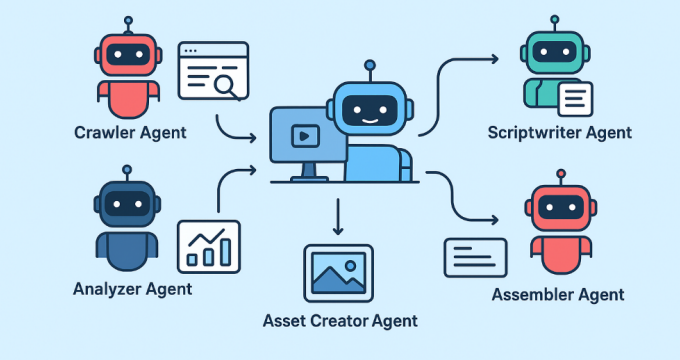


Optimizing latency in multi-agent systems

Bachelor thesis at Chalmers

Author

Sophearoth Prum

Department of Computer Science and Engineering

CHALMERS UNIVERSITY OF TECHNOLOGY

Gothenburg 2026

www.chalmers.se

Bachelor Thesis 2026

Optimizing latency in multi-agent systems

A research on lowering latency, cost while maintain quality in multi-agent system

Sophearoth Prum

CHALMERS

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg 2026

Optimizing latency in multi-agent systems

Sophearoth Prum

Supervisor: Flavio Nicoletti
Examinator: John J. Camilleri

Bachelor thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 72 92 32361

Written in LaTeX
Gothenburg 2026

Optimizing latency in multi-agent system
Sophearoth Prum
Institutionen för Data- och informationsteknik
Chalmers Tekniska Högskola

Abstract

Large Language Model (LLM)-based multi-agent systems are increasingly used to solve complex tasks through collaboration between specialized agents. However, the use of multiple agents, tool invocations, and inter-agent communication can introduce significant latency and cost, limiting practical deployment.
This thesis investigates how architectural optimizations affect the performance of an LLM-based multi-agent system. A financial analysis pipeline was implemented using the Agent-to-Agent (A2A) protocol for inter-agent communication and the Model Context Protocol (MCP) for tool use. Four cumulative optimization techniques were evaluated: agent parallelization, tool batching, schema pruning, and model assignment. Performance was assessed using end-to-end latency, inference cost, and output quality.
The results show that agent parallelization provides negligible latency improvement under the evaluated deployment conditions due to shared model endpoint contention. In contrast, tool batching reduces median latency up to 27.4% and inference cost by 54.6% while improving output quality from 4.18 to 5.00. Schema pruning and model assignment techniques further reduces inference cost up to 77.1% compared to the baseline without degrading quality. Overall, the results suggest that reducing tool invocation overhead and unnecessary context transfer provides greater benefits than agent-level parallelization in the evaluated multi-agent architecture.

Keywords: Agent-to-Agent, Multi-agent system, Optimization, Latency, MCP.

Preface

This thesis was carried out as a degree project in the Bachelor's Programme in Computer Engineering at the Department of Computer Science and Engineering, Chalmers University of Technology, Gothenburg, during the spring of 2026. The work was conducted in collaboration with *Knightec Group*, who provided mentorship and a working environment throughout the project.

I would like to thank my supervisor *Flavio* for valuable guidance and feedback throughout the project. I also thank the examiner *John J. Camilleri* for his support and constructive input during the examination process.
Furthermore, I want to express my gratitude to *Felix Freden, Rasmus Standar, and Andrey Alishev* at *Knightec Group* for their support and for providing the opportunity to carry out this work.

Göteborg, June 2026
Sophearoth Prum

Contents

List of Figures

1

Introduction

Large Language Model (LLM)-based multi-agent systems are increasingly used to solve complex tasks through collaboration between specialized agents. These systems often rely on external tools and inter-agent communication protocols such as Model Context Protocol (MCP) and Agent-to-Agent (A2A). While these architectures improve modularity and task specialization, they also introduce additional latency through tool invocations and communication overhead. As multi-agent systems become more complex, reducing latency becomes increasingly important for practical and interactive applications.

1.1 Background

Multi-agent system (MAS) where multiple LLM-based agents collaborate to solve complex problems are increasingly being used in areas such as digital assistants, cloud automation and data analysis. For these systems to be practical in interactive and latency-sensitive applications, they require low latency and responsive execution.

Multi agent systems are being increasingly adopted due to the limitations of one agent. For example, if one agent has access to all the tools to do a complex task, it could easily be overwhelmed by all the responsibilities it has, leading to a decrease in quality response. [3] Additionally, One agent is slower in general compared to Multi Agent Systems due to the lack of parallelization that MAS benefit from by enabling multiple agents to work simultaneously.

Multi agent systems requires the agents to communicate with each other and use tools. Historically, these interactions relied on fragmented and custom integration approaches.Two key standards have emerged to overcome this: Agent-to-Agent (A2A) for inter-agent communication and Model Context Protocol (MCP) for tool invocation. [4]

One of the causes to high latency in MAS are delays that arises during inter-agent communications and tool requests via protocols such as Model Context Protocol (MCP) and Agent-to-Agent (A2A).[5, 2] In systems in which many agents want to communicate with each other and use tools, these delays could accumulate to a significant waiting time such that it limits the system's practical usability.

1.2 Purpose

The purpose of this thesis is to develop and evaluate methods for reducing latency in multi-agent system where agents communicate via A2A protocol and use MCP servers for tool requests, improving responsiveness and practical usability

1.3 Goal

The goal of the project is to develop a multi-agent prototype that uses A2A as a communication protocol and MCP tools, where we implement and evaluate latency-reducing techniques such as request batching, parallelization, and/or predictive tool allocation.

The prototype's performance will be measured and compared against a naive sequential implementation to quantify the improvement.

1.4 Limitations

This thesis does not address hardware-level optimization, including techniques specific to GPUs or CPUs. Instead, this thesis focuses on optimizations of the system's architecture, such as parallelization, caching, and related techniques.

Due to time constraints, not all possible optimization methods will be explored. This thesis will focus on techniques that are expected to have the most significant impact.

Additionally, this thesis will focus primarily on system design, therefore, any graphical user interface (GUI) will be excluded, and user requests will be handled via command-line via tools such as curl.

Furthermore, since the system is deployed in a cloud environment, latency measurements include network communication overhead such as authentication and inter-agent handshakes, and therefore do not reflect pure computational performance.

1.5 Research questions

This study investigates how architectural optimizations affect the performance of multi-agent systems. The following research questions guides the study:

- **RQ1:** How do different architectural optimizations affect the latency of a multi-agent system?
- **RQ2:** Which optimization techniques provide the best cost reduction?
- **RQ3:** To what extent do these optimization techniques affect the quality of the output?

2

Theory

This chapter presents the theoretical background and key technologies that form the foundation of the system. It introduces the core concepts of large language models, multi-agent systems, agent communication protocols, retrieval-augmented generation, and cloud-based agent deployment. These concepts are necessary to understand the design decisions and optimizations presented in later chapters.

2.1 Large Language Models and Agents

Large Language Models (LLMs) are neural network-based models trained to predict the next token in a sequence of text.[16].Modern LLMs are capable of performing reasoning-like behavior through in-context learning, where task-specific behavior is induced via prompting rather than explicit retraining.[17]

A limitation of LLMs is their reliance on probabilistic text generation, which can lead to wrong reasoning in structured or numerical tasks. To solve this, modern systems extend LLMs with external tools, allowing them to perform deterministic operations such as calculations, database queries, and API calls.

Tool-augmented LLMs introduce additional overhead in terms of latency and token usage, since each tool invocation requires additional context processing. This becomes relevant for system optimization, particularly in multi-agent pipelines where tool usage is frequent.

2.2 Multi-Agent Systems

A Multi-agent system is a system that consists of autonomous agents which are connected in a structured way that allows them to communicate, coordinate and share knowledge [3]

Multi-agent systems can be organized according to different coordination and orchestration patterns.Common workflow structures include:

- **Sequential pipelines**, where agents execute in a fixed order, with each agent's output becoming the input for the next [15].
- **Graph-based workflows**, where task dependencies are represented explicitly and independent tasks may execute concurrently [15].

- **Decentralized coordination**, where agents communicate without relying on a single central orchestrator [13].

Sequential pipelines are simple to implement but may introduce unnecessary latency due to strict execution ordering. In contrast, DAG-based execution enables parallelism when dependencies allow it, reducing overall system latency. This trade-off between simplicity and performance is a key consideration in multi-agent system design.

2.3 Agent Communication Protocols

2.3.1 Agent-to-Agent (A2A) Protocol

The Agent-to-Agent (A2A) protocol is an open standard introduced by Google in 2025 that enables direct communication and interoperability between AI agents across different frameworks and vendors [4].

A2A introduces concepts such as Agent Cards for capability discovery of new agents, Tasks for delegating units of work, and Messages for exchanging information during task execution. [4].

A2A introduces abstraction layers that simplify multi-agent coordination but may introduce additional communication overhead, particularly in distributed or cloud-based environments.

2.3.2 Model Context Protocol (MCP)

The Model Context Protocol (MCP) is an open standard introduced by Anthropic that addresses how an AI model or agent accesses external data, tools, and context in a uniform way [4].

MCP improves modularity and interoperability by decoupling tool implementation from agent logic. However, each tool definition is included in the model context, which can increase token usage and inference cost. This trade-off is relevant when optimizing systems with large toolsets.

2.4 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) is a technique that enhances LLM outputs by retrieving relevant external information at inference time and injecting it into the model context. [18]

A typical RAG system consists of three components:

- **Embedding model**, which converts text into vector representations.
- **Vector database**, which stores embeddings for efficient similarity search.
- **Retriever**, which selects relevant context based on a query.

RAG reduces hallucinations by grounding responses in external data sources. However, it introduces additional latency due to embedding computation and retrieval operations, and increases token usage due to context augmentation.

In this system, RAG is used to retrieve interpretations of financial indicators, ensuring that outputs are grounded in external documentation rather than model memory.

2.5 Cloud-Based Agent Execution and Observability

Modern LLM-based systems are often deployed in cloud environments to enable scalability, monitoring, and reproducibility of experiments. Cloud execution allows for standardized measurement of latency, cost, and tool usage across multiple runs.

Observability systems such as AWS CloudWatch capture execution traces, including:

- Invocation latency per agent
- Token usage per request
- Tool call frequency
- Execution traces and dependencies

This data is essential for evaluating system performance and identifying bottlenecks in multi-agent pipelines.

The trade-off is additional overhead from network latency and authentication on each inter-agent call, which contributes to the total system latency measured in this thesis.

2.6 Technology Stack

The system is implemented in Python and built using the Strands Agents framework, which provides abstractions for multi-agent orchestration, tool integration, and A2A communication.

The system utilizes:

- Claude Haiku 4.5 and Amazon Nova Micro as LLM backends
- MCP for tool integration
- AWS AgentCore for deployment and execution
- AWS CloudWatch for monitoring and trace collection
- S3 for report storage

The choice of technologies is driven by requirements for scalability, observability, and support for tool-augmented LLM workflows.

3

Methodology

This section describes the experimental methodology used to evaluate the multi-agent system. It details the baseline system, the incremental optimization configurations, and the evaluation procedure for measuring performance, cost, and quality.

3.1 Experimental Setup

The goal of the experimentation is to study how different optimization techniques could affect latency, cost and quality of the output of the multi-agent system. This section describes the evaluation metrics and the methodology used to measure them.

3.1.1 Experimental Design

The experiment uses an incremental approach where each configuration builds on the previous one by adding a single optimization. This makes it possible to evaluate both the individual contribution of each optimization and their combined effect on the system

Each stage implements a new architectural optimization that is expected to influence the system's core performance metrics: latency, cost-efficiency, and output quality.

3.2 Evaluation Procedure

To isolate the effect of each architectural optimization, all configurations were evaluated using an identical query, following standard controlled experiment design where only one variable is changed at a time. Two queries were used to evaluate robustness across different market conditions:

- **January 2018:** "Analyze the S&P 500 in January 2018" - representing stable market conditions.
- **March 2020:** "Analyze the S&P 500 in March 2020" -representing extreme volatility during the COVID-19 market crash.

These periods were selected to assess whether optimization techniques maintain consistent performance across varying market.Additionally, a two-month stress test

was conducted using the query "Analyze the S&P 500 during the first two months of 2011" to evaluate system behaviour beyond the primary benchmark scope.

The query was selected because it activates all agents in the pipeline, including data retrieval, technical analysis, visualization generation, and report writing, ensuring that all architectural components are evaluated during each run.

3.3 Experimental Environment

All experiments are conducted where each 4 sub agents (Data Fetcher, Data science, Visualizer and Writer) are deployed to AWS Agentcore in the region us-west-2 and uses AWS bedrock for model selection. Claude haiku 4.5 was selected as the default agent for all agents in the baseline configuration.

To reduce randomness and improve reproducibility across experimental runs, all models were configured with a temperature setting of 0.

3.4 Metrics

To evaluate the impact of the proposed optimization techniques, the system is measured across three primary dimensions: latency, cost, and output quality. These metrics were selected to reflect the main trade-offs in multi-agent systems, where improvements in performance may affect inference cost or response quality.

3.4.1 Latency

Latency is the primary metric used to evaluate architectural performance. This study categorizes latency into two distinct measurements to isolate the impact of system optimizations from external factors:

1. Application-Level Latency (Wall-clock Time): This is measured locally by recording the time elapsed between the initial request sent and the final response received. This "End-to-End" value includes agent inference time, tool execution, and the transit time over the network.

2. Server-Side Latency (CloudWatch): To isolate the performance of the agents within the AWS environment, logs are retrieved from Amazon CloudWatch. These logs record the execution duration strictly within the cloud infrastructure, excluding any delays caused by the local internet connection.

3. Network Latency By calculating the difference between the application-level latency (T_{app}) and the CloudWatch latency (T_{cloud}), the **Network Overhead** ($T_{network}$) can be isolated:

$$T_{network} = T_{app} - T_{cloud} \tag{3.1}$$

This separation is necessary to identify where the bottlenecks are, whether it is the agent execution time or the network overhead like authentication and, specifically, to A2A , agent card discovery which affects the latency.

3.4.2 Cost

Cost is measured in terms of the number of input and output tokens generated by each agent. This metric is important in multi-agent systems because the agents typically make numerous API calls, which can drive up expenses. By analyzing token usage, we can evaluate the efficiency of different optimization strategies.

Costs related to deployment infrastructure such as AgentCore, CloudWatch, or S3 storage are excluded, as the focus of this study is solely on the agent system itself.

For each end-to-end execution of the system, token usage is tracked for each individual agent. The total cost is the sum of all agents' costs. Each agents' cost is calculated by multiplying its input and output tokens by their respective prices

Model pricing was obtained from the official AWS Bedrock pricing documentation [11].

3.4.3 Quality

In addition to latency and cost, this study evaluates how the different optimization techniques affect the quality of the generated financial reports. This is necessary to ensure that architectural optimizations do not substantially degrade the usefulness, correctness, or coherence of the generated output.

3.4.3.1 LLM-as-Judge

To evaluate the quality of a report generated by an LLM, human judgment is traditionally considered the most nuanced and reliable metric. However, manual review is not only time-consuming but also prone to subjective inconsistencies. To ensure a scalable and objective judgment in all experimental runs, this study uses the LLM-as-a-Judge framework [1]. This approach allows for the automated assessment of reports by ranking the report by scoring them.

To ensure the reliability of the LLM-as-a-Judge framework, the judge's prompt was designed using two key prompt engineering techniques which was identified in the [1]

Decomposition of Evaluation Steps: Suggests breaking down the evaluation task into smaller tasks, giving detailed definitions and constraints for each small step in prompts, effectively guiding the LLM through the entire evaluation pipeline.[1]

Decomposition of Evaluation Criteria: This involves breaking down the evaluation into different dimensions. For example, when evaluating a financial report, we look at how factual the report is (to ensure no numbers are hallucinated) and the quality of its structure. To evaluate the report as a whole, four dimensions that the judge will evaluate the report on:

- **Factual Grounding:** Measures whether the numerical values in the report are correct. This includes checking that financial data is taken from the provided dataset and not hallucinated by the model. To further ensure the reliability of factual grounding evaluation, the judge prompt was provided with the correct pre-calculated values for all technical indicators. This allows the judge to verify numerical accuracy against known correct answers.
- **Indicator Coverage:** Evaluates whether the required technical indicators (e.g., moving averages, RSI, P/E ratios) are included and correctly calculated in the report.
- **Narrative Coherence:** Assesses how well the report is structured and how clearly the analysis is presented. A high score indicates a coherent and logically flowing text rather than fragmented output.
- **Actionability:** Evaluates whether the report provides useful insights or conclusions that could support decision-making, rather than remaining descriptive.
- **Source Attribution:** Evaluates whether the RAG system is used properly by checking if relevant external sources (e.g., the Technical Handbook) are incorporated and referenced when explaining financial concepts.

Additionally, the same study [1] highlights that LLM-based evaluators may exhibit self-preference bias, where models tend to favor outputs that resemble their own writing style. To reduce this effect, the evaluation model used in this study is Amazon Nova Pro, which belongs to a different model family than the models used in the main agent pipeline

3.4.4 Statistical Analysis

To account for variations in LLM inferences and network latencies, each configuration is executed in multiple runs so that the result metrics can be medians, mean, and other percentiles.

In this study, each configuration was executed 8 times each. This amount will ensure that the tests are reliable and to reduce random variance introduced by LLMs non-determinism as well as variance in network latencies.

Eight runs were selected as a balance between statistical reliability and practical constraints, including API cost and total experiment duration. Given that temperature was set to zero to minimize non-determinism, and that the primary metrics are medians rather than means, eight runs are sufficient to identify consistent trends while remaining feasible within the project's time and cost budget

3.4.4.1 Calculations

The median was used as the primary aggregate metric instead of the mean because LLM-based systems are inherently non-deterministic and may produce occasional

outlier executions. Median values are therefore more robust against extreme latency spikes and unstable responses.

In addition to the median, mean and latency values were also recorded to analyze worst-case performance characteristics.

The results are visualized using box plots that show the latency distribution across runs, including the median, the interquartile range, and outliers.

4

Implementation

This section describes the design and implementation of the multi-agent system. It details the baseline implementation, the optimization techniques applied.

4.1 Design and prototype

The system design phase defined the overall architecture of the multi-agent pipeline. A high-level diagram of the system was created during this phase to illustrate the interaction between agents, their assigned tools, and the overall execution flow.(See figure 4.1)

The system follows a workflow-based collaboration pattern, where each agent performs a specialized task in a fixed sequence and passes its output to the next component. A workflow is a structured coordination pattern in which tasks are executed sequentially across multiple agents, where each step depends on the output of the previous one [6].

4.2 Baseline System implementation

The baseline system is a sequential multi-agent pipeline which consists of four specialized agents that collaborate to generate a financial report: a Data Fetcher agent, a Data Science agent, a Graph Drawer agent, and a Report Writer agent. The pipeline is orchestrated by a central Python Script in a fixed pipeline where each agent receives the output of the previous agent as its input.

4.2.1 Agent Pipeline

The pipeline follows a workflow collaboration pattern, in which each agent has access to tools or knowledge base and can perform a specialized task before passing its result to the next agent. The execution order is as follow:

1. **Data Fetcher Agent:** Responsible for fetching data on the S&P500 and US macroeconomic data. The agent has access to 4 tools or functions that is implemented via MCP protocol that enables it to be able to retrieve external data:

The schema tools `get_sp500_schema` and `get_macroeconomic_schema` allow the agent to dynamically discover the structure of the data before querying it. These tools provide metadata information such as available columns and value ranges which prevents the agent from querying out of range values, for example in the S&P500 dataset, the available date ranges are from 1927-12-30 to 2020-11-04.

The query tools `query_sp500` and `query_macroeconomic_data` retrieve the actual data from the dataset. Both tools accept SQL queries, enabling the agent to perform precise filtering and aggregation rather than loading entire datasets into memory.

2. **Data science Agent:** Responsible for performing calculations and extracting insights from the data it receives from the Data Fetcher Agent. The agent has access to two tools to be able to perform its work: The `calculator` tool which provides the agent the ability to compute financial indicators such as Simple Moving Average (SMA), Relative Strength Index (RSI). This avoids relying on the LLM's internal arithmetic capabilities, which are known to be unreliable for numerical computations.

   The agent also has access to a `retrieve` tool, which queries a knowledge base containing interpretation guidelines for financial indicators such as RSI and SMA. This is implemented using Retrieval-Augmented Generation (RAG), where indicator interpretations are dynamically retrieved at runtime rather than relying on the LLM's possibly outdated information or hardcoding the information into the agent's system prompt which keeps the prompt concise and the knowledge base independently updatable. Additionally, since the knowledge base is sourced from a PDF document, the agent is able to cite specific sections of the source material in its output, improving the traceability and reliability of the generated report.

   The financial handbook (45 pages) was stored in an AWS Bedrock Knowledge Base using Titan Text Embeddings v2 with 1024-dimensional float vector embeddings stored in Amazon S3 Vectors. Default chunking was applied, splitting text into chunks of approximately 300 tokens with 20% overlap while respecting sentence boundaries. Retrieval was performed via similarity search using default configuration

3. **Visualizer Agent:** Responsible for generating a line chart of the S&P 500 index over the requested time period. The agent produces one chart per report using the `generate_line_chart` tool.

4. **Writer Agent:** is responsible for structuring, writing, and generating the report in markdown format. The writer takes input from all 3 agents' output and structure a coherent report in a specific financial styles. It has access to the tool `generate_markdown_report` which generates a markdown report that is then saved to a S3 bucket.

The interaction between these agents and their respective toolsets is illustrated in the system diagram in Figure 4.1

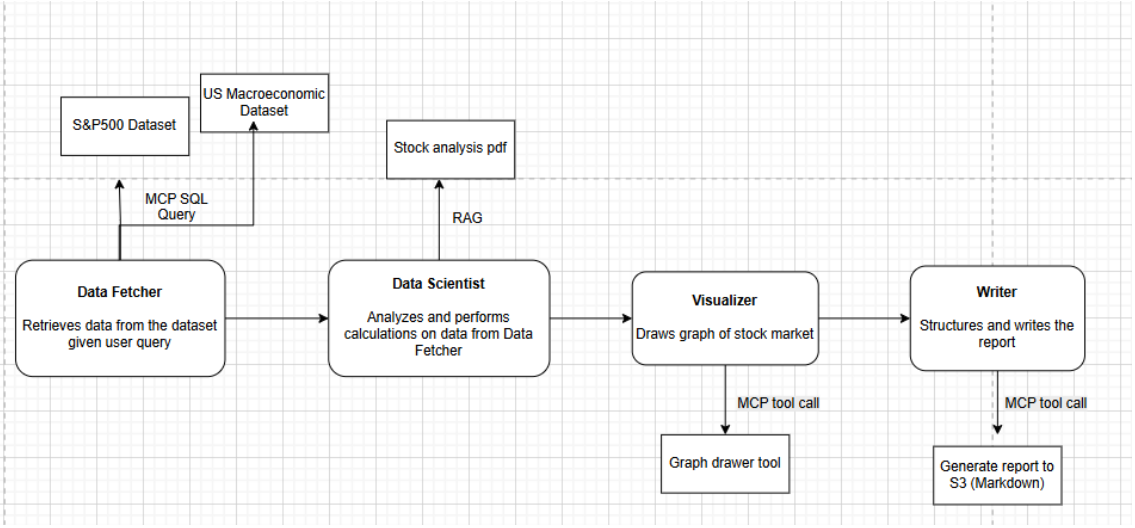


Figure 4.1: Baseline system diagram

4.3 Deployment in the cloud

In order to capture the timing, latencies, and token counts of each agent, they will be deployed to AWS Bedrock AgentCore. AgentCore will then automatically capture all those metrics after they are invoked and will display the results in AWS Cloudwatch.

Cloudwatch will also be able to capture the traces of the calls. Traces are detailed records of events and the sequence of actions each agent performed, including timing information and any interactions between components, which helps in analyzing performance and debugging the system. For example, traces could show that the Datascience agent performed nine separate calculator calls. This can be identified as non-optimal, and we could then optimize the agent to combine multiple calculations into a single call, thus saving time.

Each agent is packaged and deployed to AWS AgentCore using the `agentcore` CLI. The deployment process involves two steps: configuration and deployment [19].

First, the agent entry point is configured using the `agentcore configure` command, specifying the A2A protocol and OAuth authentication via an Amazon Cognito User Pool:

```
agentcore configure -e agent.py --protocol A2A
```

The Cognito User Pool was created separately and provides JWT-based authentication for inter-agent communication. The discovery URL and allowed client IDs from Cognito are provided during configuration.

Once configured, the agent is deployed using:

```
agentcore deploy
```

This builds a container image, pushes it to Amazon ECR, and deploy the container

and start AgentCore runtime. Upon completion, an Agent Runtime ARN is returned, which is used by the orchestrator to construct the A2A endpoint URL for each agent.

All four agents including Data Fetcher, Data Science, Visualizer, and Writer are deployed as independent runtimes following this process.

4.4 Optimization Configurations

This section describes the baseline and incremental optimization configurations that are evaluated in this study. Each configuration introduces a one modification to study its impact on latency, cost and quality.

4.4.1 Configuration 0: Baseline System

This stage utilizes the sequential system described in Section 4.2, the primary objective of which is to establish a performance baseline (or 'floor'). By executing the system in a controlled environment for a fixed number of iterations, we can calculate the mean and median for all core metrics: latency, cost, and quality.

The orchestrator invokes each agent sequentially using the A2A protocol. Each agent receives the output of the previous agent as its input context, and the next agent is only invoked after the current one completes.

4.4.2 Optimization 1: Parallel Agents

While the baseline system (Stage 1) utilizes a sequential pipeline, this optimization introduces an Asynchronous Directed Acyclic Graph (DAG) architecture. This change addresses the architectural inefficiency where the Visualizer and Data science agents were executed linearly despite having no functional dependency on one another. By recognizing that both agents depend solely on the output of the Data Fetcher, the DAG allows for concurrent execution. This same parallel technique is discussed in the Halo framework [7], where the authors reported that using parallel workers is central to their system's performance. Their research shows that without this parallel execution, the average latency decreases by 62%, validating the decision to move away from a sequential setup in this stage

Specifically, `asyncio.gather()` is used to dispatch A2A calls by the orchestrator to both the Data Science agent and the Visualizer agent simultaneously

This change is expected to reduce the latency while leaving cost and quality mostly unchanged. Since the total amount of work (tokens processed) is exactly the same as the baseline, the cost is not expected to significantly change. We are just changing when the work happens.

The modified interaction between these agents and their respective toolsets is illustrated in the system diagram in Figure 4.2

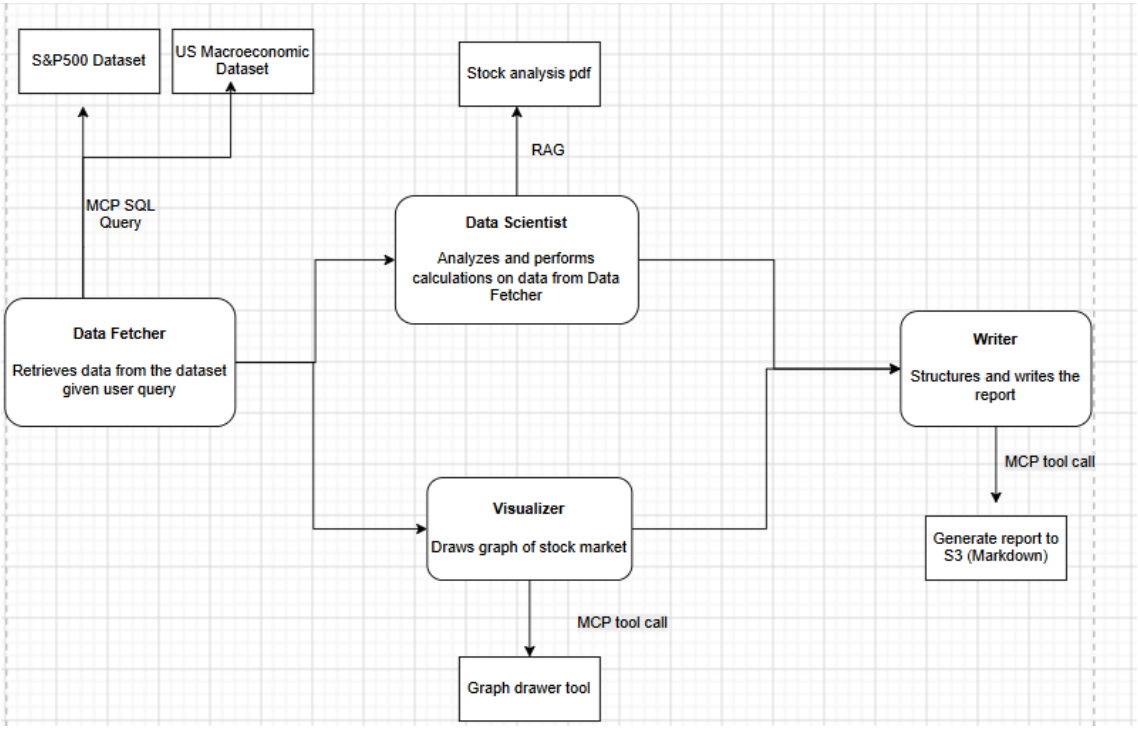


Figure 4.2: Optimization 1 diagram (parallelized agents)

4.4.3 Optimization 2: Batch tool

This optimization focuses on reducing the number of tool calls performed by the Data Science agent.

In the previous configurations, the Data Science agent is responsible for computing technical indicators such as SMA, EMA, RSI, and MACD using an atomic `calculator` tool which Strands provides. For each indicator, the agent calls the tool, receives the result then decide what to calculate next. This means that computing each indicator requires many separate calculator tool calls, and between each call the model must process the entire conversation history up to that point, costing both latency and token counts.

This configuration replaces the `calculator` tool completely with a custom python code interpreter tool that the agent can use. Instead of calculating indicator one at a time, the agent writes a single python script that computes all required indicator all at once and return the result in a single step.

This follows the principle established in [8], where the authors demonstrate that requiring an LLM to generate and execute a Python function for a class of tasks reduces the cost of inference compared to repeated calls to the atomic tool, while maintaining the equivalent output quality.

4.4.4 Optimization 3: Tool schema pruning

The visualizer agent connects to an external mcp tool, `antv/mcp-server-chart` that allows it to draw graphs. The tool exposes up to 26 different graph generating tools for the agent including mind maps, flowcharts, network graphs, and geographic maps. However, the Visualizer agent in this system only requires a small subset of these tools to complete its task. Despite this, the full tool schema for all 26 tools is loaded into the agent's context on every invocation. Each tool has its own description of how to use it, what the input should be and output structure which contributes approximately 41,000 input tokens per call regardless of which tools are actually used.

Prior work has identified tool schemas as a significant source of token overhead in MCP-based agentic systems. Recent approaches such as semantic tool discovery and dynamic tool gating reduce context size by exposing only tools relevant to the current task rather than loading the entire tool catalog into the model context [22]

This configuration reduces the number of tools loaded into the Visualizer agent's context by excluding chart types irrelevant to financial analysis. This is expected to substantially reduce the input token consumption of the Visualizer agent, thereby reducing cost.

To disable specific tools from an MCP server, the server supports an environment variable that allows selective tool exclusion. In this system, unwanted tools are disabled by passing a comma-separated string of tool names through the `DISABLED_TOOLS` environment variable during the initialization of the MCP server.

```
StdioServerParameters(
    command="npx",
    args=["@antv/mcp-server-chart"],
    env={
        "DISABLED_TOOLS": "tool_a,tool_b,tool_c"
    },
)
```

However, this optimization is not expected to improve end-to-end system latency. In the preceding configuration, the Visualizer and Data Science agents execute concurrently. Even if the Visualizer completes its task faster due to the reduced context, the pipeline must still wait for the Data Science agent to finish before proceeding to the Report Writer. Since the Data Science agent consistently takes longer to complete than the Visualizer, it remains the bottleneck of the parallel branch, and any latency reduction in the Visualizer has no effect on total pipeline duration.

4.4.5 Optimization 4: Model Assignment

In all previous configurations, every agent in the pipeline uses the same model: Claude Haiku 4.5. This uniform assignment does not account for the difference in task complexity across agents. The Data Fetcher's task is to write structured SQL queries against a fixed dataset, and the Visualizer's task is to select and call a chart

generation tool with the provided data. Both tasks are straightforward and do not require the reasoning capability that Haiku provides.

The Data Science agent, by contrast, must compute multiple technical indicators across several reasoning steps and retrieve relevant information from the knowledge base to contextualize its findings. The Report Writer receives output from all preceding agents and must synthesize it into a coherent, structured financial report. Both tasks involve multi-step reasoning and benefit from a more capable model.

This approach is inspired by recent work on heterogeneous model routing in multi-agent systems. Prior research has shown that assigning model capacity according to task complexity can substantially reduce computational cost while maintaining performance [21]. Unlike confidence-aware routing approaches, this thesis uses a static assignment strategy in which lighter models are manually assigned to agents with less demanding tasks.

This configuration assigns Amazon Nova Micro to the Data Fetcher and Visualizer agents, while the Data Science agent and Report Writer retain Claude Haiku 4.5. At $0.0175 per million input tokens and $0.07 per million output tokens, Nova Micro is approximately 46x cheaper on input and 57x cheaper on output compared to Haiku 4.5 ($0.80 and $4.00 per million respectively). This is expected to significantly reduce the overall cost of the pipeline, as the token savings on the Fetcher and Visualizer agents compound across all 15 runs. A secondary effect of using a smaller model is faster inference time, which may also reduce the latency of those two agents.

4.5 Metrics Collection

To evaluate system performance, metrics are recorded and stored in two ways

Application-level latency is measured locally by recording wall-clock time from the moment the orchestrator dispatches the first agent request to the moment the final response is received.

All other runtime metrics are extracted from AWS CloudWatch logs generated by the AgentCore runtime environment. Each agent invocation produces structured telemetry spans using the Strands tracing framework [6], which records latency, token usage, and model information per invocation.It also captures the execution trace of each agent, including the tools invoked and their order of execution.

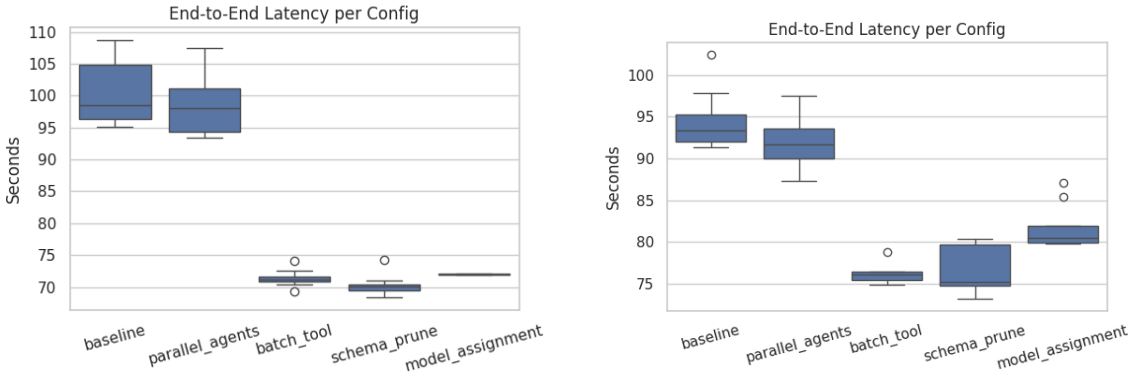
The logs are stored in the CloudWatch log group `aws/spans`, where each span contains structured attributes such as `gen_ai.usage.input_tokens`,`gen_ai.usage.output_tokens`, and execution duration.

5

Result

This chapter presents the results of all optimization configurations, evaluated across three dimensions: end-to-end latency, cost, and output quality. Each configuration builds cumulatively on the previous one. Evaluations were conducted across two market periods: January 2018, representing stable market conditions, and March 2020, representing the COVID-19 market crash.

5.1 End-to-End Latency



(a) January 2018 **(b)** March 2020

Figure 5.1: Box plot of end-to-end latency across all five configurations (Lower is better)

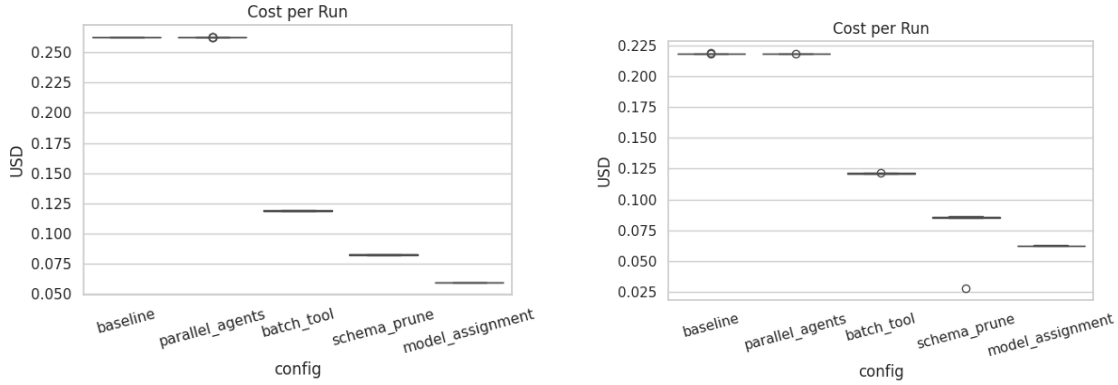
Parallel execution provides negligible latency improvement in both periods: 1.8% in January 2018 and 2.9% in March 2020.

The batch_tool configuration achieves the most meaningful latency reduction, bringing median latency from 98.0s to 71.2s in January 2018 (27.4%) and from 91.6s to 76.1s in March 2020 (16.9%). Both improvements have measurable improvements. Schema pruning provides a further improvement of 1.4% and 1.1% respectively, within one standard deviation. Model assignment slightly increases latency in both periods with median rising from 75.25 to 80.49 in 2020 and from 70.12 to 72 in 2018 which indicates that model substitution does not improve latency.

Table 5.1: Latency improvement relative to previous configuration across evaluation periods (negative values indicate regression)

Configuration	Jan 2018	Mar 2020
parallel_agents	1.8%	2.9%
batch_tool	27.4%	16.9%
schema_prune	1.4%	1.1%
model_assignment	-2.7%	-7.0%

5.2 Inference cost per report generation

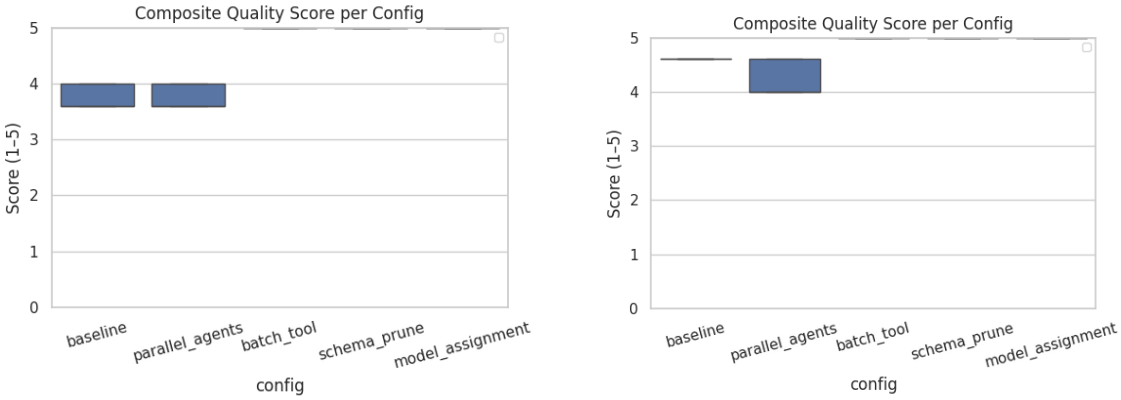


(a) January 2018 **(b)** March 2020

Figure 5.2: Box plots of inference cost per run across all five configurations. Lower values indicate better performance.

Parallel execution shows no meaningful cost change in either period. Tool batching provides the largest single cost reduction, 54.6% in January 2018 and 44.4% in March 2020. Schema pruning and model assignment further reduce cost incrementally, reaching a cumulative reduction of 77.1% in January 2018 and 71.5% in March 2020.

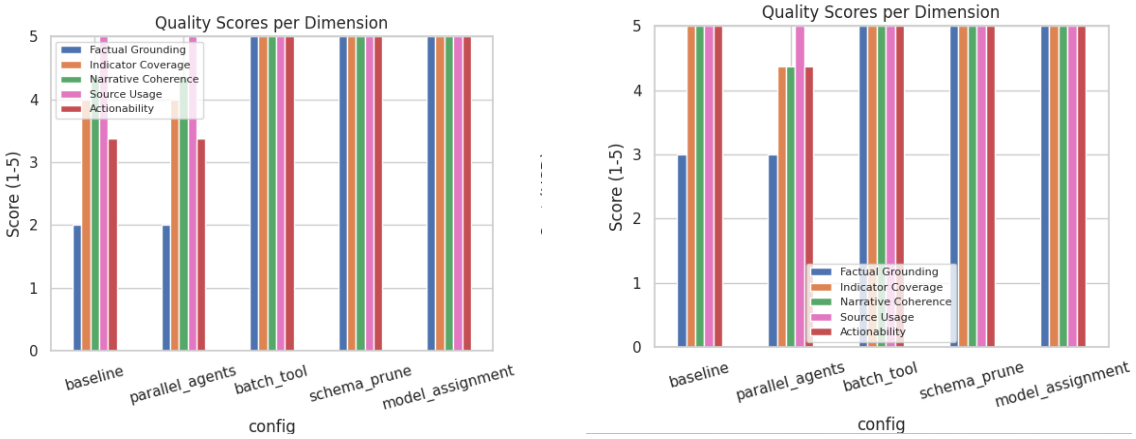
5.3 Output Quality



(a) January 2018 **(b)** March 2020

Figure 5.3: Box plot of composite quality score across all five configurations (Higher is better)

Output quality remains stable or improves across configurations. The baseline achieves 3.75 in January 2018 and 4.60 in March 2020. From batch_tool onwards, the composite score reaches 5.00 in both periods, driven primarily by improved factual grounding through deterministic Python computation.



(a) January 2018 **(b)** March 2020

Figure 5.4: Quality scores per dimension across all five configurations (Higher is better)

5.4 Stress Test: Two-Month Workload

To evaluate system behaviour beyond the primary benchmark, a two-month query horizon was tested. The baseline and parallel_agents configurations failed consistently due to malformed tool inputs under increased data complexity. Batch_tool

and schema_prune completed successfully with median latency of approximately 83-85 seconds. Model assignment failed because nova-micro exceeded its output token constraints and could not generate the complete two-month response.

Table 5.2: Two-month stress test outcomes

Configuration	**Outcome**	**Failure Mode**
baseline	Failed	Malformed tool inputs
parallel_agents	Failed	Malformed tool inputs
batch_tool	Completed	—
schema_prune	Completed	—
model_assignment	Failed	Nova-micro output limit

These findings suggest that cumulative optimizations improve system robustness under extended workloads, while model assignment introduces a reliability ceiling at scale.

6

Discussion

This chapter discusses the observed results in relation to the research questions. The focus is on explaining the causes behind the performance improvements, analyzing trade-offs between latency, cost, and quality, and identifying limitations of the proposed system.

6.1 Overview of Findings

The results show that the architectural optimizations evaluated in this multi-agent system substantially reduced end-to-end latency and inference cost while maintaining stable output quality. The largest gains were achieved through tool batching, which simultaneously reduced latency, cost, and improved output quality.

Results are consistent in direction across both evaluation periods, supporting the robustness of the findings. Parallel execution provides no meaningful improvement in either period. Tool batching provides the greatest marginal latency reduction in both periods, and model assignment achieves the greatest cumulative cost reduction. The magnitude of improvement differs between periods, partly due to differences in baseline latency (98.5s in January 2018 vs 93.3s in March 2020).

6.2 Impact on Latency (RQ1)

The results indicate that end-to-end latency is primarily influenced by sequential dependencies and tool execution overhead rather than agent-level parallelism.

The parallel_agents configuration does not provide a meaningful reduction in median latency compared to the baseline (98.5s vs 98.0s in January 2018, 93.3s vs 91.7s in March 2020), and this difference is not substantial in either period. In addition, this configuration exhibits increased variance, suggesting unstable execution behavior under concurrent workloads.

One possible explanation for this behavior is related to shared service constraints when multiple concurrent requests are sent to the same underlying model endpoint in AWS. Prior work on AWS Bedrock performance testing has shown that increased concurrency can lead to non-linear scaling effects due to service quotas, queueing, and internal request scheduling overhead rather than pure compute scaling [20].

In such conditions, concurrent requests may not execute independently, and performance can degrade or stay unchanged despite parallelization at the application level. This is seen in the table 6.1 where the Data science's latency increased in parallel configuration from baseline.

To further investigate this effect, an additional experiment was conducted in which the Visualizer agent was executed in parallel using a separate model provider. This configuration showed a measurable reduction in latency variance and improved stability compared to the shared-provider setup, suggesting that shared-model contention is a contributing factor to the observed performance behavior. (See figure 6.1)

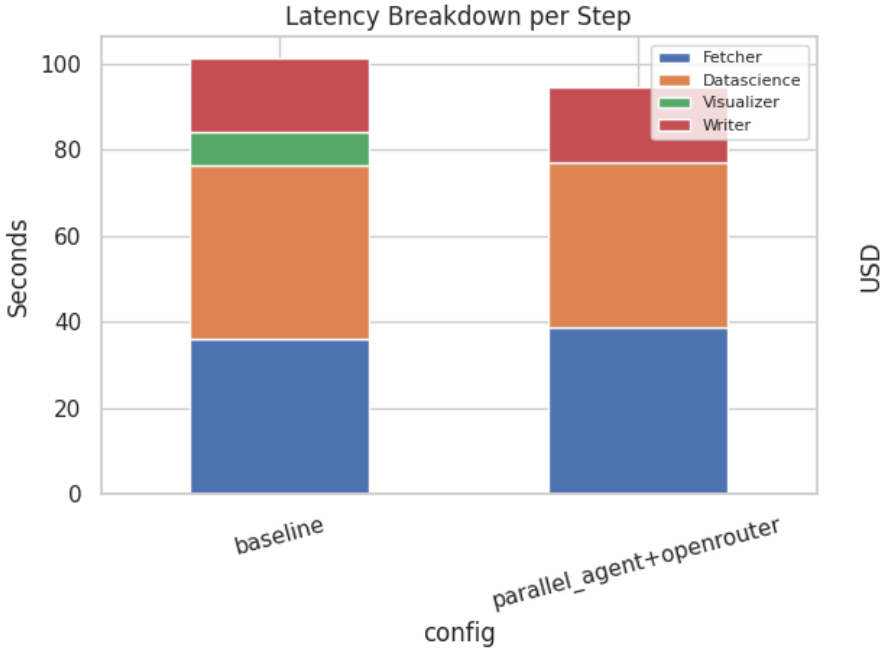


Figure 6.1: Latency comparison between baseline and parallel agent execution using a separate model provider

In contrast, the most meaningful reductions in latency are observed in configurations that reduce the number and cost of tool invocations. The batch_tool configuration reduces median latency to 71.2 seconds in January 2018 and 76.1 seconds in March 2020, corresponding to reductions of 27.4% and 18.8% respectively.

This is mainly because the pipeline is compute-heavy, particularly the Data Science agent which requires multiple sequential calculator tool calls to compute technical indicators such as RSI and SMA. Each call adding an extra LLM invocation and increasing the total response time and cost.

Table 6.1 shows that the Visualizer agent accounts for only about 9% (6.19s) of total server-side latency. Since the Visualizer runs in parallel with the Data Science agent, which accounts for approximately 60% of total latency, parallelization only attempts

to hide this minor component while leaving the dominant bottleneck untouched. In contrast, tool batching directly targets the Data Science agent, reducing its latency by 52% (41.5s to 20.1s), suggesting that optimizing the dominant bottleneck is far more effective than parallelizing agents that are not the bottleneck in the evaluated system.

Table 6.1: 5 server-side latency (excludes network latency) per agent across configurations, January 2018 (seconds)

Configuration	Fetcher	Visualizer	Datascience	Writer	Total
baseline	8.09	6.19	41.50	13.99	69.77
parallel_agents	7.99	6.10	46.86	14.17	75.12
batch_tool	8.10	6.38	20.09	17.04	51.61
schema_prune	8.06	5.69	20.25	15.27	49.27
model_assignment	8.55	4.36	22.78	13.98	49.67

Adding on Schema pruning technique onto batch_tool maintain comparable latency under standard load conditions, with differences falling within one standard deviation. It is also observed that the visualizer agent's latency decreases slightly (see Table 6.1), but since it runs in parallel with the Data Science agent, this has little impact on overall system latency.

However, trace-level analysis reveals that adding model assignment technique introduces a latency tradeoff. Although nova-micro has a lower time-to-first-token compared to Claude Haiku 4.5, it generates approximately more output tokens in the data fetcher agent. This increases the input context for downstream agents, and affecting the data science agent to take a longer than it usually does. This shows that in the evaluated multi-agent pipeline, even small changes can propagate downstream and affect other agents.

As a result, model assignment does not improve end-to-end latency despite using a faster model, and under March 2020 conditions exhibits a regression of 7.0% relative to schema_prune. as seen in Table 6.1 where the latency of the Data fetcher agent actually increase as well as the Data science agent

6.3 Impact on Cost (RQ2)

The observed reduction in inference cost is primarily driven by three mechanisms: reduction in token usage through tool batching, context pruning through schema pruning, and model substitution through model assignment.

Parallel execution does not meaningfully reduce cost, as it does not change the total number of tokens processed, but only the order of execution of the tasks. Instead, cost reduction is achieved through architectural and model-level optimizations.

The batch_tool configuration produces the largest single cost reduction, lowering mean cost by 54.6% in January 2018 and 44.4% in March 2020. By grouping calculator calls into single tool invocations, the number of prompt processing steps across

the pipeline is reduced directly. Since each tool call requires additional prompt processing, reducing the number of calls directly decreases total token consumption.

Schema pruning achieves a further 30.4% marginal cost reduction in January 2018 and 35.5% in March 2020 by reducing the input context passed to the visualizer agent from approximately 42,000 tokens to 10,000 tokens. This directly reduces token billing for the visualizer without meaningfully affecting latency, confirming that schema pruning operates primarily as a cost optimization rather than a latency optimization.

Model assignment achieves the greatest cumulative cost reduction, bringing mean cost to 0.0600 USD per end-to-end run in January 2018 which is a 77.1% reduction relative to baseline by substituting Claude Haiku 4.5 with nova-micro for the data fetcher and visualizer agents.

Overall, cost savings accumulated across the evaluated pipeline, suggesting that in this system cost efficiency was mainly driven by reducing LLM invocations, pruning redundant context, and assigning cost-efficient models to appropriate pipeline stages.

6.4 Impact on Output Quality (RQ3)

Despite reductions in latency and cost, output quality does not degrade across most configurations. The composite quality score improves from 3.75 in the January 2018 baseline and 4.60 in the March 2020 baseline to 5.00 from batch_tool onwards.

The primary driver of this improvement is factual grounding. In the baseline configuration, the Data Science agent relies on the LLM to perform numerical calculations using the calculator tool, such as computing technical indicators like RSI and SMA. This approach is prone to calculation errors, as LLMs are not inherently reliable arithmetic engines.

The batch_tool configuration addresses this by delegating computations to a Python execution environment, where calculations are performed deterministically. This eliminates the main source of factual error in the pipeline, improving the factual grounding score from 2.0 to 5.0 and lifting the composite score to its maximum value.

6.5 Stress Test: Two-Month Workload

To evaluate system behaviour beyond the primary benchmark scope, a two-month query horizon was tested.

The baseline configuration failed consistently, with trace analysis revealing that increased data volume caused the data science agent to produce malformed tool inputs to the calculator, resulting in execution errors. This indicates that unoptimized agents in the evaluated system are more prone to reasoning failures under higher data complexity specifically when using the calculator tool.

The batch_tool and schema_prune configurations were successfully completed with a median latency of approximately 83-85 seconds, compared to 70-76 seconds under a one-month workload. The additional latency is primarily caused by the increase of fetcher agent processing time under the larger dataset.

The model_assignment configuration exhibited data retrieval errors because nova-micro hit its output token limit, showing that nova-micro has a limit for longer tasks.

These findings suggest that the cumulative optimizations improve not only latency and cost but also system robustness under extended workloads. However, model assignment introduces a reliability ceiling that must be considered for production deployments requiring extended query horizons.

6.6 Limitations and Threats to Validity

Several limitations might affect the generalizability of the results.

First, the system was deployed in a controlled cloud environment (AWS us-west-2), meaning that a portion of the observed latency comes from network communication overhead between agents such as network latency, A2A handshake protocol and AWS authentication, which contributes to about 27-30% of the total end-to-end latency.

Additionally, the Strands framework does not support connection pooling or advanced request reuse mechanisms. This limits the ability to further optimize network and tool invocation overhead.

Third, large language models are inherently non-deterministic. Although temperature was set to zero to reduce randomness, variations in token length and tool usage still introduce minor fluctuations in latency measurements.

Fourth, tool performance is sensitive to prompt and schema design. Poorly specified tool instructions may lead to incorrect tool calls or retries, which introduce unnecessary latency overhead.

Furthermore, the evaluation was conducted using two fixed queries representing contrasting market conditions held constant across all runs, in order to isolate the effect of each architectural optimization and eliminate input variation. Generalizability across diverse query types remains to be validated in future work.

Finally, the quality evaluation relies entirely on an automated LLM-as-judge framework with no validation against human evaluators, making quality conclusions weaker than the latency and cost findings. Large quality improvements, such as the factual grounding score increasing from 2.0 to 5.0 at the batch_tool configuration, reflect real system-level changes and are considered reliable. However, smaller score differences between configurations should be interpreted cautiously, as minor variations in report phrasing can influence automated evaluation outcomes independently of actual quality differences.

6.7 Future Work

Several directions for future improvement can be identified.

First, more advanced scheduling strategies could be explored, including dynamic task allocation and automatic dependency resolution to increase parallel execution opportunities.

Second, automatic tool selection and routing mechanisms could reduce reliance on manually defined tool schemas and improve efficiency in multi-tool environments.

Third, prompt caching could be explored as a technique to reduce redundant context processing across repeated agent invocations. This was excluded from the current evaluation for two reasons: the Strands telemetry pipeline does not expose cache-specific token metrics, making reliable cost measurement difficult, and the system prompts of most agents in this study fall below the minimum token threshold required for prompt caching to take effect on Claude Haiku 4.5 [9].

Fourth, network-level optimizations, including connection pooling and request reuse, could further reduce communication overhead between agents and external services.

Additionally, the output token constraints observed in model_assignment under stress-test workloads suggest that a static assignment strategy is insufficient for variable complexity tasks. A confidence-aware dynamic routing approach, as proposed in [21], could address this limitation by switching to a more capable model when task complexity exceeds the smaller model's capacity.

Finally, future evaluations can include a diverse set of queries and include different time periods and market conditions to assess the generalizability of the optimization techniques

6.8 Conclusion

Performance improvements in multi-agent systems are dictated by the nature of the bottleneck. In the evaluated system, the dominant bottleneck was tool invocation overhead inside the Data Science agent.

The largest improvements were achieved through tool batching and schema pruning, while parallel execution provided limited benefit under the evaluated deployment conditions. This suggests that parallelization can improve latency more effectively when agents are distributed across separate model endpoints, avoiding contention on shared inference resources.

These findings highlight that identifying the actual bottleneck is more valuable than applying general-purpose optimizations in multi-agent pipeline design.

Bibliography

[1] J. Gu, et al., *A Survey on LLM-as-a-Judge*, arXiv preprint, 2024. [Online]. Available: https://arxiv.org/abs/2411.15594. Accessed: Mar. 19, 2026.

[2] Z. Ding, M. Zhu, Y. Liu, *Network and Systems Performance Characterization of MCP-Enabled LLM Agents*, arXiv preprint, 2025. [Online]. Available: https://arxiv.org/abs/2511.07426. Accessed: Feb. 20, 2026.

[3] A. Kumar, E. Augustinsson, M. Zethraeus, P. Sarathi, and R. Bridges, *A Practical Approach to Optimize Multi-Agent Systems*, AI Sweden, Dec. 16, 2025. [Online]. Available: https://www.ai.se/sites/default/files/2025-12/A%20Practical%20Approach%20to%20Optimize%20Multi-Agent%20Systems-v2.pdf. Accessed: Apr. 17, 2026.

[4] Q. Li and Y. Xie, *From Glue-Code to Protocols: A Critical Analysis of A2A and MCP Integration for Scalable Agent Systems*, arXiv preprint arXiv:2505.03864, 2025. [Online]. Available: https://arxiv.org/abs/2505.03864. Accessed: Feb. 20, 2026.

[5] Guild.ai Team, *Agent-to-Agent Protocol (A2A)*, Guild.ai, Jan. 2026. [Online]. Available: https://www.guild.ai/glossary/agent-to-agent-protocol-a2a. Accessed: Feb. 20, 2026.

[6] AWS Builder Center, *Strands Workflow Tool: Building Multi-Agent Business Process Automation*, AWS Builder Center, 2026. [Online]. Available: https://aws.amazon.com/builder-center/strands-workflow/. Accessed: Mar. 20, 2026.

[7] J. Shen, N. Wadlom, Y. Lu, *Batch Query Processing and Optimization for Agentic Workflows*, arXiv preprint, 2025. [Online]. Available: https://arxiv.org/abs/2509.02121 . Accessed: Apr. 15, 2026.

[8] T. Cai, et al., *Large Language Models as Tool Makers*, arXiv preprint, 2023. [Online]. Available: https://arxiv.org/abs/2305.17126. Accessed: Apr. 29, 2026.

[9] Anthropic, *Prompt Caching*, Anthropic Documentation, 2025. [Online]. Available: https://docs.anthropic.com/en/docs/build-with-claude/prompt-caching. Accessed: Apr. 22, 2026.

[10] AWS / Strands Agents, *Amazon Bedrock — Caching*, Strands Agents SDK Documentation, 2026. [Online]. Available: https://strandsagents.com/docs/user-guide/concepts/model-providers/amazon-bedrock/. Accessed: Apr. 22, 2026.

[11] Amazon Web Services (AWS), *Amazon Bedrock Pricing*, AWS Documentation, 2026. [Online]. Available: https://aws.amazon.com/bedrock/pricing/. Accessed: May. 13, 2026.

[12] X. Li, S. Wang, S. Zeng, Y. Wu, and Y. Yang, *A Survey on LLM-based Multi-Agent Systems: Workflow, Infrastructure, and Challenges*, *Vicinagearth*, vol. 1, no. 1, p. 9, 2024. [Online]. Available: https://arxiv.org/abs/2412.17481. Accessed: May. 17, 2026.

[13] T. Guo, et al., *Large Language Model based Multi-Agents: A Survey of Progress and Challenges*, arXiv preprint arXiv:2402.01680, 2024. [Online]. Available: https://arxiv.org/abs/2402.01680. Accessed: May. 17, 2026.

[14] Y. Chen, et al., *From Static Templates to Dynamic Runtime Graphs: A Survey of Workflow Optimization for LLM Agents*, arXiv preprint, 2026. [Online]. Available: https://arxiv.org/pdf/2603.22386. Accessed: May. 17, 2026.

[15] AWS / Strands Agents, *Agent Workflows: Building Multi-Agent Systems with Strands Agents SDK*, Strands Agents SDK Documentation, 2026. [Online]. Available: https://strandsagents.com/latest/user-guide/multi-agent/workflow/. Accessed: May. 17, 2026.

[16] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, *Attention Is All You Need*, arXiv preprint arXiv:1706.03762, 2017. [Online]. Available: https://arxiv.org/abs/1706.03762. Accessed: May. 18, 2026.

[17] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, *Language Models are Few-Shot Learners*, arXiv preprint arXiv:2005.14165, 2020. [Online]. Available: https://arxiv.org/abs/2005.14165. Accessed: May. 18, 2026.

[18] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*, arXiv preprint arXiv:2005.11401, 2020. [Online]. Available: https://arxiv.org/abs/2005.11401. Accessed: May. 18, 2026.

[19] K. Kamino, *Building and Deploying an A2A Server with Strands Agents + Amazon Bedrock AgentCore*, DevelopersIO, Classmethod Inc., Feb. 1, 2026. [Online]. Available: https://dev.classmethod.jp/en/articles/strands-agents-amazon-bedrock-agentcore-a2a/. Accessed: May. 18, 2026.

[20] Avahi, *Performance Testing AWS Bedrock Foundational Models*, Avahi Blog, Jan. 8, 2026. [Online]. Available: https://avahi.ai/case-study/performance-testing-aws-bedrock-foundational-models/. Accessed: May. 22, 2026.

[21] J. Wang, S. Zhao, J. Liu, H. Wang, W. Li, B. Qin, and T. Liu, *Orchestrating Intelligence: Confidence-Aware Routing for Efficient Multi-Agent Collaboration across Multi-Scale Models* arXiv:2601.04861, 2026. [Online]. Available: https://arxiv.org/abs/2601.04861

[22] S. Mudunuri, J. Wan, A. Qin, and S. Manoharan, *Semantic Tool Discovery for Large Language Models: A Vector-Based Approach to MCP Tool Selection*, arXiv preprint arXiv:2603.20313, 2026. [Online]. Available: https://arxiv.org/abs/2603.20313. Accessed: May. 22, 2026.

A

LLM-as-Judge Prompt

```
### EVALUATION TASK
You are evaluating a financial report against VERIFIED GROUND TRUTH METRICS compute

### EVALUATION STEPS (Decomposition)
Follow these steps before assigning scores:
You must:
1. Compare every numerical claim in the report against the verified metrics.
2. List all inconsistencies.
3. Evaluate qualitative analysis separately from numerical accuracy.
4. Penalize fabricated or unsupported claims.
5. Identify which technical indicators (RSI, MA, etc.) were requested vs. which wer
6. Check if the report cites the provided Technical Handbook for its definitions.
7. Assess if the final recommendation is supported by the findings.

NUMERICAL TOLERANCE RULES:
- SMA: ±0.1 acceptable
- RSI: ±2 acceptable
- MACD: ±1.0 acceptable
- Return: ±0.05% acceptable

### DIMENSIONS
1. Factual Grounding: (1-5) Are numbers accurate? Penalize hallucinations or invent
2. Indicator Coverage: (1-5) Did the agent use the technical indicators provided in
3. Narrative Coherence: (1-5) Is the report well-structured and easy to read, or ju
4. Actionability: (1-5) Does the report provide a clear decision-making signal or o
5. Source Usage: (1-5) Did the agent correctly use and cite the Technical Handbook?

### INPUT DATA
- ORIGINAL QUERY: {query}
- VERIFIED METRICS: {ground_truth}
- REPORT TO EVALUATE: {report}

### OUTPUT FORMAT
Return ONLY JSON:
{
```

```
  "audit_log": "...",
  "scores": {
    "factual_grounding": {"score": X, "reasoning": "..."},
    "indicator_coverage": {"score": X, "reasoning": "..."},
    "narrative_coherence": {"score": X, "reasoning": "..."},
    "actionability": {"score": X, "reasoning": "..."},
    "source_usage": {"score": X, "reasoning": "..."}
  }
}
```

INSTITUTIONEN FÖR Data- och informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige
www.chalmers.se

CHALMERS