversation flow traceability.
Complete Observability	The system shall provide complete transparency enabling real-time monitoring and retrospective analysis of all agent actions and system states, capturing all interactions in a unified, queryable format that preserves causal relationships.
Collective Adaptation	The system shall provide evaluation infrastructure and rollout capabilities that enable adaptation methods to convert experience into improved performance, with runtime configuration modification capabilities to support continuous improvement.
Full Auditability	The system shall ensure immutable record-keeping with monetary incentives through distributed ledger technology, enabling agent reward, alignment, and accountability while supporting regulatory compliance.

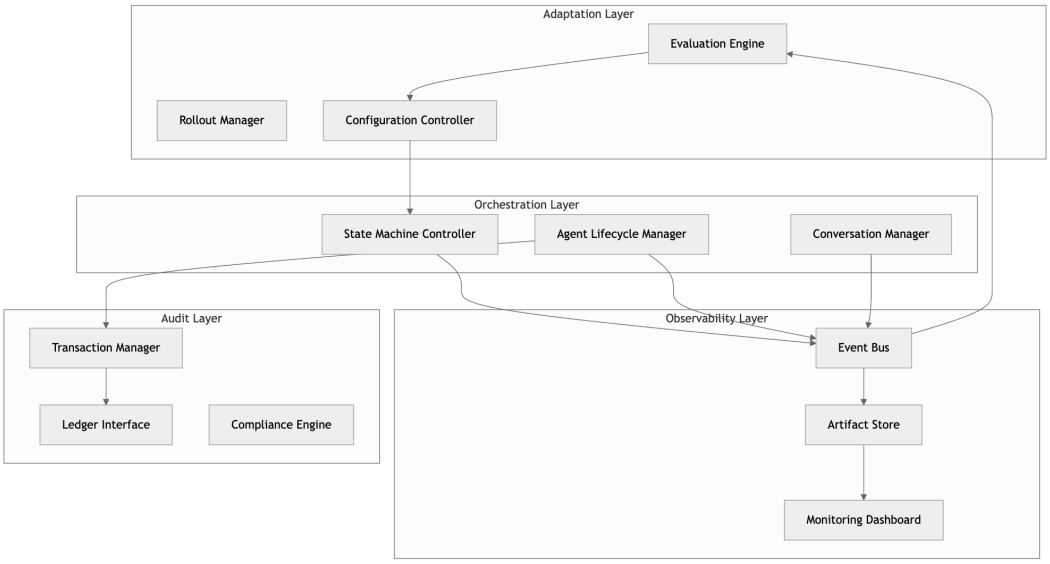


Figure 5.2.1: Four Layer System Architecture

5.2.1 Orchestration Layer

The purpose of the orchestration layer is to enable dynamic agent coordination where the system itself decides task distribution and agent deployment through emergent, configurable behavior.

The orchestration layer implements finite state machine orchestration as shown in Figure 5.2.2, ensuring coherent state progression while enabling dynamic routing decisions and handling non-deterministic outputs from agents.

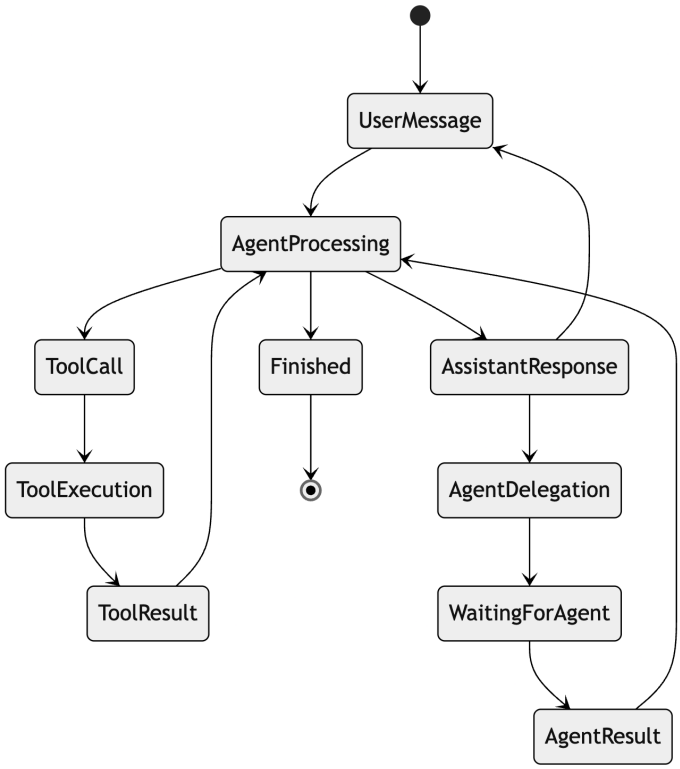


Figure 5.2.2: Finite State Machine Flow

The design rests on four interlocking ideas and is inspired fully by Erik's work (Erik 2025a; Erik 2025b; Erik 2025c; Erik 2025d). **Emergent Coordination** empowers the system to decide at runtime how many agents to deploy and how to assign tasks, moving beyond rigid sequential workflows to truly configurable behavior. **Finite State Machine Control** governs state transitions with explicit rules and handlers, ensuring deterministic progression while allowing dynamic routing based on agent outputs. **Conversation Branching** lets interaction histories fork at any point, supporting experimental workflows and comparative analyses of different coordination strategies. Finally, **Dynamic Agent Delegation** enables agents to spawn sub-agents and sub-conversations creating a hierarchical task decomposition that emerges from actual system needs rather than a predetermined plan.

5.2.2 Observability Layer

The observability layer seeks to provide complete transparency enabling real-time monitoring and retrospective analysis of all agent actions and system states.

The observability architecture ensures complete system transparency through the event flow shown in Figure 5.2.3.

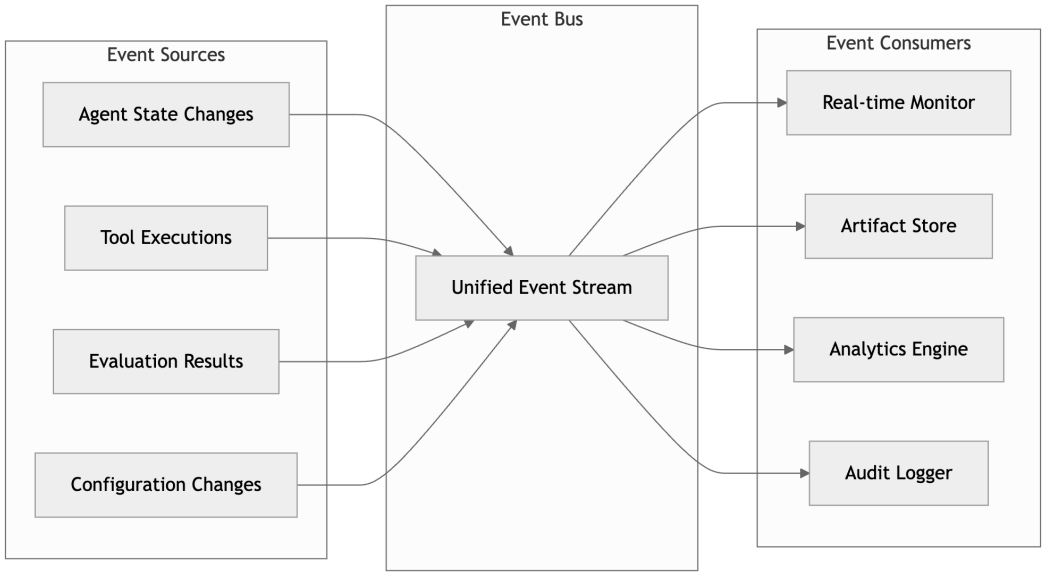


Figure 5.2.3: Observability over Event flow

The observability architecture has four core mechanisms. **Universal Event Streaming** ensures that every system activity flows through a central event stream, making agent decision-making and system-wide behavior patterns fully transparent in real time. **Complete Traceability** records both metadata and payload in each event, preserving causal links across all agent interactions for a holistic understanding. **Real-time Transparency** leverages live event streams to power operational monitoring and debugging interfaces, keeping the system observable during execution. Finally, **Event Logs** captures events exhaustively for post-hoc examination of agent behaviors and system performance.

5.2.3 Adaptation Layer

Adaptation is about converting the experience into improved performance through feedback loops operating at both collective system-wide and individual agent levels.

The adaptation layer implements closed-loop feedback mechanisms as illustrated in Figure 5.2.4.

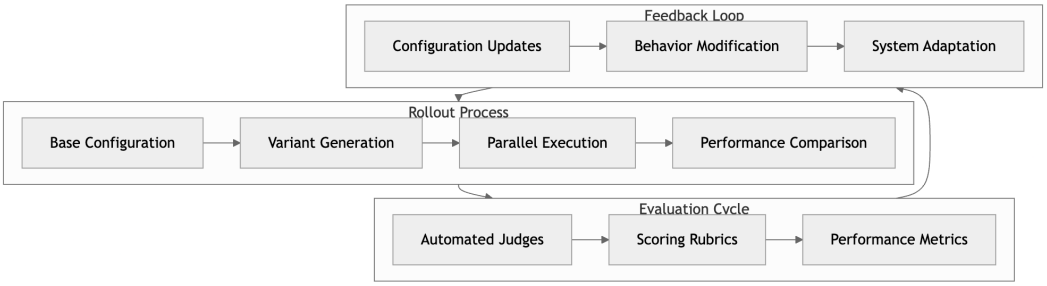


Figure 5.2.4: Adaptation Loop

Continuous improvement unfolds across multiple dimensions. **Multi-Level Learning** employs adaptation loops at both the agent and collective system-wide levels, driving iterative enhancements at every scale. **Evaluation Layer** integrates automated judge components that apply configurable rubrics to assess performance and generate structured feedback. **Rollouts** supports A/B testing via branching, thus to compare agent configurations and behavioral strategies methodically is simple. Finally, **Runtime Configuration** allows hierarchical adjustments on the fly, enabling the system to adapt and refine its behavior during execution.

5.2.4 Audit Layer

The final layer, the Audit Layer, leverages distributed-ledger technology to ensure immutable record-keeping, enabling both accountability and agent-level reward mechanisms, as well as alignment on a collective level.

The audit layer provides tamper-evident records and economic coordination through the distributed ledger architecture shown in Figure 5.2.5.

The audit layer encompasses four essential functions. **Immutable Audit Trails** leverages distributed-ledger technology to record all significant system op-

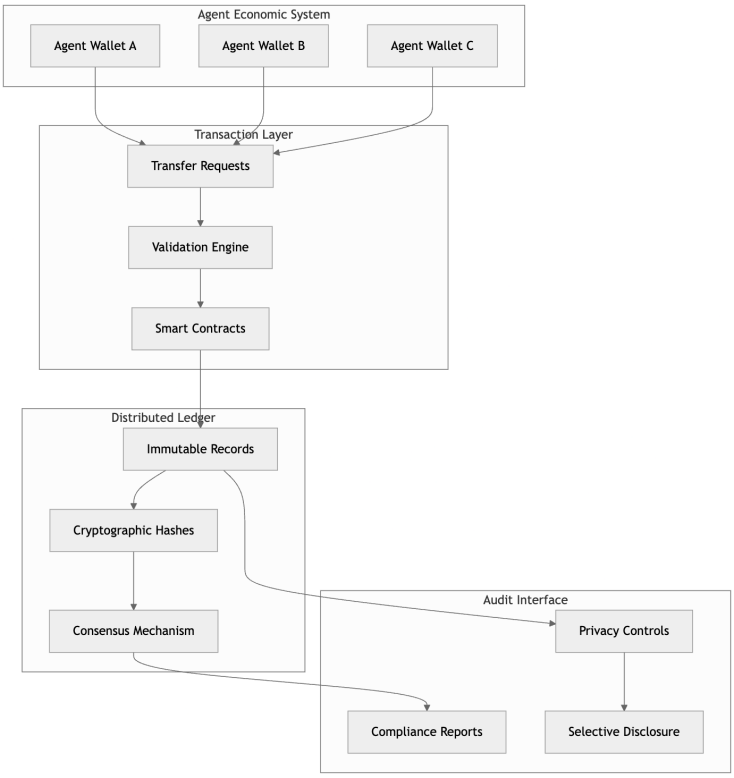


Figure 5.2.5: DLT Architecture

erations and agent actions in a tamper-evident ledger. **Monetary Incentive System** provisions individual agent wallets for performance-based reward distribution, creating an economic coordination mechanism for alignment. **Agent Accountability** arises from transparent economic transactions and performance records, incentivizing beneficial behaviors. Finally, **Privacy-Preserving Compliance** implements selective disclosure mechanisms to satisfy regulatory requirements while safeguarding sensitive operational data.

5.3 Component Interactions

5.3.1 Execution Flow

The system execution follows the sequence shown in Figure 5.3.1, demonstrating end-to-end integration across all architectural layers.

The execution unfolds through six integrated stages. **Agent Processing** consumes conversation states and produces responses or tool invocations. **Tool Execution** runs asynchronously, publishing results back into the conversation streams. **State Progression** captures every activity as state transitions flowing through the orchestration layer. **Event Publication** automatically pushes those state changes to the observability layer for real-time monitoring and analysis. **Eval-**

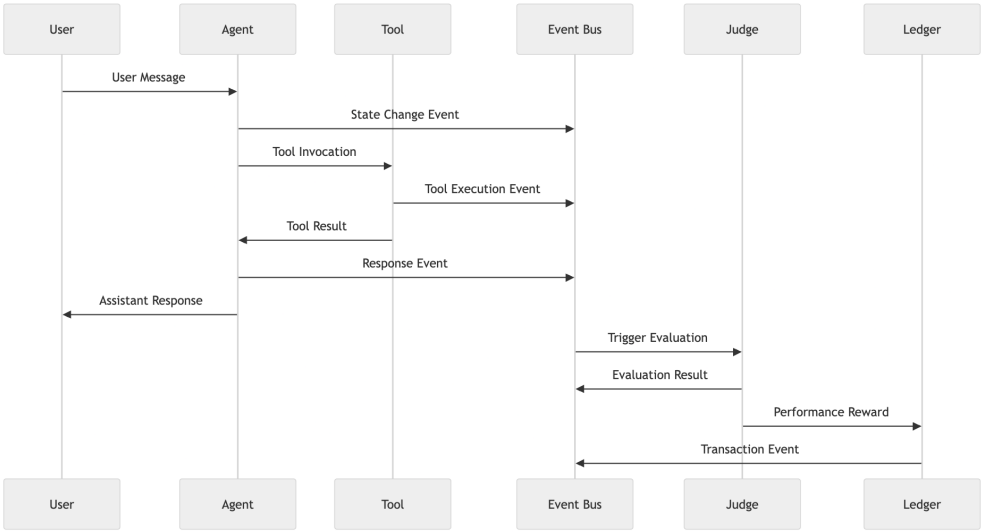


Figure 5.3.1: Execution Flow

uation Triggering fires upon conversation completion or significant milestones, invoking automated quality assessments. Finally, **Adaptation Feedback** uses those evaluation results to refine future agent configurations via the adaptation layer.

5.3.2 Data Flow Architecture

The information architecture supports the system through the data stores and processing components shown in Figure 5.3.2.

The data architecture comprises five core components. **Conversation State** persists interaction histories with branching support. **Event Streams** publish all system activities in real time with structured metadata. **Configuration Store** maintains hierarchical settings with runtime override capabilities. **Audit Trails** provide immutable logs of significant events with cryptographic integrity. Finally, **Evaluation Results** capture performance assessments linked to specific conversation instances.

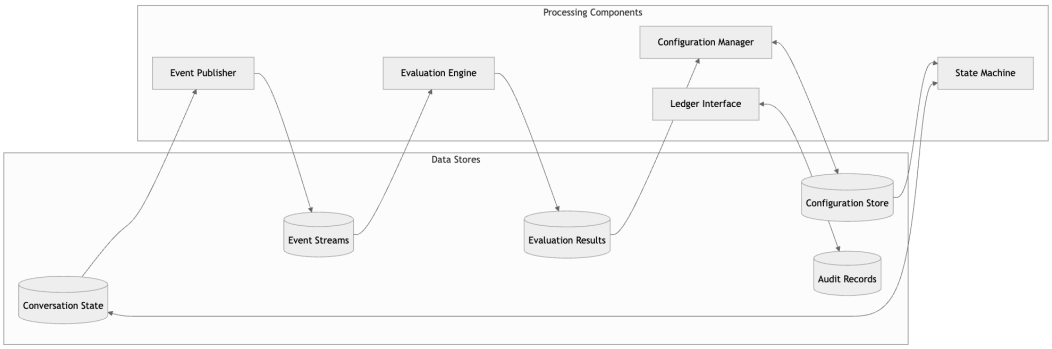


Figure 5.3.2: Flow of Data

5.4 Design Principles

The architectural design follows several key principles that ensure system coherence and extensibility, as summarized in Table 5.4.1:

5.5 Quality Attributes

The architecture addresses several critical quality attributes essential for autonomous multi-agent systems, as detailed in Table 5.5.1:

5.6 Architectural Validation

Building on the four foundational pillars introduced in Chapter 1 (Section 1.1), and the specific gaps distilled in Section 3.2, this design directly addresses the architectural deficiencies and looks to enable agentic multi-agent systems, as summarized in Table 5.6.1:

The architecture establishes an integrated framework where each layer provides essential capabilities for agentic operation while maintaining clear interfaces that enable independent development and evolution. This foundation is necessary for

Table 5.4.1: Design Principles and Implications

Principle	Implications
Separation of Concerns	Each layer addresses distinct autonomy requirements with minimal dependencies, enabling independent evolution and testing.
Event-Driven Architecture	Components communicate through asynchronous events rather than direct coupling, supporting scalability and comprehensive observability.
Complete Traceability	All system activities are captured in event streams and conversation logs, enabling comprehensive debugging and analysis of agent interactions.
Plugin Extensibility	Agent types, tools, and evaluation methods integrate through well-defined interfaces, supporting experimentation and future enhancement.
Configuration-Driven Behavior	System adaptation occurs through declarative configuration changes rather than code modification, enabling automated optimization.

Table 5.5.1: Quality Attributes and Architectural Support

Quality Attribute	Architectural Support
Reliability	Fault tolerance through distributed processing and persistent state management ensures system availability despite component failures.
Scalability	Horizontal scaling through worker queue distribution and stateless component design supports growing agent populations and conversation volumes.
Auditability	Complete event capture with immutable storage enables regulatory compliance and forensic analysis of autonomous system behavior.
Extensibility	Plugin architecture and configuration-driven behavior support research experimentation and operational adaptation without architectural changes.
Observability	Comprehensive event streaming and real-time monitoring provide complete system visibility for both operational management and research analysis.

Table 5.6.1: Research Gap and Architectural Design

Architectural Deficiency	**Conceptual Solution**
Rigid Orchestration	FSM-based coordination where the system itself decides task distribution and agent deployment, enabling emergent behavior
Fragmented Observability	Universal event streaming with complete transparency, making all agent actions and system states observable in real-time
Missing Adaptation	Integrated evaluation infrastructure with rollout capabilities provides foundation for systematic adaptation methods and continuous improvement
Absent Auditability	DLT integration with monetary incentives provides immutable records and economic coordination for agent alignment

multi-agent systems that seeks to exhibit emergent, intelligent behavior while remaining observable, adaptive, and accountable for humans.

Chapter 6 presents the concrete implementation of this conceptual design through P2Engine, detailing specific technology choices and implementation patterns that realize these design principles in a working MAS framework who seeks to be agentic.

CHAPTER

SIX

IMPLEMENTATION

6.1 P2Engine

This chapter explores P2Engine, a multi-agent system (MAS) framework that couples the emergent behavior of agents, finite state machine orchestration and distributed-ledger technology (DLT) to enable verifiable audit trails and possibilities for adaptation. The complete source code is available in the project's GitHub repository[1] (see Appendix A for details). Video demonstrations of the P2Engine framework can be found in the `/demos` directory of that repository. In addition to the core foundational pillar demos (see Chapter 7), the `/demos` folder includes plenty demonstration videos showcasing various functionalities of P2Engine. For a quick visual overview of P2Engine in action, please refer to the gallery of screenshots in Appendix B.

This chapter is organized as follows. section 6.2 offers a Quick-Start Guide; section 6.3 examines the anatomy of P2Engine revealing the core workings of the framework and some of its vital components; section 6.4 demonstrates how the framework embodies the four foundational pillars.

The GitHub repository provides a complete and standalone introduction to the framework. The primary entry point is `p2engine/README.md`, which offers setup instructions and full overview of the framework. In the repository, each major core module (e.g., `agents`, `orchestrator`, `runtime`) will include its own top-level documentation to guide fully.

[1] `https://github.com/NTNU-IDI/multi-agent-system-master-thesis-2025-adam`

6.2 Quick-Start Guide

A complete getting-started guide is available in the repository at `p2engine/README.md`. This section provides a brief setup guide to help you begin working with the framework.

Prerequisites

P2Engine requires several external dependencies and services to function properly. The following components must be installed before proceeding:

Table 6.2.1: Required dependencies for P2Engine

Component	Version	Purpose	Installation / Check
Python	3.9+	Runtime environment	`python -version`
Poetry	Latest	Dependency management	Poetry docs
Redis	Latest	Message broker & cache	`redis-server` (package mgr.)
Daml SDK	Latest	Distributed ledger (optional)	Daml install
OpenAI API Key	—	LLM provider access	OpenAI keys

Installation & Setup

The installation process involves cloning the repository, installing Python dependencies, and configuring the environment:

Listing 6.1: Complete P2Engine setup and launch

```
# 1. Clone and install
git clone https://github.com/NTNU-IDI/multi-agent-system-master-thesis-2025-adam.git
cd multi-agent-system-master-thesis-2025-adam
cd p2engine
poetry install



# 2. Configure environment
cp .env.example .env
# Edit .env and add: OPENAI_API_KEY="sk-your-api-key-here"

# 3. Launch P2Engine
# Option A: Enter the virtual environment
poetry shell
./scripts/run_project.sh

# OR Option B: Use 'poetry run' directly without activating the shell
# poetry run ./scripts/run_project.sh
```

Startup Flow

Running `./scripts/run_project.sh` performs every required step in one pass: it loads your `.env`, (optionally) starts the (Canton Network)/DAML ledger, clears and starts Redis, spins up Celery workers for four task queues, boots the runtime engine, and finally drops you into the interactive CLI. Figure 6.2.1 shows that sequence.

Basic Usage

Once P2Engine is running, you can interact with agents and explore the system. P2Engine provides several interactions via commands through its CLI interface. The primary usage patterns involve agent conversations, ledger operations, and rollouts:

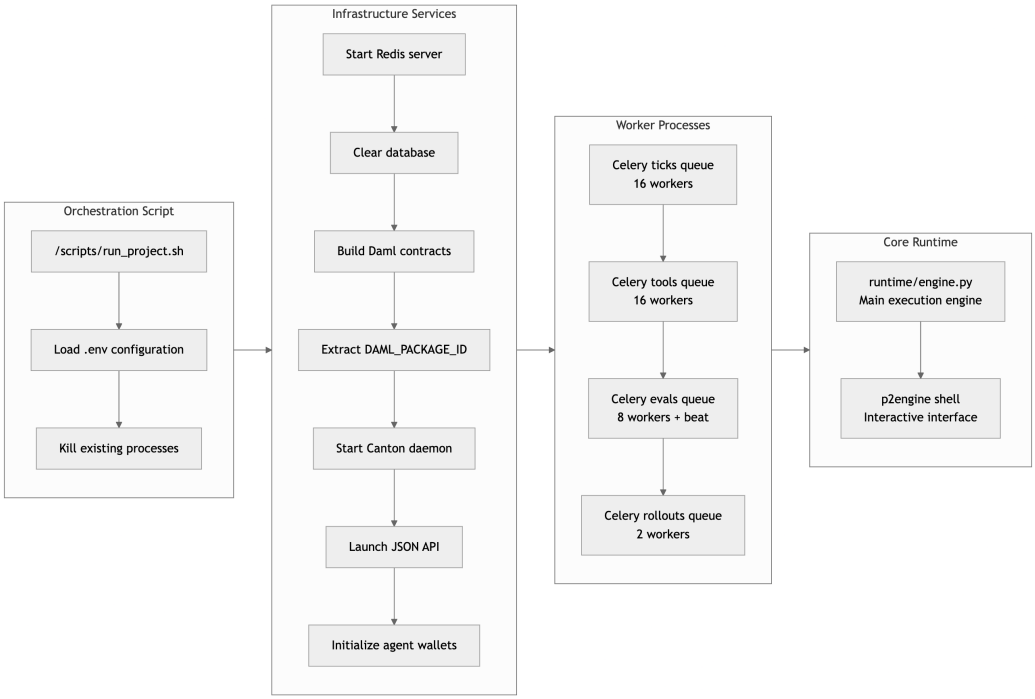


Figure 6.2.1: P2Engine Startup `run_project.sh`

Listing 6.2: Common Usage with P2Engine

```
# Start a conversation with agent
p2engine chat with agent_alpha
You> Hello! What can you do?
Agent> I can help with weather, calculations, and manage
   funds...

# Tool usage with  agent
You> What is the weather in Paris?
Agent> [Uses get_weather tool] The weather in Paris is

# Financial operations with agent
You> Transfer 25 to agent_beta for helping with analysis
Agent> [Uses transfer_funds tool] Successfully transferred
   ...

# Delegate tasks to another agent
You> Delegate to agent_helper: What is the weather in
   Tokyo? If they did well, reward them 20 units
Agent> I asked weather agent what is the weather in Tokyo.
    He said 17 degrees.

# Start rollout from configuration file
p2engine rollout start config/rollout_joke.yml
```

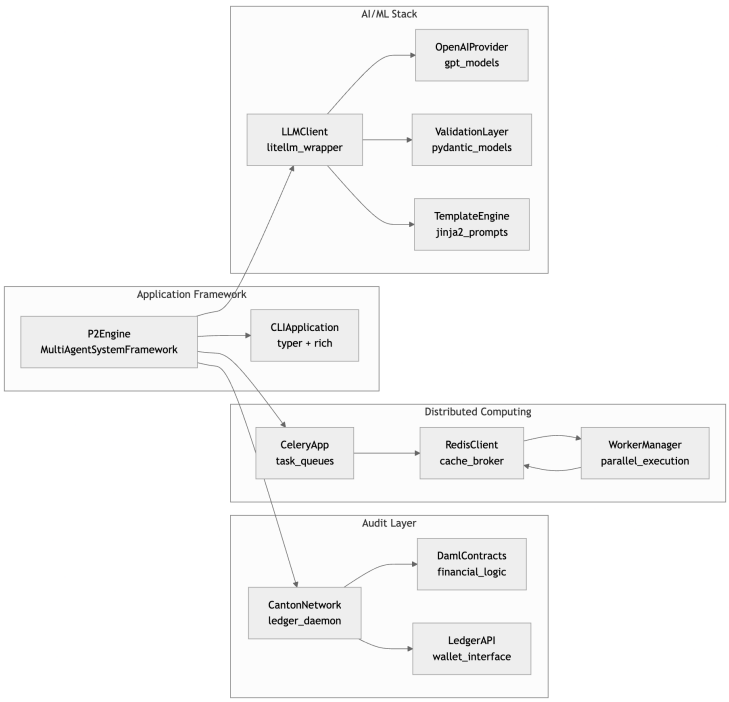


Figure 6.2.2: P2Engine Tech Stack

6.2.1 Technology Stack

P2Engine follows a bare-bones by default, only use dependencies when needed philosophy. Wherever possible, the framework stays independent of heavyweight libraries; where essential, it integrates a carefully chosen set of technologies that compliments the four foundational pillars orchestration, observability, adaptation and auditability.

The following outlines the technologies used and their respective purposes:

AI / ML layer: A thin abstraction over large-language-model providers is handled by `LiteLLM` (`infra/llm_client.py`).

Runtime: Workloads are distributed through `Celery` task queues (`runtime/tasks/`) with `Redis` (`infra/redis_client.py`) serving as both broker and cache. Together they offer a mature, straightforward, and high-throughput solution.

Audit layer: Auditability is ensured by a `Canton Network with DAML` (`services/canton_ledger_service.py`).

Application: Interaction is provided via a `Typer` + `Rich` command-line interface (`cli/`), while `Pydantic` models deliver runtime type safety throughout the codebase.

Table 6.2.2: Key Technologies in P2Engine.

Category	Technology	Purpose	Implementation
AI / ML	LiteLLM	Unified interface to multiple LLM providers and models	`infra/llm_client.py`
Task Queue	Celery	Distributed task orchestration across workers with retry semantics	`runtime/tasks/`
Caching	Redis	Message broker and in-memory cache supporting pub/sub and persistence	`infra/redis_client.py`
Ledger	Canton Network	DLT for auditability	`services/canton_ledger_service.py`
CLI	Typer + Rich	Developer-friendly command-line interface with rich formatting	`cli/`
Validation	Pydantic	Runtime data modelling and type validation throughout the codebase	Throughout codebase

6.3 Anatomy of P2Engine

This section dissects P2Engine's system anatomy, the mechanics, its internal structures and its pathways. It will highlight the framework's vital components, its key mechanics and layers, with the goal of building a clear mental model of P2Engine.

6.3.1 Agent Lifecycle

P2Engine implements a comprehensive agent lifecycle management system that handles agent creation, registration, activation, and termination. The lifecycle management ensures that agents are properly initialized with their required dependencies and maintained throughout their operational period.

Agent Registration and Discovery

The agent system uses a plugin-based architecture that supports both built-in and external agent types. Agent registration occurs through the `AgentRegistry` in `orchestrator/registries.py`, which maintains both in-memory and Redis-backed persistence of agent configurations.

Listing 6.3: Agent Factory

```
# Agent creation through factory pattern
class AgentFactory:
    def create(self, cfg: AgentConfig):
        if cfg.type == "llm":
            return LLMAgent(
                agent_id=cfg.id,
                model=cfg.llm_model,
                tool_names=cfg.tools,
                behavior_template=cfg.behavior_template,
                llm_client=self.llm_client,
                tool_registry=self.tool_registry
            )
        elif cfg.type == "rule_based":
            return RuleBasedAgent(rules=cfg.rules)
        # Additional agent types...
```

Session Management

Agent sessions are managed through the `Session` class in `infra/session.py`, which coordinates agent registration within conversation contexts. The session management system ensures that agents are properly activated when needed and cleaned up when conversations conclude.

Lifecycle Controls

The following lifecycle controls govern agent behavior during a session:

Lazy initialization Agents are created on-demand when first referenced in conversations.

Configuration override Runtime behavior modification through Redis-backed configuration patches.

Graceful termination Proper cleanup of agent resources and conversation state upon completion.

6.3.2 Finite State Machine Orchestration

P2Engine's orchestration is fundamentally built on finite state machine (FSM) principles, where each agent conversation progresses through well-defined states with explicit transition rules (Erik 2025a). This approach provides predictable behavior while enabling complex multi-agent interactions.

State Types and Transitions

The system defines several core state types in `orchestrator/interactions/states/`, each representing a specific phase of agent–user interaction:

UserMessageState Incoming user input that triggers agent processing.

AssistantMessageState Agent responses, either text or tool invocations.

ToolCallState Active tool execution with correlation tracking.

ToolResultState Tool completion results with success/failure status.

WaitingState Temporary states during asynchronous operations.

FinishedState Terminal state indicating conversation completion.

State Processing

State transitions are managed through a handler registry in `runtime/handlers.py`, where each state type has a dedicated handler function:

This handler-based approach ensures each state transition is explicit and deterministic, while the effect system enables side-effects and cross-agent communication patterns.

6.3.3 Conversation State Management

One of P2Engine's distinguishing features is its sophisticated conversation state management system (Erik 2025b) that supports branching, forking, and parallel conversation paths. This capability enables complex multi-agent interactions and experimental workflows.

Interaction Stack Architecture

The core conversation state is managed through the `InteractionStack` class in `orchestrator/interactions/stack.py`. Each conversation maintains a stack of interaction states that can be manipulated, branched, and restored.

Branching and Forking Operations

The system supports several key operations for conversation management:

Fork Create a new branch from any point in the conversation history.

Listing 6.4: State Handler

```
# Handler registration pattern
@handler(UserMessageState)
def handle_user_message(entry: StackEntry, stack:
    InteractionStack,
                        agent: IAgent, conversation_id: str,
                           agent_id: str):
    # Convert conversation history to LLM format
    ask = AskSchema(history=render_for_llm(stack),
        conversation_id=conversation_id)

    # Execute agent reasoning
    response = run_async(agent.run(ask))

    # Generate effects based on response type
    return materialise_response(stack, response,
        conversation_id, agent_id)

@handler(ToolResultState)
def handle_tool_result(entry: StackEntry, stack:
    InteractionStack,
                       agent: IAgent, conversation_id: str,
                          agent_id: str):
    # Continue conversation after tool completion
    ask = AskSchema(history=render_for_llm(stack),
        conversation_id=conversation_id)
    response = run_async(agent.run(ask))
    return materialise_response(stack, response,
        conversation_id, agent_id)
```

Listing 6.5: Branching

```
# Branch creation and management
def fork(stack: InteractionStack, idx: int) -> str:
    """Create new branch from specified conversation index"""
    new_branch_id = uuid.uuid4().hex[:8]
    interactions = stack.redis.lrange(stack.current_key, 0, idx
        )

    if interactions:
        stack.redis.rpush(stack._branch_key(new_branch_id), *
            interactions)

    stack.checkout(new_branch_id)
    return new_branch_id

def checkout(stack: InteractionStack, branch_id: str) -> None:
    """Switch to specified conversation branch"""
    if not stack.redis.exists(stack._branch_key(branch_id)):
        raise ValueError(f"Branch {branch_id} does not exist")

    stack.redis.set(stack._current_ptr_key(), branch_id)
```

Checkout Switch between different conversation branches.

Rewind Return to a previous state in the conversation history.

Merge Combine insights from multiple conversation branches (future work).

This branching capability is essential for the rollout system, enabling parallel experimentation with different agent configurations while maintaining conversation context.

6.3.4 Tool System Architecture

P2Engine implements a flexible and extensible tool system that enables agents to interact with external services, perform computations, and execute side effects. The tool architecture supports both built-in tools and dynamic plugin discovery.

Tool Registration and Discovery

Tools are automatically discovered through Python decorators and registered in the global `ToolRegistry`. The registration system supports both simple function-based tools and complex tools with input validation, caching, and post-effects.

Listing 6.6: Tool Registration Decorators

```
@function_tool(
    name="get_weather",
    description="Get current weather information for a location
        ",
    input_schema=WeatherInput,
    cache_ttl=300,  # Cache results for 5 minutes
    side_effect_free=True
)
def get_weather(location: str, unit: str = "fahrenheit") ->
    dict:
    # Tool implementation
    return {
        "status": "success",
        "data": {"location": location, "temperature": 72, "unit
            ": unit}
    }
```

Tool Execution Pipeline

Tool execution follows a standardized pipeline that includes validation, execution, caching, and post-effect processing. The pipeline ensures consistent behavior across all tools while supporting advanced features like deduplication and audit logging.

The execution pipeline includes several key stages:

Input validation Pydantic schema validation ensures type safety.

Deduplication check Prevents redundant tool calls based on configurable policies.

Cache lookup Returns cached results for side-effect-free tools when available.

Tool execution Actual tool logic execution with error handling.

Result processing Output validation and formatting.

Post-effects Execution of registered post-effect handlers.

Audit logging Recording of tool execution in the artifact bus.

6.3.5 Error Handling and Resilience

P2Engine implements comprehensive error handling and resilience mechanisms to ensure system stability and graceful degradation under failure conditions. The error handling strategy operates at multiple levels, from individual tool failures to system-wide resilience patterns.

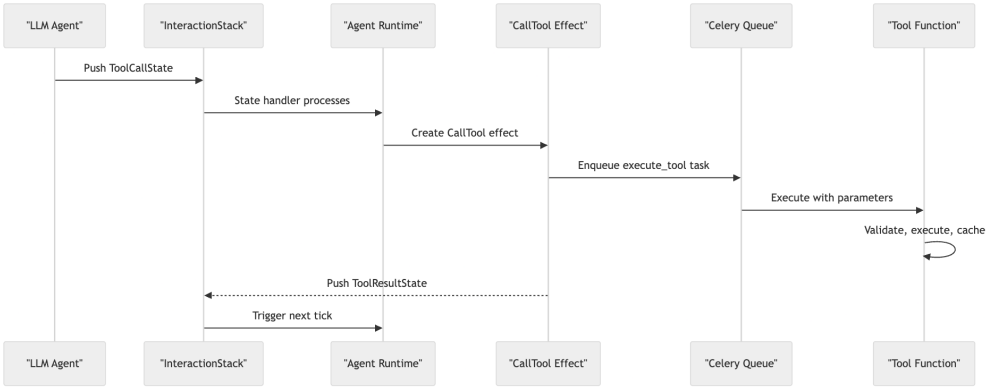


Figure 6.3.1: Tool Execution Pipeline

Multi-Level Error Handling

The system implements error handling at several architectural levels:

Tool level Individual tool failures are caught and transformed into structured error responses.

Agent level Agent execution errors trigger fallback behaviors and timeout handling.

Session level Conversation-level errors are managed through state recovery mechanisms.

System level Infrastructure failures trigger circuit breakers and graceful degradation.

Timeout and Retry Mechanisms

The system implements sophisticated timeout and retry mechanisms throughout the execution pipeline:

Listing 6.7: Timeout Handling

```
@handler(WaitingState)
def handle_waiting(entry: StackEntry, stack: InteractionStack,
                   agent: IAgent, conversation_id: str, agent_id
                      : str):
    state: WaitingState = entry.state

    if not state.is_expired():
        return []

    # Handle different timeout scenarios
    if state.kind == "tool":
        timeout_result = ToolResultState(
            tool_call_id=state.correlation_id,
            tool_name="unknown",
            result={"status": "timeout", "message": "Tool call
               exceeded timeout"}
        )
    elif state.kind == "agent":
        timeout_result = AgentResultState(
            correlation_id=state.correlation_id,
            result={"status": "timeout", "message": "Agent
               response timeout"}
        )

    stack.pop()
    stack.push(timeout_result)
    return [PublishSystemReply(conversation_id, "Operation
       timed out")]
```

Circuit Breaker Patterns

For external service integration, P2Engine implements circuit breaker patterns that prevent cascading failures and enable graceful service degradation when external dependencies become unavailable.

6.3.6 Performance and Scalability

P2Engine is designed with performance and scalability in mind, utilizing distributed task queues, intelligent caching, and efficient state management to support concurrent agent operations and large-scale rollouts.

Distributed Task Architecture

The system leverages Celery's distributed task queue system with specialized queues for different operation types:

Ticks queue High-priority agent state transitions and conversation progression.

Tools queue Tool execution with configurable concurrency levels.

Evaluations queue Automated quality assessment and scoring.

Rollouts queue Large-scale experimental execution.

Caching and Performance Optimization

Multiple caching layers optimize system performance, for example in-memory lookup, Redis caching, and filesystem persistence.

Scalability Patterns

The architecture supports horizontal scaling through several design patterns:

Stateless agents Agent instances can be created and destroyed dynamically.

Queue-based communication Loose coupling enables independent scaling of components.

Redis clustering State persistence can be distributed across Redis cluster nodes.

Rollout parallelization Experimental variants execute independently for maximum throughput.

Listing 6.8: Agent Configuration

```yaml
# agents.yml configuration
agents:
  - id: agent_alpha
    type: llm
    llm_model: "openai/gpt-4o"
    behavior_template: "weather_expert"
    tools:
      - "get_weather"
      - "delegate"
      - "transfer_funds"

  - id: treasurer
    type: llm
    llm_model: "openai/gpt-4o"
    tools:
      - "check_balance"
      - "transfer_funds"
      - "transaction_history"
```

6.3.7 Configuration Management

P2Engine implements a flexible configuration system that supports both static configuration files and dynamic runtime modifications. The configuration architecture enables easy customization while maintaining system integrity and type safety.

Hierarchical Configuration

The system uses a hierarchical configuration approach with multiple sources and precedence levels:

Default values Built-in defaults in code.

Configuration files YAML and JSON configuration files.

Environment variables Runtime environment overrides.

Dynamic overrides Redis-based runtime configuration changes.

Runtime Configuration Overrides

The system supports dynamic configuration changes through Redis-backed overrides, enabling real-time behavior modification without system restarts:

This dynamic configuration capability is essential for the adaptation pillar, enabling the system to modify agent behavior based on evaluation results and performance metrics.

Listing 6.9: Override Commands

```
# Change agent behavior template
p2engine config set-behavior conversation_1 weather_expert

# Modify tool availability
p2engine config set-tools conversation_1 "get_weather,delegate,
    reward_agent"

# Override LLM parameters
p2engine config set-override conversation_1 agent_alpha '{"
    temperature": 0.7}'
```

6.3.8 Artifact Bus

The Artifact Bus serves as the central event store for all P2Engine operations, providing searchable, persistent storage of conversation states, tool results, evaluations, and metrics:

The Artifact Bus provides comprehensive event storage and real-time observability across all system operations. As the central nervous system for P2Engine's monitoring capabilities, it captures and correlates events from every component, enabling both real-time monitoring and detailed post-hoc analysis.

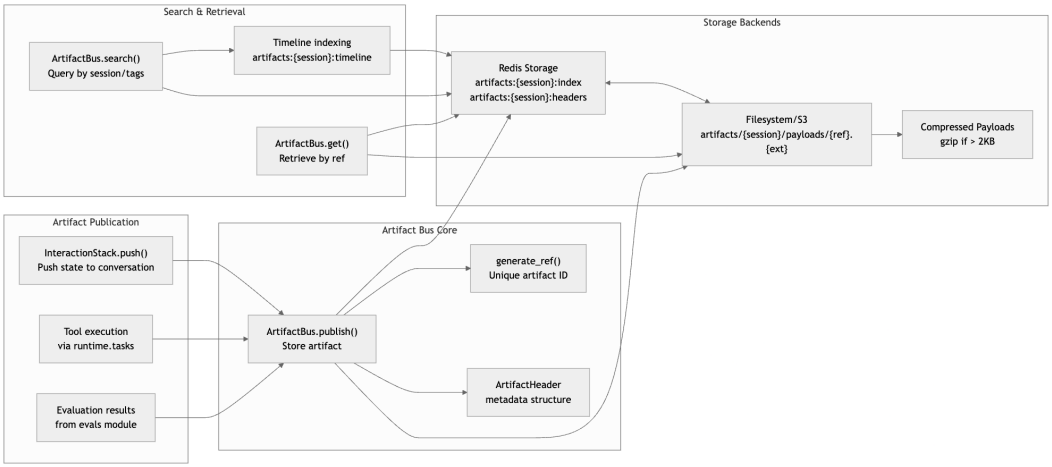


Figure 6.3.2: Artifact Bus

The infrastructure's artifact bus provides system-wide event storage through several key mechanisms:

Universal State Tracking All state changes are published as artifacts, creating a complete audit trail of system behavior

Cross-Module Correlation Events are linked via `session_id`, `agent_id`, and `branch_id`, enabling comprehensive tracing across distributed components

Metrics Collection Integrated evaluation and metrics collection support performance monitoring and system optimization

This architecture ensures that every significant operation within P2Engine is observable, traceable, and analyzable, supporting both operational monitoring and continuous improvement initiatives.

6.3.9 Post-Effects System

Post-effects are hooks that execute immediately after a tool completes successfully, enabling additional behaviours such as agent delegation, reward distribution, or follow-up evaluations. Figure 6.3.3 provides an illustrative overview of the flow.

Registration Functions decorated with `@register_post_effect` are auto-discovered at startup and stored in the `runtime/post_effects.py` registry.

Execution context Each hook receives the tool result, conversation metadata, and a Redis handle, allowing stateful side-effects.

Typical actions Triggering another tool, scheduling a Celery task, updating ledger balances, or writing configuration overrides for adaptation.

6.3.10 Rollouts

rollouts sets the stage for adaptation methods.

The rollouts provides A/B testing capabilities for systematically comparing different agent configurations, tool combinations, and behavioral parameters. Implemented in the `runtime/rollout/` module, it integrates seamlessly with the Celery task system for distributed execution, enabling parallel evaluation of multiple configuration variants.

Rollout Specification Format

Listing 6.10: YAML Rollout Specification

```yaml
# Example rollout configuration
teams:
  weather_analysis:
    initial_message: "What's the weather in Paris and provide
       travel advice?"
    base:
      agent_id: agent_alpha
      model: openai/gpt-4o
      tools: ["get_weather", "delegate"]
    variants:
      - temperature: 0.3
        tools: ["get_weather", "delegate", "reward_agent"]
        reasoning_policy: "mcts_v1"
      - temperature: 0.7
        tools: ["get_weather", "delegate"]
        reasoning_policy: "greedy"
    eval:
      evaluator_id: gpt4_judge
      metric: score
      rubric: |
        Rate the response quality (0-1) based on:
        - Accuracy of weather information (0.4)
        - Helpfulness of travel advice (0.4)
        - Clarity and organization (0.2)
```

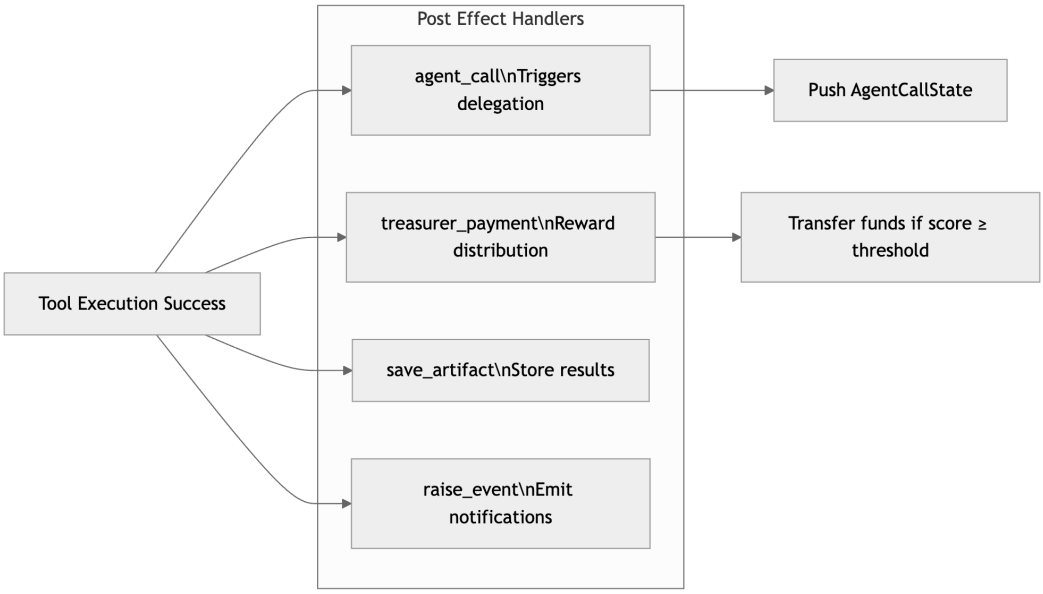


Figure 6.3.3: Post-effects Architecture

The system uses YAML-based configuration files to define rollout experiments, specifying teams, base settings, variants, and evaluation criteria. See Listing 6.10 for an example format:

Core Rollout Components

The rollout system consists of several key components that work together to enable systematic experimentation. The `RolloutEngine` in `runtime/rollout/engine.py` orchestrates the entire rollout execution process, managing variant creation and result aggregation. The `VariantExpander` in `runtime/rollout/expander.py` takes base configurations and generates all specified variant combinations for testing. The `RolloutSpec` module in `runtime/rollout/spec.py` handles specification parsing and validation, ensuring rollout configurations are well-formed. Finally, the `RolloutStore` in `runtime/rollout/store.py` manages state persistence and metrics storage throughout the rollout lifecycle.

Rollout Execution Flow

The rollout system executes through a well-defined sequence of steps, leveraging the Celery task queue system with the dedicated `rollouts` queue for parallel variant execution. The process begins with specification loading, where the system parses YAML rollout configuration via `RolloutSpec`. Next, variant generation occurs as the `VariantExpander` creates all configuration combinations based on the specification. During parallel execution, each variant runs in separate Celery

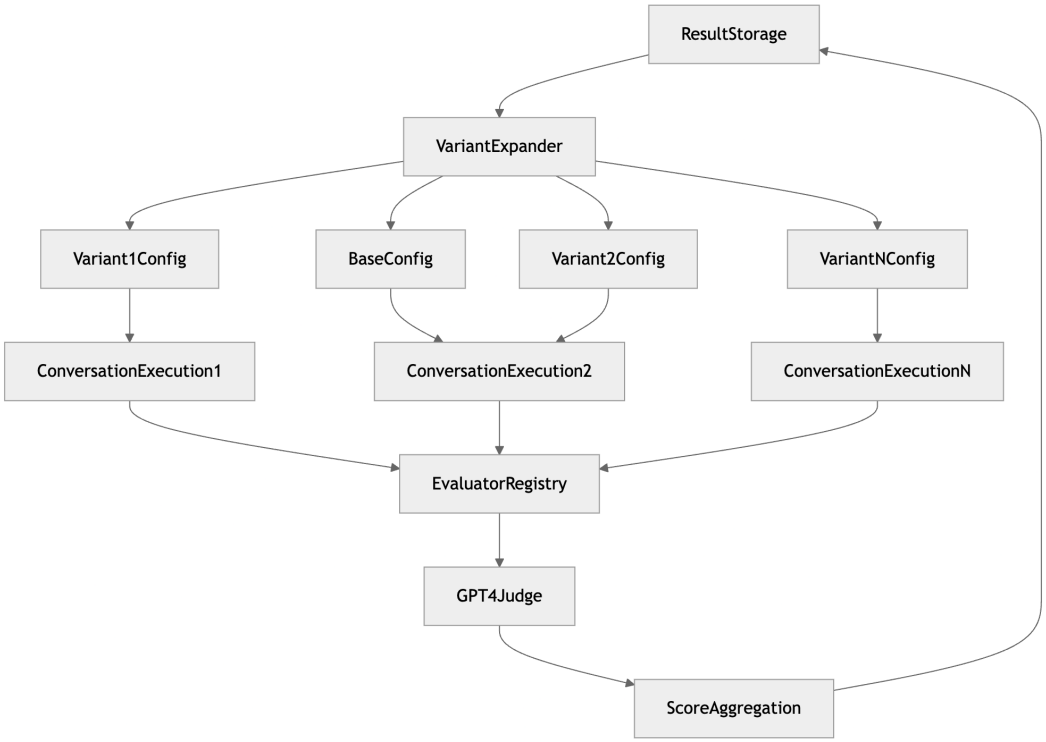


Figure 6.3.4: Variant Expansion

workers to maximize throughput. The system manages conversations by creating isolated conversation instances per variant, ensuring clean separation between experiments. Upon completion, evaluation triggering automatically initiates quality assessment for completed conversations. Finally, result aggregation collects metrics and scores across all variants, providing comprehensive performance comparisons.

6.3.11 Audit Layer

The audit layer provides distributed financial transaction capabilities for P2Engine's multi-agent system through Canton Network and DAML DLT infrastructure (*The Canton Network* 2025; *Daml: A Smart Contract Language for Distributed Applications* 2025). This integration enables agents to maintain individual wallets, execute secure transfers, and maintain complete audit trails of all financial operations within the framework. The ledger system serves as both a financial backbone and an accountability mechanism, ensuring that all agent actions involving value transfer are permanently recorded and verifiable.

Purpose and Design Principles

The Audit Layer has several key functions. Agent wallet management provides individual financial accounts for each agent, enabling economic interactions and incentive mechanisms. Secure transactions ensure immutable transfers between agents with full audit trails, preventing tampering or repudiation. The system

supports financial incentives for payment-based agent coordination and reward distribution. Accountability is achieved through tamper-evident records of all financial operations, meeting both regulatory and system integrity requirements. The integration design ensures seamless connection with agent tools and CLI operations, making financial operations a natural part of agent behavior.

Ledger Architecture

The Audit Layer is built on a robust ledger architecture that provides high integrity and traceability for all agent-related transactions. The architecture leverages Canton Network's privacy-preserving blockchain technology (*The Canton Network* 2025) combined with DAML's (*Daml: A Smart Contract Language for Distributed Applications* 2025) smart contract capabilities to create a sophisticated financial infrastructure.

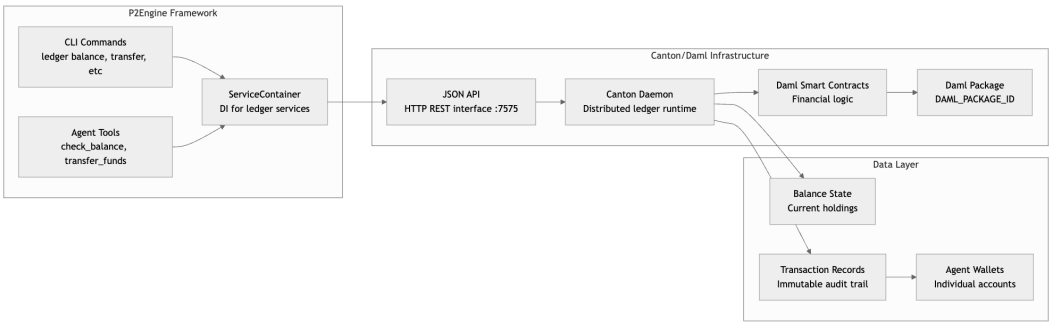


Figure 6.3.5: Autdit Layer

The integration flow connects P2Engine's agent system with the distributed ledger through multiple abstraction layers. This design ensures that agents can perform financial operations without needing to understand the underlying blockchain complexity.

Transaction Processing

The transaction system supports both direct transfers initiated through CLI commands and autonomous transfers initiated by agents during their operations. This flexibility enables both administrative control and agent autonomy in financial operations.

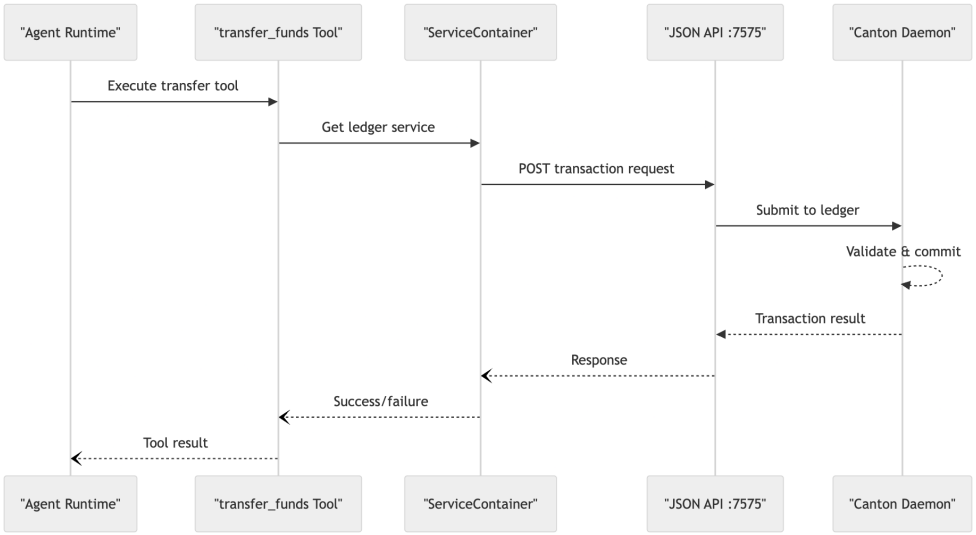


Figure 6.3.6: Ledger Flow

Listing 6.11: Direct Ledger Transfer

```
p2engine ledger transfer agent_alpha agent_beta 25.0 --reason "
    Payment for services"
```

Direct Transfer Operations

Direct transfers can be initiated through the CLI for administrative purposes or testing. The transfer command includes the source agent, destination agent, amount, and an optional reason for audit purposes. See Listing 6.11.

Agents can also initiate transfers through natural language during conversations. When chatting with an agent, financial operations can be requested conversationally, as shown in Listing 6.12.

The agent interprets the request and executes the transfer using its available tools.

Listing 6.12: Conversational Transfer

```
p2engine chat with agent_alpha
You> Transfer 50 to treasurer for management fees
```

Agent Tool Integration

The ledger provides specific tools that agents can use autonomously during their operations. The `check_balance` tool allows agents to query their current wallet balance, enabling financial decision-making. The `transfer_funds` tool enables agents to send payments to other agents, supporting economic interactions. The `transaction_history` tool provides access to the audit trail, allowing agents to review past transactions for reconciliation or analysis.

Transaction Flow Architecture

The transaction flow ensures that all financial operations follow a consistent pattern with proper validation, execution, and recording. Figure 6.3.7 illustrates the complete flow from initiation to ledger confirmation.

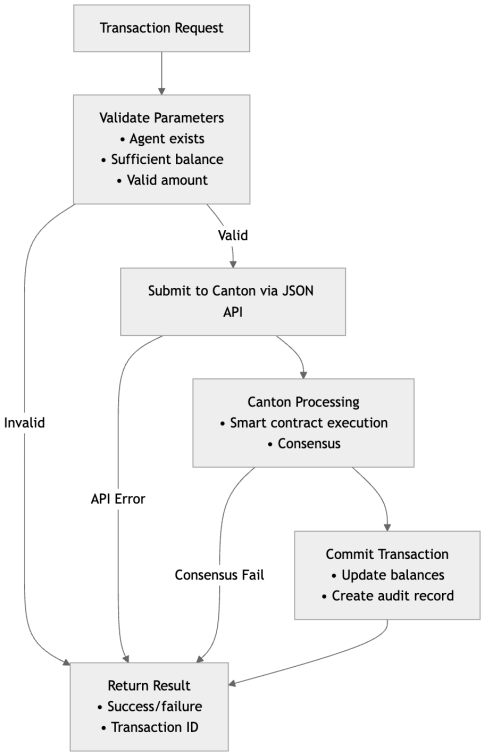


Figure 6.3.7: Transaction Flow

6.4 Realising the Four Foundational Pillars

P2Engine is built upon the four foundational pillars that work together to build the desired MAS.

6.4.1 Orchestration Pillar

The orchestration pillar implements FSM orchestration that enables precise, observable agent workflows through finite-state machine management. This pillar handles the complexities of conversation flow, state transitions, and multi-agent coordination patterns. The orchestration mechanisms ensure that agent interactions follow well-defined protocols while maintaining flexibility for emergent behavior.

Orchestration Implemented in `orchestrator/` via `InteractionStack`, `FSMOrchestrator`, and the state-machine core.

Agent types LLM, rule-based, and human-in-loop agents defined in `agents/`, all wired through the `ToolRegistry`.

Dynamic delegation Agents can delegate tasks to peers using the `delegate` tool, enabling hierarchical task distribution.

6.4.2 Observability Pillar

The observability pillar delivers real-time monitoring, comprehensive tracing, and debugging capabilities across all system components. By providing full visibility into agent behaviors, tool executions, and system-state changes, this pillar enables both operational monitoring and post-hoc analysis. The observability infrastructure captures detailed execution traces that support debugging, performance optimization, and behavioral analysis.

ArtifactBus Event pub/sub backbone implemented in `infra/ArtifactBus` for log and metric shipment.

Celery-based architecture Four specialised queues (`ticks`, `tools`, `evals`, `rollouts`) expose system activity to tracing back-ends.

Rich telemetry The CLI uses `Rich` for coloured logs and live dashboards.

6.4.3 Adaptation Pillar

The adaptation pillar provides infrastructure for continuous improvement through automated evaluation systems and rollout capabilities that enable systematic adaptation methods.

Rollout engine Systematic A/B testing managed by `runtime/RolloutExecutor`.

Judge-based evaluation Automated quality assessment using LLM-powered judges in `evaluation/`.

MCTS/RL infra nfrastructure supporting Monte-Carlo tree search and reinforcement-learning integration via the `runtime/` effect system, providing foundation for future adaptation methods.

6.4.4 Auditability Pillar

The auditability pillar integrates distributed-ledger technology to provide immutable audit trails, privacy-aware operations, and verifiable accountability for all agent actions and financial transactions. This pillar ensures that the system maintains a permanent, tamper-evident record of all significant operations, supporting both regulatory compliance and its own system verification.

Canton Network/DAML integration Ledger operations implemented in `services/canton_ledger_service.py`.

Financial accountability Agent wallets, transfers, and transaction history logged on-chain.

Complete audit trails Every interaction and side-effect routed through ledger hooks in `infra/ledger_hooks.py`.

The integration of these foundational pillars creates a unified architecture that supports complex and agentic MAS scenarios while also maintaining system integrity and performance. Figure 6.4.1 illustrates the four pillars and how they emerge in P2Engine.

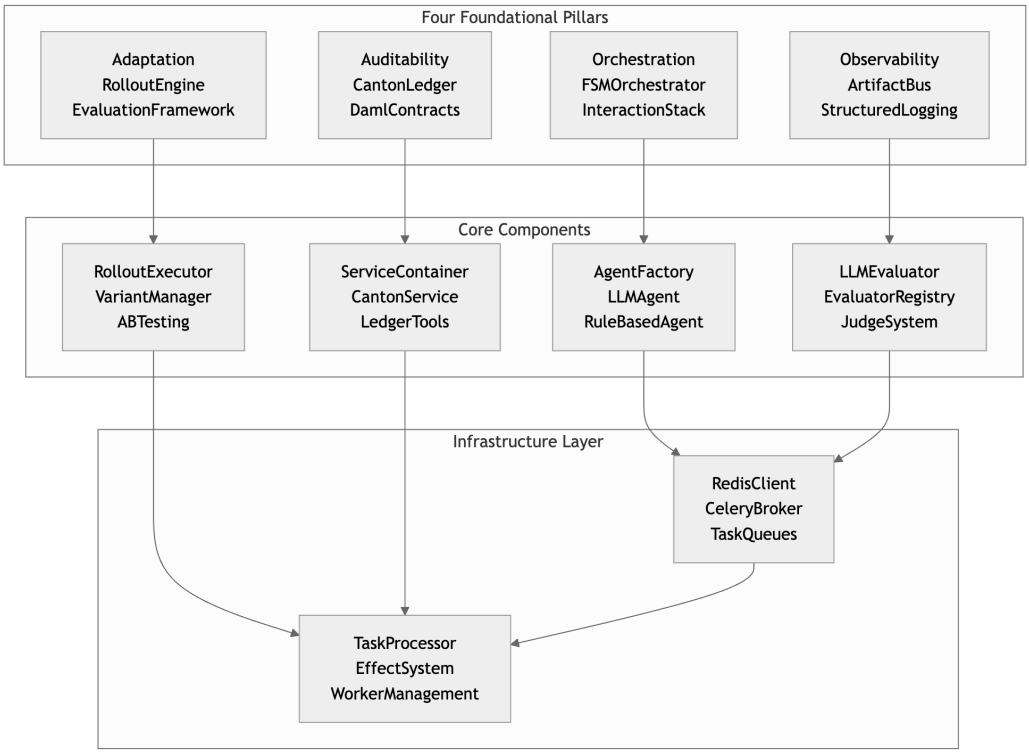


Figure 6.4.1: Foundational Pillars in P2Engine

CHAPTER

SEVEN

EXPERIMENTAL VALIDATION

This chapter validates P2Engine against the four criteria from Section 4.1.2 using video demonstrations. All demo videos are available in the project repository under `/demos`.

7.1 Validation Execution

Each criterion (E1–E4) maps directly to a Research Question (Table 7.1.1) and is illustrated by a dedicated video.

Table 7.1.1: Evaluation Criteria and Research Questions

Criterion	Focus	RQ
E1	FSM orchestration with dynamic task delegation	RQ1.1
E2	Unified event streaming for full traceability	RQ1.2
E3	Rollouts for adaptation infrastructure	RQ1.3
E4	Canton Network ledger audit trails and incentives	RQ1.4

7.2 Video Demonstrations

E1 – Orchestration (`demo_orchestration.mp4`) Agents are orchestrated via FSMs, delegate subtasks to specialists, and track conversation state through

branching.

E2 – Observability (`demo_observability.mp4`) Every agent action and tool invocation is captured in real time, with session and branch identifiers preserving causality.

E3 – Adaptation (`demo_adaptation.mp4`) YAML configured A/B tests run in parallel, GPT-4 judges score variants, and results are tracked.

E4 – Auditability (`demo_auditability.mp4`) Agents maintain wallets on Canton Network, perform autonomous transfers, and all transactions are immutably recorded.

7.3 Integration Validation

The concurrent operation of all four pillars demonstrates P2Engine's unified architecture, autonomous coordination (E1) without rigid DAGs, real-time transparency (E2) across agents and tools, adaptation infrastructure (E3) through rollouts, and verifiable accountability (E4) through distributed-ledger records.

7.4 Research Question Validation

RQ1: Can a multi-agent system framework combining finite-state orchestration, comprehensive observability, adaptation infrastructure, and distributed-ledger auditability deliver the foundations necessary to enable truly agentic multi-agent systems?

Answer: The P2Engine demonstrations indicate that the framework exhibits early agentic behaviors by orchestrating multiple agents, tracing all actions, supporting adaptation loops via rollout mechanisms and artifacts, and auditing economic events. This validates RQ1 and its subquestions.

7.5 Summary

P2Engine conclusively integrates the four foundational pillars in one framework. The video evidence confirms that finite-state orchestration can unify comprehensive event streaming, adaptation loops, and ledger-based audit trails. It is a proof of concept framework that can make multi-agent systems more agentic. All materials are open source and reproducible via the repository.

CHAPTER

EIGHT

FUTURE WORK

Adaptation Methods: P2Engine's modular core makes it straightforward to introduce continuous adaptation loops, where agents refine their policies based on execution outcomes. Lightweight reinforcement or evolutionary strategies could be layered on top of the existing rollout and state-tracking mechanisms to enable agents to learn from both successes and failures. By abstracting policy updates and transition dynamics, the system can support multi-objective tuning, balancing task performance with collaboration quality. Over time, agents could automatically discover improved coordination patterns without manual workflow redesign by humans.

Agent Research Group: Future work could focus on mirroring human research and problem-solving workflows within the tool system. Agents might learn to sequence and compose tools in ways that resemble how people gather, cross-reference, and synthesize information. The finite-state orchestration could be extended to reflect natural decision sequences, enabling smoother transitions between exploration, planning, and execution phases. This approach aims to make agent interactions more intuitive and effective by leaning on established human collaboration strategies.

Market Place: Extending P2Engine into a marketplace platform would involve embedding lightweight micropayment and staking primitives so agents can transact and back each other's services. A simple on-chain or ledger-backed staking mechanism can signal reputation and commitment, rewarding high-quality agents and demoting underperformers. The artifact bus can expose performance metrics in real time, allowing dynamic pricing and reputation calculations. Together, these elements would enable a self-regulating ecosystem where specialized agents discover, contract, and coordinate without central oversight.

CHAPTER

NINE

CONCLUSIONS

This thesis introduced **P2Engine**, a MAS framework that unifies four foundational pillars: **orchestration**, **observability adaptation**, and **auditability** in order to enable agentic multi-agent systems. Through a Design Science Research methodology, P2Engine was designed, implemented, and validated, yielding the following contributions:

C1: Architectural Analysis Identified finite-state machine orchestration as the optimal unifying abstraction for integrating all four pillars.

C2: MAS Framework Developed P2Engine, demonstrating practical integration of FSM orchestration, real-time observability, adaptation infrastructure, and DLT audit trails.

C3: DLT Integration Embedded distributed-ledger technology to provide tamper-evident financial transactions and incentive mechanisms.

C4: Adaptation Infrastructure Built evaluation and rollout capabilities enabling systematic, automated feedback loops for continuous improvement.

P2Engine dynamically allocates and coordinates agents via finite-state orchestration, captures every interaction in a unified event stream for full traceability, executes A/B rollouts with automated LLM judges to enable performance optimization, and records all economic events on-chain, ensuring immutable accountability.

By delivering a concrete, reproducible artifact and clearly articulated design principles, this work establishes a solid foundation for future research and practical deployments of agentic multi-agent systems.

REFERENCES

ART Docs: Getting Started (2025). OpenPipe. URL: https://art.openpipe.ai/getting-started/about (Accessed 17th July 2025).

Baldwin, Anne et al. (2011). *Taxonomy of Increasingly Complex Systems*. ResearchGate. URL: https://www.researchgate.net/publication/264821377_Taxonomy_of_Increasingly_Complex_Systems (Accessed 8th July 2025).

Bertalanffy, Ludwig von (1968). *General System Theory: Foundations, Development, Applications*. George Braziller. URL: https://www.panarchy.org/vonbertalanffy/systems.1968.html (Accessed 8th July 2025).

CAMEL: The first and the best multi-agent framework. Finding the Scaling Law of Agents (2025). Camel-AI. URL: https://github.com/camel-ai/camel (Accessed 15th July 2025).

Cemri, M., M. Z. Pan, S. Yang, L. A. Agrawal, B. Chopra, R. Tiwari and I. ... Stoica (2025). "Why Do Multi-Agent LLM Systems Fail?" In: *arXiv preprint arXiv:2503.13657*. URL: https://arxiv.org/abs/2503.13657 (Accessed 12th July 2025).

Dafoe, Allan, Yoram Bachrach, Mohammed Barakeh, Thore Graepel, Gillian Hadfield, Eric Lancelot, Kate Musick, Johannes Olsson, Hado von Hasselt and Alexander Virsik (2020). "Cooperative AI: machines must learn to find common ground". In: *Nature* 593, pp. 33–36. DOI: 10.1038/d41586-021-01170-0.

Daml: A Smart Contract Language for Distributed Applications (2025). Digital Asset. URL: https://www.digitalasset.com/developers (Accessed 10th Jan. 2025).

DSPy (2025). DSPy. URL: https://dspy.ai/ (Accessed 15th July 2025).

Erik (2025a). *1 : Using State Machines to Orchestrate Multi-Agent Systems — Part 1: The State Machine*. URL: https://arnovich.com/writings/state_machines_for_multi_agent_part_1/ (Accessed 20th Mar. 2025).

— (2025b). *2 : State Machines for Agents — Part 2: Steering Agents*. URL: https://arnovich.com/writings/state_machines_for_multi_agent_part_2/ (Accessed 20th Mar. 2025).

Erik (2025c). *3: State Machines for Agents — Part 3: Extending Agents*. URL: https://arnovich.com/writings/state_machines_for_multi_agent_part_3/ (Accessed 20th Mar. 2025).

— (2025d). *4: State Machines for Agents — Part 4: Improving Agent Reasoning*. URL: https://arnovich.com/writings/state_machines_for_multi_agent_part_4/ (Accessed 10th May 2025).

FSM - User can input speaker transition constraints (2025). AG2. URL: https://docs.ag2.ai/0.8.7/docs/use-cases/notebooks/notebooks/agentchat_groupchat_finite_state_machine/ (Accessed 15th June 2025).

Goldstein, Jeffrey (1999). "Emergence as a Construct: History and Issues". In: *Emergence* 11, pp. 49–72. URL: https://www.researchgate.net/publication/243786253_Emergence_as_a_Construct_History_and_Issues (Accessed 8th July 2025).

Han, Shanshan, Qifan Zhang, Yuhang Yao, Weizhao Jin, Zhaozhuo Xu and Cheng He (2024). "LLM Multi-Agent Systems: Challenges and Open Problems". In: *arXiv preprint arXiv:2402.03578*. URL: https://arxiv.org/abs/2402.03578 (Accessed 12th July 2025).

Heylighen, Francis (2007). "Five Questions on Complexity". In: *Self-organization and emergence in multi-agent systems*. Ed. by Carlos Gershenson. Automatic Press, pp. 1–16.

Holland, John H. (1995). *Hidden Order: How Adaptation Builds Complexity*. Reading, MA: Perseus Books.

Introducing Atropos (2025). Nous Research. URL: https://nousresearch.com/introducing-atropos/ (Accessed 17th July 2025).

Jessop, Bob (2003). "Governance and meta-governance: on reflexivity, requisite variety and requisite irony". In: *Governance as Social and Political Communication*. Ed. by Henrik P. Bang, pp. 101–116.

Kotzé, Paula, Alta van der Merwe and Aurona Gerber (2015). "Design Science Research as Research Approach in Doctoral Studies". In: *Proceedings of the Twenty-First Americas Conference on Information Systems*. Puerto Rico. URL: https://www.cair.org.za/sites/default/files/2019-08/Paper_2_0.pdf (Accessed 10th May 2025).

LangGraph (2025). LangChain. URL: https://www.langchain.com/langgraph (Accessed 15th July 2025).

Luo, Junyu, Weizhi Zhang, Ye Yuan, Yusheng Zhao, Junwei Yang, Yiyang Gu, Bohan Wu, Binqi Chen, Ziyue Qiao, Qingqing Long, Rongcheng Tu, Xiao Luo, Wei Ju, Zhiping Xiao, Yifan Wang, Meng Xiao, Chenwu Liu, Jingyang Yuan, Shichang Zhang, Yiqiao Jin, Fan Zhang, Xian Wu, Hanqing Zhao, Dacheng Tao, Philip S. Yu and Ming Zhang (2025). "Large Language Model Agent: A Survey on Methodology, Applications and Challenges". In: *arXiv preprint arXiv:2503.21460*. URL: https://arxiv.org/abs/2503.21460 (Accessed 12th July 2025).

Make Your AI Agents Unstoppable (2025). Orra Software. URL: https://www.orra.dev/ (Accessed 15th July 2025).

MetaGPT: The Multi-Agent Framework (2025). DeepWisdom. URL: https://www.deepwisdom.ai/ (Accessed 15th July 2025).

Mitchell, Melanie (2009). *Complexity: A Guided Tour*. New York: Oxford University Press.

PydanticAI: Agent Framework (2025). PydanticAI. URL: https://ai.pydantic.dev/ (Accessed 15th July 2025).

Scalable RL solution for advanced reasoning of language models (2025). PRIME-RL. URL: https://github.com/PRIME-RL/PRIME (Accessed 17th July 2025).

Sioud, A. D. (2025). *P2Engine development: Modular multi-agent systems framework, autonomous systems research*. URL: https://adamsioud.com (Accessed 15th July 2025).

Skyttner, Lars (2005). *General Systems Theory: Problems, Perspectives, Practice*. Singapore: World Scientific Publishing.

TensorZero: An Open-Source Stack for Industrial-Grade LLM Applications (2025). TensorZero. URL: https://github.com/tensorzero/tensorzero (Accessed 17th July 2025).

Terlson, Adam (2025). *Agentic State Machines, a pattern library*. URL: https://github.com/adamterlson/AgenticStateMachines (Accessed 20th Mar. 2025).

The AI framework that adds the engineering to prompt engineering) (2025). BoundaryML. URL: https://github.com/BoundaryML/baml (Accessed 15th July 2025).

The Canton Network (2025). Canton Network. URL: https://www.canton.network/ (Accessed 15th July 2025).

The Leading Multi-Agent Platform (2025). CrewAI. URL: https://www.crewai.com/ (Accessed 15th July 2025).

verl: Volcano Engine Reinforcement Learning for LLMs (2025). Volcengine. URL: https://github.com/volcengine/verl (Accessed 17th July 2025).

Wiener, Norbert (1948). *Cybernetics: Or Control and Communication in the Animal and the Machine*. Cambridge, MA: MIT Press.

Xie, Tengyang and Yurun Yuan (2025). "Reinforce LLM Reasoning through Multi-Agent Reflection". In: *arXiv preprint arXiv:2506.08379*. URL: https://arxiv.org/abs/2506.08379 (Accessed 10th July 2025).

Yan, Walden (2025). *Don't Build Multi-Agents*. URL: https://cognition.ai/blog/dont-build-multi-agents (Accessed 12th July 2025).

Yehudai, Asaf, Lilach Eden, Alan Li, Guy Uziel, Yilun Zhao, Roy Bar-Haim, Arman Cohan and Michal Shmueli-Scheuer (2025). "Survey on Evaluation of LLM-based Agents". In: *arXiv preprint arXiv:2503.16416*. URL: https://arxiv.org/abs/2503.16416 (Accessed 12th July 2025).

Zhang, Yaolun, Xiaogeng Liu and Chaowei Xiao (2025). “MetaAgent: Automatically Building Multi-Agent Systems Based on Finite State Machines”. In: URL: https://openreview.net/forum?id=a7gfCUhwdV (Accessed 12th Mar. 2025).

APPENDICES

APPENDIX

A

GITHUB REPOSITORY

The complete `p2engine` framework developed for this master's thesis is available on GitHub. The repository contains all relevant code and documentation for testing, evaluation and future development.

Repository Link

- `https://github.com/NTNU-IDI/multi-agent-system-master-thesis-2025-adam`

Note: The `JULY-2025` branch is the official deliverable. Its contents are frozen as of the thesis deadline and will not be modified further. Future updates and ongoing development of `p2engine` will take place on the `main` branch.

APPENDIX

B

P2ENGINE APPLICATION SCREENSHOTS

Figures B.1–B.19 showcase key p2engine CLI interactions. Covering shell startup, agent chats, ledger operations, agent tool usage, and rollouts. Providing a comprehensive view of the framework in action.

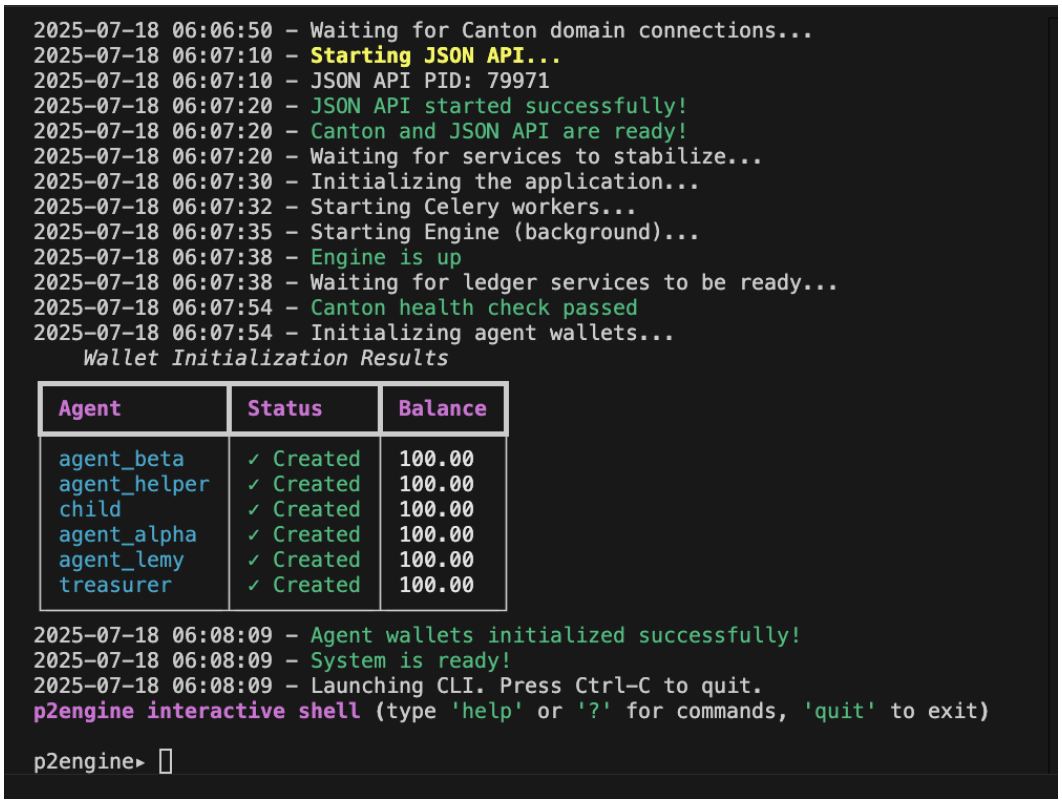


Figure B.1: p2engine CLI startup sequence showing version information and system checks, and the initial interactive shell screen.

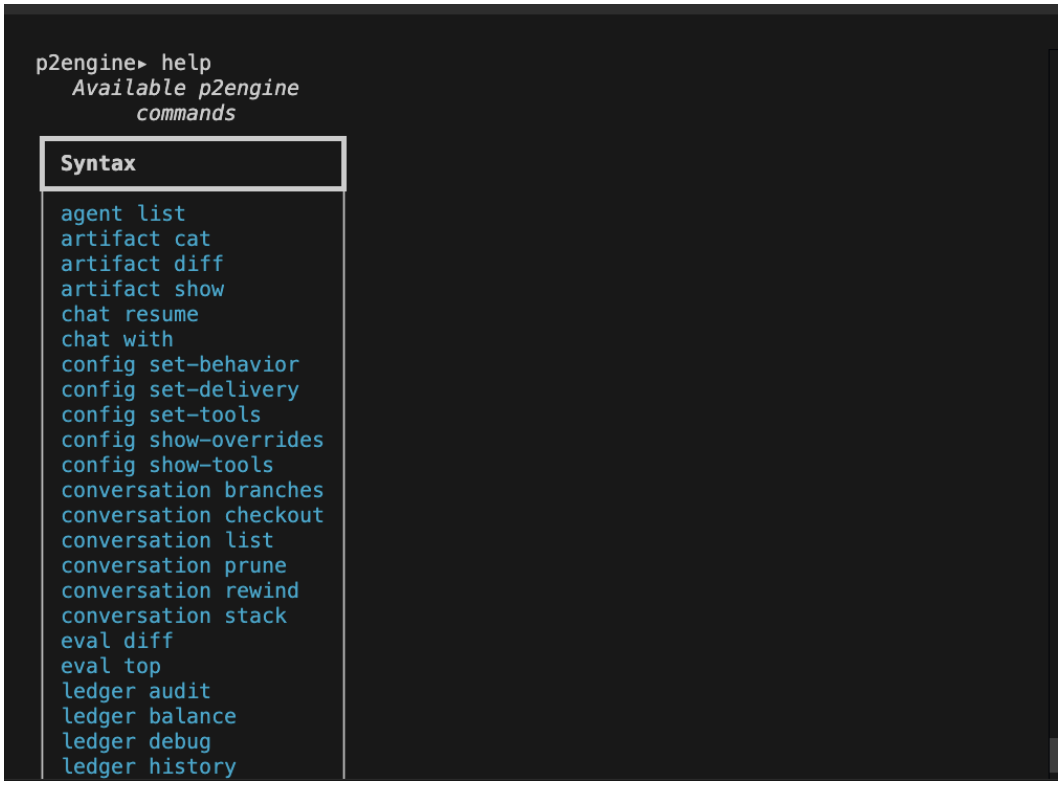


Figure B.2: p2engine `help` command and its output shows the list of available commands in p2engine.

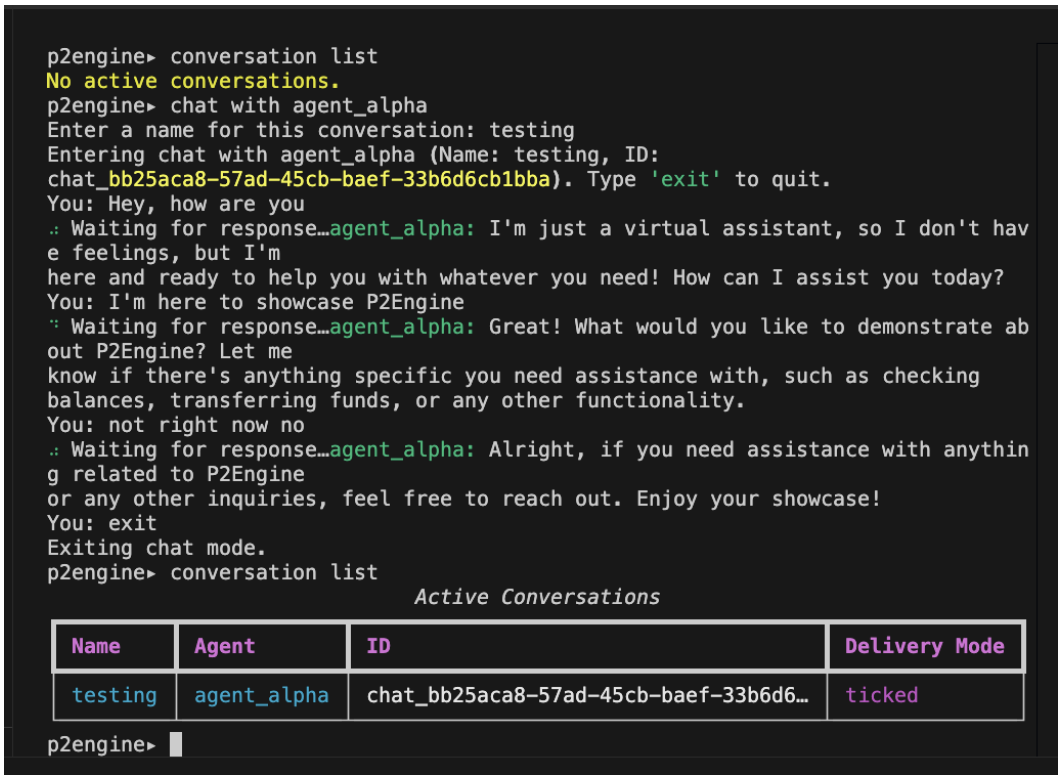
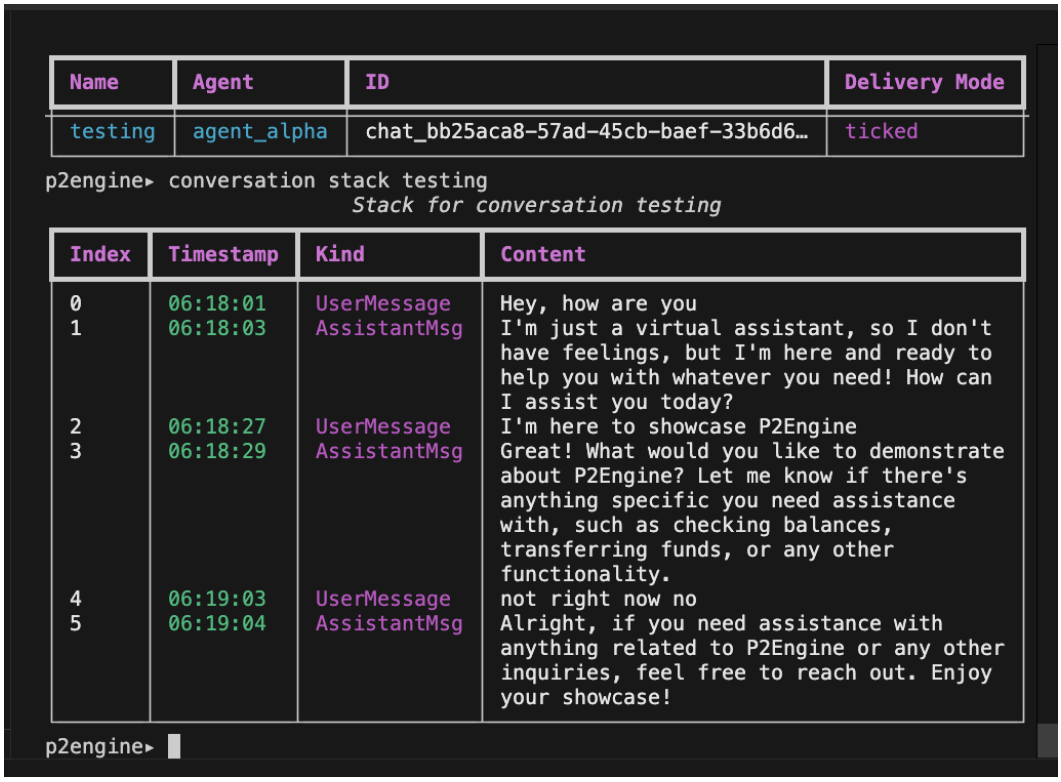


Figure B.3: Chat with Agent
shows starting a chat with `agent_alpha`, the live dialogue, exiting chat mode, and then the updated `active-conversations` list.



Name	Agent	ID	Delivery Mode
testing	agent_alpha	chat_bb25aca8-57ad-45cb-baef-33b6d6…	ticked

p2engine▸ conversation stack testing

Stack for conversation testing

Index	Timestamp	Kind	Content
0	06:18:01	UserMessage	Hey, how are you
1	06:18:03	AssistantMsg	I'm just a virtual assistant, so I don't have feelings, but I'm here and ready to help you with whatever you need! How can I assist you today?
2	06:18:27	UserMessage	I'm here to showcase P2Engine
3	06:18:29	AssistantMsg	Great! What would you like to demonstrate about P2Engine? Let me know if there's anything specific you need assistance with, such as checking balances, transferring funds, or any other functionality.
4	06:19:03	UserMessage	not right now no
5	06:19:04	AssistantMsg	Alright, if you need assistance with anything related to P2Engine or any other inquiries, feel free to reach out. Enjoy your showcase!

p2engine▸

Figure B.4: Conversation Stack
displays the conversation stack, the indexed message history (user and assistant) for the `testing` conversation, as shown by the `conversation stack testing` command.

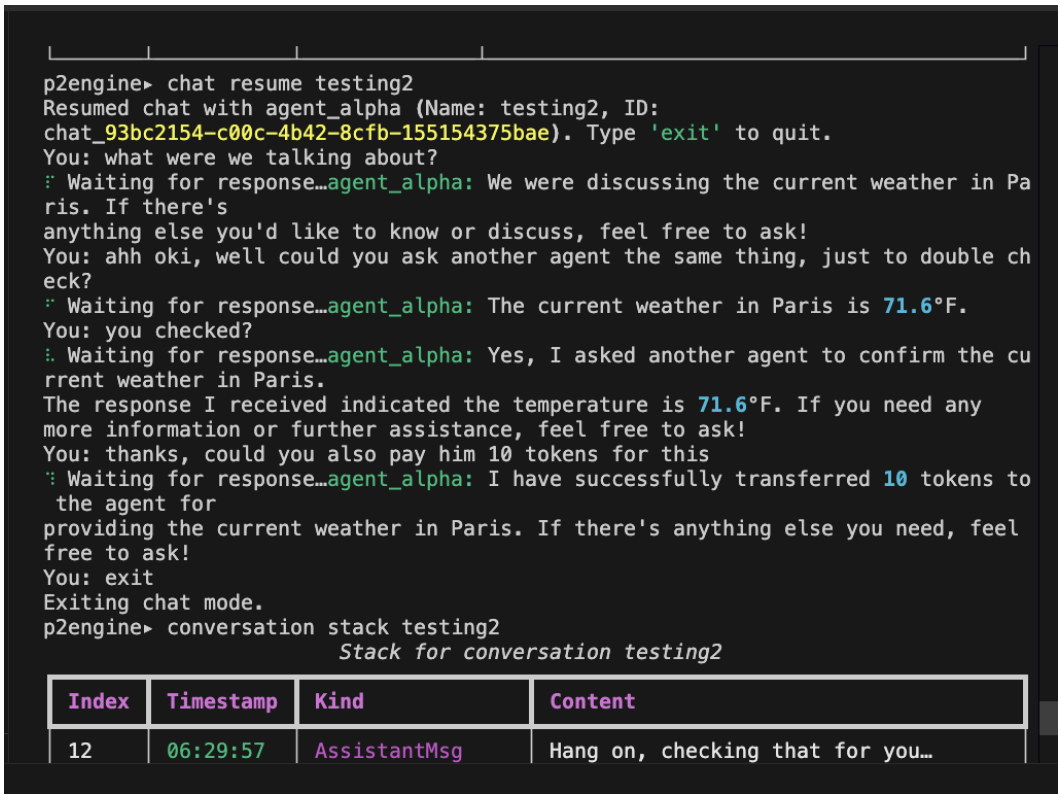


Figure B.5: Resume Chat & Delegation
shows resuming a previous chat (`chat resume testing2`), an agent delegating the `weather-check` tool to another agent, transferring 10 tokens for that service, exiting chat mode, and then viewing the updated `conversation stack`.

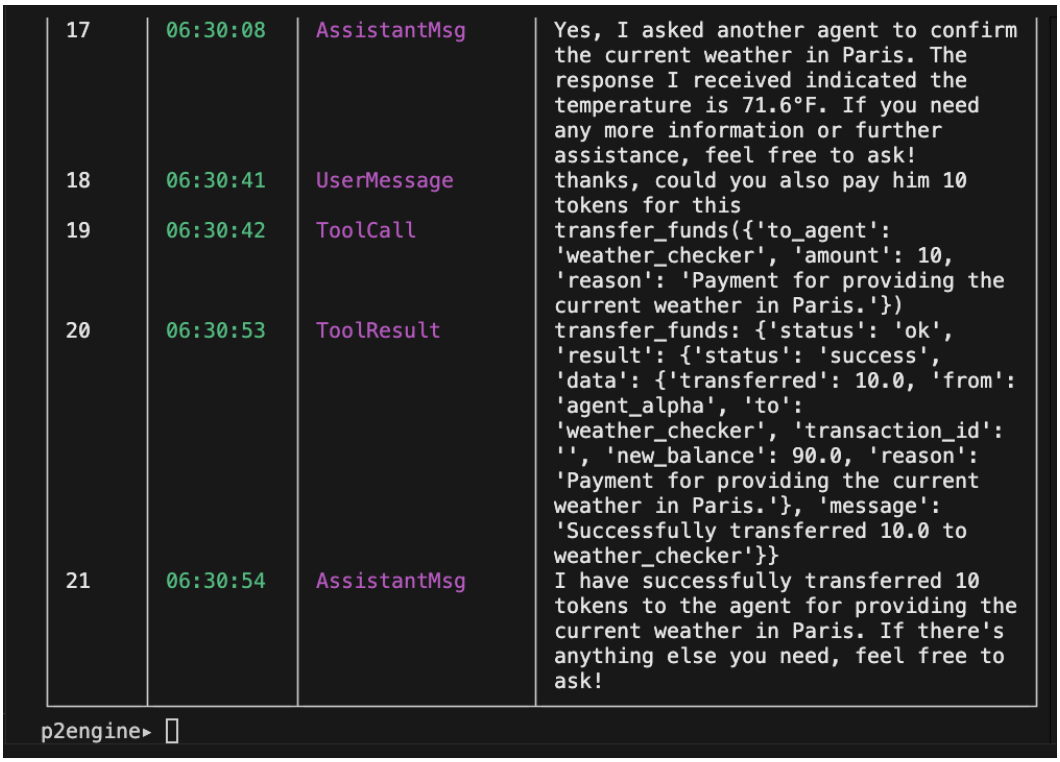


Figure B.6: ToolCall in Stack View
shows the `ToolCall` entry for `transfer_funds({'to_agent':'weather_checker',...})` at index 19, the `ToolResult` at index 20 confirming a successful transfer of 10 tokens, and the assistant's follow-up message at index 21.

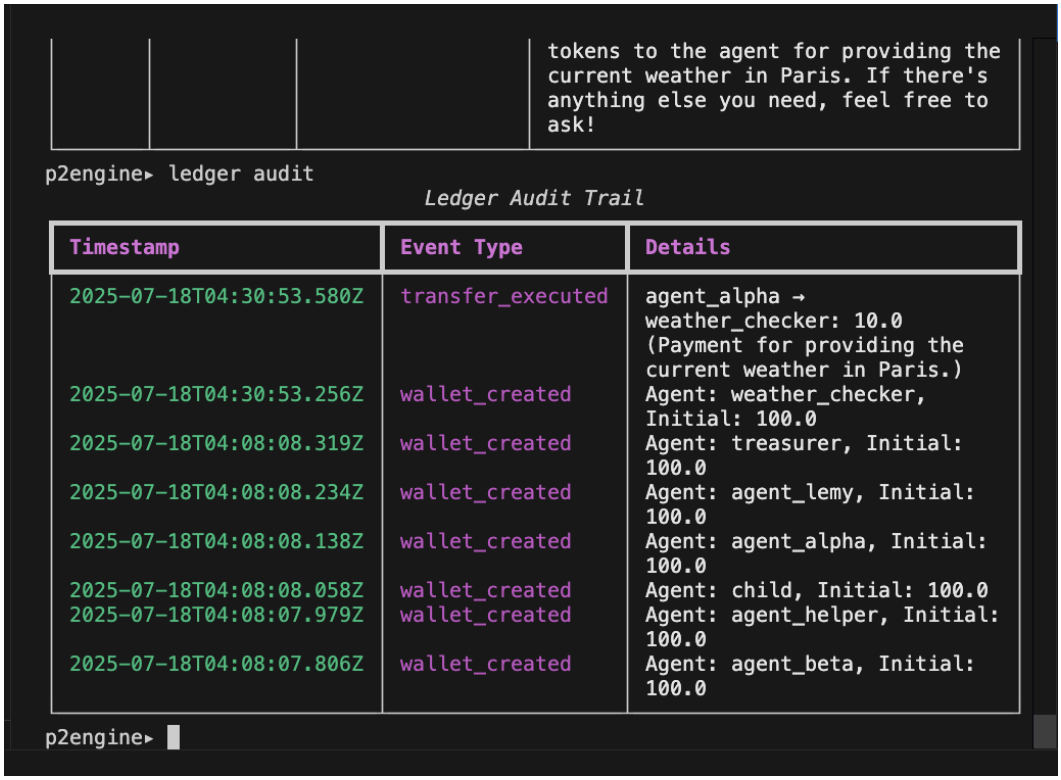


Figure B.7: Audit Trail
shows the `transfer_executed` event with timestamp (`2025-07-18T04:30:53.580Z`) and details of the 10-token payment from `agent_alpha` to `weather_checker`, confirming the transaction on the Canton Network, along with the initial `wallet_created` events.

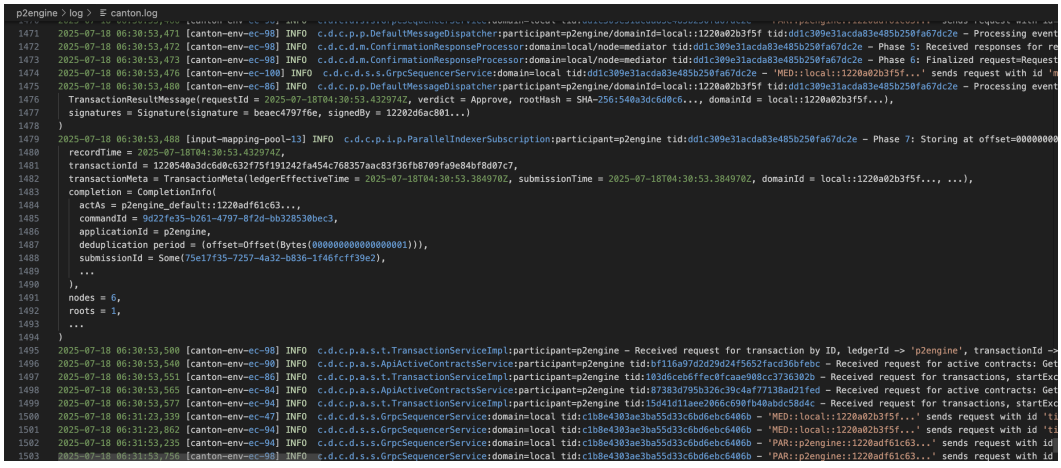


Figure B.8: Canton Network Log File Excerpt
shows INFO-level entries for transaction processing and service events, confirming that the Canton Network is running correctly.

Figure B.9: Journal JSON Excerpt
shows the `journal.ndjson` artifact showing the detailed sequence of state entries, user messages, synthetic events, tool calls, and tool results (including timestamps, step indices, and payload contents) for one conversation, useful for debugging and tracing every action in the session.

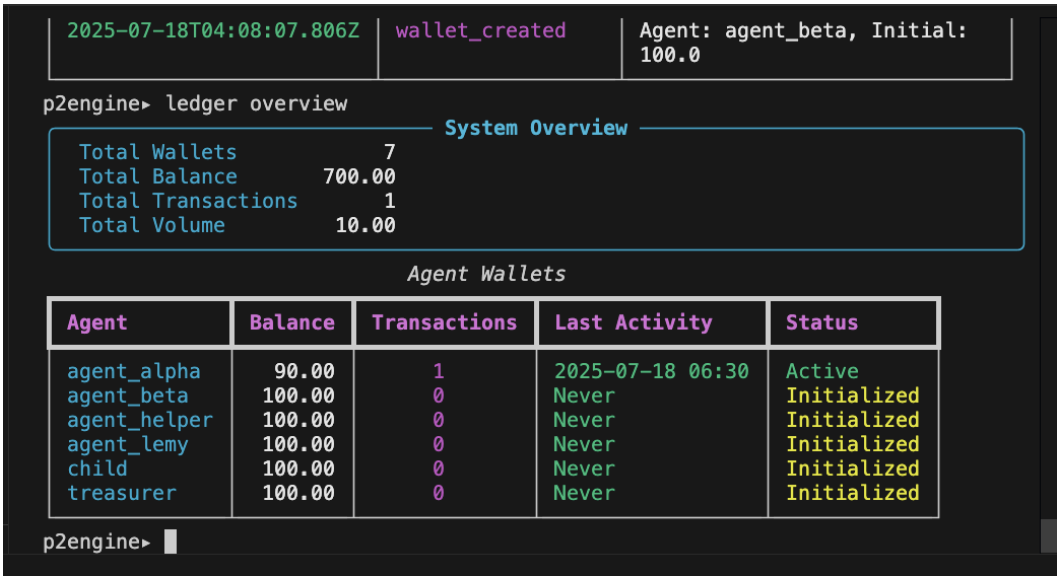


Figure B.10: Ledger Overview Display
shows the system-wide wallet summary (total wallets, balance, transactions, volume) and per-agent wallet details (balance, transaction count, last activity, status) as produced by the `ledger overview` command.

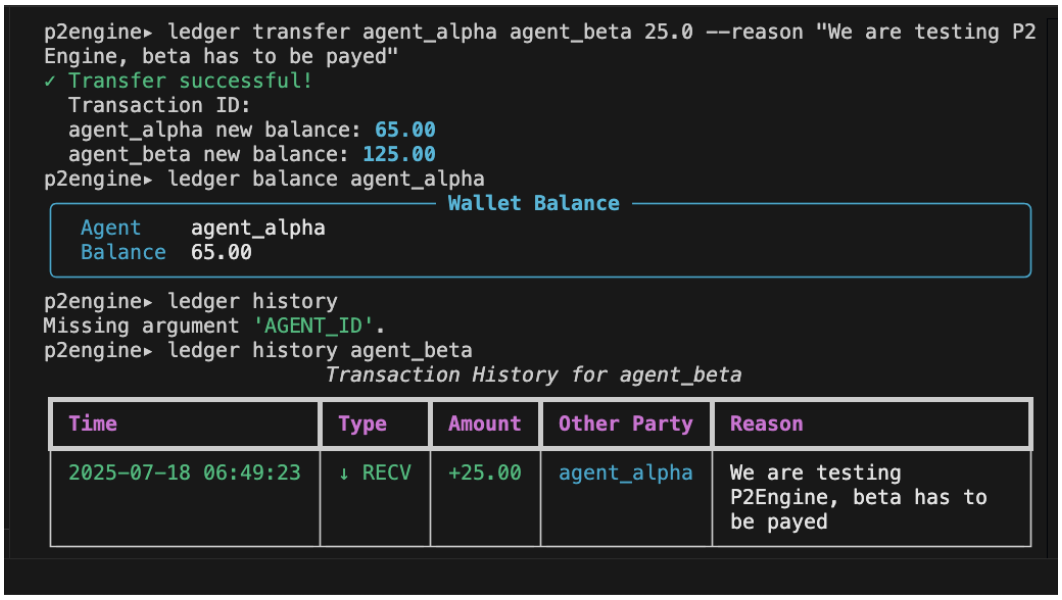


Figure B.11: Ledger Transfer & History
demonstrates the `ledger transfer agent_alpha agent_beta 25.0 -reason "We are testing P2Engine, beta has to be payed"`, showing a successful transfer and updated balances, followed by `ledger balance agent_alpha` and `ledger history agent_beta` displaying the transaction record with timestamp, amount, counterparty, and reason.

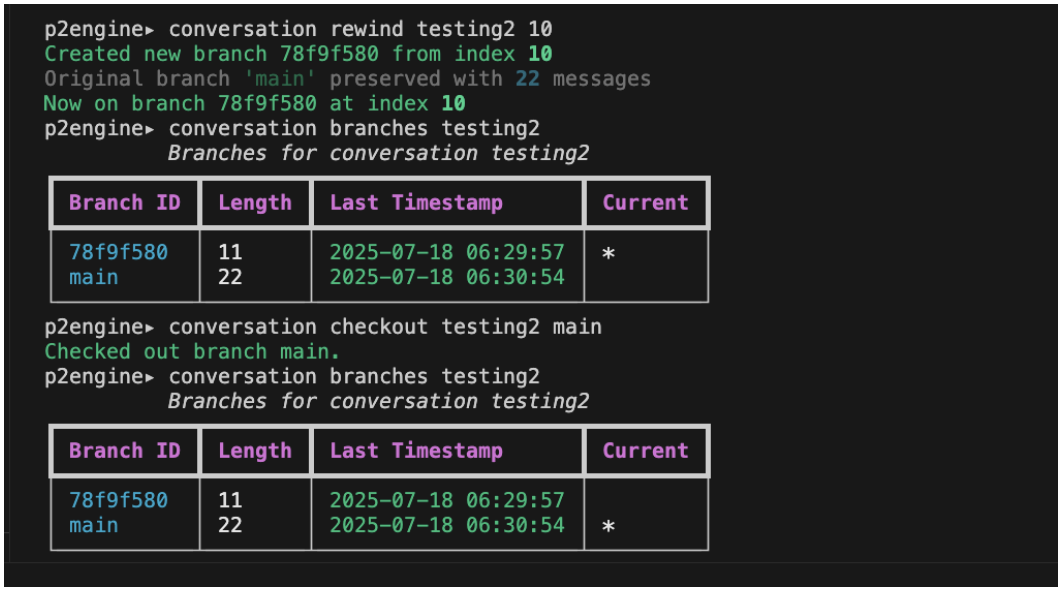


Figure B.12: Conversation Branching & Rewind
shows using `conversation rewind testing2 10` to create a new branch at message index 10, listing both the new branch and the original `main` branch, then checking out `main` and verifying the current branch via `conversation branches testing2`.

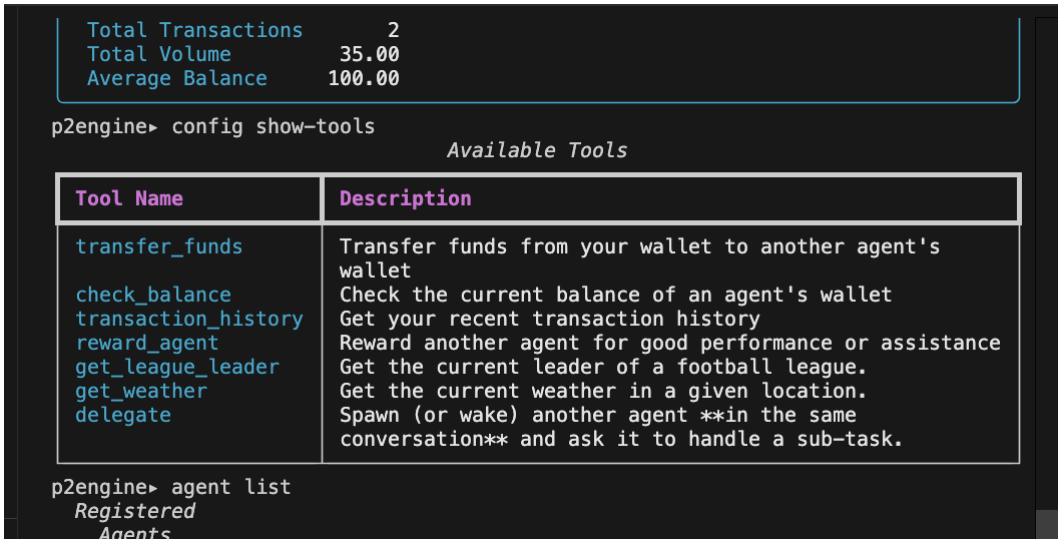


Figure B.13: Available Tools Listing
displays the output of `config show-tools`, listing each tool name (e.g. `transfer_funds`, `delegate`, `get_weather`) alongside its description of functionality.

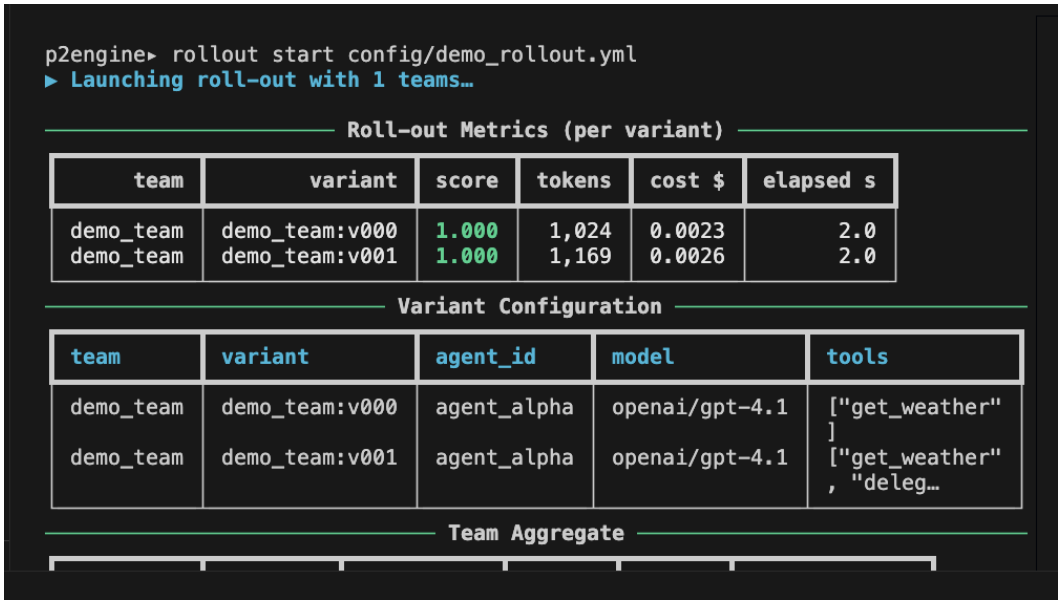


Figure B.14: Rollout Execution & Cost Metrics
shows the results of `rollout start config/demo_rollout.yml`, including per-variant scores, token counts, and the cost ($) column, which reflects actual OpenAI API charges for each variant, alongside elapsed time and the detailed variant configuration.

p2engine > config > ! demo_rollout.yml

```yaml
teams:
  demo_team:
    initial_message: "What's the weather in Paris right now?"

    base:
      agent_id: agent_alpha
      model: openai/gpt-4.1

    variants:
      - tools: ["get_weather"]
      - tools: ["get_weather", "delegate"]

    eval:
      evaluator_id: gpt4_judge
      judge_version: "0.4"
      metric: score
      timeout_sec: 1800
      prompts:
        - |
          You are the judge. Give a score between 0–1 that reflects how helpful
          and correct the assistant's final answer is.
```

Figure B.15: Demo Rollout Configuration
shows the `demo_rollout.yml` file defining one team (`demo_team`) with an initial message, base agent/model settings, two tool-based variants (`get_weather` vs. `get_weather+delegate`), and evaluation parameters using the `gpt4_judge`.

```
p2engine▸ rollout start config/rollout_joke.yml -f
▶ Launching roll-out with 2 teams…
```

team	variant	score	tokens	cost $	elapsed s
joke_team	joke_team:v000	1.000	1,029	0.0043	8.0
joke_team	joke_team:v001	0.750	987	0.0007	2.0
joke_team	joke_team:v002	0.000	1,491	0.0089	20.1
pun_team	pun_team:v000	0.000	881	0.0028	4.0
pun_team	pun_team:v001	0.000	900	0.0005	2.0
pun_team	pun_team:v002	0.000	1,148	0.0055	10.0

Roll-out Metrics (per variant)

team	variant	score	tokens	cost $	elapsed s
joke_team	joke_team:v000	1.000	1,029	0.0043	8.0
joke_team	joke_team:v001	0.750	987	0.0007	2.0
joke_team	joke_team:v002	0.000	1,491	0.0089	20.1
pun_team	pun_team:v000	0.000	881	0.0028	4.0
pun_team	pun_team:v001	0.000	900	0.0005	2.0
pun_team	pun_team:v002	0.000	1,148	0.0055	10.0

Figure B.16: Comparative Rollout Results for Joke & Pun Teams
shows the output of `rollout start config/rollout_joke.yml -f`, comparing two teams (`joke_team` and `pun_team`) across three variants each (v000–v002). For each variant it reports the judge "score," token usage, actual OpenAI API "cost ($)," and elapsed time, allowing direct comparison of performance and cost between approaches.

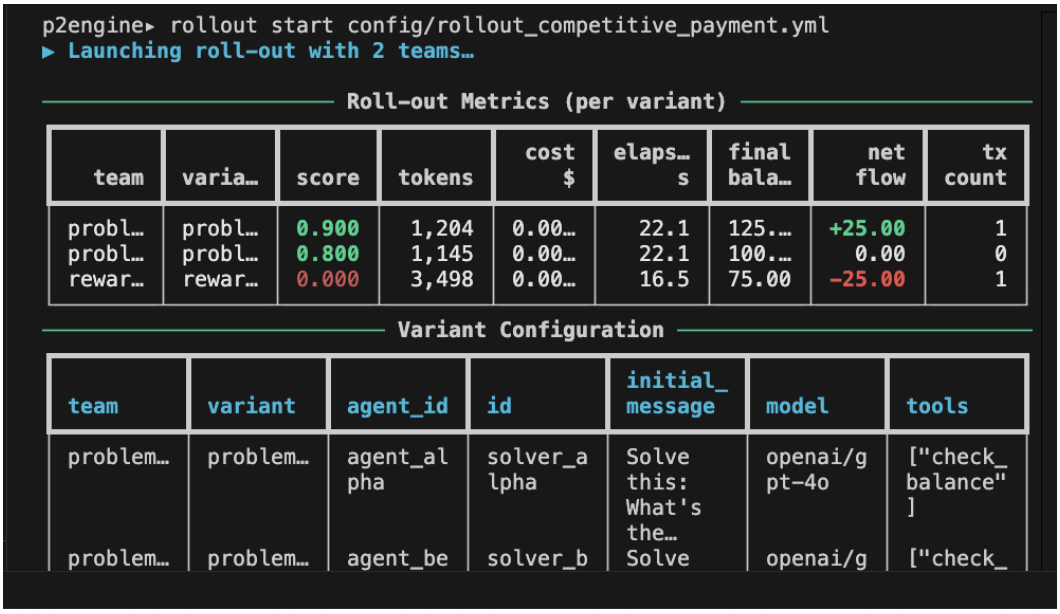


Figure B.17: Competitive Payment Rollout Metrics
displays the results of `rollout start config/rollout_competitive_payment.yml`, including per-variant judge scores, token usage, actual API cost, elapsed time, final balances, net token flows, and transaction counts, enabling side-by-side comparison of "solver" versus "rewarder" strategies.

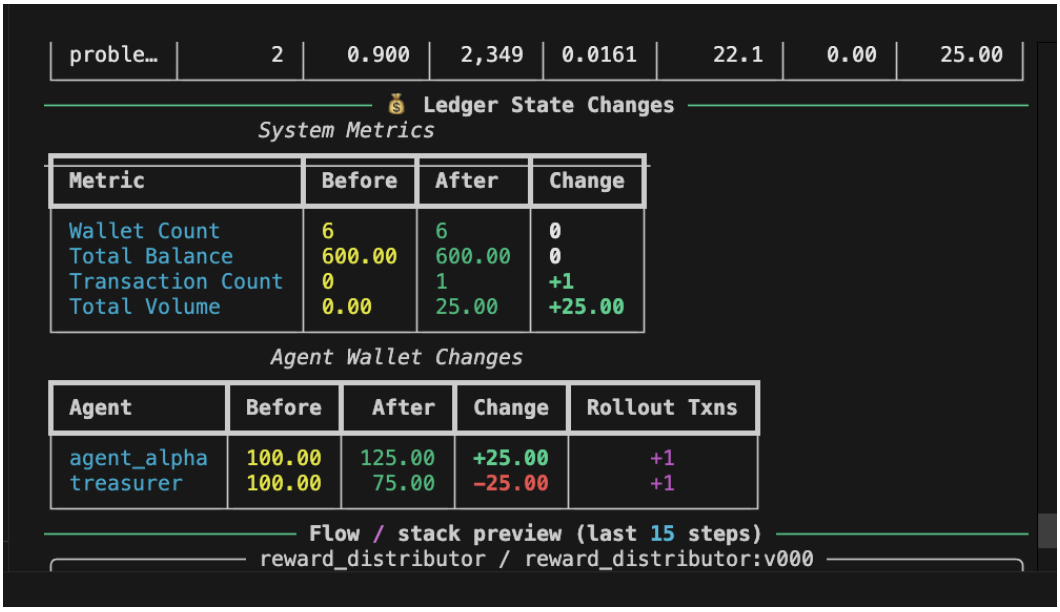


Figure B.18: Ledger State Changes
displays the before/after comparison of the ledger following the competitive-payment rollout: system metrics (wallet count, total balance, transaction count, total volume) and per-agent wallet changes (balances before and after, net change, and number of rollout transactions).

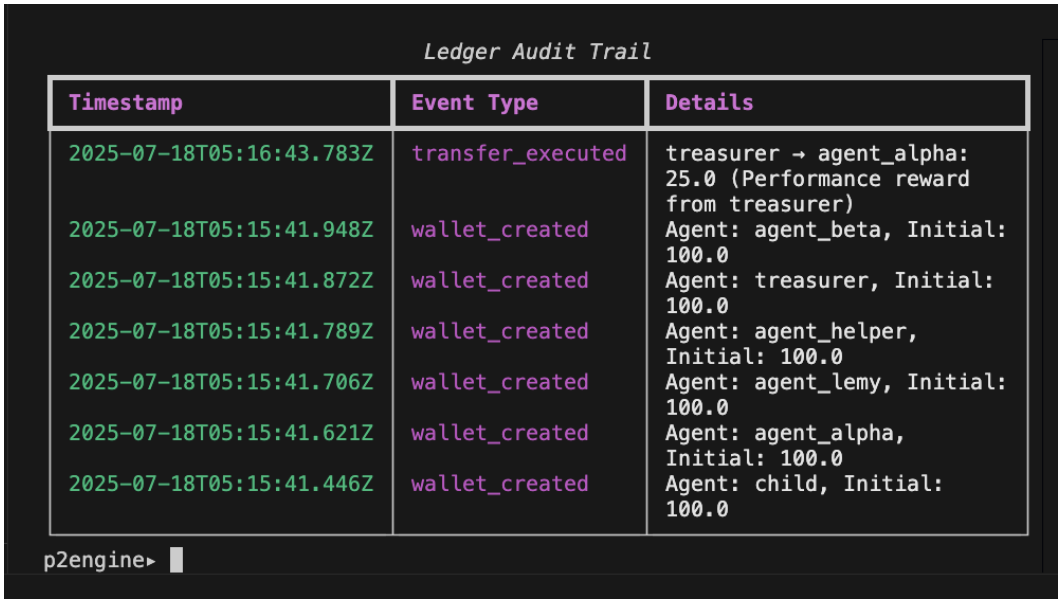


Figure B.19: Ledger Audit Trail Post Rollout
shows the `transfer_executed` event for the 25-token performance reward from `treasurer` to `agent_alpha` with its timestamp, followed by the initial `wallet_created` entries, confirming the audit trail after the rollout.