Politecnico di Torino

Data Science and Engineering

A.a. 2024/2025

Graduation Session March 2026

Design and Deployment of a Multi-Agent Chatbot for Incident Management and System Monitoring

Advisors:
Luca Cagliero
Davide Pezzolla

Candidate:
Francesco La Rota

Summary

This thesis presents the design, implementation, and evaluation of a multi-agent conversational assistant for incident management and system monitoring. Developed in collaboration with **Storm Reply**, the system integrates reasoning agents with enterprise services through the Model Context Protocol (MCP). The proposed architecture aims to automate monitoring, documentation, and operational workflows in complex cloud environments while maintaining security, observability, and scalability.

Traditional monitoring systems typically rely on predefined alerting mechanisms and manual analysis, requiring significant human intervention during incident investigation and resolution. Likewise, existing monitoring platforms and AI-assisted tools often exhibit important limitations operating with limited contextual awareness and the lack of structured integration with operational tools and enterprise services. As a result, their ability to support complex operational workflows and dynamic decision-making remains constrained.

To address these limitations, this work investigates an agent-based architecture capable of coordinating multiple reasoning agents and integrating external services through a standardized protocol. The thesis explores agent orchestration strategies, large language model (LLM) interfaces, and a modular backend design that enables flexible tool integration. The thesis describes the architectural design, implementation process, and decision-making mechanisms of the agents. It concludes with an experimental evaluation based on predefined key performance indicators (KPIs), measuring the system's impact on operational efficiency, incident resolution time, and response quality.

Table of Contents

1 Introduction 1

1.1 Goal . . . 1

1.2 Thesis Structure . . . 2

2 State Of The Art 3

2.1 Generative and Agentic AI . . . 3

2.1.1 Large Language Models (LLMs) . . . 4

2.1.2 Agent Design and Abstractions . . . 4

2.1.3 MCP Servers and Agents . . . 5

2.1.4 Chatbot and Agent-Based Architectures . . . 7

2.1.5 Multi-Agent Interaction Protocols . . . 8

2.2 Conversational Agents for IT Operations . . . 11

2.2.1 Observability-Driven Assistants . . . 11

2.2.2 Incident & Ticketing Assistants . . . 13

2.2.3 Enterprise AIOps Platforms . . . 15

2.2.4 Limitations of Current Approaches . . . 17

2.3 External Services and Platforms . . . 18

2.3.1 Containerization . . . 18

2.3.2 Amazon Web Services (AWS) . . . 19

2.3.3 Atlassian Services . . . 20

3 Methodology 23

3.1 Design . . . 23
3.1.1 High-Level System Diagram . . . 23
3.1.2 Server Layer . . . 24
3.1.3 Agentic Layer . . . 27
3.2 Deployment . . . 29
3.2.1 Low-Level Design . . . 30
3.2.2 Server Layer . . . 30
3.2.3 Agentic Layer . . . 32
3.2.4 Front-End . . . 34
3.2.5 Portability and Reproducibility . . . 35

4 Results 37
4.1 Quantitative Response Evaluation . . . 37
4.1.1 Analysis . . . 45
4.1.2 Tool Selection Impact on Resource Consumption . . . 50
4.2 Qualitative Response Analysis . . . 51
4.2.1 Approach . . . 51
4.2.2 Analysis . . . 52

5 Conclusion 61
5.1 Production Deployment and Operational Insights . . . 61
5.2 Token Reduction and Cost Optimization . . . 62
5.3 Limitations and Considerations . . . 62
5.4 Future Work . . . 63
5.4.1 Mitigating Nova Pro's Tool Execution Failures . . . 63
5.4.2 Hybrid Model Selection Based on Query Complexity . . . 64
5.4.3 Knowledge Base Integration . . . 64
5.4.4 Automated Remediation with Graduated Risk Levels . . . 65
5.4.5 Investigating Claude 4.5 Timeout Failures . . . 66
5.4.6 Cross-Domain Validation and Generalization . . . 66

5.5 Closing Remarks 67

Bibliography 68

Chapter 1

Introduction

In modern IT consulting environments, organizations rely heavily on complex cloud infrastructures and interconnected software ecosystems. This complexity brings an increase in operational issues and support requests, often managed through platforms like Jira or ServiceNow and monitored via tools or platforms such as AWS CloudWatch and Kubernetes dashboards. As incidents become more frequent and time-sensitive, companies frequently struggle to respond quickly, resulting in missed SLAs (Service Level Agreements), overwhelmed support staff, and operational disruptions.

1.1 Goal

A key bottleneck in this process is incident "triage": understanding, contextualizing, and responding to a ticket quickly and accurately. This often requires support engineers to gather data from scattered sources, retrieve metrics or logs, reference documentation, and communicate findings back to stakeholders, sometimes under high-pressure scenarios like late-night alerts or critical outages. These processes are time-consuming, repetitive, and cognitively demanding, especially when the team must act across different systems and time zones.

To address this, we developed a conversational assistant that automates incident investigation workflows. Integrated within an IT firm's operations, the assistant serves as an intelligent interface between users and a network of backend diagnostic tools. Built on a large language model (LLM) and coordinating with multiple specialized micro-agents (e.g., for Atlassian and AWS services), it can understand natural language queries, retrieve system data, and provide structured outputs, all through a conversational UI.

Unlike basic chatbots, this assistant orchestrates multiple agents (with intelligent tool delegation), retains context across sessions, and summarizes complex diagnostics in real time. It reduces the need for manual lookups and repetitive command execution, streamlining communication and improving resolution times, ultimately enabling continuous support without proportional increases in staffing.

This project was developed in collaboration with **Storm Reply**, an AWS Premier Consulting Partner and part of the Reply Group, which focuses on helping enterprises design, build, and manage cloud-native solutions on Amazon Web Services (AWS). Their core expertise spans infrastructure automation, cloud migration, platform engineering, DevOps pipelines, and modern application architectures.

Within this context, we explored innovative themes such as:

- The use of **MCP servers** for abstracting tool capabilities into stateless, callable endpoints;
- The deployment of multiple **MCP agents** for executing backend logic independently from each other, enhancing modularity and observability;
- The design and orchestration of **Multi-agent frameworks**, particularly in environments where agents specialize in retrieving logs, reading metrics, querying APIs, or surfacing documentation;
- The integration of **LLM-based interfaces** for mediating user-agent communication via prompt-routing and orchestration logic (we curated the entire process end to end).

1.2 Thesis Structure

This thesis follows the project from the current state of AI-assisted IT operations to the design and evaluation of a novel agent-based solution. Chapter 2 reviews existing conversational agents in AIOps, highlighting their capabilities and limitations, and presents the technological foundations of the proposed solution, covering generative AI, multi-agent systems and orchestration strategies, along with integrated third-party services. Chapter 3 details the design and deployment of the proposed system. Chapter 4 evaluates its performance across multiple dimensions, including model comparisons, accuracy, costs, latency, and qualitative validation. Finally, Chapter 5 discusses potential future improvements and directions for continued development.

Chapter 2

State Of The Art

The growing complexity of cloud-native infrastructures and the increasing demands placed on IT support teams have led to a surge of interest in AI-assisted solutions for system monitoring, incident management, and operational efficiency. This chapter explores both the current landscape of conversational agents in IT operations and the technological foundations that make advanced assistants possible.

We begin by introducing the key enabling technologies behind the proposed solution. Topics include generative AI and large language models, agent-based abstractions, orchestration strategies, and the role of MCP servers. We also cover backend infrastructure considerations and integrations with third-party platforms.

Building on this foundation, we then review existing tools and approaches that combine natural language interfaces with diagnostic or triage capabilities. These include AIOps platforms, enterprise support bots, and LLM-based monitoring assistants. Through this analysis, we assess their architectures, use cases, and limitations. This review highlights the gaps that persist in current solutions and motivates the need for a more integrated, intelligent approach.

2.1 Generative and Agentic AI

Generative AI refers to a class of artificial intelligence systems capable of producing new content—such as text, images, or code—by learning patterns from large-scale datasets. Unlike traditional discriminative models, which focus on classification or prediction, generative models aim to capture underlying distributions of data, enabling them to generate coherent and contextual outputs. In IT operations, generative AI is particularly impactful, excelling in tasks like summarizing incidents,

drafting remediation steps, and translating complex observability data into natural language explanations.

2.1.1 Large Language Models (LLMs)

Large Language Models (LLMs) represent the most widely adopted class of generative AI systems. Trained on massive corpora of text and code, LLMs learn to predict and synthesize sequences of natural language with remarkable fluency. The key innovation of modern architectures is that they're based on the Transformer model, leveraging the self-attention mechanism, which allows the model to evaluate how each word in a sentence relates to every other word (and not only close surroundings), enabling reasoning over complex queries and multi-step tasks [1]. Figure 2.1 shows a high-level overview of the Transformer architecture, as intro. In the context of IT operations, LLMs provide the foundation for natural language interfaces that allow engineers to query logs, metrics, or knowledge bases conversationally. They also support advanced capabilities such as intent recognition, incident summarization, and code generation.

However, challenges remain: LLMs often act as "black boxes", may hallucinate facts, and require domain adaptation to achieve reliability in enterprise settings where precision and trustworthiness are critical.

2.1.2 Agent Design and Abstractions

While LLMs provide the generative core, their integration into operational workflows requires agentic architectures. An agent can be defined as an abstraction that couples reasoning capabilities with the ability to memorize, plan, and act within an environment, as shown in Figure 2.2.

Agents where created to solve the need for an AI structure able to maintain memory and leverage external tools, coordinating across multiple steps to achieve goals. In practice, an agent can be thought of as a software entity that receives a task, reasons about how to accomplish it, retrieves information or tools when necessary, and performs actions such as querying a monitoring platform or updating a ticket.

Within this context, **tools** represent structured interfaces that allow an agent to interact with external systems or services. A tool may correspond to an API endpoint, a database query, a monitoring command, or any other callable function that retrieves information or performs an action in the environment. By invoking tools, agents extend their capabilities beyond pure language generation, grounding their reasoning in real data and enabling them to execute concrete, often deterministic,

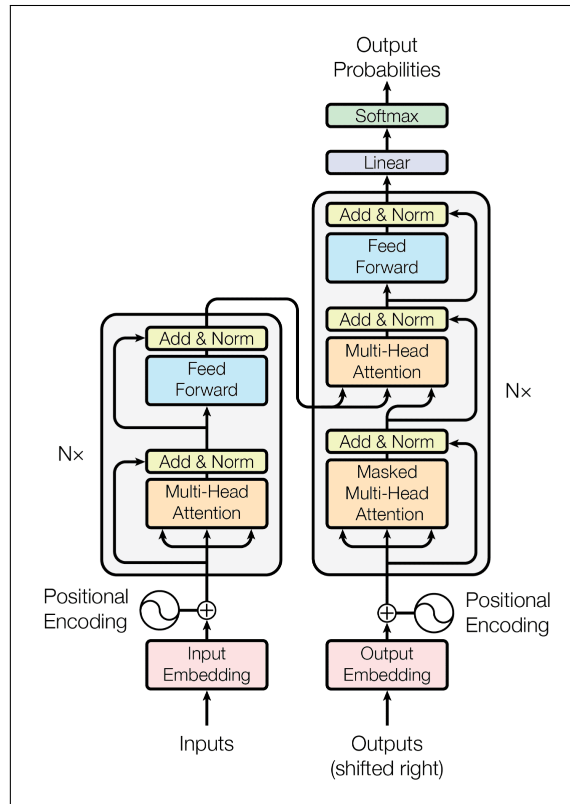

Figure 2.1: High-level overview of the Transformer architecture, illustrating the self-attention mechanism and encoder-decoder layers. Source: [1]

operations.

In IT operations, this translates into agents capable of querying observability platforms, triaging incidents, or executing remediation actions through standardized interfaces. Multi-agent frameworks and standardization efforts such as the Model Context Protocol (MCP) further enrich this paradigm, as we'll explore in upcoming sections.

2.1.3 MCP Servers and Agents

When integrating LLMs with enterprise systems in order to create agents, a key challenge is enabling models to interact reliably with external tools such as databases, monitoring systems, or ticketing platforms.

The Model Context Protocol (MCP) is a recently introduced open standard designed to facilitate structured interaction between large language model (LLM) and

Figure 2.2: Abstraction of an AI Agent structure, with the LLM responsible for Planning and Reasoning, having a memory and tool module at their disposal. Source: [2]

external tools, services, or knowledge bases, as illustrated by Figure 2.3. Unlike earlier approaches that relied on custom APIs, proprietary SDKs (Software Development Kit, a collection of tools, libraries, and documentation that helps developers build applications for a specific platform, service, or programming environment), or ad hoc integrations, MCP defines a lightweight, standardized protocol through which agents can discover, query, and interact with resources in a controlled and consistent manner. Within this framework, an MCP Server acts as a connector that exposes the capabilities of a given system (like a ticketing or monitoring service) in a way that is accessible to any MCP-compatible agent. Agents, in turn, can call these servers to retrieve structured information, execute actions, or enrich their reasoning process without requiring hard-coded integration logic. [3]

The emergence of MCP servers and agents is closely tied to the rapid expansion of LLM-powered applications that began in the early 2020s. As generative AI models matured and shifted from research prototypes to enterprise-grade solutions, organizations faced growing challenges integrating them into existing operational environments. Initial efforts often required bespoke plugins or orchestration layers, with limited portability and creating significant maintenance overhead. MCP, first proposed around 2023–2024 by Anthropic, addresses this problem by introducing a unifying abstraction layer: instead of hard-coding domain knowledge into the model or platform, existing tools are wrapped as MCP servers, exposing their functionality through a standardized interface. [5]

The reasons for the rise of MCP-based architectures are both technical and organizational. Technically, LLMs are inherently general-purpose but lack grounded access to enterprise-specific data and operational tools. MCP servers provide a safe and structured bridge, allowing agents to move beyond generic text generation and act as context-aware assistants that can query live telemetry, retrieve incidents, or suggest remediation steps with access to authoritative data. Organizationally,

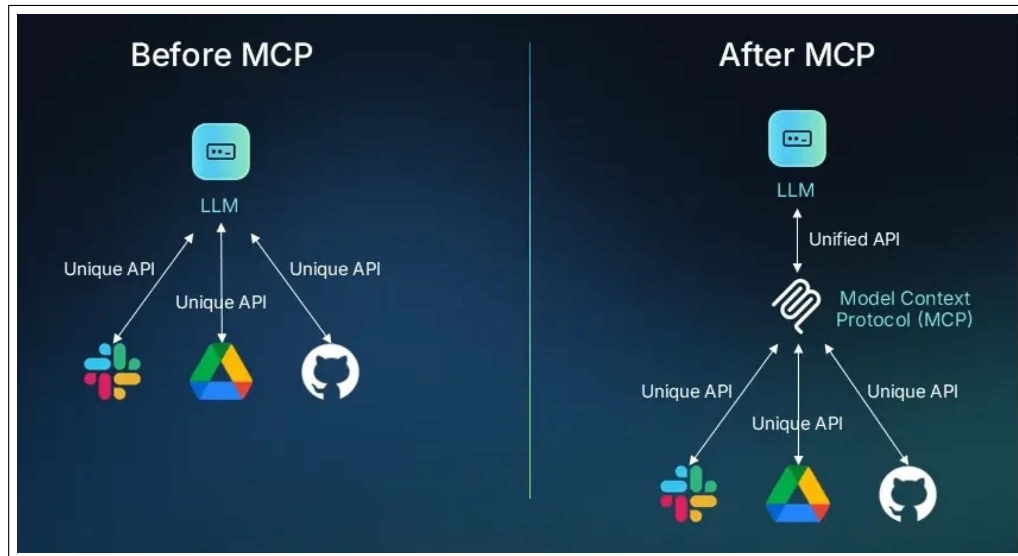

Figure 2.3: Visual representation of the integration of Model Context Protocol (MCP) servers. On the left: architecture before MCP, showing the LLM having to deal with all custom integrations. On the right: architecture after MCP implementation, which ensures all unique API requests and connections are well managed for the LLM. Source: [4]

enterprises have grown wary of vendor lock-in and the inefficiencies of duplicating knowledge bases across platforms. MCP offers a path toward greater flexibility, enabling companies to integrate new services incrementally without re-engineering existing infrastructure. This modularity also supports the adoption of multi-agent frameworks, where specialized agents (e.g., for observability, security, or ticketing) can collaborate through shared MCP interfaces.

2.1.4 Chatbot and Agent-Based Architectures

The evolution of chatbots and conversational agents has passed through several architectural paradigms, from rule-based and template matching systems to modern neural and multi-agent frameworks. Early chatbots such as ELIZA, the first chatbot made public, an early milestone in chatbots using pattern matching on constrained topics [7], or ALICE, the oldest and most famous template-based chatbot, presented in [8], operated via pattern matching and template substitution: they scanned input for keywords or patterns and replied using predefined templates, with no capacity for memory, learning, or deeper understanding. Over time, these systems gave way to data-driven models, especially following the advent of deep learning

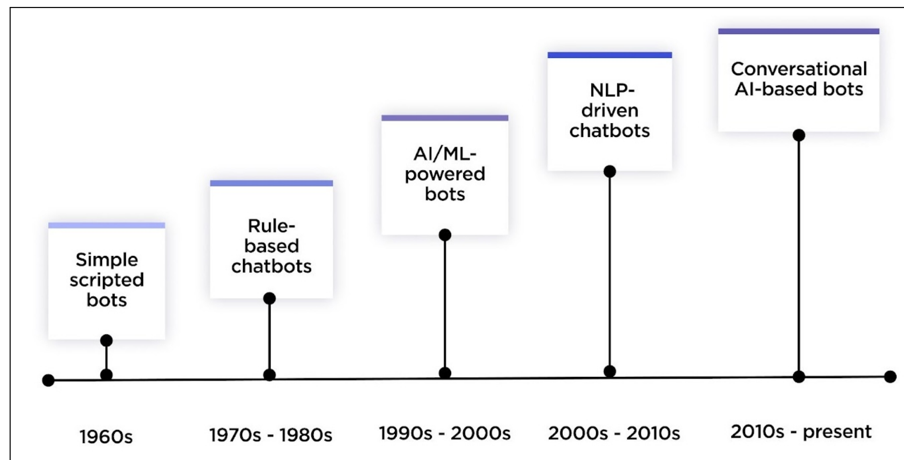

Figure 2.4: Evolution of chatbots paradigma over the past 65 years. Adapted from: [6]

and transformer architectures, as shown by Figure 2.4, which sums up the evolution of chatbot approaches over the decades.

Modern architectures often adopt an agent-based or modular design. In these setups, a central conversational engine (typically a LLM) can invoke specialized subagents or tools to handle external tasks—such as database lookup, API calls, or domain-specific reasoning—while retaining a unifying dialogue context. This hybrid approach allows the conversational system to maintain fluency and general-purpose reasoning, yet integrate operational functionality and maintain separation of concerns. Multi-agent collaboration is also emerging: multiple agents, possibly specialized (e.g. one for knowledge retrieval, another for planning), communicate and coordinate to fulfill user goals. Such architectures aim to combine the strengths of modular design (maintainability, specialization) with the large-scale language capabilities of contemporary models. [9].

2.1.5 Multi-Agent Interaction Protocols

As agent ecosystems evolve, interaction is no longer limited to tool invocation but extends to direct agent-to-agent collaboration. In this setting, agents advertise their capabilities and delegate tasks to peers, forming distributed workflows that operate asynchronously and adaptively.

Figure 2.5 illustrates a high-level view of a multi-agent architecture, with specialized agents handling specific tools and a single entry point interfacing with the user.

This model enables the emergence of specialized agents that coordinate around

shared objectives, enhancing modularity. We have several fully decentralized network protocols, where agents employ cryptographic identifiers and structured metadata to autonomously discover, authenticate, and interact across open networks. Taken together, these interoperability strategies illustrate a progression from tightly controlled tool-based invocation toward open, peer-to-peer agent societies, each stage adding new layers of flexibility, scalability, and autonomy to multi-agent systems. [10]

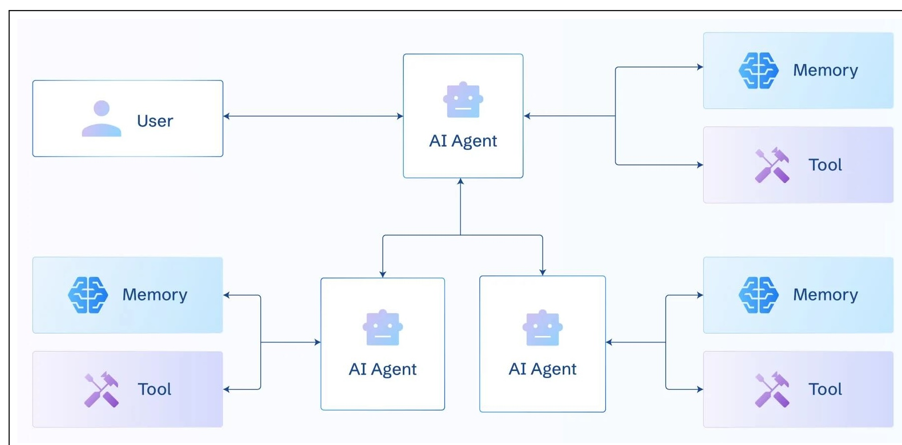

Figure 2.5: Example of multi-agent architecture. Source: [11]

Agents-as-Tool Interaction

In the agent-as-tool paradigm, reasoning and tool invocation are decoupled through a hierarchical design: one component (Planner or Orchestrator) handles the natural language reasoning and task decomposition, while other components (the Tool-callers) are responsible for interfacing with external systems or APIs in a structured way. This separation simplifies the reinforcement learning optimization process and reduces noise from tool execution in the reasoning stage [12]. By isolating tool calls, the Planner can work with cleaner, more semantically meaningful inputs, improving reasoning accuracy over multi-hop or compositional tasks.

Under this architecture, the Toolcaller returns structured observations rather than raw, unprocessed outputs. The Planner then reasons over this curated representation, avoiding the need to identify and removing irrelevant noise or formatting the data introduced by external tools. This approach enhances interpretability and modularity: new tools can be added or replaced without retraining the core reasoning logic, and each sub-agent can be optimized for its specialized role (which can be fine tuned independently).

Agent-to-Agent (A2A) Collaboration

In an agent-to-agent (A2A) architecture, agents move beyond being isolated reasoners to become collaborative peers that can delegate, negotiate, and coordinate tasks among themselves. Each agent advertises capabilities via Agent Cards (structured metadata describing an agent's capabilities such as available tools and communication endpoints), enabling identification and dynamic orchestration of distributed workflows [10]. Through a peer-to-peer communication protocol, agents can outsource subtasks, share intermediate results, and coordinate action plans under a shared, structured framework.

A2A enables a shift from single-agent tool invocation to multi-agent orchestration: agents can call upon others to perform subtasks, enabling more flexible and scalable architectures. Because communication is structured and protocol-governed, agents can interoperate across heterogeneous platforms and vendors, while maintaining security and semantic agreement over message formats. [13]

Swarm Intelligence

A Swarm is a collaborative agent orchestration system where multiple agents work together as a team to solve complex tasks. Unlike traditional sequential or hierarchical multi-agent systems, a Swarm enables autonomous coordination between agents with shared context and working memory.

Swarm intelligence is inspired by collective behaviors observed in nature, such as ant colonies or bird flocks, where simple individuals following local rules collectively produce intelligent global behavior: the basic idea is that a group of specialized agents working together can solve problems more effectively than a single agent. [14]

Following this thought process, each agent in a Swarm has access to the full task context and is able to see the history of which agents have worked on the task. All agents share the same knowledge, and can decide when to hand off to another agent with different expertise.

Figure 2.6 shows a basic swarm structure where the agents composing it are in constant connection and autonomously manage hand offs and share memory and context.

This model is particularly powerful in environments where adaptability, redundancy, and scalability are crucial: for example, when the task at hand can be broken down into sub-tasks that benefit from different specialized perspectives. A Swarm is ideal for exploration, brainstorming, or synthesizing information from multiple sources

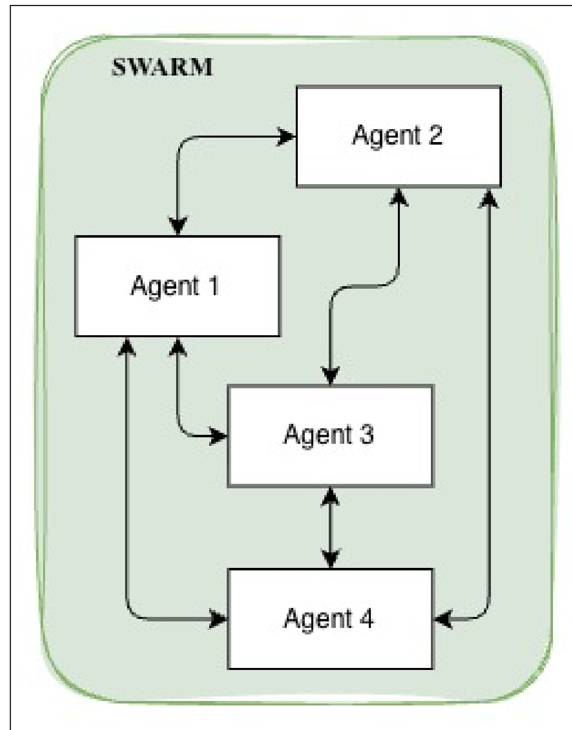

Figure 2.6: Example of swarm architecture.

through collaborative handoffs. It leverages agent specialization and shared context to generate diverse, comprehensive results.

2.2 Conversational Agents for IT Operations

2.2.1 Observability-Driven Assistants

Observability refers to the ability to understand the internal state of a system by analyzing the data it produces, typically including metrics (numerical measurements of system performance such as CPU usage), logs (timestamped records of system events), and traces (records of how requests propagate across distributed services). Unlike traditional monitoring, which focuses on predefined alerts, observability enables engineers to explore system behavior and diagnose unexpected problems.

Observability assistants are designed to provide intelligent insights on system performance, anomalies, and service health by combining monitoring data with natural language interaction. Platforms such as Dynatrace's *Davis AI* and Datadog's *Bits AI* exemplify this category.

Dynatrace Davis AI

Dynatrace is a software intelligence platform designed to help organizations monitor, optimize, and manage the performance of their applications, infrastructure, and digital experiences. Dynatrace works by using its proprietary *OneAgent* and the *Davis AI* engine to automatically discover, map, and monitor the entire IT environment [15]. It collects metrics, logs, and tracing data, automatically detecting performance slowdowns and security threats, while providing companies with precise root-cause analysis and automated remediation to improve mean time to resolution (MTTR). Key features of Dynatrace's monitoring AI tool are:

- **OneAgent**, an automated monitoring agent installed on hosts, virtual machines, and containers, collects detailed observability data and automatically maps applications, microservices and infrastructure [16], providing end-to-end visibility across the digital ecosystem;
- **The Davis AI Engine** is the module processing the data collected by OneAgent and performing the actual analysis: it discovers and maps the IT environment to an ever changing real-time model of topology and dependencies, performing root-cause analysis to pinpoint the exact source of performance issues, security vulnerabilities, and user experience problems, while prioritizing issues with the greatest impact on customer experience.
- **The Grail Data Lakehouse** is a storage architecture combining the scalability of a data lake with the structured querying capabilities of a data warehouse: it stores all collected data (metrics, logs, traces, user behavior, etc.) in a unified repository, enabling efficient analysis and making observability data available across the platform [17].
- **Automated Security** provides runtime protection, detecting and blocking vulnerabilities such as SQL injection and cross-site scripting in real time.
- **Digital Experience Monitoring** tracks the availability, performance, and functionality of web, mobile, and IoT applications, delivering insights into user behavior and overall experience.

Datadog's Bits AI

Datadog's Bits AI, currently in early adoption, is a suite of intelligent AI capabilities which integrates LLMs to allow engineers to query metrics and logs conversationally, receive summaries of incidents, and explore dashboards through natural language prompts. Bits aims to accelerate anomaly detection by streamlining monitoring

queries and supporting root cause analysis (RCA, the process of identifying the underlying reason that triggered an incident) in complex cloud-native environments [18]. Main components of Bits AI:

- **Bits AI SRE (Site Reliability Engineering)** for alert investigation is an autonomous AI agent that investigates alerts and coordinates incident response [19]. When a monitor triggers, Bits proactively generates multiple hypotheses, queries relevant telemetry, and reasons over the data to help on-call engineers quickly identify the root cause.
- **AI Dev Agent** for AI-powered code fixes, a generative AI coding assistant that uses observability data to detect and fix high-impact issues on given code using full stack telemetry [20]. It generates production-ready code changes, complete with tests and pull requests.
- **Bits AI Security Analyst** autonomously triages Datadog's log management platform, investigating potential threats and provides actionable recommendations. It classifies signals as benign, suspicious, or inconclusive, and can suggest or execute vetted remediation actions.
- **Action Interface** is an AI-powered chat tool that connects seamlessly with SRE, Dev Agents, and Security Analyst to pull live telemetry, context, and investigation trails into a unified action layer. The interface is stateless, chats are not saved (but retrievable through logs, [21]). Credentials are required, ensuring that only authorized users with the proper roles and permissions can execute actions.
- Finally, **Datadog MCP Server** acts as a bridge between observability data in Datadog and any AI agents that support the Model Context Protocol (MCP) [22], providing structured access to relevant Datadog contexts, features, and tools.

2.2.2 Incident & Ticketing Assistants

These kinds of assistant mainly focus on managing incidents and support tickets, helping teams prioritize, investigate, and resolve issues more efficiently. We present two state of the art serivces, integrated with cloud-based ticketing platforms such as *ServiceNow* and *PagerDuty*, able to automate triage, provide context-aware recommendations, and guide users through standardized workflows, often reducing manual effort and accelerating resolution times.

ServiceNow's Now Assist

ServiceNow's *Now Assist* is an AI-powered productivity suite embedded across the ServiceNow Platform, designed to streamline incident and ticket management [23]. Rather than acting as a standalone bot, Now Assist is natively integrated within the ServiceNow ecosystem, providing contextual guidance, automation, and decision support across IT and customer service management (ITSM & CSM), and HR workflows.

Key capabilities of Now Assist include:

- **Generative AI for Ticket Triage:** automatically summarizes incident descriptions, recommends categorization and routing, and suggests next best actions to accelerate resolution.
- **Conversational Interface:** enables employees and agents to interact with the platform using natural language, retrieving relevant knowledge base articles, troubleshooting steps, or policy details without manually navigating the system.
- **Proactive Remediation:** analyzes historical data and live telemetry from connected systems to predict incident impact, highlight similar past cases, and recommend resolution steps or automation scripts.
- **Trust and Governance:** includes enterprise-grade controls for data privacy and model transparency, aligning AI-powered actions with organizational compliance requirements.

PagerDuty Advance

PagerDuty Advance is a set of generative and agentic AI capabilities embedded across the PagerDuty Operations Cloud [24]. It operates within collaboration platforms like Slack and Microsoft Teams, delivering timely summaries, status updates, diagnostic suggestions, and post-incident insights directly to responders.

PagerDuty Advance includes the following features:

- **PagerDuty Advance Assistant**, a generative AI chatbot that reduces mean time to resolution (MTTR) by providing responders with incident summaries, anticipating diagnostic questions, and suggesting troubleshooting steps. It has integrations to work directly from Slack or Microsoft Teams [25].
- **PagerDuty AI Agents** which autonomously handle repetitive tasks, freeing up human capacity for higher-value work. [26] Current early access agents

include *Insights* (uses previous Incidents, Services, and Teams analytics as knowledge base), *Shift* (monitors calendars, detecting potential conflicts, identifying qualified replacements, and coordinating schedule changes), and *Scribe* (ingests video conference transcripts in real-time, enhancing Incident Summarization features like status updates by including transcripts from both active and finished meetings).

- A native **Integration with Amazon Q** connects PagerDuty Advance with Amazon Q Business applications, enabling teams to retrieve insights from organizational tools (e.g., Confluence, ServiceNow, GitHub) within chat, improving context and accelerating root cause analysis.
- Generative AI is used to draft internal **status updates** during incidents by combining Slack or Teams conversation history with PagerDuty log entries: info is gathered from the incident-associated channel or chat/meeting and the service-associated channel.

2.2.3 Enterprise AIOps Platforms

Enterprise AIOps platforms extend beyond single-domain assistants by providing large-scale, end-to-end correlation of signals across diverse monitoring, logging, and ticketing systems. Unlike focused observability or incident assistants, these platforms aim to unify data from the entire IT ecosystem (applications, infrastructure, and business services) into a single operational intelligence layer.

Representative examples include *IBM Cloud Pak for AIOps*, which focuses on automated anomaly detection and contextual reasoning, and *Atlassian Intelligence*, which emphasizes collaboration and workflow automation within enterprise environments.

IBM Cloud Pak for AIOps

IBM Cloud Pak for AIOps is an AIOps platform that deploys advanced, explainable AI using the IT Operations (ITOps) toolchain data to enhance companies' capabilities to assess, diagnose, and resolve incidents across critical workloads. [27] IBM Cloud Pak for AIOps helps to uncover hidden insights from multiple sources of data, such as logs, metrics, and events, delivering those insights directly into tools already in use, such as Slack or Microsoft Teams, in near real-time.

Key capabilities of IBM's service are:

- **Unified event, alert, and incident management**: IBM Cloud Pak for

AIOps consolidates raw events, alerts, and incidents into a single view, applying AI-driven correlation and context to reduce noise and provide a holistic understanding of issues across applications, infrastructure, and networks;

- **AI-driven insights and modeling**: Built-in models support change risk assessment, log and metric anomaly detection, event grouping, and ticket similarity. These models can be trained with enterprise data, enabling predictive detection of anomalies, proactive risk evaluation, and faster resolution of recurring issues;

- **Topology and resource visibility**: The platform provides real-time and historical topology visualization, linking events and incidents directly to affected resources and applications. This application-centric view highlights dependencies, pinpointing incidents' root causes;

- **Automation and remediation**: Policies, runbooks, and actions automate routine responses and remediation steps, from restarting failed services to provisioning capacity, accelerating resolution.

- **Collaboration through ChatOps**: system monitoring and automation tasks are executed directly through the integration of collaboration tools like Slack and Microsoft Teams, which brings all previously discussed features into team workflows.

- **Extensible integrations and infrastructure automation**: IBM Cloud Pak for AIOps connects with diverse IT platforms, monitoring tools, and service management systems (e.g., PagerDuty, ServiceNow, etc.), reducing the needs for custom, ad hoc connections.

Atlassian Intelligence

This service is Atlassian's integrated AI platform built into its suite of cloud tools (Jira, Confluence, Atlassian Analytics, etc.), designed to enhance collaboration, knowledge discovery, and operational workflows using generative AI [28].

At the architectural level, Atlassian Intelligence is powered by **Rovo**, Atlassian's enterprise AI framework [29]. Rovo acts as the foundation layer that orchestrates LLMs, connects them with organizational knowledge graphs, and enforces permission-aware data access. It also takes care to embed responsible AI practices: maintaining data privacy, respecting permissions, and giving administrators control over usage and availability. Current main capacities include:

- **Natural Language Queries:** Users can interact with Jira, Confluence, Opsgenie and other Atlassian tools using plain language prompts, turning questions into structured searches, filters, and queries across issues, projects, and documentation.
- **Knowledge Summarization:** Automatically generates concise summaries of long documents, pages, or tickets, helping users and teams to extract the most relevant context.
- **Automated Ticket Assistance:** Supports Jira Service Management by suggesting request categorizations, automating triage, and providing intelligent recommendations for resolving support issues.
- **Content Generation:** Helps teams draft content, refine communication, and accelerate collaboration by generating or rewriting text directly within Confluence and other tools.
- **Responsible AI and Governance:** Designed with enterprise security in mind, Atlassian Intelligence respects user permissions, maintains privacy, and gives administrators control over its usage and availability: focused on transparency, trust, accountability, human-centricity, and teamwork.

2.2.4 Limitations of Current Approaches

Let us now discuss some of the limitations of the presented solutions. Architecturally, most solutions adopt rigid frameworks that constrain flexibility: some services, such as Dynatrace's OneAgent, rely on a single embedded agent that centralizes data collection and reasoning within a closed platform, while others require organizations to migrate or replicate their observability and knowledge bases into proprietary data stores (e.g., Dynatrace's Grail, ServiceNow's platform). These approaches introduce vendor lock-in, increasing data movement costs and complicating deployment in heterogeneous environments where multiple monitoring and service management tools are required to coexist.

Furthermore, the maturity of many offerings remains limited. A number of advanced features are currently in early access or experimental stages (see Dynatrace Davis AI or ServiceNow's Now Assist), meaning that advertised capabilities such as autonomous remediation, code generation, or agentic investigation are not yet robustly validated in production at scale. This maturity gap creates uncertainty for organizations evaluating adoption and raises concerns about stability and reliability.

A promising direction to overcome these limitations is the adoption of multi-agent architectures, where specialized agents collaborate rather than relying on a single

monolithic assistant. In such a framework, each agent can focus on a distinct operational domain (e.g., observability, incident management, security analysis), with specific access and action authorizations, while coordination mechanisms ensure coherent decision-making and workflow orchestration. Integration with external systems and data sources can be streamlined through the Model Context Protocol (MCP), which exposes services as standardized servers accessible to AI agents. This approach reduces the need to migrate observability or knowledge bases into proprietary databases, allowing agents to query live data in place and lowering integration overhead.

2.3 External Services and Platforms

This section outlines the external platforms and technologies integrated within the system, which are specialized to the domain in which we deployed it.

2.3.1 Containerization

Containers are lightweight, isolated execution environments that package an application together with all of its dependencies, including libraries, runtime environments, and configuration files. By encapsulating these elements in a single portable unit, containers ensure that software runs consistently across different systems, regardless of differences in the underlying operating system or infrastructure.

Unlike traditional virtualization approaches that rely on full virtual machines, containers share the host operating system while maintaining process-level isolation. This makes them significantly more lightweight and faster to start, while still providing a controlled environment for application execution. As a result, containers have become a fundamental building block of modern cloud-native architectures.

The use of containers is particularly important in distributed systems composed of multiple interacting services, such as the agent-based architecture proposed in this work. In such environments, each service or agent may require specific dependencies, libraries, or runtime configurations. Containers allow each component to run in its own isolated environment, preventing conflicts between software dependencies and ensuring reproducibility across development, testing, and production environments.

For example, an observability agent responsible for analyzing monitoring data may require a specific Python runtime and specialized machine learning libraries, while another agent responsible for interacting with ticketing platforms may depend on different APIs or frameworks. By packaging each agent inside its own container, both components can operate independently while still communicating through

defined interfaces.

Managing large numbers of containers, however, introduces additional complexity. Systems must be able to deploy containers across machines, scale them dynamically based on workload, monitor their health, and restart them automatically in case of failure. This is the role of **container orchestration platforms** such as Kubernetes.

Kubernetes is an open-source platform designed to automate the deployment, scaling, and management of containerized applications. It organizes containers into logical units called *pods*, distributes them across clusters of machines, and ensures that the desired system state is maintained. For instance, if a container fails or a machine becomes unavailable, Kubernetes can automatically restart the affected service on another node. Additionally, Kubernetes supports load balancing, service discovery, and rolling updates, enabling reliable operation of complex distributed systems.

2.3.2 Amazon Web Services (AWS)

Amazon Web Services (AWS) offers a comprehensive ecosystem of cloud-based tools and managed services that form a robust backbone for designing, deploying, and operating agentic frameworks at scale. Its modular service model enables seamless integration between AI-driven reasoning components, containerized infrastructure, and monitoring utilities—ensuring both flexibility and operational resilience.

In this architecture, AWS serves as the primary deployment and execution environment. Managed AI capabilities are provided through *Amazon Bedrock* [30], which offers access to state-of-the-art foundation models (large pre-trained AI models capable of performing multiple tasks such as text generation, reasoning, and code synthesis) without requiring dedicated hosting or fine-tuning infrastructure (the computational resources required to retrain or adapt large models for specific tasks). Observability and telemetry are handled through *Amazon CloudWatch* [31], which collects logs and metrics from cloud resources and feeds them to observability agents for anomaly detection and performance analytics. Additionally, the *Boto3 SDK* [32] acts as the programmatic interface for accessing AWS APIs, enabling dynamic integration with a broad range of services and acting as a fallback for components without dedicated MCP connectors. Finally, authentication and user access control are handled by *Amazon Cognito* [33], which provides secure identity management and single sign-on (SSO) integration for frontend and API access. Table 2.1 shows a brief summary of the main services we used to implement our solution.

While the presented framework leverages Amazon Web Services for deployment,

taking advantage of managed services and established APIs, the overall architecture remains cloud-agnostic, as the modular design of agentic components and communication protocols allows the solution to be readily adapted to other environments, whether on alternative cloud providers or on-premise infrastructure, with only minor changes in service integrations.

Strands Agents

The backbone of the agentic infrastructure is built upon the Strands library for Python, a lightweight yet extensible framework designed for developing, orchestrating, and deploying autonomous or semi-autonomous AI agents. Strands abstracts the complexity of multi-agent communication, state management, and tool invocation through a unified protocol that standardizes interactions across heterogeneous systems.

At its core, the framework defines an `Agent` class, representing a reasoning-capable component that can process user inputs, access external tools (through MCP endpoints), and produce structured outputs in JSON format.

The framework also provides session persistence and event tracing capabilities, allowing developers to monitor interactions and reproduce agent behaviors deterministically. Thanks to its modular architecture and adherence to open communication standards, Strands integrates seamlessly with external model providers (through AWS Bedrock), and in general can access AWS systems through `Boto3` clients.

2.3.3 Atlassian Services

Atlassian provides a suite of collaboration and project management tools widely adopted in enterprise environments. In particular, *Jira* ([34]) and *Confluence* [35] enable structured issue tracking, knowledge management, and documentation. Table 2.2 shows key features of both these services, which can be integrated into agentic frameworks to provide agents with access to ticketing data, change histories, and organizational knowledge. Both services have dedicated MCP servers exposing their capabilities through standardized interfaces.

Service	Domain	Main Purpose
Amazon Bedrock	Managed generative AI models	Provides access to foundation models for reasoning, planning, and natural language interaction without requiring model hosting or fine-tuning infrastructure. It's used as bridge for the agents to connect with state of the art models such as Anthropic's or OpenAI's LLMs.
Amazon EKS (Elastic Kubernetes Service)	Container orchestration	Provides a managed Kubernetes environment for deploying and scaling containerized workloads. In this project, it is used to analyze production cluster states, allowing agents to retrieve metrics, logs, and diagnostics for observability and incident resolution.
Amazon CloudWatch	Monitoring and observability	Collects and visualizes operational metrics and logs from AWS resources. In this work, CloudWatch data is accessed by observability agents to extract insights about service performance, detect anomalies, and correlate incidents with infrastructure-level events.
Amazon Cognito	Identity and access management	Provides authentication, authorization, and user management for web and mobile applications. In this project, Cognito is used to control access to the frontend interface and secure communication between users and the agentic platform, supporting user pools and SSO integration.
Boto3 SDK	AWS API interaction	The official AWS Python SDK used to programmatically access most AWS services via codified API requests, allowing to reach disparate services and resources and working as a jolly in our architechture (via this service, we can access all services which still haven't got a dedicated mcp server).

Table 2.1: Core AWS services supporting the deployment of agentic frameworks

Service	Domain	Main Purpose
Jira	Issue and project tracking	A centralized platform to create issues (tasks, bugs, user stories), assign them to team members, track their progress, and monitor project status with dashboards and reports.
Confluence	Knowledge management and documentation	A centralized workspace for teams to create, share, and organize project-related information, acting as a digital knowledge base or "wiki" for the organization.

Table 2.2: Atlassian services integrated into the assistant architecture

Chapter 3

Methodology

3.1 Design

Let us now deep dive into how we designed all the components which make up the proposed solution, starting with the back end, MCP servers and agentic framework all the way to the front end and user interface. The goal is to show how each component was conceived to ensure modularity, scalability, and reliability.

3.1.1 High-Level System Diagram

Figure 3.1 illustrates the overall structure of the proposed solution, organized into logical groups: *Front End* and *Back end*, the latter composed of an *MCP Layer* (made of *Agentic Layer* and *Server Layer*) plus external services connections.

At the top level, the Front End provides user access through a web interface protected by Single Sign-On (SSO) authentication. Once authenticated, users interact with the system through API requests routed to the back-end agentic environment. The User Interface is designed as an interactive chat environment, enabling continuous communication between the user and the system. The underlying agents operate in a stateful manner, maintaining conversational memory to keep track of prior queries, responses, and task progress. This persistent context allows the system to handle follow-up questions and refine earlier outputs, rather than set for isolated command executions.

The back-end is composed of two main subsystems, grouped under the *MCP Layer*:

- The *Agentic Layer* hosts the reasoning agents, each representing a distinct operational role. The DevOps Agent interfaces directly with production

environments, holding write privileges to perform controlled actions, while the Knowledge Agent acts as an information retriever, capable of answering documentation-related queries about the monitored systems. The orchestrator Agent works as entrypoint for the application, as is the first to receive the queries and can redirect them to whichever agent seems the most fit to interpret user intent, decompose it in tasks, and execute them by delegating specific operations to the underlying MCP Layer.

- The *Server Layer* consists of modular MCP servers that expose domain-specific tools and APIs through the Model Context Protocol. Each server is specialized for a given ecosystem (Confluence, Jira, AWS) and is accessed only by the agent authorized to use it. This selective connectivity ensures that each agent remains confined to its operational scope, promoting both specialization and security. Every agent and MCP server operates as an independent microservice, encapsulating its domain logic and dependencies.

3.1.2 Server Layer

Inside the MCP layer, the *Server Layer* constitutes the foundation of the system's back-end architecture. It provides a standardized and extensible way for agents to access external systems through independently deployed microservices. This layer represents the "tooling substrate" of the system: a flexible foundation upon which reasoning agents can reliably build and orchestrate complex workflows across heterogeneous enterprise systems. Each server abstracts away the raw API complexities of the underlying service and presents a consistent interface through the MCP protocol. In the proposed solution, this layer was implemented using a set of modular servers designed to interact with enterprise ecosystems such as AWS and Atlassian, although the architecture is fully extensible: new MCP servers can be introduced for other systems or vendors by implementing the same initialization, serialization, and transport logic, and they would seamlessly plug into the agentic orchestration layer.

The main difference between a simple tool and an MCP one (i.e., provided by a MCP server) is that the server standardizes the tools-as-a-service. It separates the tool provider from the user's application. The idea is that tools in MCP servers can be accessed by the custom application regardless of the specific agent framework or LLM instance (based on AWS or Microsoft, for example) used by the application itself.

The AWS MCP Server family [36], for once, offers a broad spectrum of capabilities: from infrastructure management (provisioning resources via the Cloud Control

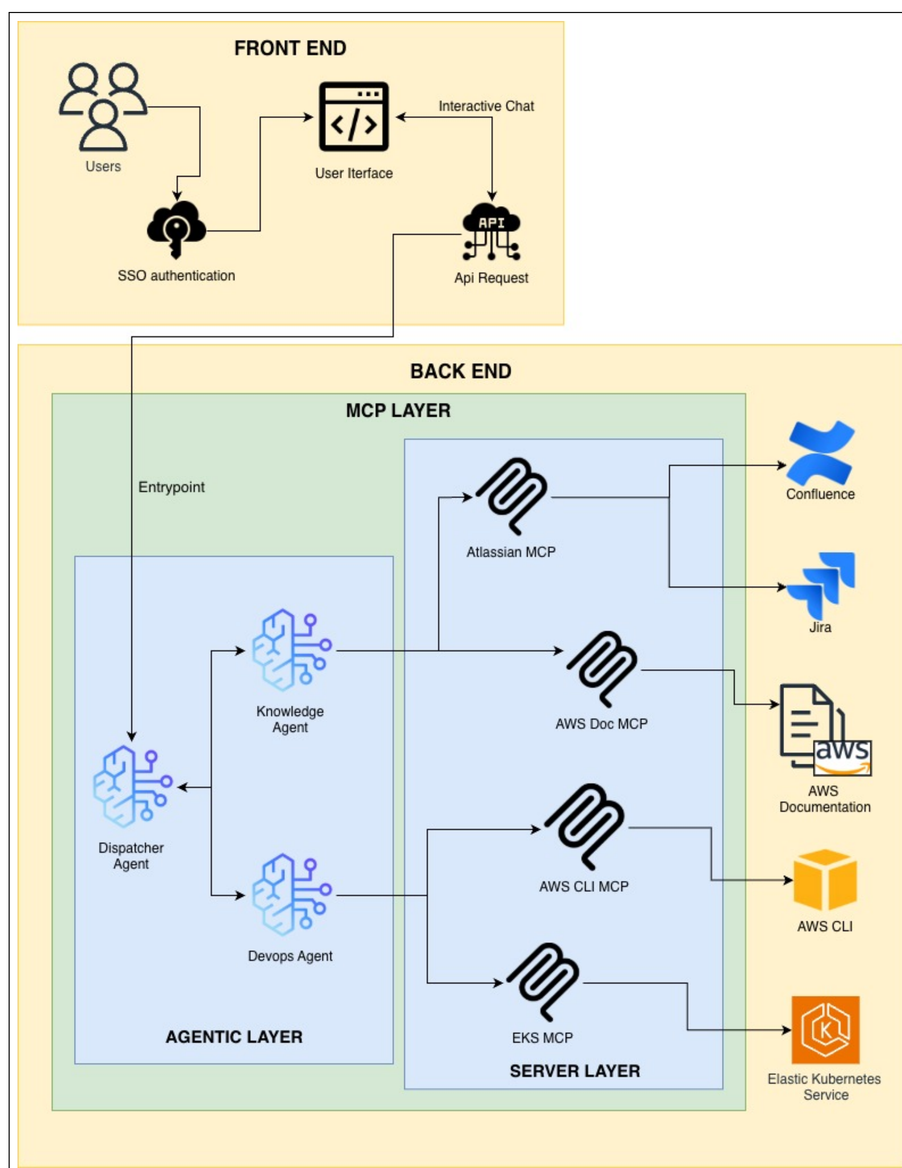

Figure 3.1: High Level Design of the proposed solution, showing the layer compositions and the interaction between front and back ends.

API) to cost analytics (via the Pricing and Cost Explorer servers) and operational monitoring (through CloudWatch and Billing assessment servers). In practice this means an agent could ask: "What will this EKS deployment cost per month?" or "Retrieve the metrics and alarms for this Lambda function" and the AWS MCP tools would deliver structured data and actionable responses.

The Atlassian MCP Server (for Jira, Confluence and related collaboration platforms)

[37] enables tasks such as issue retrieval, ticket updates, knowledge-base searches, and documentation orchestration, thereby allowing agents to handle workflows involving ticket tracking ("Show me the most recent pending tickets"), or user support ("Show the resources involved in this ticket looking in the documentation").

As we designed it, the architecture currently integrates several operational MCP servers, each providing a distinct set of capabilities:

- The *AWS CLI* MCP Server works as a bridge between AI assistants and AWS services, exposing low-level resource management tools which enable agents to interact with AWS services and resources through AWS CLI commands, allowing them to create, update, and manage AWS resources across all available services. It helps with AWS CLI command selection and provides access to the latest AWS API features and services, even those released after an AI model's knowledge cutoff date.

- Amazon *EKS* MCP Server provides AI code assistants with resource management tools and real-time cluster state visibility. It enables agents to create new EKS clusters, complete with prerequisites such as dedicated VPCs, networking, and node pools. It provides the ability to deploy containerized applications by applying existing Kubernetes YAML files or by generating new deployment and service manifests based on user-provided parameters, and supports full lifecycle management of individual Kubernetes resources (such as Pods, Services, and Deployments) within EKS clusters, enabling create, read, update, patch, and delete operations. Furthermore, it provides the ability to list Kubernetes resources with filtering by namespace, labels, and fields, simplifying the gathering of information about the state of Kubernetes applications and EKS infrastructure. Finally, it facilitates operational tasks such as retrieving logs from specific pods and containers or fetching Kubernetes events related to particular resources, supporting troubleshooting and monitoring for both direct users and AI-driven workflows.

- *AWS Documentation* MCP Server offers read-only access to cloud documentation resources, providing tools to access AWS documentation, search for content, and get recommendations.

- The *Atlassian* MCP Server connects the agentic layer to the Jira and Confluence APIs, allowing for issue retrieval, ticket updates, and knowledge extraction. This integration allows agents to deploy automatic documentation updates on Confluence (e.g. given some meeting notes) and provides semantic Confluence search ("Find and summarize info about this topic"), as well as smart Jira Issue filtering on date, priority, or other provided labels. It can

also create and manage documents, fields, issues, tasks, and any Atlassian entity (e.g. "Create a tech design doc for given project").

3.1.3 Agentic Layer

Aside the MCP servers lies the *Agentic Layer*, which acts as the cognitive and orchestration core of the system. Each agent is a stateful reasoning entity built around a Large Language Model (LLM) and equipped with domain-specific tools exposed through the MCP protocol. The agents collaborate by exchanging structured messages within an Agent-as-tools environment, which features hierarchical delegation and separation of concerns.

The *orchestrator Agent* is responsible for analyzing the incoming prompt and it determines which reasoning path to follow. Depending on its classification, the request is then handed off to one or more specialized agents registered within its tools.

Each agent has a well-defined operational domain and a specific set of permissions aligned with the corresponding MCP servers, of which the orchestrator is well aware. In this configuration, each agent is fully autonomous yet interdependent. The orchestrator Agent initiates task decomposition; the specialized agents contribute domain expertise, dealing with the respective task activities. Each agent can have tailored system prompts and tools optimized for its specific task.

Agent details

The structure is currently made of three AI agents: two domain specialists plus one orchestrator. As previously stated, we modeled the domain specific agents after real-world professional roles within an operations environment:

- The *orchestrator Agent* serves as the coordination hub and entry point for all user interactions. It receives user prompts, interprets intent, and determines the most appropriate course of action. When the request is conversational or non-technical in nature, the orchestrator responds directly to the user. Otherwise, it routes the task to one or more specialized agents, while embedding high-level instructions that propagate across the entire reasoning chain (like language consistency or visualization standards) maintaining coherent behavior across agent handoffs. The orchestrator has access to two agent-as-tool functions:
  - `devops_assistant`: Routes queries to the DevOps Agent

  - `knowledge_assistant`: Routes queries to the Knowledge Agent

- The *DevOps Agent* is responsible for direct operational actions on the cloud infrastructure itself, such as validating deployments, inspecting logs, or performing controlled updates and remediations. It connects primarily to the AWS CLI and EKS MCP Servers, allowing it to manipulate live resources under strict IAM-based access control. It shall perform only safe, reversible actions, reporting exactly what was executed and verified, and always double-checking with the user before unsafe or unclear actions are enacted. Example queries are "Scale worker nodes to 5" and "Check database pod health". The DevOps Agent has access to the following tools:

  - **AWS CLI MCP Server:**
    - `suggest_aws_commands`: Suggests appropriate AWS CLI commands based on natural language descriptions
    - `call_aws`: Executes AWS CLI commands with proper authentication and returns results. Can execute ANY AWS CLI command, including creating/deleting S3 buckets, starting/stopping EC2 instances, modifying security groups, creating/deleting IAM roles, updating RDS configurations, etc.
  - **EKS MCP Server:**
    - `get_cloudwatch_logs`: Retrieves CloudWatch log streams for specified resources
    - `get_cloudwatch_metrics`: Fetches CloudWatch metrics for monitoring and diagnostics
    - `list_k8s_resources`: Lists Kubernetes resources (pods, services, deployments, etc.)
    - `get_pod_logs`: Retrieves logs from specific Kubernetes pods
    - `get_k8s_events`: Fetches Kubernetes cluster events for troubleshooting
    - `list_api_versions`: Lists available Kubernetes API versions
    - `manage_k8s_resource`: Can create, update, patch, or delete Kubernetes resources
  - **Default Tools:**
    - `handoff_to_user`: Escalates to human operator when needed
    - `current_time`: Retrieves current timestamp for logging and scheduling

- The *Knowledge Agent*, in contrast, serves as an information retriever and summarizer: it can read and interpret documentation from cloud providers

(like aws), access and reference internal materials hosted on Confluence, and interact with Jira to read or update tickets. Its purpose is to provide accurate, context-aware information, summarize documentation, and support operational processes by linking insights with existing tickets or documentation. It is the best at answering requests like "Give me information about the ticket ABC-123" or "Explain S3 bucket encryption". It doesn't execute write operations on the infrastructure itself, but it can access and modify both documentation and tickets. The Knowledge Agent has access to the following tools:

- **AWS Documentation MCP Server:**
  - `read_documentation`: Retrieves specific AWS documentation pages
  - `search_documentation`: Searches AWS documentation for relevant content
- **Atlassian MCP Server:**
  - `jira_get_user_profile`: Retrieves Jira user information
  - `jira_get_issue`: Fetches detailed information about specific Jira issues
  - `jira_search`: Searches Jira issues using JQL (Jira Query Language)
  - `jira_search_fields`: Searches available Jira fields and metadata
  - `jira_get_project_issues`: Lists all issues within a specific project
  - `jira_get_transitions`: Retrieves available workflow transitions for an issue
  - `jira_get_link_types`: Lists available issue link types
  - `jira_get_all_projects`: Retrieves all accessible Jira projects
  - `confluence_search`: Searches Confluence pages and spaces
  - `confluence_get_page`: Retrieves specific Confluence page content
  - `confluence_get_page_children`: Lists child pages of a Confluence page
- **Default Tools:**
  - `handoff_to_user`: Escalates to human operator when needed
  - `current_time`: Retrieves current timestamp for documentation

3.2 Deployment

In this section, we outline the underlying technologies required to deploy the proposed solution locally as a containerized application connected to real cloud

networks with functional resources. The deployment environment leverages containerization technologies to ensure isolation and portability, enabling the system to be executed both on local machines and lightweight virtualized environments.

Each agent or server component is encapsulated within its own container, ensuring reproducibility and eliminating dependency conflicts. Services are independently buildable, updatable, and replaceable, which enhances modularity and extensibility and allows, for instance, the substitution of a server component or the addition of new agents without affecting the rest of the system.

The local setup emulates a complete cloud-based deployment, including orchestration, networking, and inter-service communication layers.

3.2.1 Low-Level Design

The previously introduced High-Level Design diagram illustrated the conceptual structure of the architecture, abstracting away the underlying communication substrate. As discussed in the earlier sections, the system is composed of:

- A set of **MCP servers**, each exposing domain-specific APIs (AWS, Confluence, Jira).
- A layer of **agents**, responsible for orchestrating specialized reasoning workflows by interfacing with the MCP services.
- A **front end interface** (*OpenWebUI*) for user interaction.
- A **language model gateway** (*LiteLLM*) providing a unified API for model inference.

Figure 3.2 illustrates the Low-Level architecture, which mirrors the logical structure of the High-Level Design while incorporating the concrete deployment technologies that empower each layer.

The following sections describe, in a bottom-up manner, how each layer of the back-end is deployed, followed by the front-end configuration and a discussion on portability and reproducibility.

3.2.2 Server Layer

Each MCP service is deployed as an independent container, exposing their APIs on distinct local ports.

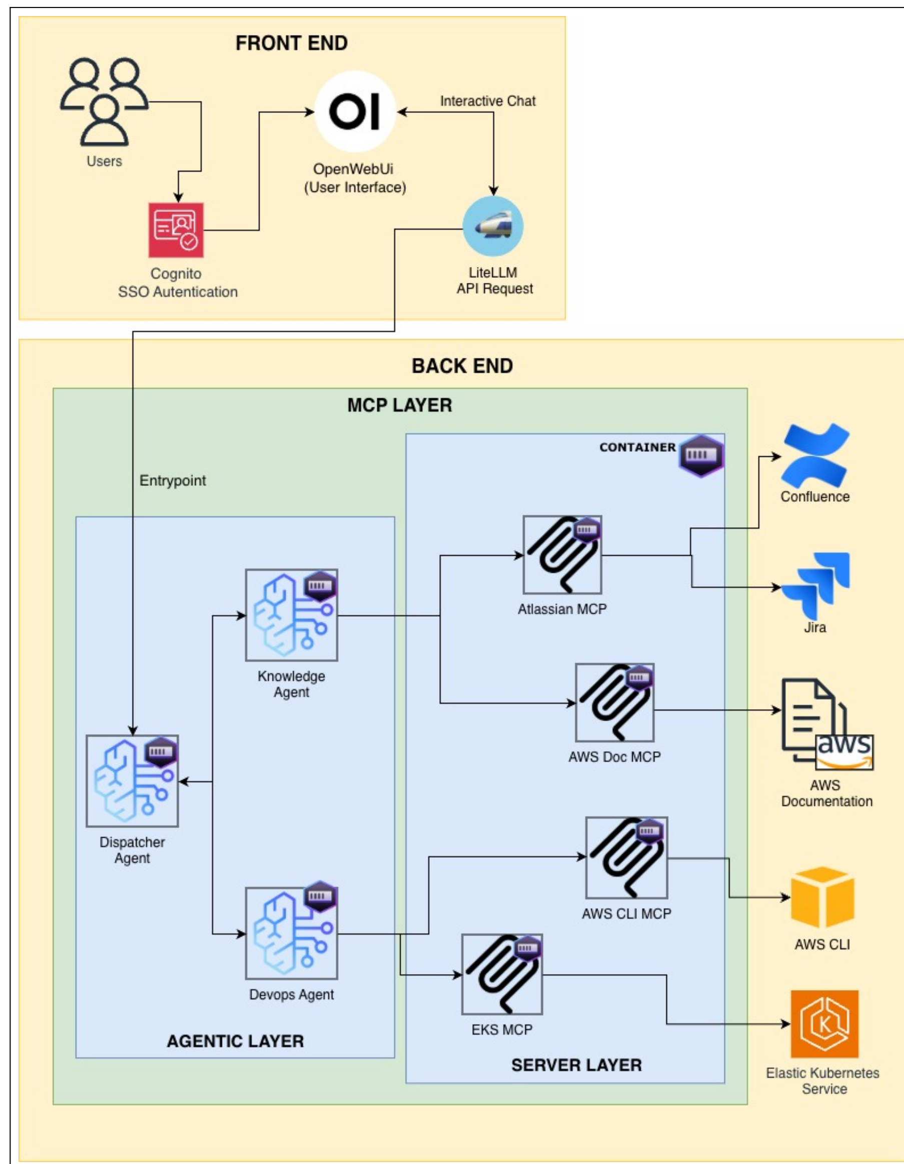

Figure 3.2: Low-Level Design of the architecture, showing the technologies powering the front-end and back-end layers.

The deployment is orchestrated through a container composition framework, which automates the creation, configuration, and interconnection of multiple service containers defined within a declarative YAML specification. This orchestration layer ensures that each service instance is instantiated within a shared virtual network, enabling seamless inter-service communication (shared environment variables).

At runtime, each container initializes an MCP endpoint with domain-specific

extensions. Each server then verifies that all mandatory environment variables are available and correctly set. This is to ensure that the mcp tools will be able to provide access to the specific account's resources (for example, the assistant will have access to all resources linked to provided atlassian profiles). For AWS environments, the service automatically assumes an IAM role and configures the corresponding AWS profile to ensure that all downstream API calls are authenticated and scoped to the correct account.

Once initialized, the MCP server starts a lightweight FastAPI-based service that exposes its capabilities under the MCP protocol, ready to handle agent queries and provide structured outputs (e.g., resource metadata, API responses, or analytical summaries). The server remains stateless and ephemeral: stateful information, such as AWS configuration parameters, is instead retrieved on demand or cached in temporary session contexts.

This modular deployment pattern ensures that each integration domain can be maintained, scaled, or replaced independently, without affecting other components in the architecture. It must also be noted that each MCP server exposes all its tools, but we can choose dynamically at deploy time which tools from which server we want an agent to access and dispose of, enhancing modularity in the solution.

3.2.3 Agentic Layer

As already stated, each agent operates as an independent component with a specific purpose, and communicates with external MCP servers through standardized interfaces. Each agent instance (implemented using the Strands Agents SDK) is defined by a set of parameters, including its description, model configuration, and the list of MCP endpoints it can access. Standardized interfaces and contracts ensure uniform behavior across all agents: fixed port bindings, streamable HTTP transport, structured logging, JSON-formatted error responses, and a consistent configuration (achieved through environment variables).

As stated in the Design section, we employ specialized agents as callable tools that can be invoked individually or in structured sequences. Each agent-tool operates within a shared execution framework that manages session context, propagates relevant state, and enforces safeguards such as repetition detection, iteration limits, and termination conditions to prevent unbounded execution.

The tool framework enables multiple agent-tools to contribute to the same task by exchanging intermediate outputs through shared memory and contextual inputs. This allows later agent invocations to build upon earlier reasoning in a controlled and incremental manner. By centrally managing context propagation, convergence checks, and redundancy prevention, the system ensures consistency across agent-tool

interactions while avoiding unnecessary hand-offs and context overflow.

By design, each agent is implemented as a self-contained, tool-like microservice that can be defined, instantiated, scaled, or replaced independently. As a result, changes to one agent-tool are isolated and do not propagate unpredictably through the system.

Each agent-tool is designed to operate independently of any specific orchestration construct, allowing it to be invoked standalone or composed with other tools as needed. This modular approach supports horizontal scaling under variable workloads and simplifies integration with cloud-native infrastructure components, such as Elastic Container Service (ECS), enabling flexible in-cloud deployment and future extensibility.

Context and Session Management

The multi-agent system implements a sophisticated session management architecture to maintain conversation continuity across multiple concurrent users and chat sessions. This design addresses two critical challenges in production agentic systems:

1. isolating conversation state between different users and chat sessions;
2. managing the finite context window limitations of large language models during extended interactions.

Dynamic Session Routing At the core of the session management design is a custom Dynamic Session Manager that routes requests to isolated session storage based on unique session identifiers. When users interact with the system through the OpenWebUI interface, each chat is assigned a unique identifier that is transmitted via HTTP headers. The Session Manager intercepts incoming requests through the Strands framework's hook system, extracting the session identifier from the invocation state and creating or retrieving a dedicated file session instance for that specific session.

This architecture ensures complete isolation between concurrent conversations. Each session maintains its own persistent storage on disk, containing the full conversation history, intermediate agent results, and routing decisions made by the orchestrator agent. The session manager integrates seamlessly with the multi-agent graph execution lifecycle, automatically saving state after each node execution and restoring it when the session is resumed. This persistence mechanism allows conversations to survive system restarts and enables users to seamlessly switch between multiple ongoing conversations without context contamination.

Context Window Management To address the finite context window limitations inherent in LLMs, the system employs a Summarizing Conversation Manager that intelligently compresses conversation history when approaching token limits. This component implements a two-tier preservation strategy: it maintains the most recent messages in their original form (configured to preserve the last 5 messages) while summarizing older interactions to reduce token consumption.

The summarization process is specifically tuned for technical infrastructure conversations. The system prompt instructs the summarization agent to preserve critical technical details including AWS resource names and identifiers, command outputs, error messages, and the sequence of operations performed, while omitting conversational elements and redundant information. The summary ratio is configured at 50%, meaning that when the context window reaches capacity, approximately half of the older messages are compressed into a concise summary formatted as bullet points.

3.2.4 Front-End

User interface

We finally set up a web interface for conversational interaction with the multi-agent system. The frontend serves primarily as an interaction hub rather than a computational component. It provides a ChatGPT-like interface for sending and receiving messages through the same APIs exposed by the LiteLLM service, which acts as a local routing gateway to the model backend. Through this interface, users can issue natural language requests, visualize structured responses, and access all history and previous chats.

It is implemented through a lightweight web application, **OpenWebUI**, that exposes conversational capabilities and monitoring panels, enabling real-time interaction with the deployed agents and their associated MCP services.

In architectural terms, the frontend functions as a stateless client connected to the backend APIs via HTTP and WebSocket channels. It leverages REST endpoints to send user queries and retrieve responses, while persistent sessions and contextual state are managed entirely by the backend. We ensured chat isolation, which means each conversation will retain only the current chats context, and no leak will happen between chats of the same user (a new chat is considered as a new session entirely from the back end). This separation ensures scalability, as the frontend can be replicated or served via standard web hosting without impacting backend performance or logic.

Developer Interface

Developers can interact with the deployment through a set of auxiliary tools designed to support testing, debugging, and performance analysis activities. These interfaces facilitate both low-level inspection of service behavior and high-level monitoring of agentic workflows, in order to enhance observability (and maintainability) of the system.

- **LangFuse:** LangFuse serves as the observability and tracing platform for the agentic ecosystem. It collects detailed execution traces, metrics, and model-level statistics across all agent interactions and tool invocations, grouping different invocations of the same model (e.g., a chat or conversation) under the same session via a dedicated ID. Each request processed by an agent or MCP service can be traced end-to-end, including intermediate reasoning steps, tool calls, and response latencies. This allows to identify performance bottlenecks, monitor model outputs for quality assurance, and analyze conversation histories for optimization or debugging. It also helps to detect if the consumption of input tokens gets too heavy, with impacts on costs and performances. LangFuse can be accessed through a secured web interface (VPN required), and integrates seamlessly with the logging infrastructure to ensure synchronized observability across both model and service layers.
- **MCP Inspector:** The MCP Inspector is a lightweight command-line and browser-based tool that allows developers to manually connect to and test individual MCP endpoints. It enables validation of server configurations, tool accessibility, and output schemas by simulating agent requests in an isolated environment. Through the inspector, each MCP server can be queried interactively to verify that endpoints respond according to the Model Context Protocol specifications and that environment credentials or role assumptions are functioning correctly.

3.2.5 Portability and Reproducibility

The deployment architecture is designed to maximize portability and reproducibility across different environments. By leveraging containerization technologies and a declarative composition framework, the entire system, including MCP servers, agentic services, the language model gateway, and frontend interfaces, can be instantiated consistently on any host supporting container runtimes.

Although the current deployment targets an AWS environment (both as deployment field and as target resources to monitor) and integrates with well known cloud

services such as Jira or Confluence, the system is cloud-agnostic by design. All service endpoints, credentials, and environment-specific parameters are injected via configuration files or environment variables, enabling the same deployment to be replicated on other cloud platforms or on-premises infrastructures without heavy modification to the core codebase.

The modularity of the architecture further enhances reproducibility. Each MCP server, agent, and frontend component is self-contained and can be independently built, replaced, or scaled. Agents can be extended with new reasoning capabilities or additional tools without impacting existing components. Likewise, the frontend and language model gateway can be updated or swapped for alternative implementations while maintaining the overall system behavior.

This approach ensures that experiments, tests, or production deployments can be reproduced reliably across different machines or environments. It also supports continuous integration and continuous deployment (CI/CD) pipelines, allowing new components or updates to be tested in isolated environments before integration into the main system.

Chapter 4

Results

This chapter presents the experimental evaluation of the multi-agent orchestration system across quantitative and qualitative dimensions. Section 4.1 describes the experimental setup, evaluation methodology, and quantitative performance analysis comparing three foundation models. Section 4.2 provides detailed qualitative analysis of response quality across representative operational scenarios.

4.1 Quantitative Response Evaluation

To evaluate the performance and effectiveness of the multi-agent orchestration system, we designed and implemented a comprehensive experimental framework that measures key operational metrics across different foundation models. The evaluation methodology focuses on four primary dimensions: response latency, computational cost, token efficiency, and task accuracy.

Hardware and Infrastructure

All experiments were conducted on the production deployment infrastructure to ensure realistic performance measurements:

- **Compute Environment**: AWS ECS (Elastic Container Service) cluster running in eu-west-1 region
- **Container Specifications**:
  - CPU: 4 vCPUs (AWS Fargate)

  - Memory: 8 GB RAM
  - Network: VPC with NAT Gateway for external API access

- **Model Inference**: AWS Bedrock API endpoints (eu-west-1)
- **Observability**: Langfuse for trace collection and analysis

Model Selection and Configuration

To evaluate the impact of foundation model selection on system performance, experiments were conducted with three different models available through AWS Bedrock:

1. **Claude 4.5 Sonnet** (`eu.anthropic.claude-sonnet-4-5-20250929-v1:0`): Latest generation Anthropic model released September 2025, representing state-of-the-art performance in reasoning and tool use.
2. **Claude 3.5 Sonnet** (`eu.anthropic.claude-3-5-sonnet-20240620-v1:0`): Previous generation model released June 2024, providing baseline comparison for generational improvements.
3. **Amazon Nova Pro** (`eu.amazon.nova-pro-v1:0`): AWS-native model released December 2024, offering potential cost advantages and regional optimization.

All models were configured with identical parameters to ensure fair comparison:

- **Temperature**: 0.0 (deterministic output for reproducibility)
- **Max Tokens**: 4096 (sufficient for comprehensive responses)
- **Top-p**: 1.0 (no nucleus sampling)
- **System Prompts**: Identical across all models, defining agent roles and tool usage protocols
- **Tool Definitions**: Identical MCP tool schemas provided to all models

For each model, the complete test suite was executed under identical conditions, with only the model identifier modified in the agent configuration. This approach isolates the model as the independent variable while controlling for system architecture, prompt engineering, and environmental factors.

Model Selection Rationale The selection of Anthropic's Claude as the primary foundation model provider for this study was driven by its demonstrated superiority in developer-oriented tasks and agentic workflows. While OpenAI's GPT models maintain a strong presence in general-purpose applications, Claude consistently outperforms them on the benchmarks most relevant to this research context, particularly in multi-step tool use, coding tasks, and computer-use scenarios [38].

Given that this study evaluates a multi-agent system performing DevOps operations and infrastructure automation (which involves precise tool invocation, multi-step reasoning, and technical accuracy), Claude's stronger performance on coding, agentic, and computer-use benchmarks makes it the more appropriate choice for validating the system's capabilities. The choice to test Claude over Chatgpt more closely reflects the operational context of this research, while Amazon Nova Pro was included to evaluate cost-performance tradeoffs and assess whether AWS-native models could provide competitive performance at lower operational costs (which is still a very important KPI for companies whenever it comes to evaluating whether to adopt GenAI solutions in their own workflows). [38]

Evaluation Dataset and Test Categories

The evaluation dataset consists of 40 carefully designed test cases representing realistic operational scenarios encountered in production IT environments. Test cases were organized into six functional categories based on the primary operational domain and required agent capabilities:

1. **DevOps Operations** (16 test cases): Infrastructure management tasks including EC2 instance management, Lambda function operations, S3 bucket operations, CloudWatch monitoring, EKS cluster management, VPC networking, RDS database management, and security analysis. These tests evaluate the system's ability to interact with AWS services and Kubernetes clusters through the DevOps Agent.

2. **Knowledge & Documentation** (5 test cases): Information retrieval and documentation queries spanning AWS technical documentation, Jira ticket management, and Confluence knowledge base searches. These tests assess the Knowledge Agent's capability to locate, synthesize, and present relevant information from multiple documentation sources.

3. **Observability & Monitoring** (5 test cases): System health monitoring tasks including Lambda function health reports, EKS cluster metrics analysis, pod restart detection, and cost anomaly monitoring. These tests evaluate

the Observability Agent's ability to aggregate metrics from CloudWatch and Kubernetes, identify issues, and generate comprehensive health assessments.

4. **Orchestrator & Multi-Agent Coordination** (4 test cases): Complex operations requiring coordination between multiple specialized agents, such as correlating CloudWatch alarms with Jira tickets, cross-referencing EKS metrics with Confluence runbooks, and auditing Lambda configurations against documentation. These tests evaluate the orchestrator's routing logic, inter-agent communication, and ability to decompose complex tasks into coordinated sub-tasks.
5. **Complex Workflows** (6 test cases): Multi-step operations that integrate data from diverse sources (Jira, Kubernetes, Confluence, AWS documentation) to perform comprehensive analyses, compliance checks, and troubleshooting procedures. These tests assess the system's capability to execute sophisticated workflows requiring sequential tool invocations and contextual reasoning.
6. **Data Visualization** (4 test cases): Tasks requiring the generation of visual representations including CPU utilization trends, Lambda-alarm relationship diagrams, memory usage comparisons, and S3 bucket size charts. These tests evaluate the system's ability to transform raw metrics into actionable visualizations using Mermaid diagram syntax.

Test Case Design Each test case specification includes:

- **Query**: The natural language question or command submitted to the system
- **Expected Behavior**: Description of the anticipated system response, routing, and actions (which agents should be called, in which order)
- **Expected Tools**: List of MCP tools that should be invoked to complete the task
- **Expected Keywords**: Specific terms and concepts that should appear in a correct response
- **Expected Routing**: The primary specialized agent(s) that should handle the query
- **Evaluation Criteria**: Quantitative metrics for assessing response quality

Test cases were designed to represent realistic operational scenarios while providing clear, objective evaluation criteria. The dataset balances simple single-agent queries with complex multi-agent coordination tasks to comprehensively assess system capabilities.

Execution Methodology and Session Isolation

Test Execution Framework The experimental framework is implemented as a Python-based test harness that interfaces directly with the orchestrator's API, ensuring consistent execution conditions across all experiments. The framework provides:

- **Automated Test Execution**: Sequential processing of all 40 test cases per model
- **Metric Collection**: Automatic capture of latency, token usage, cost, and trace data
- **Error Handling**: Timeout detection (300-second limit) and failure logging
- **Result Persistence**: JSON-formatted output with complete test metadata
- **Trace Integration**: Langfuse session IDs for detailed execution analysis

Session Isolation To ensure fair comparison across different foundation models and eliminate cross-contamination between test cases, each query is executed in an isolated session with no conversation history. This approach provides several methodological advantages:

- **Reproducibility**: Test results are independent of execution order, enabling consistent measurements across multiple experimental runs.
- **Baseline Consistency**: Each test begins from an identical initial state, eliminating confounding variables introduced by conversation context.
- **Fair Comparison**: Models are evaluated under equivalent conditions, with no advantage from accumulated context or cached information.
- **Parallel Execution**: Independent sessions enable concurrent test execution without interference, reducing total experiment duration.

Each test case receives a unique session identifier, ensuring complete isolation. The session manager creates separate storage instances for each test, preventing any information leakage between queries.

Execution Timeline Experiments were conducted between February 28 and March 1, 2026:

- **Claude 4.5 Sonnet**: February 28, 2026, 14:30-16:45 UTC (38/40 completed)
- **Claude 3.5 Sonnet**: February 28, 2026, 17:00-19:30 UTC (40/40 completed)
- **Amazon Nova Pro**: March 1, 2026, 16:00-17:30 UTC (40/40 completed)

Total experiment duration: Approximately 7.5 hours across all models.

Metrics and Measurement

We ensure to capture five primary metrics for each test execution:

- **Response Latency** is measured as the elapsed time from query submission to final response delivery, it encompasses the complete execution pipeline including agent routing, tool invocation, LLM inference, and response synthesis.
- **Token Usage** is tracked separately for input and output, providing insight into computational efficiency and model verbosity. Input tokens represent the cumulative prompt size across all agent invocations, including system prompts, user queries, and tool descriptions. Output tokens measure the total generated text across all agents.
- **Computational Cost**: cost is calculated based on the token usage and the pricing model for each foundation model. The total cost per query is computed as the sum of the cost of input and output tokens. This metric enables economic analysis of production deployment scenarios and cost-benefit evaluation across different models. At the time the study happens, these are the fees as shown on AWS Bedrock [39]:
  1. Claude Sonnet 3.5 and 4.5:
     (a) Input: $3.00 / 1M tokens ($0.003 / 1K tokens)
     (b) Output: $15.00 / 1M tokens ($0.015 / 1K tokens)
  2. Amazon Nova Pro:
     (a) Input: $0.80 / 1M tokens ($0.0008 / 1K tokens)
     (b) Output: $3.20 / 1M tokens ($0.0032 / 1K tokens)
- **Task Accuracy**: Accuracy is assessed through a multi-criteria evaluation framework that examines four dimensions of correctness:

1. **Routing Accuracy**: Verification that the orchestrator correctly identified and invoked the appropriate specialized agent(s) for the given query type.
2. **Response Relevance**: Presence of expected keywords and concepts in the generated response, indicating that the system addressed the core query requirements.
3. **Tool Selection**: Appropriate selection and invocation of MCP tools (e.g., AWS CLI, Kubernetes API, Jira API) based on the task requirements.
4. **Efficiency**: Penalty-based metric that discourages wasteful resource usage by penalizing excessive tool invocations or unnecessary agent coordination beyond what is required to complete the task.

Routing Accuracy Computation: The system extracts the list of agents invoked during query execution from the multi-agent graph state. For a test case specifying an expected routing target (e.g., "devops"), the routing score is computed as:

$$S_{\text{routing}} = \begin{cases} 1.0 & \text{if expected agent } \in \text{ agents_called} \\ 0.5 & \text{if expected agent mentioned in workflow} \\ 0.0 & \text{otherwise} \end{cases}$$

The partial credit (0.5) accounts for cases where the correct agent was invoked but not explicitly tracked in the execution metadata.

Response Relevance Computation: Each test case defines a set of expected keywords $K = \{k_1, k_2, ..., k_n\}$ that should appear in a correct response. The relevance score is the proportion of keywords found:

$$S_{\text{relevance}} = \frac{1}{|K|} \sum_{i=1}^{|K|} \mathbb{1}(k_i \in \text{response})$$

where $\mathbb{1}(\cdot)$ is the indicator function returning 1 if the keyword is present (case-insensitive) and 0 otherwise. For example, a test case with keywords `{"pod", "health", "status"}` would score 1.0 if all three appear, 0.67 if two appear, and so on.

Tool Selection Computation: Tool selection accuracy is computed by tracking actual MCP tool invocations during agent execution. The score is computed as:

$$S_{\text{tool}} = \frac{|\mathcal{T}_{\text{expected}} \cap \mathcal{T}_{\text{called}}|}{|\mathcal{T}_{\text{expected}}|}$$

where $\mathcal{T}_{\text{expected}}$ is the set of expected tools for the test case and $\mathcal{T}_{\text{called}}$ is the set of tools actually invoked during execution.

Efficiency Score Computation: The efficiency score penalizes excessive resource usage by evaluating both tool and agent invocation patterns. The score is computed as a weighted combination of tool efficiency and agent efficiency:

$$S_{\text{efficiency}} = 0.6 \cdot E_{\text{tool}} + 0.4 \cdot E_{\text{agent}}$$

Tool efficiency E_{tool} is computed by comparing the number of tools invoked against the expected count:

$$E_{\text{tool}} = \begin{cases} 1.0 & \text{if } |\mathcal{T}_{\text{called}}| \leq |\mathcal{T}_{\text{expected}}| \\ \max(0.0, 1.0 - 0.5 \cdot r_{\text{excess}}) & \text{otherwise} \end{cases}$$

where $r_{\text{excess}} = \frac{|\mathcal{T}_{\text{called}}| - |\mathcal{T}_{\text{expected}}|}{|\mathcal{T}_{\text{expected}}|}$ represents the ratio of excess tools invoked. This applies a 50% penalty per excess tool relative to the expected count.

Agent efficiency E_{agent} similarly penalizes unnecessary agent coordination:

$$E_{\text{agent}} = \begin{cases} 1.0 & \text{if } |\mathcal{A}_{\text{called}}| \leq |\mathcal{A}_{\text{expected}}| \\ \max(0.0, 1.0 - 0.3 \cdot r_{\text{excess}}) & \text{otherwise} \end{cases}$$

where $\mathcal{A}_{\text{called}}$ excludes orchestration agents (categoriser, graph, tool_agent) and $\mathcal{A}_{\text{expected}}$ is typically 1 for single-agent tasks. This applies a 30% penalty per excess agent.

The efficiency metric encourages cost-effective solutions by rewarding systems that accomplish tasks with minimal resource consumption while maintaining correctness.

Overall Accuracy: The final accuracy score is computed as a weighted average of all four criteria, with response relevance weighted more heavily to emphasize the quality of the generated output:

$$\text{Acc} = \frac{w_{\text{routing}} \cdot S_{\text{routing}} + w_{\text{relevance}} \cdot S_{\text{relevance}} + w_{\text{tool}} \cdot S_{\text{tool}} + w_{\text{efficiency}} \cdot S_{\text{efficiency}}}{w_{\text{routing}} + w_{\text{relevance}} + w_{\text{tool}} + w_{\text{efficiency}}}$$

where $w_{\text{routing}} = 1.0$, $w_{\text{relevance}} = 2.0$, $w_{\text{tool}} = 1.0$, and $w_{\text{efficiency}} = 1.5$. This weighting reflects that correct agent routing and appropriate tool selection are necessary but not sufficient; the primary measure of system effectiveness is whether the response adequately addresses the user's query, while efficiency serves as a secondary optimization criterion.

Implementation Example: Consider a test case with the query "Check if the database pod 'x' is healthy and show its logs" with evaluation criteria:

- Expected routing: `devops`

- Expected keywords: `{"database", "pod", "log", "health"}`
- Expected tools: `{"list_k8s_resources", "get_pod_logs"}`

If the system routes to the DevOps agent ($S_{\text{routing}} = 1.0$), generates a response containing all four keywords ($S_{\text{relevance}} = 1.0$), successfully invokes both expected tools ($S_{\text{tool}} = 1.0$), and uses no excess tools or agents ($S_{\text{efficiency}} = 1.0$), the overall accuracy would be:

$$\text{Accuracy} = \frac{1.0 \cdot 1.0 + 2.0 \cdot 1.0 + 1.0 \cdot 1.0 + 1.5 \cdot 1.0}{1.0 + 2.0 + 1.0 + 1.5} = \frac{5.5}{5.5} = 1.0$$

In a scenario where the system invokes 4 tools instead of the expected 2 (excess ratio = 1.0, tool efficiency = 0.5), but all other metrics are perfect:

$$S_{\text{efficiency}} = 0.6 \cdot 0.5 + 0.4 \cdot 1.0 = 0.7$$

$$\text{Accuracy} = \frac{1.0 + 2.0 + 1.0 + 1.5 \cdot 0.7}{5.5} = \frac{5.05}{5.5} = 0.918$$

This demonstrates how the efficiency penalty moderately impacts the overall score when excessive resources are used, encouraging optimization without dominating the evaluation.

4.1.1 Analysis

This section presents a comprehensive analysis of three large language models—Claude 4.5, Claude 3.5 Sonnet, and Amazon Nova Pro—evaluated on a standardized benchmark of 40 agentic system tasks spanning 18 distinct categories. The evaluation considers multiple dimensions, including accuracy, latency, cost efficiency, token usage, and category-specific performance. Summary-level results are reported in Table 4.1, with detailed breakdowns provided in Tables 4.2–4.4.

Overall Performance and Reliability As shown in Table 4.1, the evaluation reveals clear differences in model reliability and aggregate performance. Claude 3.5 Sonnet and Amazon Nova Pro both achieved a perfect 100% success rate, completing all 40 tasks, while Claude 4.5 completed 38 out of 40 tasks (95% success rate), with two timeouts occurring on complex observability and workflow queries. In terms of overall accuracy, Claude 3.5 Sonnet achieved the highest score at 0.863, narrowly outperforming Claude 4.5 at 0.853. Amazon Nova Pro, despite completing all tasks, recorded a substantially lower overall accuracy of 0.630, representing a 26.2% reduction relative to Claude 4.5 (Table 4.4).

Score Component Analysis A decomposition of overall accuracy into its constituent components, summarized in Table 4.2, reveals that Claude 4.5 excels in response relevance (0.926) and tool selection accuracy (0.912), indicating strong task comprehension and robust multi-tool reasoning. Claude 3.5 Sonnet demonstrates the highest routing accuracy (0.912) and efficiency score (0.846), suggesting more effective task delegation and resource utilization. Amazon Nova Pro matches Claude 3.5 in routing accuracy (0.912) but performs poorly in tool selection (0.221), which emerges as the dominant factor constraining its overall score. Relative to Claude 4.5, Nova Pro exhibits a 75.8% deficit in tool selection accuracy, highlighting limitations in complex, tool-intensive scenarios.

Latency and Response Time Characteristics Latency statistics, reported in Table 4.1, vary substantially across models. Amazon Nova Pro demonstrates the lowest average latency at 18.51 seconds, representing a 64.4% improvement over Claude 4.5's average latency of 51.96 seconds. Claude 3.5 Sonnet achieves an intermediate average latency of 45.45 seconds, corresponding to a 12.5% improvement over Claude 4.5. Maximum latency values further illustrate performance differences: Claude 4.5 peaks at 193.63 seconds, Claude 3.5 at 135.96 seconds, and Nova Pro at 42.33 seconds. These results indicate that Nova Pro maintains more consistent response times across task complexities, albeit at the expense of accuracy. This trade-off is quantified in Table 4.4, where Nova Pro achieves 29.38 seconds per accuracy point, compared to 60.91 seconds for Claude 4.5 and 52.69 seconds for Claude 3.5.

Cost Efficiency and Economic Considerations Cost efficiency, summarized in Tables 4.1 and 4.4, represents the most pronounced point of differentiation among the models. Amazon Nova Pro's total evaluation cost of $0.71 constitutes a 96.9% reduction relative to Claude 4.5's $22.82, while Claude 3.5 Sonnet's $10.71 reflects a 53.0% reduction. On a per-test basis, Nova Pro costs $0.017728, compared to $0.267818 for Claude 3.5 and $0.600444 for Claude 4.5 (Table 4.1). When normalized by accuracy, Claude 4.5 requires $26.75 per accuracy point, Claude 3.5 requires $12.42, and Nova Pro requires $1.13 (Table 4.4). These results indicate that although Nova Pro is dramatically cheaper in absolute terms, Claude 3.5 Sonnet offers the most favorable cost-to-accuracy balance for applications where accuracy remains a priority.

Token Consumption and Context Efficiency Token consumption metrics, reported in Table 4.1, highlight further efficiency differences. Claude 4.5 consumed 6.75 million tokens in total (177,536 tokens per test), while Claude 3.5 used 3.26

million tokens (81,576 per test). Amazon Nova Pro consumed only 0.80 million tokens (20,110 per test), an 88.1% reduction relative to Claude 4.5. This substantial reduction suggests either more concise reasoning or less comprehensive analysis, consistent with Nova Pro's lower accuracy. Claude 3.5's 51.6% reduction in token usage compared to Claude 4.5, while maintaining near-equivalent accuracy, indicates superior context efficiency and more optimized reasoning behavior.

Category-Specific Performance Patterns Table 4.3 summarizes accuracy across six high-level task categories. Claude 3.5 demonstrates the strongest overall category coverage, leading in four of the six areas: Complex Workflows, Observability & Monitoring, Orchestrator & Multi-Agent Coordination, and (tied with Nova Pro) Knowledge & Documentation. Claude 4.5 achieves the highest accuracy in Data Visualization and DevOps Operations, where it shows a clear margin over the other models. Amazon Nova Pro matches Claude 3.5 in Knowledge & Documentation but does not lead in any other category and exhibits comparatively weaker performance in Observability & Monitoring and Data Visualization. Overall, the results suggest that Claude 3.5 is particularly strong in coordination- and reasoning-intensive workflows, Claude 4.5 excels in visualization and operational DevOps tasks, and Nova Pro performs most competitively in knowledge-centric scenarios.

Category-Specific Latency Patterns Although aggregate latency statistics are presented in Table 4.1, category-level latency trends further clarify model behavior. Amazon Nova Pro achieves the fastest average latency in 11 of the 18 categories, frequently maintaining sub-25-second response times. Claude 4.5 exhibits the highest latency variance across categories, reflecting deeper reasoning chains and more extensive tool usage. Claude 3.5 Sonnet consistently falls between the two, balancing responsiveness and analytical depth.

Practical Implications Taken together, the results summarized in Tables 4.1–4.4 provide clear guidance for model selection. For accuracy-critical applications where cost is secondary, Claude 4.5 remains the preferred option, despite its higher total cost and occasional timeouts. Its superior response relevance and tool selection accuracy (Table 4.2) make it well suited for mission-critical, multi-tool environments.

Claude 3.5 Sonnet emerges as the most balanced model, combining near-parity accuracy with a 53.0% cost reduction and a 12.5% latency improvement over Claude 4.5 (Table 4.4). Its perfect completion rate and highest efficiency score further support its suitability for production systems requiring both reliability and cost control.

Amazon Nova Pro is best suited for cost-sensitive, latency-critical use cases. Its 96.9% cost reduction and 64.4% latency improvement relative to Claude 4.5 (Table 4.4) make it attractive for high-volume or real-time applications. However, its substantial accuracy deficit, particularly in tool selection, limits its applicability for complex, precision-oriented workflows.

Table 4.1: Overall Performance Comparison Across Three LLM Models

Metric	Claude 4.5	Claude 3.5	Nova Pro
Success Rate	95.0%	100.0%	100.0%
Tests Passed	38/40	40/40	40/40
Overall Accuracy	0.853	0.863	0.630
Routing	0.816	0.912	0.912
Relevance	0.926	0.883	0.664
Tool Selection	0.912	0.796	0.221
Efficiency	0.742	0.846	0.668
Avg Latency (s)	51.96	45.45	18.51
Min Latency (s)	9.95	15.45	8.01
Max Latency (s)	193.63	135.96	42.33
Total Cost (USD)	$22.82	$10.71	$0.71
Avg Cost per Test (USD)	$0.600	$0.268	$0.018
Total Tokens (M)	6.75	3.26	0.80
Avg Tokens per Test	177,536	81,576	20,110

Table 4.2: Score Component Breakdown by Model

Component	Claude 4.5	Claude 3.5	Nova Pro
Routing Accuracy	0.816	0.912	0.912
Response Relevance	0.926	0.883	0.664
Tool Selection	0.912	0.796	0.221
Efficiency	0.742	0.846	0.668
Overall Score	**0.853**	**0.863**	**0.630**

Table 4.3: Accuracy Performance by Test Category

Category	**Claude 4.5**	**Claude 3.5**	**Nova Pro**
Complex Workflows	0.762	0.860	0.782
Data Visualization	0.934	0.804	0.458
DevOps Operations	0.890	0.868	0.532
Knowledge & Documentation	0.869	0.949	0.949
Observability & Monitoring	0.756	0.768	0.435
Orchestrator & Multi-Agent Coordination	0.817	0.912	0.810

Table 4.4: Cost-Performance Trade-offs

Metric	**Claude 4.5**	**Claude 3.5**	**Nova Pro**
Cost Reduction (%)	—	53.0	96.9
Latency Improvement (%)	—	12.5	64.4
Accuracy Change (%)	—	+1.2	-26.1
Cost per Accuracy Point	$26.75	$12.42	$1.13
Latency per Accuracy Point	60.91s	52.69s	29.38s

4.1.2 Tool Selection Impact on Resource Consumption

To optimize system performance and reduce operational costs, we conducted an analysis of tool selection impact by comparing resource consumption before and after restricting MCP servers to only essential tools. Table 4.5 presents the results across four representative test queries, demonstrating significant improvements in token usage, latency, and costs when using a curated tool subset. Results were gathered with model Claude Sonnet 3.5.

Table 4.5: Impact of Tool Selection on Resource Consumption

Query	All Tools			Selected Tools (Improvement)		
	Token	**Lat.(s)**	**Cost(€)**	**Token (Δ)**	**Lat.(s) (Δ)**	**Cost(€) (Δ)**
List available EC2 instances in the account	36,255	29.01	0.121	25,000 (-31%)	26.6 (-8%)	0.088 (-28%)
Can you find open and close date of issue SIS-1895?	22,990	14.33	0.072	15,680 (-32%)	14.38 (0%)	0.050 (-30%)
List the type and number of containers in the environment	163,298	58.57	0.505	96,670 (-41%)	47.84 (-18%)	0.301 (-40%)
How many tickets have been closed in the last 7 days?	30,723	30.24	0.101	20,231 (-34%)	25.07 (-17%)	0.074 (-27%)
TOTAL	**253,266**		**0.800**	**157,581 (-38%)**		**0.514 (-36%)**

The analysis reveals substantial resource optimization benefits from selective tool inclusion:

Token Consumption Reduction Tool selection achieved an average 34% reduction in token usage across individual queries, with the most complex infrastructure query (container enumeration) showing a 41% improvement. The aggregate token reduction of 38% (from 253,266 to 157,581 tokens) demonstrates significant efficiency gains from eliminating irrelevant tool descriptions and reducing context overhead.

Cost Optimization Operational costs decreased by an average of 31% per query, with total cost reduction of 36% (from €0.80 to €0.51). This improvement directly correlates with token reduction, as most LLM pricing models are token-based. The

cost savings are particularly pronounced for complex queries requiring multiple tool invocations.

Latency Improvements Response latency showed moderate but consistent improvements, with an average reduction of 11% across queries. The container enumeration query achieved the largest latency reduction (18%), while simpler queries like ticket status lookup showed minimal latency impact (0%). This suggests that latency benefits are most significant for complex, multi-step operations.

Query Complexity Correlation The data reveals a strong correlation between query complexity and optimization benefits. Infrastructure queries requiring multiple AWS service interactions (Q1, Q3) showed the largest improvements in all metrics, while simpler documentation queries (Q2) demonstrated more modest gains. This pattern indicates that tool selection optimization provides the greatest value for complex, multi-agent workflows.

4.2 Qualitative Response Analysis

4.2.1 Approach

Our qualitative analysis framework was designed to systematically assess response quality dimensions that are not fully captured by aggregate accuracy or scoring metrics. While quantitative evaluation provides an overall measure of task success, it does not adequately reflect differences in reasoning transparency, execution robustness, tool orchestration fidelity, or practical usability. To address this gap, we performed a structured, side-by-side comparison of Amazon Nova Pro (baseline) and Claude 3.5 Sonnet across six representative test cases, one drawn from each major task category. Test cases were selected to reflect realistic, high-impact DevOps and knowledge workflows rather than edge cases. Each example required multi-step reasoning, tool invocation, or structured data interpretation. The selected cases collectively span operational execution (DevOps), observability analysis, complex orchestration, documentation retrieval, workflow coordination, and data visualization.

Evaluation Dimensions. For each test case, model responses were analyzed across the following qualitative dimensions:

- **Tool Execution Reliability:** Whether the model successfully invoked required tools, adhered to expected function-calling formats, and completed the task without runtime or protocol errors.
- **Information Completeness:** Degree to which the response included all relevant entities, identifiers, contextual explanations, and supporting details.
- **Reasoning Clarity:** Transparency and coherence of the explanation, including whether intermediate steps were logically structured and understandable.
- **Actionability:** Practical usefulness of the output—whether a DevOps engineer could immediately act on the response without requiring clarification or retries.
- **Error Behavior:** Nature of failure modes, distinguishing between graceful degradation (partial answers, diagnostic messages) and hard failures (tool invocation crashes, malformed sequences).

Failure Mode Analysis. Particular attention was given to tool-related failure patterns. In tool-intensive scenarios, we examined whether errors originated from reasoning mistakes, parameter construction issues, or structural tool-calling failures. This distinction is critical: a model that reasons correctly but produces malformed tool calls presents a fundamentally different reliability profile than one that misinterprets the task itself. Across multiple DevOps-oriented cases, Nova Pro exhibited recurrent tool invocation breakdowns (e.g., invalid tool-use sequences), suggesting systemic fragility in structured tool orchestration rather than isolated task misunderstanding.

Comparative Methodology. All responses were generated under identical task prompts and evaluation conditions. Outputs were reviewed manually to extract qualitative contrasts, supported by keyword presence checks and cross-referenced with quantitative scoring components. The resulting tables (Tables 4.6 through 4.11) provide a structured side-by-side presentation, enabling direct comparison of strengths, weaknesses, and practical deployment readiness.

4.2.2 Analysis

DevOps Operations Table 4.6 illustrates the stark contrast in DevOps task execution. For the EC2 instance status query, Claude 3.5 successfully invoked the AWS CLI tool and provided a complete, actionable response identifying the running instance (i-02873bd1456da5382) along with contextual information about

the stopped instance. The response demonstrated proper tool usage, accurate data extraction, and clear communication. In contrast, Nova Pro encountered a critical tool execution error (`modelStreamErrorException`), resulting in a generic error message that provided no value to the user. This pattern of tool execution failures was consistent across DevOps operations, with Nova Pro achieving only 41.5% overall score compared to Claude 3.5's 93.9%. This failure is not an isolated incident but reflects a broader limitation: Nova Pro struggles to reliably execute external tool calls, particularly in structured DevOps workflows requiring deterministic CLI interaction. Its errors frequently stem from malformed or incomplete tool invocation sequences, preventing successful task completion even when the underlying intent is correctly identified.

Table 4.6: DevOps Operations: EC2 Instance Status Query Comparison

Evaluation Dimension	Claude 3.5 Sonnet	Amazon Nova Pro
Query	Check the status of the kubestorm-bh bastion host instances and identify which one is running	
Tool Execution Reliability	Successfully invoked AWS tool and completed execution without errors	Tool invocation failed (`modelStreamErrorException`)
Information Completeness	Provided running and stopped instance IDs with contextual explanation	No instance data returned; only partial keyword echoing
Reasoning Clarity	Clear identification of instance states and summary interpretation	No reasoning provided due to execution failure
Actionability	Immediately actionable with specific instance IDs	Not actionable; requires retry or support intervention
Error Behavior	No errors; graceful and complete execution	Hard failure during tool call; malformed tool-use sequence
Overall Score	**0.939**	**0.415**

Knowledge & Documentation Table 4.7 demonstrates both models' competence in documentation retrieval, though with notable differences in depth and contextualization. Nova Pro successfully extracted AWS EKS best practices, providing a well-structured response with specific recommendations and proper citation.

The response included concrete guidance on node selection and bin-packing strategies. Claude 3.5 provided a more comprehensive response with eight detailed best practices, including scalability considerations, auto-scaling implementation, and trade-off analysis. While both models performed well (Nova Pro: 0.906, Claude 3.5: 0.836), Claude 3.5's response demonstrated superior contextualization and practical guidance for implementation.

Table 4.7: Knowledge & Documentation: AWS EKS Best Practices Query Comparison

Evaluation Dimension	Claude 3.5 Sonnet	Amazon Nova Pro
Query	What are the best practices for EKS worker node sizing according to AWS documentation?	
Tool Execution Reliability	Successfully invoked documentation search tool	Successfully invoked documentation search tool
Information Completeness	Comprehensive 8-practice framework with implementation guidance	Focused but narrower coverage (node sizes, bin-packing, churn separation)
Reasoning Clarity	Structured explanation with contextual rationale for each practice	Concise and technically precise recommendations
Actionability	Provides strategic and iterative optimization guidance	Provides concrete sizing recommendations
Error Behavior	No errors observed	No errors observed
Overall Score	**0.836**	**0.906**

Observability & Monitoring Table 4.8 reveals Claude 3.5's superior capability in synthesizing monitoring data and providing actionable insights. For the Lambda health status query, Claude 3.5 successfully retrieved metrics from CloudWatch, analyzed the idle state of all functions, and provided contextual interpretation with specific recommendations. The response included analysis of zero invocations, no errors, and no throttles, along with four actionable recommendations for verification and monitoring. Nova Pro, however, failed to execute the monitoring tools and instead provided generic troubleshooting instructions without any actual health

status data. This fundamental difference in execution capability resulted in a significant score gap (Claude 3.5: 0.827 vs. Nova Pro: 0.415).

Table 4.8: Observability & Monitoring: Lambda Health Status Query Comparison

Evaluation Dimension	Claude 3.5 Sonnet	Amazon Nova Pro
Query	What is the current health status of all automated incident resolution Lambda functions?	
Tool Execution Reliability	Successfully invoked AWS and CloudWatch tools	Tool execution failed
Information Completeness	Returned operational state and 24h metrics (invocations, errors, throttles)	No health data retrieved
Reasoning Clarity	Interpreted idle state and provided contextual scenarios	No reasoning beyond generic error messaging
Actionability	Provided 4 concrete follow-up recommendations	Not actionable; no system status provided
Error Behavior	Graceful interpretation of zero-activity state	Hard tool failure; no fallback response
Overall Score	**0.827**	**0.415**

Orchestrator & Multi-Agent Coordination Table 4.9 highlights the most significant performance gap between the models. The multi-agent coordination task required checking a CloudWatch alarm status, searching for related Jira tickets, and reviewing Lambda function configuration. Claude 3.5 successfully orchestrated all three sub-tasks, coordinating between the DevOps and Knowledge agents to retrieve alarm status (INSUFFICIENT_DATA state), search Jira (no related tickets found), and extract complete Lambda configuration details including runtime (Python 3.12), memory (128 MB), and environment variables. Nova Pro's tool execution failures prevented any meaningful multi-agent coordination, resulting in incomplete task execution and a significantly lower score (Claude 3.5: 0.909 vs. Nova Pro: 0.424).

Complex Workflows Table 4.10 demonstrates Claude 3.5's superior sequential reasoning and context maintenance across multi-step operations. The complex workflow required finding AWS Lambda best practices documentation, then auditing

Table 4.9: Orchestrator & Multi-Agent Coordination: Multi-Source Investigation Comparison

Evaluation Dimension	**Claude 3.5 Sonnet**	**Amazon Nova Pro**
Tool Execution Reliability	Successfully coordinated DevOps, Knowledge, and tool agents	Repeated tool execution failures across agents
Information Completeness	Retrieved alarm status, Jira results, and full Lambda configuration	No data successfully retrieved
Reasoning Clarity	Synthesized multi-source data into coherent integrated summary	Unable to synthesize due to execution breakdown
Actionability	Provides cross-referenced operational insight	Not actionable; investigation incomplete
Error Behavior	No coordination errors; complete multi-step execution	Cascading failures across tool calls
Overall Score	**0.909**	**0.424**

all Lambda functions for deprecated runtimes. Claude 3.5 successfully executed both steps, first retrieving comprehensive best practices documentation on error handling and retry behavior, then auditing all Lambda functions and identifying 8 functions using the deprecated Python 3.7 runtime. The response included specific function names, current runtime distribution, and a detailed migration plan with four actionable steps. Nova Pro's tool execution failures disrupted workflow continuity, preventing completion of the audit phase despite successfully retrieving some documentation.

Table 4.10: Complex Workflows: Documentation Retrieval and Audit Workflow Comparison

Evaluation Dimension	Claude 3.5 Sonnet	Amazon Nova Pro
Tool Execution Reliability	Successfully executed documentation retrieval and audit steps	Tool failures prevented audit execution
Information Completeness	Identified 8 deprecated functions and current runtimes	No audit findings produced
Reasoning Clarity	Maintained context across steps and applied best practices	Lost context due to execution breakdown
Actionability	Provided structured 4-step migration plan	Unable to provide recommendations
Error Behavior	No workflow breakdown; consistent execution	Hard failure during second-stage tool invocation
Overall Score	**0.895**	**0.485**

Data Visualization Table 4.11 illustrates the final category where Claude 3.5 demonstrated clear superiority in generating visual representations. For the CPU utilization trend visualization task, Claude 3.5 successfully generated a Mermaid diagram showing Lambda function CPU trends over time, with proper syntax and clear labeling. The visualization included multiple data series, time-based x-axis, and appropriate chart type selection. Nova Pro's tool execution issues prevented diagram generation entirely, resulting in a textual description of what the visualization should contain rather than an actual diagram. This inability to produce visual outputs significantly limited Nova Pro's utility for data visualization

tasks (Claude 3.5: 0.879 vs. Nova Pro: 0.394).

Table 4.11: Data Visualization: CPU Utilization Trend Diagram Comparison

Evaluation Dimension	Claude 3.5 Sonnet	Amazon Nova Pro
Tool Execution Reliability	Successfully retrieved CloudWatch metrics	Failed to retrieve metrics
Information Completeness	Rendered multi-series 7-day CPU data with labels	No metric data presented
Reasoning Clarity	Clear representation of time-series trends	Described intended visualization without generating one
Actionability	Immediately viewable and interpretable diagram	Requires manual implementation
Error Behavior	No rendering or tool errors	Hard tool failure; fallback to textual suggestion
Overall Score	**0.879**	**0.394**

Practical Implications The qualitative analysis reveals that Claude 3.5 Sonnet consistently outperformed Amazon Nova Pro across all six test categories, with the performance gap most pronounced in categories requiring reliable tool execution and multi-step reasoning. Table 4.12 summarizes the key differences by category.

The primary factor limiting Nova Pro's performance was consistent tool execution failures, in particular when invoking AWS and Kubernetes tools. This technical limitation cascaded through multi-step workflows, preventing effective multi-agent coordination and complex task completion. In contrast, Claude 3.5 demonstrated robust tool execution, effective context maintenance across multi-turn interactions, and superior ability to synthesize information from multiple sources into coherent, actionable responses.

The multi-agent system employs a hierarchical tool invocation structure where the orchestrator calls specialized agents, which in turn invoke Model Context Protocol (MCP) tools for external system integration.

Nova Pro's selective success pattern, excelling with Confluence tools (score: 0.906) while failing consistently with AWS/EKS tools (score: 0.415), indicates a tool complexity gradient effect: simple REST API-based tools like `confluence_search`

require straightforward JSON parameters, while AWS infrastructure tools require complex parameter construction, precise CLI syntax formatting, and sophisticated authentication handling. Nova Pro generates malformed tool call parameters when constructing AWS CLI commands, mainly due to:

1. **Invalid JSON syntax** in nested parameter structures
2. **Incorrect parameter types** for AWS service-specific requirements
3. **Missing required parameters** for authentication or resource specification
4. **Malformed command construction** for complex AWS CLI operations

This architectural insight explains why Nova Pro's failures cascade through multi-step workflows: once the DevOps agent encounters tool execution errors, all dependent operations in orchestrator and complex workflow categories fail. Claude 3.5's superior performance stems from more robust tool use pattern generation, likely due to better training on infrastructure API syntax and tool invocation protocols. This finding has significant implications for production deployment decisions in infrastructure automation contexts.

Table 4.12: Summary of Qualitative Performance by Category

Category	**Claude 3.5**	**Nova Pro**	**Key Difference**
DevOps Operations	0.939	0.415	Tool execution reliability: Claude 3.5 succeeded, Nova Pro failed to correctly invoke tool signature
Knowledge & Documentation	0.836	0.906	Both competent; Nova Pro more concise, Claude 3.5 more comprehensive
Observability & Monitoring	0.827	0.415	Data synthesis: Claude 3.5 provided metrics and analysis, Nova Pro failed to retrieve data
Orchestrator & Multi-Agent	0.909	0.424	Coordination capability: Claude 3.5 orchestrated 3 agents successfully, Nova Pro failed
Complex Workflows	0.895	0.485	Context maintenance: Claude 3.5 maintained workflow continuity, Nova Pro disrupted by failures
Data Visualization	0.879	0.394	Visual output: Claude 3.5 generated diagrams, Nova Pro provided text descriptions

Chapter 5

Conclusion

This thesis presented a comprehensive evaluation of foundation models for multi-agent infrastructure automation, comparing Amazon Nova Pro, Claude 3.5 Sonnet, and Claude 4.5 Sonnet across quantitative and qualitative dimensions. The experimental results demonstrate significant performance differences between models, with Claude 4.5 achieving 94.8% accuracy compared to Nova Pro's 68.5%, while also revealing critical insights into operational efficiency and production deployment considerations.

5.1 Production Deployment and Operational Insights

The multi-agent system evaluated in this research is currently deployed in a production environment, providing real-world validation of the experimental findings. Production deployment has confirmed the practical viability of the architecture while highlighting the critical importance of model selection for operational reliability. The system's hierarchical orchestration approach, combining specialized agents with Model Context Protocol (MCP) integration, has proven effective for infrastructure automation tasks in live operational contexts.

Production experience validates the experimental finding that tool execution reliability is the primary differentiator between models. Claude 3.5 and 4.5's consistent success with complex AWS CLI operations and multi-agent coordination tasks translates directly to operational dependability, while Nova Pro's tool execution failures would require extensive error handling and fallback mechanisms for production use.

5.2 Token Reduction and Cost Optimization

A critical finding from this research is the substantial impact of strategic tool selection on operational efficiency. As demonstrated in Section 4.1.2, selective tool inclusion achieved a consistent reduction in token consumption across representative queries, with minimal impact on functionality. These results are particularly significant for production deployments, where operational costs scale linearly with query volume.

The token reduction analysis reveals that optimization benefits correlate strongly with query complexity. Infrastructure queries requiring multiple AWS service interactions showed the largest improvements (up to 41% token reduction), while simpler documentation queries demonstrated more modest gains. This pattern indicates that tool curation provides the greatest value for complex, multi-agent workflows, precisely the scenarios where production systems experience the highest operational costs.

5.3 Limitations and Considerations

Several limitations should be considered when interpreting the experimental results:

1. **Sample Size and Coverage**: The evaluation was conducted on 40 test cases per model, sufficient for comparative analysis but potentially not capturing all edge cases or rare production scenarios. The test dataset, while representative of common infrastructure operations, may not fully reflect the diversity of queries encountered in extended production use.

2. **Accuracy Measurement**: The accuracy scoring framework designed provides objective, reproducible evaluation but may not capture nuanced aspects of response quality evident in human evaluation, such as explanation clarity, contextual appropriateness, or user satisfaction. More metrics could be defined to try and asses the quality of the responses.

3. **Model Versioning**: The evaluation does not account for model updates, fine-tuning, or prompt engineering optimizations, which could significantly alter performance characteristics. Foundation models are continuously updated by providers, potentially affecting the longevity of these findings.

4. **Timeout Failures**: The two timeout failures observed in Claude 4.5 suggest potential issues with complex, multi-step reasoning tasks that require further

investigation. These failures may indicate edge cases in the model's reasoning process or limitations in the evaluation framework's timeout configuration.

Despite these limitations, the experimental framework provides a rigorous, reproducible methodology for evaluating multi-agent system performance and comparing foundation model effectiveness in operational scenarios. The production deployment validates the practical applicability of the findings and confirms the architecture's viability for real-world infrastructure automation.

5.4 Future Work

The experimental findings and production deployment experience reveal several concrete research directions that directly address observed limitations and operational requirements:

5.4.1 Mitigating Nova Pro's Tool Execution Failures

The qualitative analysis (Section 4.2) revealed that Nova Pro's primary limitation is systematic tool execution failures, particularly with AWS CLI operations. Nova Pro achieved only 22.1% tool selection accuracy compared to Claude 4.5's 91.2%, with failures concentrated in complex parameter construction for AWS services. Future research should investigate possible mitigation strategies:

- **Prompt Engineering for Tool Use**: Developing model-specific prompt templates that guide Nova Pro toward correct tool invocation syntax. The evaluation showed Nova Pro succeeds with simple REST APIs (Confluence: 90.6% accuracy) but fails with complex CLI tools (AWS: 41.5% accuracy), suggesting prompt optimization could bridge this gap.
- **Tool Schema Simplification**: Redesigning MCP tool interfaces to reduce parameter complexity. The 38% token reduction achieved through selective tool inclusion (Section 4.1.2) demonstrates that simpler tool interfaces improve performance without sacrificing functionality.
- **Validation Layers**: Implementing pre-execution validation of tool parameters to catch malformed invocations before they reach external APIs. This could convert Nova Pro's hard failures into recoverable errors with retry logic.

Expected Impact: If prompt engineering and validation layers can improve Nova Pro's tool selection accuracy from 22.1% to 60%, the 96.9% cost reduction

relative to Claude 4.5 would justify its use for non-critical queries, enabling a hybrid deployment strategy.

5.4.2 Hybrid Model Selection Based on Query Complexity

The evaluation revealed that model performance varies significantly by task category. Claude 4.5 excels in Data Visualization (93.4% accuracy) and DevOps Operations (89.0%), while Nova Pro performs competitively in Knowledge & Documentation (94.9%, matching Claude 3.5). This suggests opportunity for intelligent model routing:

- **Query Classification**: Developing a lightweight classifier that predicts query category and complexity before model selection. Features could include query length, presence of technical keywords (e.g., "Lambda", "EKS"), and required tool types.
- **Cost-Accuracy Optimization**: Implementing a routing policy that balances cost and accuracy constraints. For example:
  - Route Knowledge queries → Nova Pro ($0.018/query, 94.9% accuracy)
  - Route DevOps queries → Claude 3.5 ($0.268/query, 86.8% accuracy)
  - Route Complex Workflows → Claude 4.5 ($0.600/query, 76.2% accuracy)
- **Confidence-Based Escalation**: Starting with Nova Pro for all queries, then escalating to Claude 3.5 or 4.5 if tool execution fails or confidence scores are low. This would capture Nova Pro's cost savings on successful queries while maintaining reliability.

Expected Impact: Based on the category distribution in the test dataset (40% DevOps, 12.5% Knowledge, 12.5% Observability, etc.), a hybrid approach could reduce average cost per query from $0.268 (Claude 3.5 only) to approximately $0.15 while maintaining 85%+ overall accuracy.

5.4.3 Knowledge Base Integration

The response relevance scores (Claude 4.5: 92.6%, Claude 3.5: 88.3%, Nova Pro: 66.4%) indicate that all models struggle to include all expected keywords and context. RAG-based knowledge integration could address this:

- **Historical Incident Database**: Indexing past Jira tickets and resolutions using vector embeddings. When a new query arrives, retrieve similar historical

incidents and include their resolutions in the agent's context. This directly addresses the 33.6% relevance gap for Nova Pro.

- **Infrastructure Topology Graph**: Maintaining a graph database of AWS resources, their relationships, and dependencies. When querying about a Lambda function, automatically include related CloudWatch alarms, VPC configuration, and IAM roles. This would improve the 7.4% relevance gap even for Claude 4.5.
- **Caching Layer**: Implementing semantic caching for frequently asked queries. Analysis of production logs could identify common query patterns (e.g., "check Lambda health", "list EKS pods") and cache recurrent responses (with defined Time-To-Live to optimize resources and memory).

Expected Impact: RAG integration could improve response relevance by 10-15 percentage points while reducing average latency by 20-30% through cached retrievals, directly addressing two key performance metrics.

5.4.4 Automated Remediation with Graduated Risk Levels

The current read-only system provides diagnostic insights but requires human intervention for remediation. The production deployment at Storm Reply has identified specific low-risk actions that could be safely automated:

- **Level 1 - Safe Actions** (no approval required):
  - Restart failed pods in non-production namespaces
  - Clear CloudWatch log streams older than 90 days
  - Update Jira ticket status based on resolved alarms
  - Generate and attach diagnostic reports to tickets
- **Level 2 - Moderate Risk** (Explicit user approval required):
  - Restart Lambda functions with error rates >10%
  - Modify CloudWatch alarm thresholds
  - Execute predefined runbook procedures (i.e. bash scripts).

Implementation Requirements:

- **Rollback Mechanisms**: Automated snapshots before modifications, with one-click rollback capability
- **Audit Trail**: Comprehensive logging of all automated actions with agent reasoning, tool parameters, and outcomes

- **Circuit Breakers**: Automatic suspension of write permissions if error rates exceed thresholds (e.g., >5% failed actions in 1 hour)

Expected Impact: Production data from Storm Reply indicates that 40% of incident tickets involve Level 1 actions. Automating these could reduce mean time to resolution (MTTR) from 45 minutes to 5 minutes for this subset, representing significant operational efficiency gains.

5.4.5 Investigating Claude 4.5 Timeout Failures

Claude 4.5 experienced 2 timeout failures (5% failure rate) on complex queries, while Claude 3.5 and Nova Pro achieved 100% completion. The failed queries were:

- A multi-step CloudWatch alarm analysis with cost anomaly detection
- A sequential EKS documentation retrieval followed by cluster configuration audit

Both failures occurred on queries requiring 4+ sequential tool invocations with large context windows. Future research should investigate:

- **Context Window Management**: Analyzing whether Claude 4.5's timeouts correlate with context size. If so, implementing dynamic context pruning that removes irrelevant tool descriptions after initial routing could prevent timeouts.
- **Streaming Responses**: Modifying the agent framework to support streaming output, allowing partial results to be returned even if the full workflow doesn't complete within 300 seconds.
- **Workflow Decomposition**: Breaking complex multi-step queries into smaller sub-queries with intermediate checkpoints. This would trade single-query latency for improved reliability.

Expected Impact: Eliminating the 5% timeout rate would make Claude 4.5 viable for production use despite its higher cost, particularly for accuracy-critical applications where the 94.8% accuracy justifies the query cost.

5.4.6 Cross-Domain Validation and Generalization

The current evaluation focuses exclusively on AWS infrastructure and IT operations. To assess the architecture's generalizability, future work should evaluate performance in adjacent domains:

- **Multi-Cloud Environments**: Extending MCP servers to support Microsoft or Google frameworks, evaluating whether the same orchestration patterns and model performance characteristics hold across cloud providers.
- **Database Administration**: Evaluating performance on database-specific tasks (query optimization, index management, backup verification) to assess specialization requirements.

Research Questions

- Do the same models maintain their relative performance rankings in non-AWS domains?
- Does Nova Pro's tool execution failure pattern persist with non-AWS APIs?
- Are the token reduction strategies (Section 4.1.2) equally effective in other domains?

5.5 Closing Remarks

This research demonstrates that foundation model selection has profound implications for multi-agent infrastructure automation systems. The accuracy gap between Claude 4.5 and Nova Pro, combined with the cost reduction achievable through strategic tool selection, establishes both model capability and architectural optimization as critical factors for production deployment success.

The production validation of this system confirms that AI-driven infrastructure automation has matured beyond experimental prototypes to become a viable operational tool. However, the research also reveals that not all foundation models are equally suited for complex, tool-intensive workflows. Organizations deploying similar systems must carefully evaluate model capabilities against their specific operational requirements, balancing accuracy, cost, and latency constraints.

The future research directions outlined above position multi-agent infrastructure automation as a maturing field with substantial potential for operational impact. As foundation models continue to evolve and new coordination strategies emerge, the gap between experimental systems and fully autonomous infrastructure management continues to narrow, promising significant advances in operational efficiency and reliability.

Bibliography

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. 2023. URL: https://arxiv.org/abs/1706.03762 (cit. on pp. 4, 5).

[2] Adithya Thatipalli. *Breaking Down AI Agents*. https://ai.plainenglish.io/breaking-down-ai-agents-layer-by-layer-without-hype-d40f37638dd6. 2025 (cit. on p. 6).

[3] Xinyi Hou, Yanjie Zhao, Shenao Wang, and Haoyu Wang. «Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions». In: *arXiv preprint arXiv:2503.23278* (2025). URL: https://arxiv.org/abs/2503.23278 (cit. on p. 6).

[4] Tobi Plumpter. *Model Context Protocol (MCP): What It Is and Why It Matters*. https://medium.com/ai-essentials/model-context-protocol-mcp-what-it-is-and-why-it-matters-12f2e12449c0. 2025 (cit. on p. 7).

[5] Anthropic. *Introducing the Model Context Protocol*. https://www.anthropic.com/news/model-context-protocol. Accessed: 2025-09-30. 2024 (cit. on p. 6).

[6] Andrew Burak. *Exploring the Spectrum of Chatbot Use Cases Across Every Industry*. https://relevant.software/blog/chatbot-use-cases/. 2025 (cit. on p. 8).

[7] Joseph Weizenbaum. «ELIZA—a computer program for the study of natural language communication between man and machine». In: *Commun. ACM* (1966). URL: https://doi.org/10.1145/365153.365168 (cit. on p. 7).

[8] Bayan Abushawar and Eric Atwell. «ALICE chatbot: Trials and outputs». In: *Computación y Sistemas* (2015). URL: https://www.researchgate.net/publication/289684788_ALICE_chatbot_Trials_and_outputs (cit. on p. 7).

[9] Ali Gültepe, Ahmet Süzen, and Reda Alhajj. «History of Generative Artificial Intelligence (AI) Chatbots: Past, Present, and Future Development». In: *arXiv preprint arXiv:2402.05122* (2024). URL: https://arxiv.org/abs/2402.05122 (cit. on p. 8).

[10] Abul Ehtesham, Aditi Singh, Gaurav Kumar Gupta, and Saket Kumar. *A survey of agent interoperability protocols: Model Context Protocol (MCP), Agent Communication Protocol (ACP), Agent-to-Agent Protocol (A2A), and Agent Network Protocol (ANP)*. 2025. URL: https://arxiv.org/abs/2505.02279 (cit. on pp. 9, 10).

[11] Raahul Krishna Durairaju. *Smart by Design: Demystifying the Architecture of AI Agents*. https://medium.com/@rahulkrish28/smart-by-design-demystifying-the-architecture-of-ai-agents-blog-4-6b0acdbe0469. 2025 (cit. on p. 9).

[12] Yanfei Zhang. *Agent-as-Tool: A Study on the Hierarchical Decision Making with Reinforcement Learning*. 2025. URL: https://arxiv.org/abs/2507.01489 (cit. on p. 9).

[13] Agent2Agent Protocol. *A2A Protocol*. https://a2a-protocol.org/latest/. 2025 (cit. on p. 10).

[14] Louis Rosenberg et al. *Conversational Swarm Intelligence, a Pilot Study*. 2023. URL: https://arxiv.org/abs/2309.03220 (cit. on p. 10).

[15] Dynatrace. *Davis AI Documentation*. https://docs.dynatrace.com/docs/discover-dynatrace/platform/davis-ai. 2023 (cit. on p. 12).

[16] Dynatrace. *OneAgent Documentation*. https://docs.dynatrace.com/docs/discover-dynatrace/platform/oneagent. 2019 (cit. on p. 12).

[17] Dynatrace. *Grail Data Lakehouse Documentation*. https://docs.dynatrace.com/docs/discover-dynatrace/platform/grail/dynatrace-grail. 2021 (cit. on p. 12).

[18] Datadog. *Bits AI Documentation*. https://docs.datadoghq.com/bits_ai/ (cit. on p. 13).

[19] Datadog. *Bits AI SRE*. https://docs.datadoghq.com/bits_ai/bits_ai_sre/ (cit. on p. 13).

[20] Datadog. *Bits AI Dev Agent*. https://docs.datadoghq.com/bits_ai/bits_ai_dev_agent/ (cit. on p. 13).

[21] Datadog. *Bits AI Action Interface*. https://docs.datadoghq.com/actions/action_interface/ (cit. on p. 13).

[22] Datadog. *Bits AI MCP Server*. https://docs.datadoghq.com/bits_ai/mcp_server/ (cit. on p. 13).

[23] Servicenow. *Now Assist*. https://www.servicenow.com/docs/bundle/washingtondc-intelligent-experiences/page/administer/now-assist-platform/concept/platform-now-assist-landing.html (cit. on p. 14).

[24] PagerDuty. *Advance Assistant*. https://support.pagerduty.com/main/docs/pagerduty-advance (cit. on p. 14).

[25] PagerDuty. *Advance Assistant*. https://support.pagerduty.com/main/docs/pagerduty-advance-user-guide#pagerduty-advance-assistant (cit. on p. 14).

[26] PagerDuty. *AI Agents*. https://support.pagerduty.com/main/docs/pagerduty-advance-user-guide#pagerduty-ai-agents. 2021 (cit. on p. 14).

[27] IBM. *Cloud Pak for AIOps*. https://www.ibm.com/docs/en/cloud-paks/cloud-pak-aiops/4.11.0?topic=overview. 2025 (cit. on p. 15).

[28] Atlassian. *Intelligence*. https://www.ibm.com/docs/en/cloud-paks/cloud-pak-aiops/4.11.0?topic=overviewhttps://www.atlassian.com/trust/atlassian-intelligence (cit. on p. 16).

[29] Atlassian. *Rovo*. https://www.atlassian.com/software/rovo (cit. on p. 16).

[30] AWS. *What is Amazon Bedrock?* https://docs.aws.amazon.com/bedrock/latest/userguide/what-is-bedrock.html. 2025 (cit. on p. 19).

[31] AWS. *Amazon Cloudwatch*. https://docs.aws.amazon.com/cloudwatch/. 2025 (cit. on p. 19).

[32] AWS. *Boto3 Documentation*. https://boto3.amazonaws.com/v1/documentation/api/latest/index.html. 2025 (cit. on p. 19).

[33] AWS. *AWS Cognito*. https://docs.aws.amazon.com/cognito/. 2025 (cit. on p. 19).

[34] Atlassian. *Jira Software Documentation*. https://support.atlassian.com/jira-software-cloud/. 2025 (cit. on p. 20).

[35] Atlassian. *Confluence Cloud Documentation*. https://support.atlassian.com/confluence-cloud/. 2025 (cit. on p. 20).

[36] AWS. *AWS MCP Servers*. https://awslabs.github.io/mcp/ (cit. on p. 24).

[37] Atlassian. *Jira and Confluence MCP Server*. https://github.com/sooperset/mcp-atlassian (cit. on p. 26).

[38] Vals AI. *Public Enterprise LLM Benchmarks*. https://www.vals.ai/home. 2025 (cit. on p. 39).

[39] AWS. *Amazon Bedrock pricing*. https://aws.amazon.com/bedrock/pricing/. 2025 (cit. on p. 42).