A Modular Framework for Scalable and Auditable LLM-Based Multi-Agent Systems

MASTER'S THESIS

Mohammad Ayaz Alam

Submitted on 16 January 2026

Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Prof. Dr. Dirk Riehle, M.B.A.
Prof. Dr. Frauke Liers
Dr. Robert Grewe (Siemens Energy)

Friedrich-Alexander-Universität
Faculty of Engineering

Declaration of Originality

I confirm that the submitted thesis is original work and was written independently by me. Digital tools were used exclusively for language refinement, grammar correction, and structural improvements. The scientific content, design decisions, implementation, evaluation, and conclusions are entirely my own.

Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

Erlangen, 16 January 2026

License

Erlangen, 16 January 2026

Abstract

Large Language Models have recently enabled new forms of intelligent systems that go beyond traditional question answering or text generation. When embedded into agent-based systems, these models can reason, plan, and interact with external tools. However, deploying such systems in real-world and industrial contexts remains challenging. Existing agent implementations often lack transparency, scalability, and reliable mechanisms for human oversight. As a result, their adoption in safety critical or operational environments is limited.

This thesis presents a modular framework for scalable and auditable LLM-based multi-agent systems. The framework is designed to treat orchestration, memory management, observability, and human intervention as first class system concerns. Rather than focusing on model level improvements, the framework emphasizes system level design that enables controlled execution, traceability, and reuse across different applications.

The framework is evaluated through two applied case studies. The first case study focuses on document driven analysis for a methanol factory construction scenario, where multiple specialized agents collaborate to interpret requirements and generate structured outputs. The second case study addresses operational diagnostics and communication support for a Terminal Management System, where selective human approval is required before executing external actions.

The results show that the proposed framework improves transparency, supports scalable knowledge access, and enables safe integration of automated reasoning into real workflows. The thesis demonstrates that effective deployment of LLM-based multi-agent systems depends primarily on architectural and engineering decisions rather than on model capabilities alone.

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface

CLI Command Line Interface

HITL Human In The Loop

LLM Large Language Model

MAS Multi Agent System

RAG Retrieval Augmented Generation

REST Representational State Transfer

TMS Terminal Management System

UI User Interface

1 Introduction

1.1 Background and Context

Large Language Models have become a central component of modern artificial intelligence systems. Advances in neural network architectures and large scale training have enabled models to process natural language with a level of fluency and contextual understanding that was previously unattainable. The introduction of attention based architectures made it possible to train models on vast text corpora and to generalize across a wide range of tasks. Subsequent work showed that increasing model size and data scale leads to emergent capabilities such as few shot learning and contextual adaptation.

These developments have led to the rapid adoption of Large Language Models in many domains, including document analysis, code generation, conversational interfaces, and decision support. In many applications, language models are no longer used as isolated components but are embedded within larger systems that combine reasoning, data access, and action execution.

At the same time, the increasing autonomy of such systems raises new challenges. When language models are allowed to select actions, invoke tools, or interact with external systems, traditional software engineering assumptions no longer fully apply. System behavior becomes probabilistic, difficult to debug, and hard to audit after execution.

1.2 LLM Based Agents and System Evolution

To extend the capabilities of Large Language Models beyond static text generation, recent research and practice have explored agent-based paradigms. In these systems, a language model is embedded within an agent that can reason about tasks, decide which actions to take, and interact with external tools. This approach allows models to decompose complex problems, retrieve information dynamically, and perform multi step workflows.

Agent-based systems have demonstrated impressive capabilities in experimental settings. However, many early implementations were built as exploratory prototypes. They often relied on implicit control flow embedded in prompts and lacked explicit mechanisms for managing execution state, memory, or safety constraints. As a result, these systems were difficult to scale and unsuitable for deployment in professional or industrial environments.

More recent frameworks have introduced abstractions for agent orchestration and tool integration. While these frameworks simplify development, they still leave many system level concerns unresolved. In particular, issues related to auditability, observability, and human oversight are often handled outside the core execution model.

1.3 Problem Statement

Despite significant progress in LLM-based agents, several fundamental problems remain unresolved.

First, transparency is limited. In many systems, it is unclear which agent made a particular decision, which data was used, or which tools were invoked during execution. This lack of traceability makes debugging difficult and undermines trust in system outputs.

Second, scalability remains a challenge. Passing long documents or entire conversation histories directly to language models leads to excessive token usage and performance degradation. Many systems rely on ad hoc truncation strategies rather than principled approaches to knowledge access.

Third, safety and control mechanisms are often insufficient. Systems that can trigger external actions such as sending messages or modifying data require reliable human oversight. Without integrated approval mechanisms, such systems pose operational risks.

Finally, reuse across applications is limited. Many agent systems are tightly coupled to a specific use case, making it difficult to transfer the same system to a different domain without substantial reimplementation.

1.4 Motivation

The motivation for this thesis arises from practical experience building and deploying LLM-based systems in applied contexts. In industrial and operational settings, correctness, traceability, and safety are often more important than raw model performance. Stakeholders require explanations, logs, and clear control over automated actions.

In addition, real-world systems evolve over time. New requirements, new data sources, and new tools must be integrated without rewriting the entire system. This motivates a framework oriented approach rather than a collection of isolated agent implementations.

The framework presented in this thesis is motivated by the need to bridge the gap between experimental agent systems and production ready engineering practices. The goal is not to replace existing agent frameworks, but to build on them in a way that makes system behavior explicit, observable, and controllable.

1.5 Objectives and Contributions

The primary objective of this thesis is to design and implement a modular framework for scalable and auditable LLM-based multi-agent systems.

The main contributions of this work are:

- a framework architecture that separates orchestration, agent execution, memory, and tooling
- a configuration driven approach that enables reuse across different applications
- an explicit execution model that supports traceability and debugging
- an integrated human in the loop mechanism for safe action execution
- applied validation through two real-world inspired case studies

Rather than focusing on optimizing individual model prompts or tasks, the thesis emphasizes system level design and engineering decisions.

1.6 Development Approach

This thesis follows an agile engineering approach. While an initial set of requirements and architectural goals was defined at the beginning of the work, several requirements were refined and clarified during implementation based on technical constraints, operational feedback, and evaluation results.

As a result, the structure of this thesis reflects iterative development rather than a strictly linear process. Architectural decisions influenced implementation choices, while implementation experience informed the refinement of requirements and evaluation criteria. This approach is consistent with the nature of engineering work involving complex software systems and aligns with the framework's development in an industry context.

1.7 Structure of the Thesis

The remainder of this thesis is structured as follows.

Chapter 2 reviews related work on Large Language Models, agent based systems, and existing frameworks. Chapter 3 derives the functional and non functional requirements for the proposed framework. Chapter 4 presents the system architecture. Chapter 5 describes the design and implementation of the framework and its application to two projects. Chapter 6 evaluates the framework through applied case studies. Chapter 7 concludes the thesis and outlines directions for future work.

2 Literature Review

This chapter reviews existing work related to Large Language Models, agent based systems, and frameworks for orchestrating such systems. The purpose of this review is not to list every recent publication in the field, but to identify the ideas and limitations that directly motivate the design of the framework proposed in this thesis.

The review progresses from language models as reasoning components, to agent based systems, to multi agent coordination and existing frameworks. The chapter concludes by identifying gaps that are not sufficiently addressed by current approaches.

2.1 Large Language Models as Reasoning Components

Large Language Models were originally developed as probabilistic models for next token prediction. The introduction of attention based architectures enabled models to process long sequences efficiently and to capture complex contextual relationships in text. The Transformer architecture in particular made it possible to scale model size and training data substantially, leading to rapid improvements in language understanding and generation capabilities.[1]

Subsequent work demonstrated that sufficiently large language models exhibit emergent behaviors such as few shot learning and task generalization without explicit task specific training.[2] These findings shifted the role of language models from narrow task solvers to general purpose reasoning components that can be adapted through prompting.

Research on prompting strategies further showed that encouraging models to externalize intermediate reasoning steps improves performance on tasks that require multi step reasoning. Chain of thought prompting revealed that models can

[1] Vaswani et al., 2017.
[2] Brown et al., 2020.

internally represent structured reasoning processes when guided appropriately.[3] This insight laid the foundation for treating language models as components that can reason, rather than merely generate fluent text.

However, while these approaches improve reasoning quality, they remain limited to text based outputs and do not address interaction with external systems, persistent state, or execution control.

2.2 Tool Use and Action Oriented Language Models

To extend the capabilities of language models beyond text generation, several approaches introduced mechanisms for tool use and action execution. The ReAct framework proposed interleaving reasoning and acting, allowing models to decide when to invoke tools based on intermediate reasoning steps.[4] This approach demonstrated that combining symbolic actions with language reasoning improves performance on tasks requiring external knowledge or computation.

Toolformer further explored this direction by training language models to autonomously decide when to use tools, based on augmented training data that includes tool usage examples.[5] These approaches highlight the potential of language models to act as controllers within larger systems.

Despite these advances, most work on tool use focuses on single agent settings. Questions related to orchestration across multiple agents, control over action execution, and traceability of tool usage are often left unaddressed. In particular, the ability to inspect which tools were invoked, with what inputs, and for what reason is critical in applied settings.

2.3 Agent-Based Systems Driven by Language Models

Building on reasoning and tool use, agent-based systems embed language models into execution loops that allow planning, memory access, and iterative action selection. Early systems such as BabyAGI and AutoGPT demonstrated that language models can be used to decompose goals into sub tasks and execute them sequentially.[6]

[3] Wei et al., 2022.
[4] Yao et al., 2023.
[5] Schick et al., 2023.
[6] Nakajima, 2023; Richards, 2023.

These systems attracted attention because they exhibited a form of autonomy that was previously difficult to achieve. However, many such implementations relied on implicit control flow encoded in prompts and lacked clear boundaries between reasoning, state management, and execution. As a result, system behavior was often brittle and difficult to debug.

More recent work explored richer agent architectures. Generative agent models demonstrated how agents with memory and social interaction rules can produce emergent behavior in simulated environments.[7] While these studies provide valuable insights into agent behavior, they are primarily focused on simulation and do not directly address deployment concerns such as safety, auditability, or integration with real systems.

Overall, the literature shows that language model driven agents are powerful, but that naive implementations struggle with reliability and control.

2.4 Multi-Agent Systems and Coordination

To address the limitations of single agent approaches, recent work has explored multi-agent systems where tasks are distributed across multiple specialized agents. Surveys of LLM-based multi-agent systems identify common patterns such as task decomposition, agent collaboration, and hierarchical coordination.[8]

Hierarchical coordination in particular has been shown to reduce complexity by assigning supervisory roles to coordinating agents that delegate work to specialized sub agents. This approach mirrors established principles from software architecture and organizational design.

However, these surveys also highlight significant challenges. Coordinating multiple agents introduces overhead, non determinism, and new failure modes. Without explicit orchestration and shared state management, agent interactions can become difficult to reason about.

Most existing work focuses on conceptual architectures or benchmark tasks. System level concerns such as persistent memory, execution tracing, and controlled action execution are often treated as secondary concerns or left to application specific implementations.

[7] Park et al., 2023.
[8] Chen et al., 2024; Li et al., 2024.

2.5 Existing Frameworks for LLM-Based Agents

Several open source frameworks aim to simplify the development of LLM-based agents. LangChain provides abstractions for prompt management, tool invocation, and agent execution, enabling rapid prototyping of agent workflows.[9] These abstractions reduce boilerplate code and encourage reuse.

LangGraph extends this approach by modeling agent execution as explicit graphs, allowing developers to define stateful workflows with branching and conditional execution.[10] This graph based execution model makes control flow more explicit and improves debuggability compared to prompt only approaches.

A key observation for this thesis is that these frameworks provide powerful primitives, but they do not prescribe a complete engineering runtime for operational deployments. In particular, developers still need to assemble and maintain cross-cutting concerns such as configuration management, persistent conversation state, audit logs, controlled execution for irreversible actions, and deployable interfaces for real users (CLI/API). As a result, production projects typically evolve into application-specific ”glue layers” around LangChain and LangGraph.

The framework proposed in this thesis intentionally builds on LangChain v1 and LangGraph for core agent execution, but contributes an engineering layer on top: configuration-driven agent hierarchies, reusable project templates, integrated memory persistence, structured logging for traceability, and human-in-the-loop policies that are enforced by the runtime rather than by prompt conventions. This shifts recurring operational complexity away from each individual application into a reusable framework core.

While these frameworks provide valuable building blocks, they focus primarily on developer convenience and experimentation. Concerns such as persistent audit logs, human approval workflows, and long term state management are typically left to the application layer.

As a result, developers must extend these frameworks significantly when building systems intended for operational or industrial use.

2.6 Identified Gaps and Motivation for This Work

The reviewed literature demonstrates rapid progress in language model capabilities and agent architectures. However, several gaps remain that limit the practical deployment of LLM-based multi-agent systems.

[9] Chase, 2022.
[10] LangGraph, 2023.

First, auditability is rarely treated as a core requirement. While reasoning traces may be generated, structured execution logs that enable post hoc analysis are often missing.

Second, safety mechanisms such as human oversight are frequently implemented externally or inconsistently, rather than being integrated into the execution model.

Third, scalability concerns related to document handling and knowledge access are often addressed through ad hoc context truncation rather than systematic retrieval based approaches.

Finally, reuse across applications is limited. Many systems are tightly coupled to a specific domain, making it difficult to transfer the same architecture to a new use case.

These gaps motivate the framework proposed in this thesis. Building on the motivation introduced in Chapter 1, the framework operationalizes these concerns through explicit architectural decisions: orchestration logic is separated from agent execution, memory access is structured through configurable backends, auditability is achieved through persistent execution traces, and human oversight is integrated into the runtime rather than deferred to external tooling.

3 Requirements

This chapter derives the requirements for the proposed framework. The requirements are grounded in limitations identified in the literature review and in practical constraints observed while building and applying LLM-based agent systems. Rather than treating requirements as fixed assumptions, they are presented as outcomes of an iterative engineering process that balances theoretical insights with real-world usage.

The requirements are divided into functional requirements, non functional requirements, and guiding design principles. Together, they define what the framework must provide before architectural and implementation details are discussed.

3.1 Functional Requirements

Functional requirements describe the capabilities that the framework must support in order to enable practical LLM-based multi-agent applications.

3.1.1 Support for Multi-Agent Systems

The framework must support the definition and execution of multiple agents within a single system. Each agent should encapsulate a distinct responsibility and be able to operate independently while still collaborating with other agents.

This requirement follows directly from findings in the literature that show improved robustness and flexibility when complex tasks are decomposed across specialized agents rather than handled by a single monolithic agent.[1]

In practice, this means that agents must be first class entities within the framework rather than ad hoc prompt templates.

[1] Chen et al., 2024; Li et al., 2024.

3.1.2 Explicit Agent Orchestration

The framework must provide explicit orchestration logic that controls how agents are selected, invoked, and coordinated. Orchestration decisions must not be embedded implicitly within prompts or hidden inside agent implementations.

Explicit orchestration is required to make system behavior understandable and debuggable. It also enables consistent execution across different projects and reduces unintended interactions between agents.

3.1.3 Shared Execution Context

All agents participating in a user session must operate over a shared execution context. This context must include conversation history, intermediate results, and execution metadata.

Without shared context, agent interactions become fragmented and repetitive. A shared context enables agents to build on each other's outputs and supports multi step workflows without requiring repeated user input.

3.1.4 Controlled Tool Invocation

Agents must be able to invoke external tools such as document processors, data retrieval services, or communication systems. Tool invocation must be explicit and mediated by the framework rather than executed directly by the agent.

Research on tool using language models shows that tool integration significantly expands system capabilities, but also introduces safety and traceability challenges if left uncontrolled.[2] The framework must therefore treat tool calls as observable system events.

3.1.5 Human Oversight Capability

The framework must support human intervention for actions that have external effects or operational risk. Human approval must be integrated into the execution flow and not handled as an external or manual process.

This requirement is particularly important in applied settings, where incorrect automated actions can have significant consequences. Research on human oversight emphasizes the importance of keeping humans meaningfully involved in automated decision making systems.[3]

[2] Schick et al., 2023; Yao et al., 2023.

[3] Kandikatla and Radeljić, 2023; Laux, 2023.

3.2 Non Functional Requirements

Non functional requirements describe quality attributes that are essential for deployment in real-world environments.

3.2.1 Auditability

The framework must provide sufficient information to reconstruct how a particular output was produced. This includes visibility into agent selection, tool usage, intermediate results, and human approval decisions.

Auditability is a recurring challenge in LLM-based systems and is frequently identified as a barrier to adoption in professional contexts.[4] For this reason, auditability is treated as a core requirement rather than an optional feature.

3.2.2 Observability and Debugging

Closely related to auditability, the framework must expose its internal state and execution flow in a way that supports debugging and analysis. Developers should be able to inspect system behavior during development and after deployment.

Observability is particularly important in systems that involve non deterministic components, where unexpected behavior cannot always be reproduced easily.

3.2.3 Scalability of Knowledge Access

The framework must scale to large document collections and long running interactions without relying on passing entire documents or histories to language models.

Prior work on retrieval augmented generation highlights the importance of selective context retrieval for reducing token usage and improving performance.[5] The framework must therefore support structured knowledge access mechanisms.

3.2.4 Safety and Control

The framework must prevent uncontrolled execution of potentially harmful actions. This includes restricting tool access, enforcing approval requirements, and providing conservative default behavior.

Safety mechanisms must be part of the framework itself rather than added on a per application basis. This ensures consistent behavior across projects and reduces the likelihood of configuration errors.

[4] Li et al., 2024.
[5] Li et al., 2024.

3.2.5 Reusability and Extensibility

The framework must be reusable across different applications and domains. Adding new agents, tools, or interfaces should not require changes to the core framework logic.

This requirement is motivated by the need to support multiple projects with a shared execution engine, as demonstrated later in the thesis.

3.3 Design Principles

Based on the functional and non functional requirements, several design principles guide the development of the framework.

3.3.1 Configuration Driven Behavior

System behavior should be defined primarily through configuration rather than hard coded logic. Agent roles, permissions, and execution constraints should be modifiable without changes to the framework code.

This principle supports rapid iteration and reuse across projects.

3.3.2 Separation of Concerns

The framework separates orchestration, agent execution, memory management, tool integration, and user interaction into distinct components. This separation improves clarity and reduces coupling between parts of the system.

3.3.3 Explicit Control Flow

Execution decisions such as routing, tool invocation, and approval checks should be explicit and inspectable. Implicit control flow embedded in prompts is avoided where possible.

This principle directly supports auditability and observability goals.

3.3.4 Safe Defaults

The framework favors conservative default behavior. Potentially risky actions require explicit configuration and often human approval.

Safe defaults reduce the risk of unintended actions and support responsible deployment of automated systems.

3.4 Summary

This chapter defined the requirements and design principles for the proposed framework. The requirements reflect both insights from prior research and practical constraints encountered during development.

These requirements provide the foundation for the architectural design presented in the next chapter, which shows how they are realized in a concrete system structure.

4 Architecture

This chapter presents the architecture of the proposed framework. The goal of the architecture is to translate the requirements defined in Chapter 3 into a coherent system structure that supports scalable, auditable, and controllable LLM-based multi-agent systems. The architecture focuses on separation of concerns, explicit orchestration, and system level observability rather than on individual model behavior.

The framework is designed as a reusable execution system. It does not encode domain specific logic directly, but instead provides architectural mechanisms that can be instantiated through configuration and project specific components. This approach allows the same architecture to support different applications without modification of the core framework.

4.1 Architectural Overview

At a high level, the framework follows a layered architecture. User interactions enter the system through standardized interfaces and are processed by a central orchestration layer that coordinates agent execution, memory access, tool invocation, and human oversight. Persistent storage and observability components support traceability and post execution analysis.

Figures 4.1 and 4.2 present the static layered architecture of the framework and introduce the main architectural components and their responsibilities.

This layered view establishes a conceptual separation between concerns. Each layer has a clear responsibility and interacts with other layers through explicit interfaces.

4.2 Framework Architecture Principles

The framework architecture is guided by a small set of explicit principles that shape both structure and behavior.

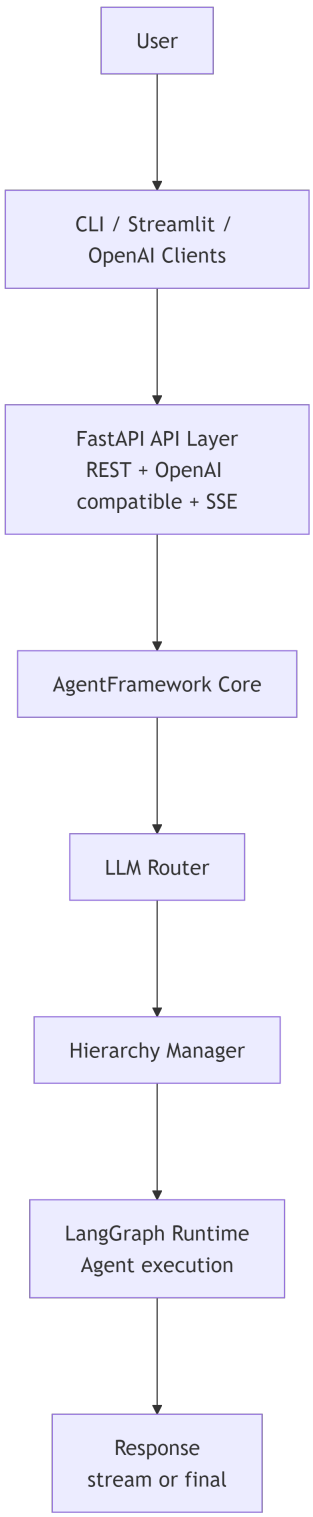


Figure 4.1: Framework overview showing external interfaces, API layer, and core runtime execution

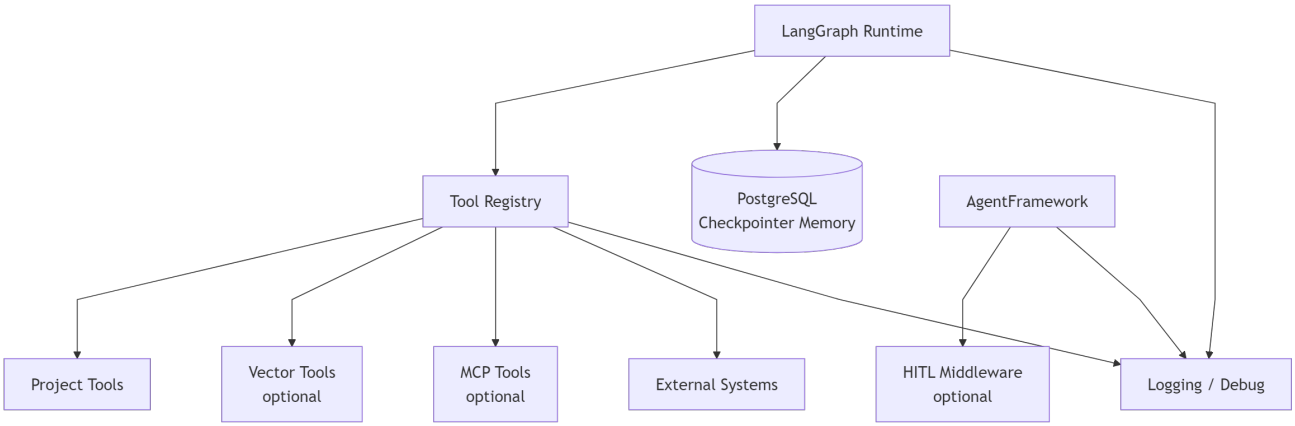


Figure 4.2: Framework supporting components including tool registry, memory management, and operational services

4.2.1 Configuration Driven Architecture

Agents, permissions, and execution constraints are defined declaratively through configuration rather than hard coded logic. The architecture assumes that system behavior will evolve over time and must be adjustable without changes to the framework core.

This principle enables reuse across projects and supports rapid adaptation to new domains.

4.2.2 Separation of Responsibilities

The framework separates orchestration, agent execution, memory management, tool integration, and user interaction. Agents focus on reasoning and decision making, while the orchestration layer manages execution flow and coordination.

This separation reduces coupling and makes system behavior easier to understand and debug.

4.2.3 Explicit Control Flow

Execution decisions such as agent routing, tool invocation, and approval checks are modeled explicitly rather than embedded implicitly in prompts. This allows execution paths to be inspected, logged, and reasoned about.

Explicit control flow is a central architectural choice that directly supports auditability.

4.2.4 Auditability as a First Class Concern

The architecture treats auditability as a core requirement. Persistent state, execution traces, and approval records are part of the system design rather than optional extensions.

This principle reflects the need for traceability in applied and industrial contexts.

4.3 Layered Architecture Description

The framework consists of several logical layers, each with a clearly defined role.

4.3.1 User Interaction Layer

The user interaction layer provides entry points to the system. These include command line interfaces, web based interfaces, and external clients interacting through standardized APIs. This layer is responsible only for input and output handling.

No agent logic or orchestration decisions are implemented at this level.

4.3.2 API and Compatibility Layer

The API layer exposes the framework functionality through REST based endpoints. In addition to internal APIs, the framework provides an OpenAI compatible interface that allows existing client tools to interact with the system without modification.

This compatibility layer reduces integration friction and enables reuse of existing client ecosystems.

4.3.3 Orchestration Layer

The orchestration layer is the central coordination component of the framework. It is responsible for routing requests to agents, managing execution flow, invoking tools, and enforcing approval requirements.

Rather than delegating these responsibilities to agents, the architecture centralizes orchestration to ensure consistent and inspectable behavior.

4.3.4 Agent Execution Layer

The agent execution layer contains the individual agents that perform reasoning and decision making. Each agent is configured with a role, a set of allowed tools, and execution constraints.

Agents do not manage system state directly. They operate within the execution context provided by the orchestration layer.

4.3.5 Persistence and Memory Layer

Persistent storage supports conversation memory, execution traces, and approval records. This layer enables post execution analysis, reproducibility, and auditability.

The architecture assumes that memory is shared across agents within a session.

4.3.6 Tool and Integration Layer

The tool layer provides controlled access to external systems such as document processing services, data sources, and communication interfaces. Tools are invoked only through the orchestration layer.

This design prevents uncontrolled side effects and makes external interactions observable.

4.4 Deployment and Runtime Topology

While the layered architecture describes logical responsibilities, the deployment topology describes how components are executed at runtime.

The architecture supports multiple interaction modes simultaneously. Web interfaces, command line clients, and OpenAI compatible clients all communicate with the same execution backend. Persistent storage and vector based retrieval services are shared across all execution paths.

4.5 Hierarchical Multi-Agent Architecture

The framework supports hierarchical multi-agent systems. In this architecture, a coordinating agent is responsible for delegating tasks to specialized agents and aggregating their results.

Hierarchy is enforced by the orchestration layer rather than by informal conventions. Agents can delegate only to authorized sub agents, and delegation paths are recorded as part of the execution trace.

This architectural choice reduces coordination complexity and aligns with established patterns in multi agent system design.

4.6 Knowledge Access and Retrieval Architecture

To address scalability concerns, the architecture integrates retrieval based knowledge access. Documents are indexed into vector representations and queried dynamically during execution.

Rather than passing full documents to agents, only relevant segments are retrieved and included in the execution context. This approach reduces token usage and supports large document collections.

Retrieval is treated as a system service rather than an application specific feature.

4.7 Human Oversight Architecture

Human oversight is integrated directly into the architectural execution model. Approval steps are modeled as part of the execution flow and are enforced by the orchestration layer.

Approval requests are handled asynchronously to avoid blocking system execution. Approval decisions are persisted and associated with specific execution steps, enabling traceability and accountability.

This architecture allows selective human oversight, where only specific actions require approval.

4.8 Summary

This chapter presented the architecture of the proposed framework. By combining layered separation of concerns, explicit orchestration, persistent state management, and integrated human oversight, the architecture provides a foundation for building scalable and auditable LLM-based multi-agent systems.

Beyond enumerating the individual layers, the architecture is intentionally designed around two integrative ideas: (i) *explicit control flow* and (ii) *traceable side effects*. Control flow is centralized in the orchestration and execution layers, which prevents prompt-level logic from implicitly deciding when to call tools, persist state, or trigger external actions. Side effects (e.g., sending emails, writing files, retrieving documents) are mediated through the tool layer and recorded as structured events, enabling auditability and deterministic replay at the session level.

This separation is also what makes the framework reusable across domains: project-specific behavior is expressed through configuration (agent topology, tool

access policies, routing rules) and reusable services (memory, retrieval, approvals), rather than hard-coded orchestration. The implementation chapter therefore focuses on *how* these architectural decisions are realized in code, and how the same core runtime can be instantiated for two distinct industrial use cases without changing the underlying execution model.

The next chapter describes how this architecture is realized in practice and how it is instantiated in two concrete projects using the same framework core.

5 Design and Implementation

This chapter describes the concrete implementation of the proposed framework and explains how the architectural decisions presented in Chapter 4 were realized in software. The focus is on engineering aspects such as modularity, configuration-driven behavior, and robustness under real execution conditions. Rather than presenting abstract capabilities, this chapter documents how these capabilities are implemented, constrained, and reused across projects.

The framework was implemented iteratively following an agile engineering approach. Requirements were refined during development based on implementation feedback and operational constraints, while maintaining architectural consistency. Design decisions were evaluated not only for correctness but also for maintainability, debuggability, and suitability for controlled deployments.

5.1 Implementation Strategy

The framework is implemented as a long running execution service that exposes a consistent interface to different clients. At runtime, the framework loads agent configurations, initializes shared services such as memory and retrieval, and waits for incoming requests.

At the implementation level, all incoming requests are normalized at the API boundary before execution begins. Independent of the interaction mode, each request is transformed into a unified internal representation consisting of a session identifier, message payload, and configuration context. This normalization ensures that downstream components operate independently of the client interface and allows the execution pipeline to be tested and instrumented consistently.

A key design decision is that agents do not communicate with each other directly. Instead, all coordination is handled by the orchestration component. This makes execution flow explicit and prevents hidden dependencies between agents. This constraint is enforced in the implementation by restricting agent interfaces to stateless input and output functions and by preventing direct access to shared memory or peer agent references.

5.2 API Layer and Execution Control

The framework exposes a unified API layer that serves as the primary execution interface for all clients, including command-line tools, web interfaces, and OpenAI-compatible clients. API endpoints are responsible for initializing execution sessions, invoking agent orchestration, and streaming or returning execution results.

Execution control is centralized in the API layer, ensuring consistent handling of session state, error propagation, and execution metadata. This design prevents client-side logic from bypassing safety or orchestration constraints. The API does not expose internal agent implementations but instead operates on execution contracts defined at the framework level.

Detailed API schemas and representative request and response formats are documented in Appendix B, providing concrete evidence of how execution is initiated and controlled.

5.3 Framework Implementation Overview

Agent orchestration in the framework is defined declaratively through configuration files rather than hard-coded control logic. Agent hierarchies, routing relationships, and tool availability are specified externally and loaded at runtime. This design allows the same framework core to support multiple projects with different agent structures without modification to the execution engine.

Each agent definition specifies its role, accessible tools, and position within a hierarchical routing structure. Parent agents are responsible for delegating tasks to specialized child agents based on routing strategies defined in configuration. This approach reduces coupling between agent logic and orchestration behavior, at the cost of increased reliance on correct configuration.

Representative configuration artifacts illustrating agent hierarchies and runtime options are provided in Appendix A, which substantiates the configuration-driven design described in this section.

Each incoming request is transformed into an internal execution context. This context contains conversation state, metadata, and references to shared services. The execution context is passed between components rather than reconstructed at each step, which ensures consistency and traceability.

Routing decisions, tool invocations, and approval checks are treated as first class execution events. Each event is recorded as part of an execution trace that can be inspected after completion.

At this point in the chapter, the generic execution lifecycle is introduced.

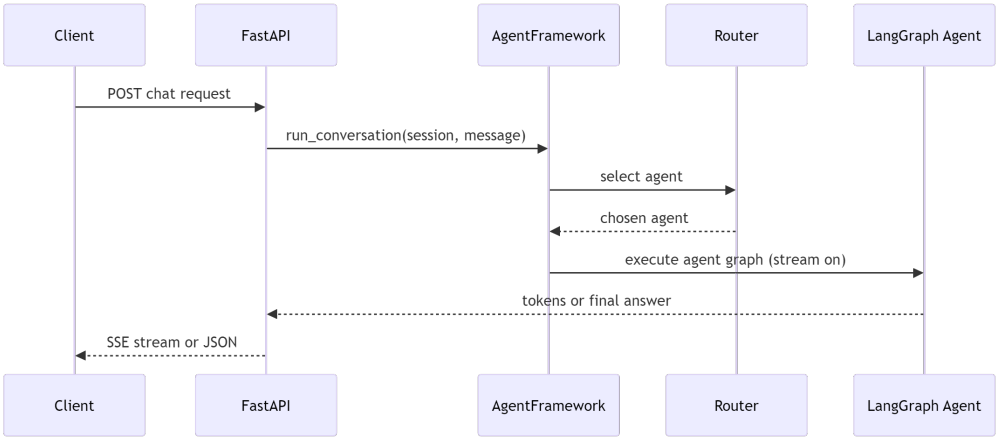


Figure 5.1: End to end request flow showing routing and agent execution

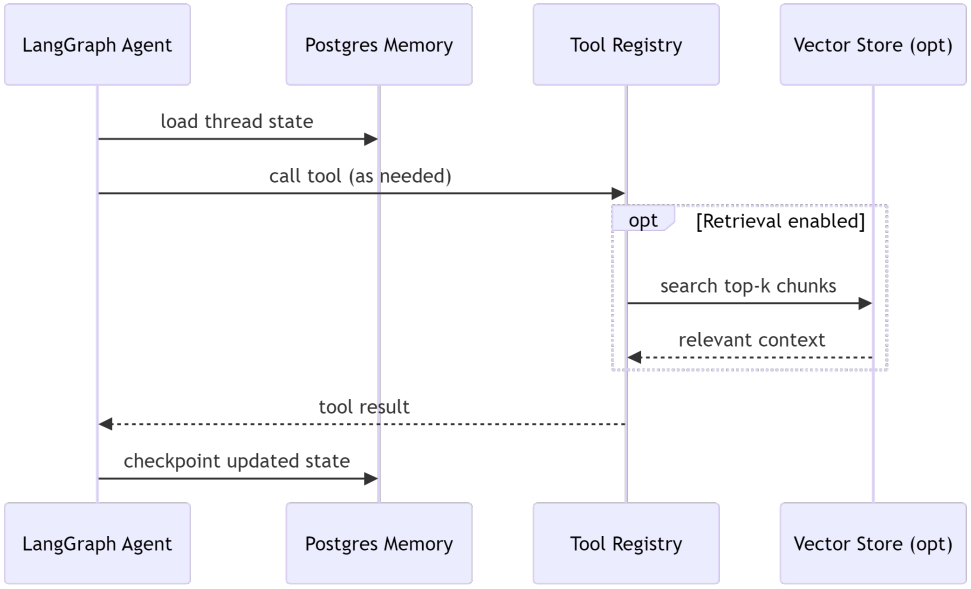


Figure 5.2: Tool invocation and memory interaction during execution

Figures 5.1 and 5.2 illustrate the common execution flow that is reused by both projects described later in this chapter.

5.4 Agent Orchestration and Hierarchy

The framework supports hierarchical agent systems through enforced delegation rules. A coordinating agent is responsible for task interpretation and delegation, while specialized agents focus on narrow responsibilities.

Hierarchy is not implicit. Delegation paths are defined in configuration and enforced by the orchestration layer. Agents cannot invoke other agents arbitrarily. This constraint reduces complexity and ensures predictable execution.

The orchestration layer maintains control over execution order, handles branching logic, and aggregates intermediate results. Agents remain focused on reasoning and decision making within their assigned scope.

Agent execution is implemented as an explicit state machine rather than an implicit conversational loop. Each agent invocation advances the system through a well-defined execution state, allowing intermediate states to be checkpointed and inspected.

This design was chosen to address failure recovery and traceability concerns. If an agent execution fails due to tool errors, model timeouts, or rejected human approvals, the framework can resume execution from the last consistent checkpoint rather than restarting the entire interaction. The trade-off of this approach is additional state-management complexity and storage overhead, which was accepted in favor of improved robustness and debuggability.

This approach allows complex workflows to be expressed through composition of simple agents without embedding control logic inside prompts.

Step-by-step orchestration flow. To make orchestration concrete, the runtime executes each user request as a deterministic pipeline that combines routing, state handling, and graph execution:

1. **Request ingestion (API layer).** A client sends a chat request to the FastAPI endpoint. The API normalizes the request into an internal message format and binds it to a *session/thread identifier* (Figure 5.1).
2. **Conversation entry point (framework runtime).** The API invokes the framework entry method (e.g., `run_conversation(session, message)`), which becomes the single gateway for execution, logging, and policy enforcement.
3. **Routing and agent selection.** The router evaluates the request against configured routing rules (agent capabilities, allowed tools, project context) and selects an agent graph to execute. The selected agent is returned to the runtime as a concrete graph definition.
4. **State restoration (checkpoint/memory).** Before executing the next step, the runtime loads the latest checkpoint for the thread (if enabled). This yields a reproducible state snapshot (messages, intermediate artifacts, approvals, tool results).

5. **Graph execution (LangGraph state machine).** The chosen agent runs as an explicit state machine. Each node produces a structured state update and may emit tool calls, retrieval requests, or approval requests.

6. **Tool execution via registry.** Tool calls are never executed directly by the LLM. Instead, calls are validated and dispatched through the tool registry (Figure 5.2), which enforces allow-lists, input schemas, and logging of inputs/outputs.

7. **Optional retrieval augmentation.** If enabled, retrieval is invoked as a shared service to fetch top-k relevant chunks, which are attached to the agent state for grounded generation (Section 5.8).

8. **Approval gating (HITL).** For safety-critical actions (e.g., sending an email), the execution pauses (blocking CLI) or creates a pending approval (async web), and resumes only after a human decision is recorded (Figure 5.3).

9. **Streaming and completion.** Tokens are streamed (SSE) or returned as a final JSON response. Independently, the runtime persists the final state and the full trace so that the execution can be inspected after completion.

This explicit orchestration design is the core mechanism that ensures predictable execution, controlled side effects, and post-hoc auditability across different multi-agent applications.

5.5 Shared Memory and Execution State

All agents participating in a session operate over a shared execution context. Conversation messages, intermediate outputs, and metadata are stored centrally and made available to subsequent execution steps.

Persistent execution state is managed through a checkpoint-based memory mechanism. At defined execution boundaries, the framework records intermediate state to a database-backed storage system. This enables inspection of partial execution, recovery after interruption, and post-hoc analysis of agent behavior.

Memory persistence is configured externally and can be enabled or disabled per deployment. While this mechanism improves auditability and robustness, it introduces storage overhead and requires careful management of retention policies. Configuration excerpts related to memory backends and checkpointing behavior are provided in Appendix A.

Persistent memory is implemented using a relational checkpoint store that records agent state transitions, tool invocations, and intermediate outputs. Each

checkpoint is associated with a unique execution identifier, enabling precise reconstruction of prior system behavior.

While this approach significantly improves auditability, it introduces non-trivial storage growth over time. In longer-running deployments, log compaction and retention policies become necessary to control disk usage. This limitation was observed during extended test runs and is considered an operational concern rather than a design flaw.

The framework avoids passing full histories to agents indiscriminately. Instead, the orchestration layer determines which context elements are relevant for a given step.

5.6 Tool Integration and Controlled Execution

External tools are integrated through a central registry. Each tool is described using structured metadata that defines its purpose, inputs, and outputs. Agents are granted access only to tools explicitly assigned to them.

When an agent requests a tool invocation, the orchestration layer intercepts the request, executes the tool, and records the result. Tool inputs and outputs are logged as part of the execution trace.

Tools are registered explicitly through a centralized tool registry that defines tool interfaces, required parameters, and expected outputs. During execution, agents may only invoke tools that have been explicitly registered for their role, enforcing a strict separation between reasoning and side-effecting actions.

This constraint prevents uncontrolled tool usage and limits the blast radius of agent errors. However, it also introduces an engineering limitation: adding new tools requires updating both the registry configuration and associated validation logic. This deliberate friction was considered acceptable to reduce accidental misuse of critical tools such as email dispatch or external system access.

This design ensures that all external actions are observable and that unintended side effects are minimized.

5.7 Human Oversight and Approval Workflow

Human-in-the-loop control is implemented as a first-class framework component rather than an agent-level convention. Safety-critical tools are marked as protected and require explicit human approval before execution may proceed. Approval requests suspend execution deterministically until a decision is recorded.

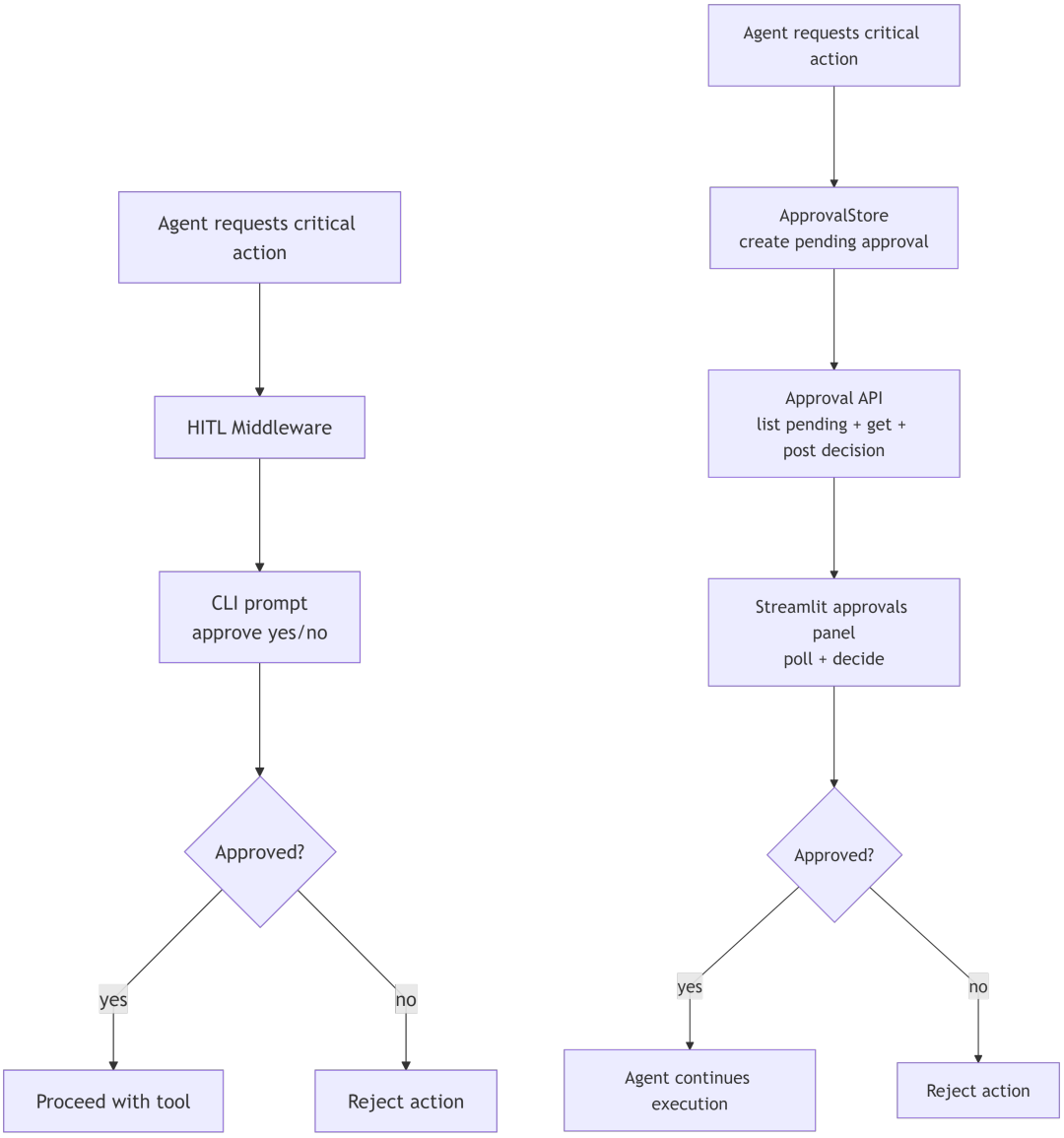
The framework supports both synchronous and asynchronous approval modes, allowing different interaction patterns depending on deployment context. These mechanisms are enforced uniformly across all agents and cannot be bypassed through prompt manipulation or tool misuse.

Concrete approval records and state transitions generated during execution are documented in Appendix C, providing auditable evidence of enforced human oversight.

Human-in-the-loop controls are implemented as a middleware layer that intercepts safety-critical actions before execution. Depending on the deployment mode, approvals are either blocking, as in command-line interfaces, or asynchronous, as in web-based deployments.

An important engineering trade-off emerged during implementation: blocking approvals provide strong safety guarantees but significantly increase end-to-end latency, whereas asynchronous approvals improve responsiveness but introduce complexity in execution resumption. Both modes are supported to allow system operators to select the appropriate balance between safety and usability.

At this point in the chapter, the human oversight mechanism is introduced.



(a) CLI blocking approval workflow for synchronous human oversight

(b) Web async approval workflow for non-blocking human oversight

Figure 5.3: Human-in-the-loop approval workflows

Figures 5.3a and 5.3b illustrate how the framework enforces human approval for safety-critical actions.

What happens first: an agent proposes a critical action (e.g., *send support email*, *execute external command*). Instead of executing the tool directly, the runtime routes the request into an approval gate.

Data flow and control flow: includes two modes. In the *blocking CLI* flow (Figure 5.3a), the runtime pauses the state machine at a well-defined checkpoint and prompts the operator for a yes/no decision. The operator decision is written back into the execution state and the graph continues deterministically with either (i) tool execution (approved) or (ii) safe termination/rejection (rejected). In the *asynchronous web* flow (Figure 5.3b), the runtime creates a pending approval record and returns control to the UI immediately. A web panel polls pending approvals and posts the operator decision via an approval API endpoint. Once recorded, the runtime resumes the paused execution from the checkpoint.

Why this matters: the design prevents prompt-level manipulation from bypassing approvals and ensures that all critical actions have an explicit operator decision attached to the execution trace. This balances automation with accountability, which is required for industrial environments where actions may have operational or compliance impact.

5.8 Knowledge Retrieval and Context Management

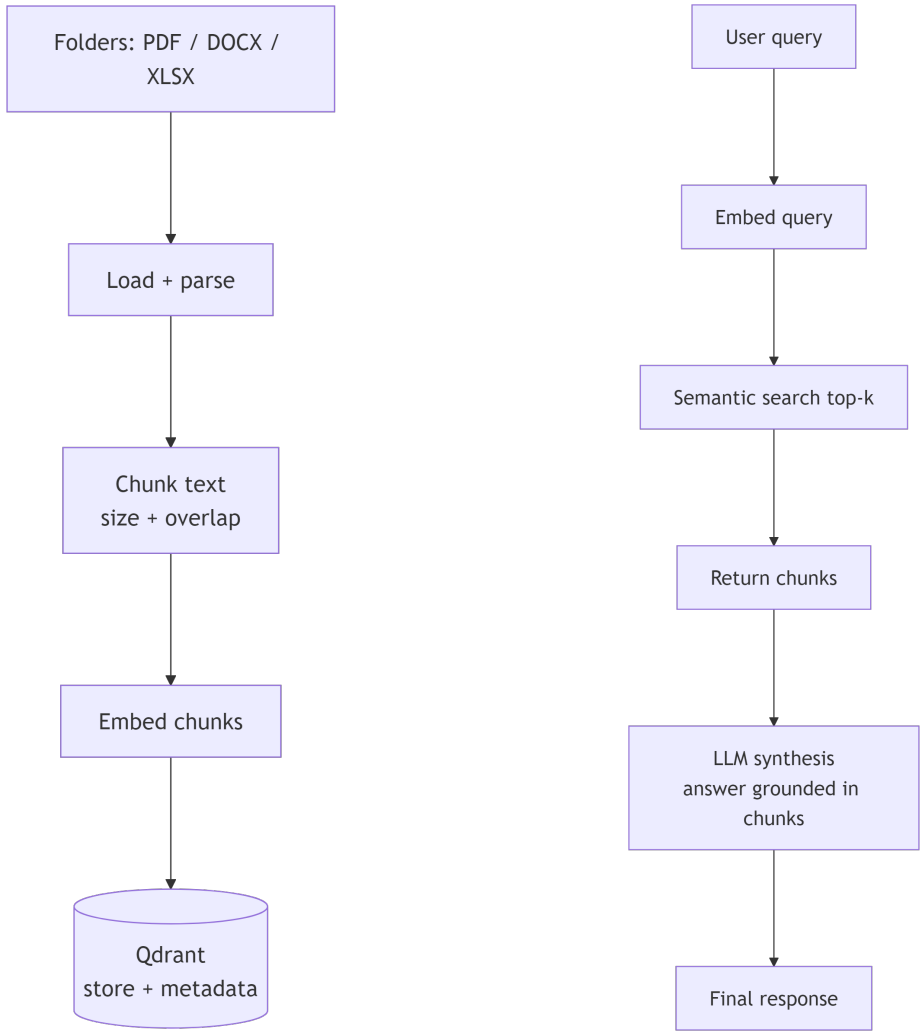
To support scalability, the framework integrates vector based retrieval for document access. Documents are indexed once and retrieved dynamically during execution.

Rather than loading entire documents into the language model context, only relevant segments are selected. This reduces token usage and improves consistency.

Knowledge retrieval is implemented as an optional capability rather than a mandatory component of the framework. When enabled, documents are chunked using a fixed-size window with overlap to balance retrieval recall and embedding cost.

A key limitation observed during implementation is that retrieval quality degrades when document structure is highly irregular, such as in poorly formatted spreadsheets or scanned PDFs. In such cases, manual preprocessing or alternative retrieval strategies are required. This limitation informed the decision to keep retrieval optional and project-specific rather than enforcing it as a core framework dependency.

Retrieval is implemented as a shared service and reused across agents and projects.



(a) Indexing pipeline: document processing and embedding generation

(b) Query pipeline: retrieval and context construction

Figure 5.4: Vector store pipeline architecture

Figure 5.4 shows the retrieval pipeline in two phases.

Indexing (left): the system loads documents from project folders (PDF/DOCX/XLSX), parses them into text, and splits the content into chunks using a configurable chunk size and overlap. Each chunk is embedded and stored in the vector store together with metadata (document source, section/page identifiers, timestamps). This metadata is later used for traceability and for reconstructing which evidence supported a given answer.

Query-time retrieval (right): when an agent requires external context, the user query (or an internally generated sub-query) is embedded and used for semantic top-k search. The retrieved chunks are returned as a compact context bundle and injected into the agent state. The final LLM response is then synthesized *grounded in the retrieved chunks*, rather than relying on full-document prompting.

Why this design matters: retrieval is implemented as an optional shared service so that projects can enable it when document scale makes full-context prompting infeasible, while preserving a consistent execution model. It also improves reproducibility: both the retrieved evidence and the model output are traceable in the execution trace.

5.9 Case Study Implementations

These two case studies were selected because they were real, actively needed use cases within the project teams, and together they validate the same core framework under contrasting conditions: a document-heavy planning scenario (Methanol Factory Assistant) that stresses retrieval and structured reporting, and a safety-critical operational scenario (Terminal Management System Assistant) that stresses auditability, controlled side effects, and human-in-the-loop approvals.

Both case studies presented in this chapter reuse the same framework runtime, API layer, memory backend, and safety mechanisms. Differences between projects are expressed exclusively through configuration and project-specific tools. This validates the separation between framework infrastructure and application logic.

Project-specific configuration excerpts and artifacts supporting this claim are included in Appendix
refappendix:config, while execution behavior is evaluated in Chapter
refchapter:evaluation.

5.9.1 Project One: Methanol Factory Construction Analysis

The Methanol Factory Assistant is a multi-agent system designed to support decision making during the planning and construction of a methanol production facility. Large-scale industrial construction projects involve multiple specialized departments, such as electrical systems, automation, roofing, and structural components, each of which must analyze extensive requirement documents and determine whether their technical capabilities satisfy project demands. This ana-

lysis is traditionally performed manually, making the process time consuming, difficult to coordinate, and prone to inconsistencies across departments.

The proposed assistant addresses this challenge by acting as a centralized AI-driven coordination system that distributes construction requirements to specialized agents, each representing a specific department. These agents analyze requirement documents and compare them against department-specific capability descriptions using semantic document understanding. The result is a structured and transparent assessment that identifies which requirements can be fulfilled, which require deviations, and which must be excluded.

This interaction can be compared to the construction of a residential house, where an architect coordinates the project while electricians, plumbers, and builders contribute domain-specific expertise. Similarly, in the proposed system, a coordinator agent manages the overall workflow, while specialized agents provide focused analyses within their respective domains. This approach enables consistent, department-specific evaluations, reduces coordination overhead, and supports informed planning decisions in complex industrial projects.

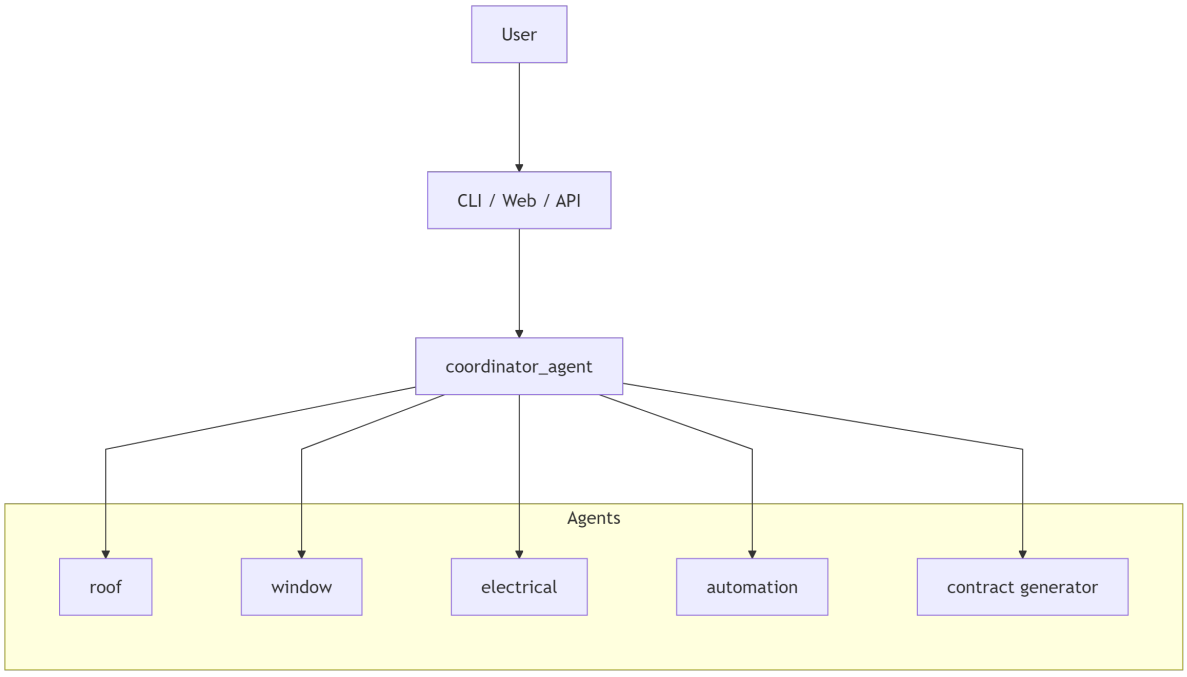


Figure 5.5: Methanol factory system agent topology and hierarchy

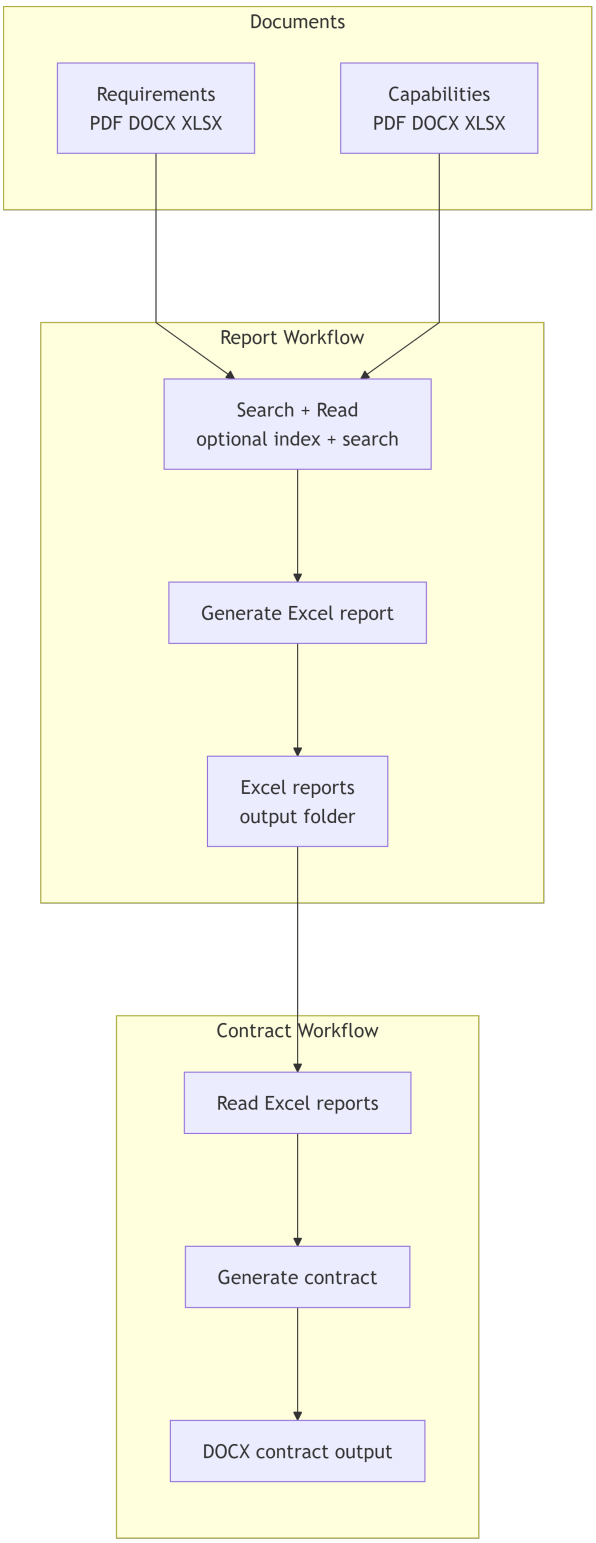


Figure 5.6: Methanol factory analysis workflows and data flow

Figures 5.5 and 5.6 illustrate the internal agent structure and information flow of the Methanol Factory Assistant. Each agent represents a specific department involved in the factory construction process, such as electrical systems or automation.

What happens first: the user submits a construction-related request (or selects a department for analysis). The coordinator agent interprets the request and activates the corresponding specialist agent based on the configured hierarchy.

What data flows where: the specialist agent retrieves relevant sections from the construction requirement documents and the department capability files, performs a semantic comparison, and categorizes each requirement as accepted, deviated, or excluded. The classification and rationale are stored as structured artifacts.

Why this design matters: the output is a transparent Excel report that documents each requirement alongside its classification and explanation. This supports traceability and consistency across departments, which are critical quality attributes in large-scale industrial construction projects.

5.9.2 Problem Context

Construction requirements and capability descriptions are provided in heterogeneous document formats. The task requires interpreting requirements and matching them against department specific capabilities.

Manual analysis is time consuming and error prone, while rule based automation lacks flexibility.

5.9.3 Agent Structure

The system is implemented as a hierarchical multi-agent system. A coordinating agent manages task flow and delegates work to specialized department agents.

Each department agent retrieves relevant document segments, interprets requirements, and produces structured outputs. Results are aggregated into standardized reports.

No project specific orchestration logic is added to the framework. Differences are expressed entirely through configuration and tools.

5.9.4 Output Generation

The system generates structured Excel outputs that categorize requirements and provide explanatory comments. These outputs are then used by a contract generation agent to produce formal documents.

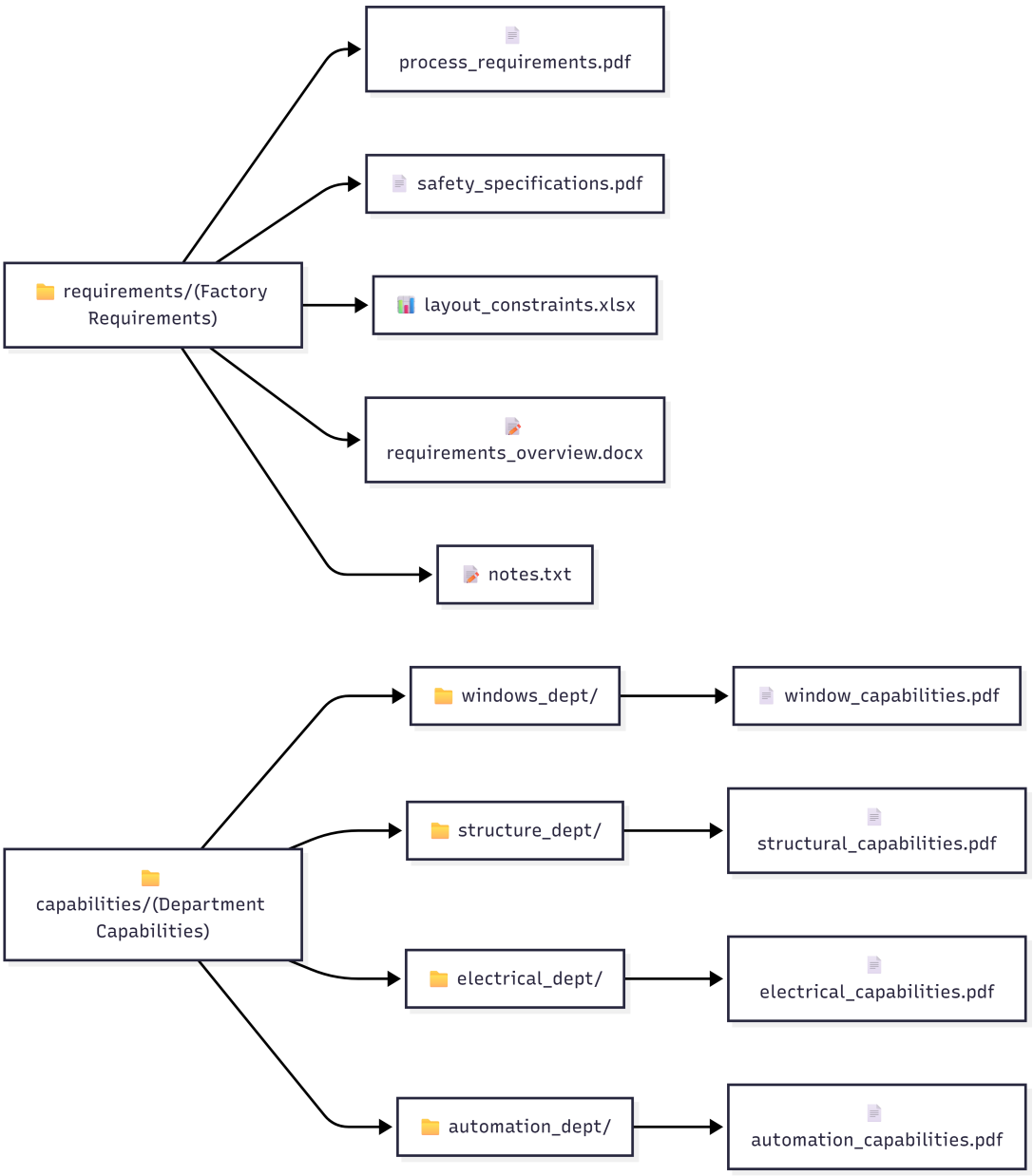


Figure 5.7: Methanol Factory Assistant: Example input requirement excerpt (anonymized)

Requirement ID/Description	Accepted	Deviated	Excluded	Comments	Reference (Page)
The contractor shall install power supply circuits and cabinet wiring to support the distributed control system and other automation equipment provided by the automation system supplier.	✓			We can fully provide power supply circuits and cabinet wiring as per the requirement.	Req: p.1, Cap: p.1
The contractor shall design and install general lighting systems for buildings, indoor working areas, office areas and general outdoor non process areas.	✓			We can provide lighting systems for all specified areas, including compliance with illumination levels.	Req: p.1, Cap: p.2
The contractor shall install cable ladders, cable trays and conduits in buildings and process areas.	✓			We have the capability to install cable ladders, trays, and conduits as specified.	Req: p.1, Cap: p.3
The contractor shall design and supply medium voltage switchgear for the eleven kilovolt main distribution system.		✓		We can design and supply medium voltage switchgear for the specified distribution system.	Req: p.1, Cap: p.4
The contractor shall assist the client in preparing grid connection documentation requested by the utility.	✓			We can assist with grid connection documentation preparation and provide technical details.	Req: p.1, Cap: p.5
The contractor shall prepare inspection and test plans for installation of electrical panels, cable trays, wiring and terminations.			✓	We can prepare inspection and test plans for all specified installations.	Req: p.1, Cap: p.6

Figure 5.8: Methanol Factory Assistant: Generated Excel assessment report (anonymized)

Figures 5.7 and 5.8 illustrate representative input documents and generated outputs using anonymized demonstration data.

This demonstrates how outputs of one agent become inputs to another without manual intervention.

5.9.5 Project Two: Terminal Management System Assistant

The Terminal Management System (TMS) is an industrial software solution developed by Siemens Energy and used by Saudi Aramco to manage oil terminal operations. The system controls and monitors loading bays where trucks collect oil and communicates with preset controllers, channels, and field sensors. For security and reliability reasons, the system operates locally within the terminal infrastructure and is not connected to the internet.

In the existing operational workflow, when an issue occurs, such as a failure at a specific bay or preset, the terminal operator observes an error message on the Human Machine Interface. The operator reports the issue to a local support team, which then contacts Siemens Energy technical support. Siemens Energy engineers typically request diagnostic log files, guide the operator through manual log generation, and analyze the data remotely. This multi-step process can take several weeks, leading to prolonged system downtime and significant financial impact.

The proposed Terminal Management System AI Assistant is a locally deployed multi-agent system designed to significantly reduce this response time. When an operator encounters an issue, for example at bay 9 and preset 9, the operator can directly query the assistant. The system automatically identifies the relevant configuration, retrieves and analyzes trace log files, and determines potential root causes based on known failure patterns and system behavior.

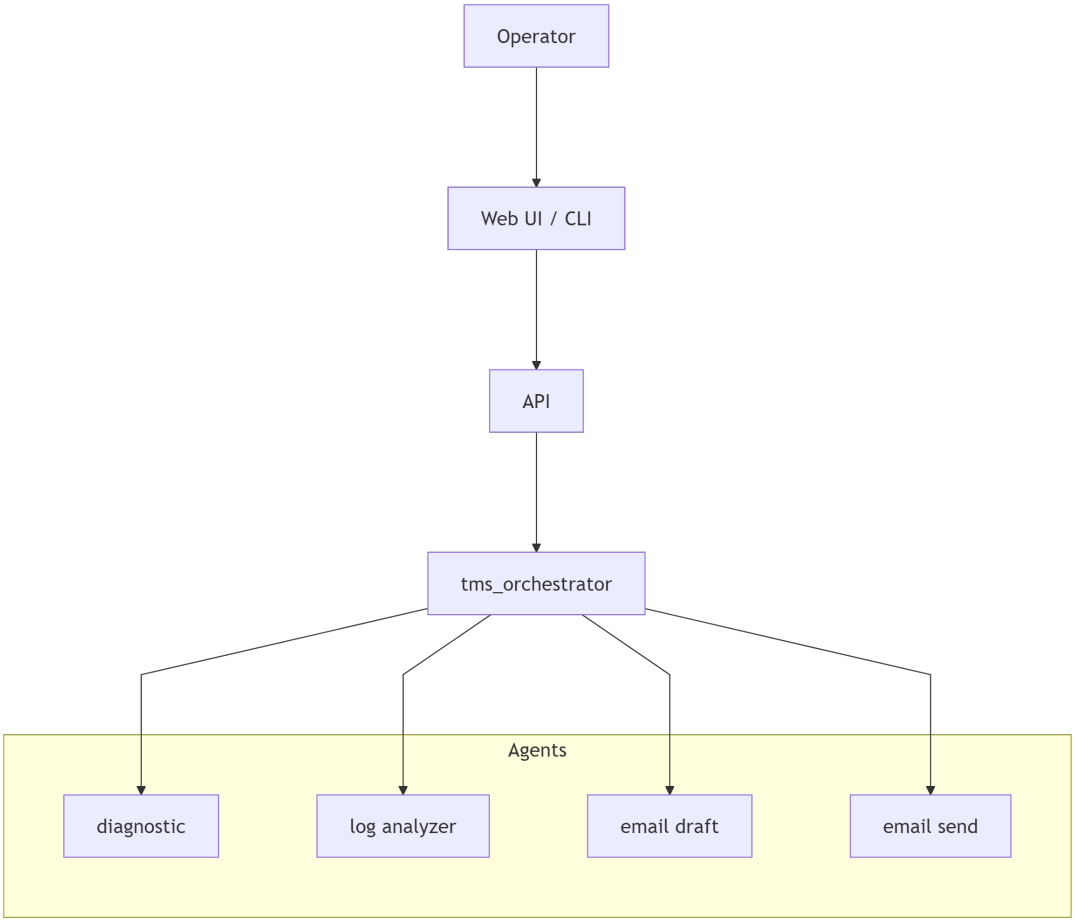


Figure 5.9: TMS agent topology showing diagnostic and communication agents

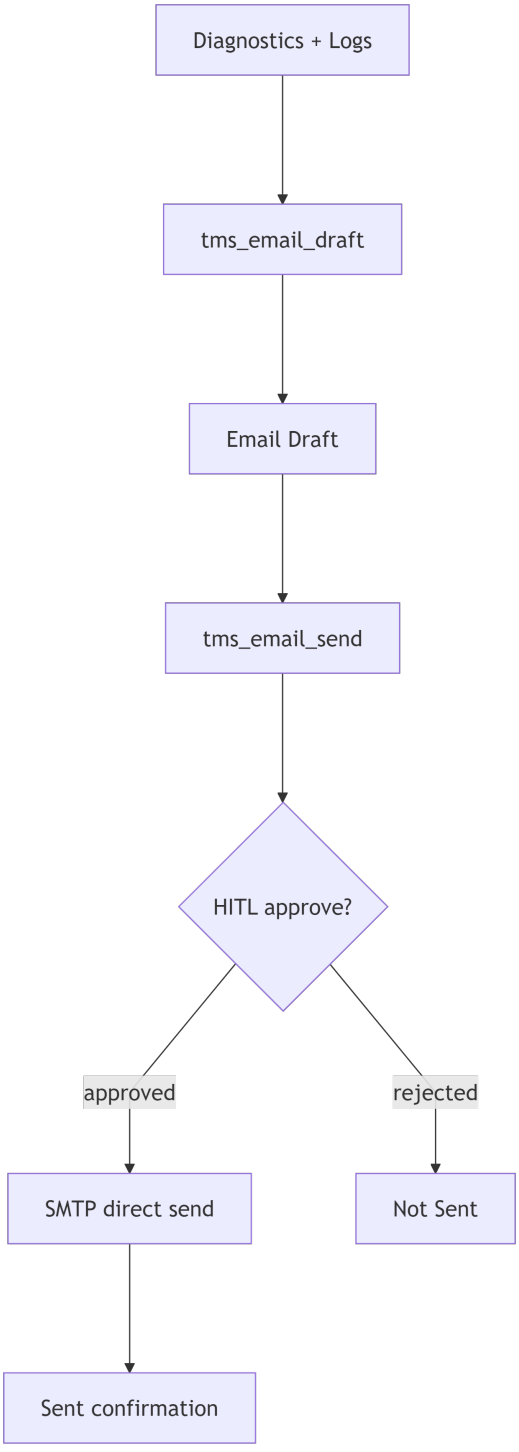


Figure 5.10: TMS email sending workflow with human oversight for safety critical actions

Figures 5.9 and 5.10 illustrate the agent structure and the separation between diagnostic reasoning and externally visible actions requiring human approval.

5.9.6 Problem Context

Operators diagnose issues related to presets, bays, and channels using configuration files and logs. The system must also support drafting and sending support emails.

Sending emails has operational impact and therefore requires human oversight.

5.9.7 Agent Structure and Safety Controls

The system consists of diagnostic agents, an email drafting agent, and an email sending agent. The orchestrator routes requests based on intent.

Only the email sending agent requires human approval. Drafting actions are considered safe and execute automatically.

This selective application of human oversight improves usability while maintaining control.

5.9.8 Inputs and Outputs

The primary inputs to the system include Terminal Management System configuration files (e.g., XML mappings between bays, presets, and channels), trace log files generated by the system, and operator-provided context such as bay numbers and preset identifiers.

Based on these inputs, the assistant produces a structured diagnostic explanation that identifies likely causes and recommends corrective actions. In addition, it generates a draft support email addressed to Siemens Energy containing a concise diagnostic summary and references to the relevant artifacts (e.g., trace logs) to enable immediate expert review.

Before any email is dispatched, a human-in-the-loop approval mechanism requires explicit operator confirmation. This preserves accountability and operational control while still benefiting from automated analysis and communication support.

5.9.9 Transparency and Traceability

All diagnostic steps, tool invocations, and approval decisions are logged. Email content is presented to the user before approval.

Figures 5.11 and 5.12 show an example diagnostic input, the generated email draft, and the corresponding approval decision.

```
17:12:13.372 tmac_101_01s0  204     3 T: CUtiSocket::_OnReadComplete -->  ERROR_OPERATION_ABORTED ignored
17:12:13.372 tmac_101_01s0  204   255 T: CUtiSocket::SocketMainLoop --> WSAWait.. state: VALID, timeout = 0
17:12:13.372 tmac_101_01    203   255 T: CUtiCmTimer::Timeout -- timer call for BTN 215100 in 243 sec
17:12:13.372 tmac_101_01    203   255 T: CUtiCmTimer::Timeout -- timer call for BTN 215100 in 243 sec
17:12:13.372 tmac_101_01    203   255 T: CUtiCmTimer::Timeout -- timer call for BTN 215100 in 243 sec
17:12:13.372 tmac_101_01    203   255 T: CUtiCmTimer::Timeout -- timer call for BTN 215100 in 243 sec
17:12:13.372 tmac_101_01    203   255 T: CUtiCmTimer::Timeout -- timer call for BTN 215100 in 243 sec
17:12:13.372 tmac_101_01    203   255 T: CUtiCmTimer::Timeout -- timer call for BTN 215100 in 243 sec
17:12:13.372 tmac_101_01    203   255 T: CUtiCmTimer::Timeout -- timer call for BTN 215100 in 243 sec
17:12:13.372 tmac_101_01    203   255 T: CUtiCmTimer::Timeout -- timer call for BTN 215100 in 244 sec
17:12:13.372 tmac_101_01    203   255 T: CUtiCmTimer::Timeout -- timer call for BTN 215002 in 1 sec
```

Figure 5.11: Terminal Management System Assistant: Trace log/configuration excerpt (anonymized)

Example Support Email

To: Siemens Energy Support Team
From: Saudi Aramco Operations <tms.support.agent@siemens-energy.com>
Subject: TMS Issue – Bay 9 Preset 9
Date: 2025-12-03 09:59:05

Issue: Socket timeout detected on Channel 01, Preset Controller 9, Bay 9.

Root Cause Analysis:

- Network connectivity issue between TMS server and preset controller
- Port 502 (Modbus) may be blocked or controller unresponsive

Recommended Actions:

1. Verify network connectivity and port 502 status
2. Inspect physical connections and controller power
3. Review attached trace log: Channel_3_autoTraceExport_7312025_5-20-29_PM.txt

Urgency: Please respond within 4 hours.

Best regards,
Saudi Aramco Operations Team

Figure 5.12: Terminal Management System Assistant: Generated diagnostic email draft (anonymized)

This transparency supports trust and operational safety.

By automating diagnostic log analysis and support communication directly at the terminal, the assistant removes several manual steps in the escalation process. Siemens Energy support engineers receive complete, structured diagnostic information earlier, enabling faster issue resolution. As a result, downtime can be reduced from weeks to hours or days, leading to substantial cost savings and improved operational efficiency.

5.10 Cross Project Observations

Across both projects, the same framework core is reused without modification. Differences arise from agent configuration, tools, and data sources rather than changes to orchestration logic.

This demonstrates that the framework supports reuse, scalability, and consistent behavior across domains.

5.11 Implementation Constraints and Observed Limitations

Several implementation constraints were identified during framework development. First, agent orchestration complexity grows non-linearly with the number of interacting agents, making deeply nested hierarchies harder to reason about. Second, extensive logging and checkpointing improve auditability but incur measurable performance and storage costs.

Finally, while configuration-driven design improves flexibility, it also increases the risk of misconfiguration. To mitigate this, validation checks were added, but complete prevention of configuration errors remains an open engineering challenge.

5.12 Summary

This chapter described how the proposed architecture is implemented and how it is instantiated in two different projects. The framework translates architectural principles into a reusable execution system that supports transparency, safety, and scalability.

The next chapter evaluates the framework and the two projects with respect to engineering quality attributes such as reliability, maintainability, and observability.

6 Evaluation

This chapter evaluates the developed framework from an engineering perspective. Unlike research-oriented evaluations that focus on hypothesis testing or quantitative performance benchmarks, the goal here is to assess the framework as a software system: how robustly it operates, how well it satisfies the stated requirements, and what trade-offs emerge during real usage.

The evaluation is based on the implementation experience, execution behavior in two case study projects, test results, and operational artifacts such as logs, configuration files, and approval records. Both successful behaviors and observed limitations are discussed to provide an honest assessment of the framework.

6.1 Evaluation Scope and Criteria

The evaluation focuses on the following engineering criteria:

- Functional correctness of multi-agent orchestration
- Robustness and failure handling
- Configuration effort and maintainability
- Auditability and observability
- Operational overhead introduced by safety mechanisms
- Reusability across projects

These criteria directly reflect the requirements defined in Chapter 3 and the architectural decisions described in Chapter 4.

6.2 Functional Correctness and Stability

Functional correctness was assessed by executing the framework across two independent projects: the Methanol Factory Assistant and the Technical Maintenance

Support system. In both cases, the framework successfully orchestrated multiple agents, managed shared state, and coordinated tool usage without requiring project-specific changes to the core runtime.

Stability was evaluated through repeated execution of agent workflows, including long-running conversations and multi-step tool interactions. The checkpoint-based memory mechanism ensured that intermediate execution state was preserved across steps and could be recovered after interruptions. No execution deadlocks or inconsistent agent states were observed during normal operation.

However, functional correctness was contingent on valid configuration. Misconfigured agent hierarchies or missing tool registrations resulted in early execution failures. While these failures were explicit and logged, they highlight that correctness depends strongly on configuration discipline rather than compile-time guarantees.

Representative execution traces and persisted intermediate states used to assess functional correctness are provided in Appendix E.

6.3 Robustness and Failure Handling

Robustness was evaluated by intentionally triggering failure scenarios, including tool execution errors, rejected human approvals, and unavailable external services.

The framework handled tool failures by propagating structured error states through the execution graph rather than terminating silently. In the Technical Maintenance Support system, rejected email send approvals resulted in immediate termination of the affected execution path while preserving the conversation state for inspection.

Human-in-the-loop rejections represent a deliberate failure mode rather than an error. The framework correctly enforced execution blocking or termination depending on the configured approval mode. This behavior confirms that safety constraints are enforced at the orchestration level rather than relying on agent compliance.

A limitation observed during robustness testing is that recovery from external service outages is currently manual. While failures are logged and surfaced, automated retries or fallback strategies are not implemented in the current version.

Representative failure cases, approval rejections, and corresponding execution traces are provided in Appendix C and Appendix E.

6.4 Configuration Effort and Maintainability

One of the primary engineering goals of the framework is configuration-driven extensibility. This was evaluated by measuring the effort required to adapt the framework to new projects.

Both case studies required only configuration changes to define agent hierarchies, enabled tools, and safety policies. No modifications to the framework core were necessary. This confirms that the architectural separation between runtime and project-specific logic is effective.

At the same time, configuration complexity grows with system size. Large agent hierarchies increase the risk of misconfiguration, particularly when dependencies between agents and tools are implicit. While configuration files improve flexibility, they shift some complexity from code to documentation and validation.

This trade off is acceptable for an engineering system targeting controlled deployments but may require additional tooling such as configuration validation or schema enforcement in future iterations.

Configuration artifacts illustrating agent hierarchies, memory backends, and safety settings are included in Appendix A.

6.5 Auditability and Observability

Auditability was evaluated by inspecting execution logs, approval records, and memory checkpoints generated during operation.

The logging system produces structured, timestamped entries that capture agent selection, tool invocations, approval decisions, and execution termination reasons. These logs make it possible to reconstruct execution paths after the fact without relying on agent output alone.

Checkpointed memory enables inspection of intermediate states, which is particularly valuable when diagnosing unexpected behavior or reviewing rejected actions. This capability proved essential during the evaluation of safety-critical workflows in the Technical Maintenance Support system.

Figures 6.1 and 6.2 illustrate how observability is integrated into the system architecture through structured logging and multiple output sinks.

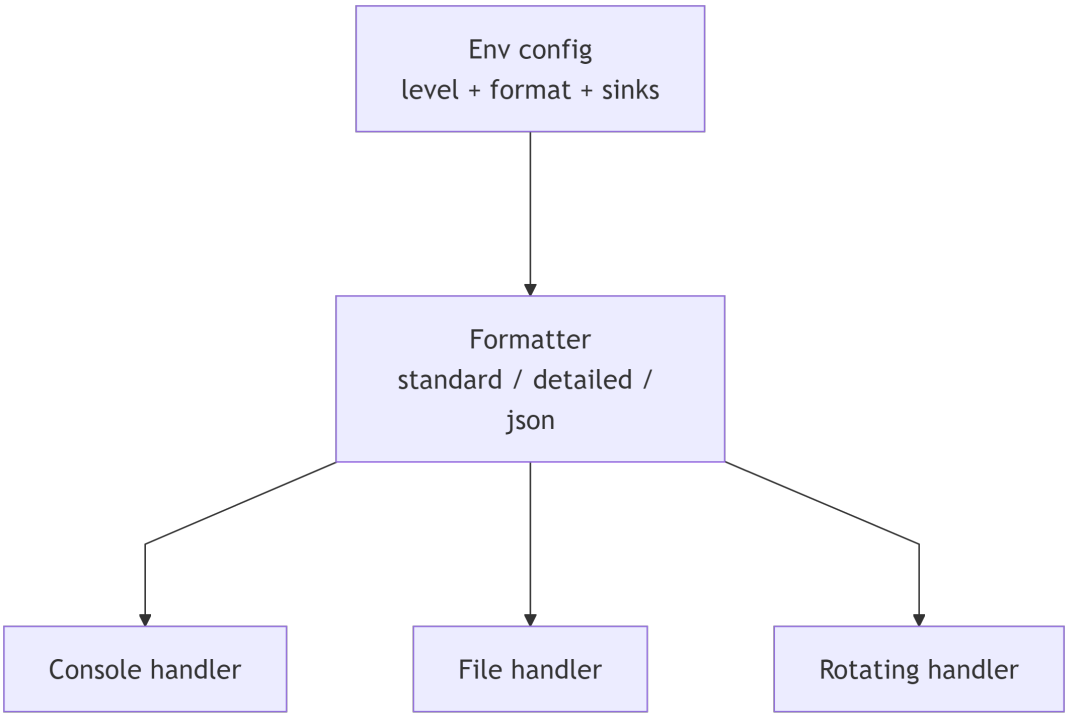


Figure 6.1: Logging configuration, format specifications, and handler architecture

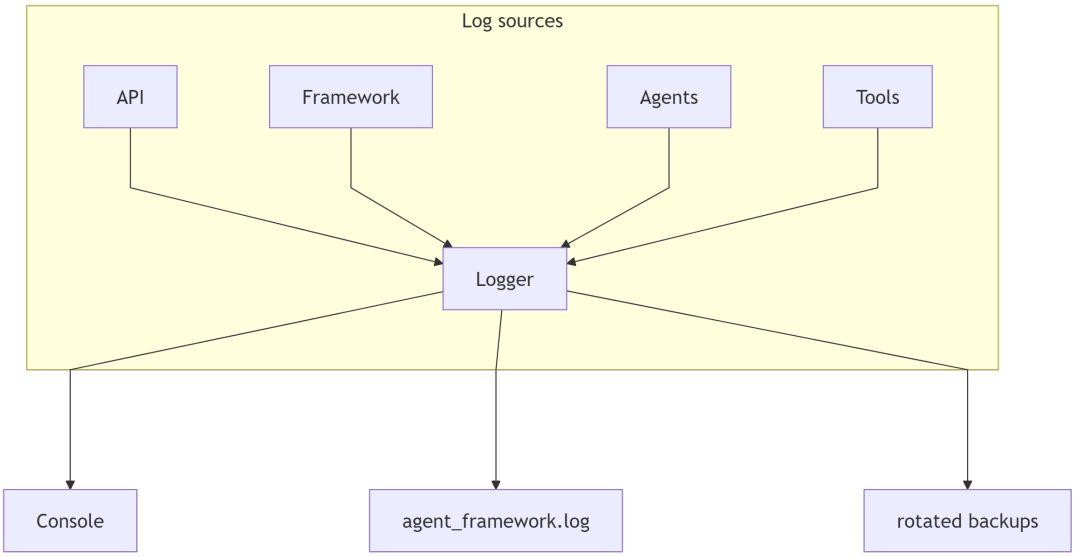


Figure 6.2: Logging data sources, logger instances, and output sinks supporting traceability

A practical limitation is log verbosity. Detailed logging is valuable during debugging and evaluation but introduces storage and readability overhead in production environments. Selecting appropriate logging levels is therefore an operational concern rather than a purely technical one.

Concrete logging examples and execution traces supporting this evaluation are presented in Appendix E.

6.6 Impact of Human-in-the-Loop Mechanisms

Human-in-the-loop mechanisms significantly improve safety for irreversible actions but introduce measurable operational overhead.

In synchronous workflows, blocking approvals increase response latency and reduce system autonomy. This is acceptable for safety-critical actions but unsuitable for high-throughput or real-time scenarios. The asynchronous approval mode mitigates this issue but introduces additional system complexity and delayed execution.

The evaluation confirms that human oversight is effective when applied selectively. Overuse of HITL mechanisms can degrade usability and throughput, emphasizing the importance of careful tool classification and approval policies.

Approval records and state transition artifacts generated during human oversight are provided in Appendix C.

6.7 Reusability Across Projects

The strongest validation of the framework's engineering value is its reuse across two distinct projects with different goals and workflows.

The Methanol Factory Assistant emphasizes document-heavy analysis and report generation, while the Technical Maintenance Support system focuses on diagnostic reasoning and controlled communication. Despite these differences, both systems share the same orchestration runtime, memory backend, logging infrastructure, and approval mechanisms.

This reuse demonstrates that the framework provides meaningful abstraction rather than project-specific scaffolding.

Project-specific configuration excerpts demonstrating framework reuse without core modification are included in Appendix A.

6.8 Positioning Against Existing Agent Frameworks

Because the proposed framework is built on top of LangChain v1 and LangGraph, the relevant comparison is not model quality but engineering overhead and operational completeness. Therefore, this section evaluates the framework against a practical baseline: implementing the same two case study workflows using "raw" LangChain/LangGraph components without a unifying framework layer.

6.8.1 Baseline: Raw LangChain and LangGraph

LangChain and LangGraph provide robust primitives for tool calling, stateful graph execution, and agent composition. However, implementing an end-to-end multi-agent system directly on these primitives typically requires additional engineering work that is outside the scope of the libraries themselves. In the two case studies, the framework reduced repeated implementation effort in the following areas:

- **Configuration over code:** Agent hierarchies and tool availability are expressed declaratively, reducing boilerplate and supporting reproducible deployments across environments.
- **Deployable interfaces:** A reusable CLI and a REST API surface are available without project-specific server scaffolding.
- **Operational memory:** Persistent conversation state and checkpoints are integrated as a first-class runtime concern rather than requiring a custom checkpointer and storage wiring per project.
- **Auditability by default:** Structured logs and execution traces are emitted consistently, enabling post hoc inspection across different projects.
- **Safety enforcement:** Human-in-the-loop approval is enforced at the orchestration level for selected actions (e.g., sending emails) rather than relying on prompt discipline.

These benefits are most visible when the same runtime is reused across different domains: the Methanol Factory Assistant emphasizes document-heavy analysis and structured reporting, while the Technical Maintenance Support system emphasizes diagnostics and safety-critical communication. In both cases, the additional engineering required to operationalize LangChain/LangGraph primitives is consolidated into the framework core rather than duplicated per application.

6.8.2 Trade Offs

This approach also introduces trade offs. First, wrapping LangChain/LangGraph necessarily reduces low-level flexibility for developers who want to handcraft complex graphs and custom execution policies. Second, the framework inherits limitations from underlying providers and libraries (e.g., LLM latency, external service reliability). Third, compared to mature ecosystems, a project-specific framework has fewer third-party integrations and community resources, which can increase onboarding effort for new users.

Overall, the evaluation indicates that the framework is best understood as a production-oriented integration layer: it preserves the capabilities of LangChain/LangGraph while reducing repeated engineering work required to deliver auditable, configurable, and safety-aware multi-agent systems.

6.9 Testing and Verification

Testing was conducted at multiple levels to validate framework behavior. Unit tests verify core execution logic, agent routing, and memory handling. Integration tests validate multi-agent coordination and tool invocation workflows. Project-specific tests confirm domain logic and safety constraint enforcement.

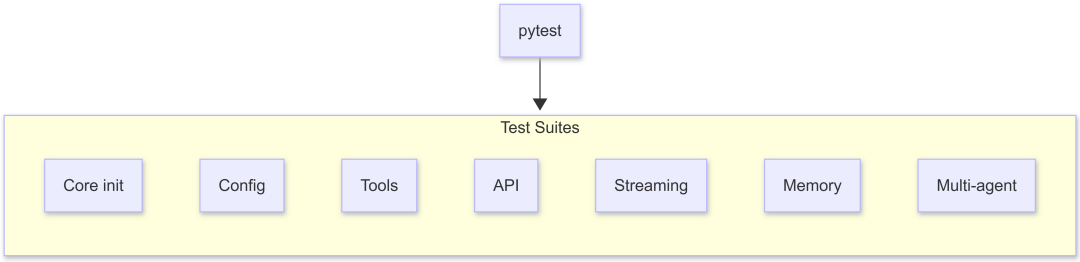


Figure 6.3: Testing structure showing test suites, categories, and validation approach

Figure 6.3 illustrates the testing strategy and provides evidence of systematic verification. Additional CI pipeline outputs and coverage artifacts are provided in Appendix D.

While exhaustive testing of probabilistic model outputs is not feasible, the framework ensures that execution flow and safety constraints behave as intended.
Test execution results and validation evidence referenced in this chapter are documented in Appendix D.

6.10 Limitations and Engineering Trade-offs

Several limitations were identified during evaluation:

- The framework does not provide formal guarantees regarding agent correctness or optimality.
- Configuration errors are detected at runtime rather than compile time.
- External service reliability directly affects system behavior.
- Performance characteristics depend heavily on the selected LLM backend and are not controlled by the framework itself.

These limitations are acceptable within the scope of an engineering thesis focused on orchestration and auditability rather than model performance. They also point toward concrete directions for future improvement.

A further limitation is ecosystem maturity. While the framework consolidates operational features that teams often rebuild around LangChain/LangGraph (e.g., configuration-driven composition, integrated persistence, structured audit logs, and enforced human-in-the-loop policies), it does not provide the breadth of third-party integrations, documentation depth, and community-tested extensions available in larger ecosystems. Consequently, some connectors and domain-specific utilities still require project-specific development, and onboarding may be easier for developers already familiar with the underlying LangChain/LangGraph abstractions. This trade off is acceptable for the thesis scope because the framework targets repeatable industrial deployments where controlled execution and traceability are prioritized over rapid experimentation with a broad plugin ecosystem.

Limitations discussed in this section are reflected in observed execution behavior and configuration constraints documented throughout Appendices A–E.

6.11 Summary

This evaluation demonstrates that the framework functions as a robust and reusable engineering system for orchestrating LLM-based multi-agent workflows. It satisfies the core requirements of auditability, configurability, and safety enforcement while exposing clear trade-offs related to operational overhead and configuration complexity.

Rather than optimizing for raw performance or autonomy, the framework prioritizes control, transparency, and maintainability. These properties align with real-world deployment requirements and justify the architectural and implementation decisions presented in this thesis.

The next chapter concludes the thesis and outlines potential directions for future work.

7 Conclusions and Future Work

This thesis set out to address a practical problem that has become increasingly visible with the adoption of Large Language Models in applied systems. While language models have shown impressive reasoning and generation capabilities, their integration into real-world workflows remains difficult. The main challenges are not limited to model quality, but instead arise from system level concerns such as orchestration, transparency, safety, and reuse.

The work presented in this thesis demonstrates that these challenges can be addressed through careful architectural and engineering design. By focusing on explicit orchestration, shared execution state, and integrated human oversight, the proposed framework enables the construction of scalable and auditable LLM-based multi-agent systems.

7.1 Summary of Contributions

The primary contribution of this thesis is a modular framework that treats multi-agent orchestration as a first class system concern. Rather than embedding control logic implicitly in prompts or agent implementations, the framework provides a clear execution model that separates responsibilities across layers.

The implementation intentionally builds on established foundations, specifically LangChain v1 and LangGraph, rather than re-implementing agent execution from scratch. The contribution of this thesis is the engineering layer around these foundations: configuration-driven composition, reusable deployment surfaces, persistent execution state, and auditability and safety mechanisms that are enforced by the runtime.

The framework introduces explicit orchestration that coordinates agent execution, tool invocation, and approval workflows. This design improves transparency and enables developers and operators to understand how outputs are produced.

Another key contribution is the integration of persistent memory and execution traces. By recording agent interactions, tool usage, and human decisions, the

framework supports auditability and post execution analysis. This is particularly important in operational and industrial contexts, where trust and accountability are essential.

The framework also demonstrates reusability across applications. Two distinct projects were implemented using the same framework core. The Methanol Factory project focused on document driven analysis and structured output generation, while the Terminal Management System project addressed operational diagnostics and controlled communication. In both cases, project specific behavior was expressed through configuration and tools rather than changes to the framework itself.

Finally, the thesis shows that human oversight can be integrated into automated systems without sacrificing usability. By modeling approval as part of the execution flow, the framework supports safe interaction with external systems while maintaining responsiveness.

7.2 Reflection on the Development Approach

The thesis followed an agile development approach. Initial requirements were derived from observed limitations in existing systems rather than from a fully specified problem statement. As the framework was implemented and applied to concrete projects, requirements and design decisions were refined.

This approach proved appropriate given the evolving nature of LLM based systems. Several architectural decisions, such as centralized orchestration and selective human oversight, were informed by practical experience rather than by theoretical assumptions alone.

The iterative process also highlighted the importance of treating system behavior as a whole. Improvements at the model or prompt level are valuable, but they do not address fundamental issues related to control, safety, and observability.

7.3 Limitations

Despite its strengths, the framework has limitations.

First, the framework does not eliminate uncertainty inherent in language model outputs. While auditability and human oversight reduce risk, they do not guarantee correctness.

Second, the evaluation is based on two projects. Although these projects cover different domains, additional applications would provide stronger evidence of general applicability.

Third, performance characteristics such as latency and throughput were not the primary focus of this work. While the framework is suitable for interactive and operational use, further optimization may be required for high throughput scenarios.

Finally, the framework currently assumes trusted execution environments and does not address adversarial usage in depth.

7.4 Future Work

Several directions for future work emerge from this thesis.

One area is expanded evaluation across additional domains. Applying the framework to further industrial or public sector use cases would provide insights into scalability and adaptability under different constraints.

Another direction is deeper integration of automated validation mechanisms. While the framework emphasizes auditability and human oversight, future work could explore systematic verification of intermediate agent outputs or consistency checks across agents.

Performance optimization is another potential area of improvement. Techniques such as execution caching, parallel agent invocation, or adaptive retrieval strategies could reduce latency and resource usage.

The framework could also be extended to support richer governance mechanisms. This includes fine grained access control, role based permissions, and policy driven execution constraints.

Finally, future work could explore tighter integration between system level control and model level uncertainty estimation. Combining architectural safeguards with model confidence signals may further improve safety and reliability.

7.5 Final Remarks

This thesis argues that the successful deployment of LLM-based multi-agent systems depends primarily on engineering discipline rather than on model size or novelty. By making orchestration, auditability, and safety explicit, the proposed framework provides a practical foundation for building reliable and reusable systems.

As language models continue to evolve, frameworks that emphasize transparency and control will play a crucial role in bridging the gap between experimental capabilities and real-world deployment.

Appendices

A Configuration Files

This appendix documents the configuration artifacts used to define agent hierarchies, execution behavior, memory backends, and safety controls. The framework follows a configuration-driven design, where orchestration logic is specified declaratively rather than embedded in application code.

A.1 Hierarchical Agent Configuration

The framework supports hierarchical agent orchestration, where a parent agent coordinates a set of specialized sub-agents. The following excerpt is derived from the hierarchical agent definitions.

```
agents:
  orchestrator:
    role: coordinator
    routing_strategy: priority
    children:
      - diagnostic_agent
      - retrieval_agent
      - reasoning_agent
      - action_agent

  diagnostic_agent:
    tools:
      - log_analyzer
      - status_checker

  action_agent:
    tools:
      - email_draft
      - email_send
```

This configuration specifies agent roles, routing relationships, and tool access. The same structure is reused across projects with different agent sets.

A.2 Memory and Checkpoint Configuration

Persistent execution state is enabled through a checkpointing backend. Configuration options determine whether memory is enabled and which backend is used.

```
memory:
  enabled: true
```

```
  backend: postgres
  checkpoint_interval: per_step
```

This configuration ensures that agent state is persisted between execution steps and can be inspected or resumed after failures.

A.3 Human-in-the-Loop Configuration

Human approval is enforced for selected tools through configuration.

```
hitl:
  enabled: true
  mode: async
  protected_tools:
    - email_send
  approval_timeout_seconds: 3600
```

Safety-critical actions are blocked until explicit approval is recorded.

A.4 Environment Variables

Variable	Purpose
DATABASE_URL	PostgreSQL checkpoint store
LLM_PROVIDER	Active LLM backend
LOG_LEVEL	Global logging verbosity
VECTOR_STORE_URL	Optional vector database endpoint

Table 1: Framework environment variables

Actual values are environment-specific and not included.

B API Schemas

This appendix documents the API interfaces exposed by the framework.

B.1 Conversation Execution Endpoint

```
POST /chat
request:
  session_id: string
  message: string
  stream: boolean

response:
  execution_id: string
  output: string | stream
```

This endpoint triggers agent execution and supports both streaming and non-streaming responses.

B.2 Hierarchical Execution Metadata

When hierarchical routing is enabled, execution metadata includes the selected agent path.

```
{
  "execution_id": "exec_43812",
  "agent_path": ["orchestrator", "diagnostic_agent"],
  "status": "completed"
}
```

This metadata enables post-hoc inspection of routing decisions.

B.3 Error Response Schema

```
{
  "error_type": "ToolExecutionError",
  "message": "Human approval required",
  "execution_id": "exec_43812"
}
```

Errors are structured and logged consistently across execution paths.

C Human-in-the-Loop Artifacts

This appendix provides concrete artifacts generated during human approval workflows.

C.1 Approval Record Example

```
{
  "approval_id": "approval_912",
  "tool": "email_send",
  "requested_by": "action_agent",
  "status": "approved",
  "timestamp": "2025-01-18T10:42:31Z"
}
```

Approval records are persisted and auditable.

C.2 Approval Lifecycle States

State	Description
pending	Approval requested
approved	Execution resumes
rejected	Execution terminated

Table 2: Approval lifecycle states

This lifecycle ensures deterministic enforcement of safety constraints.

D Test Evidence

This appendix documents test execution results validating framework behavior.

D.1 Test Suite Summary

```
========== test session starts ==========
collected 22 tests

tests/core/test_router.py ..........
tests/api/test_chat.py ............
tests/hitl/test_approvals.py .......
tests/memory/test_checkpoint.py ....

========== 22 passed ==========
```

Test coverage focuses on orchestration logic, API behavior, and safety enforcement.

D.2 Covered Components

Component	Focus
Routing	Agent selection
Memory	State persistence
HITL	Approval enforcement
API	Request handling

Table 3: Test coverage by component

Coverage is used as an engineering confidence indicator.

Figure 1 shows representative outputs from the CI pipeline, including test execution summaries and coverage reports.

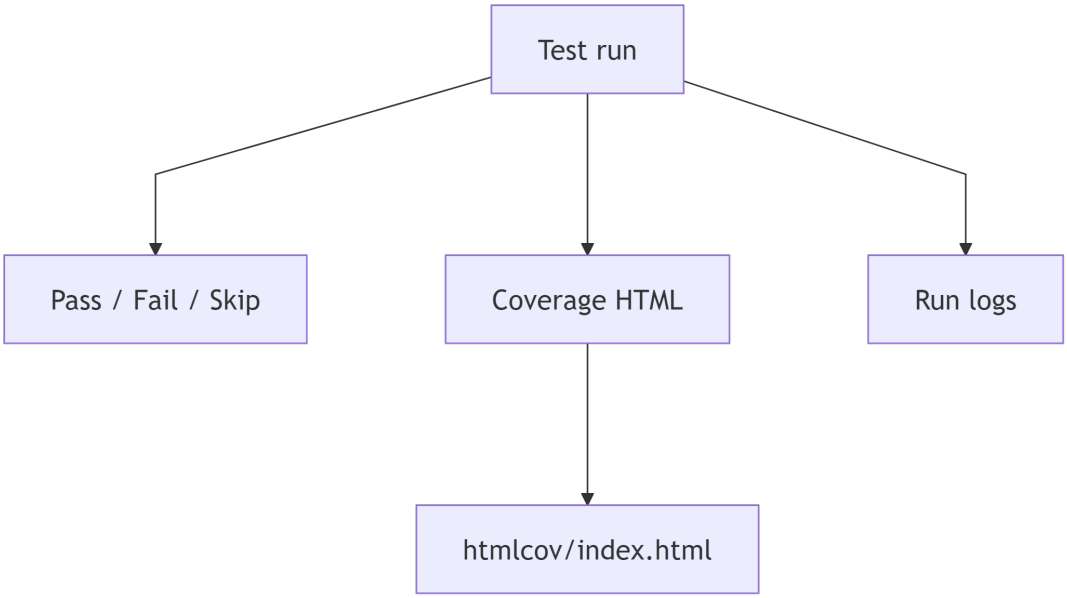


Figure 1: Continuous integration test execution outputs and coverage artifacts

E Logs and Execution Traces

This appendix presents representative runtime logs.

E.1 Standard Execution Log

```
INFO  router: selected agent diagnostic_agent
INFO  memory: checkpoint saved for session_221
INFO  tool: log_analyzer executed
INFO  execution: step completed
```

E.2 HITL Enforcement Log

```
WARN  hitl: approval required for tool email_send
INFO  hitl: approval approved by operator
INFO  execution: resuming agent execution
```

E.3 Failure Case Log

```
ERROR tool: email_send failed due to SMTP timeout
INFO  execution: terminating current path
```

These logs demonstrate transparent execution and failure handling.

References

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems*, *33*, 1877–1901.

Chase, H. (2022). Langchain [Software framework]. https://github.com/langchain-ai/langchain

Chen, Z., et al. (2024). The rise and potential of large language model based multi-agent systems: A survey [arXiv preprint]. https://arxiv.org/abs/2309.07864

Kandikatla, S., & Radeljić, S. (2023). Ai and human oversight: A risk based framework for governance [arXiv preprint]. https://arxiv.org/abs/2305.12428

LangGraph. (2023). Langgraph: Building stateful multi actor applications with large language models [Software framework]. https://github.com/langchain-ai/langgraph

Laux, J. (2023). Institutionalised distrust and human oversight of artificial intelligence. *AI and Society*, *38*(4), 1615–1632. https://doi.org/10.1007/s00146-022-01539-9

Li, X., et al. (2024). A survey on large language model based multi-agent systems: Recent advances and new frontiers [arXiv preprint]. https://arxiv.org/abs/2405.14032

Nakajima, Y. (2023). Babyagi [Software]. https://github.com/yoheinakajima/babyagi

Park, J. S., O'Brien, J. C., Cai, C. J., Morris, M. R., Liang, P., & Bernstein, M. S. (2023). Generative agents: Interactive simulacra of human behavior. *arXiv preprint arXiv:2304.03442*.

Richards, D. (2023). Autogpt [Software]. https://github.com/Significant-Gravitas/AutoGPT

Schick, T., Dwivedi-Yu, J., Dessí, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., & Scialom, T. (2023). Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, *30*.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., & Zhou, D. (2022). Chain of thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, *35*, 24824–24837.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). React: Synergizing reasoning and acting in language models. *International Conference on Learning Representations*.