CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

An Empirical Comparison of Multi-Agent LLM Architectural Patterns for Automated Unit Test Generation

Master's Thesis in Computer science and engineering

Mohammed Agha
Ayah Miqdad

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's Thesis 2026

An Empirical Comparison of Multi-Agent LLM Architectural Patterns for Automated Unit Test Generation

Mohammed Agha
Ayah Miqdad

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

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

An Empirical Comparison of Multi-Agent LLM Architectural Patterns for Automated Unit Test Generation

Supervisor: Robert Feldt, Department of Computer Science and Engineering
Examiner: Christian Berger, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Mohammed Agha
Ayah Miqdad
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Software engineers are increasingly adopting multi-agent large language model (LLM) systems, yet the controlled empirical evidence comparing the underlying architectural patterns' trade offs on software engineering tasks is still lacking. This thesis applies a two phase mixed-methods design. The first phase conducts a structured review of 31 articles published between 2023 and 2026, resulting in a taxonomy of 10 coordination patterns for multi-agent LLM systems. In the second phase, an empirical experiment comparing three patterns a Single-Agent Baseline, a Sequential architecture, and a Hierarchical architecture is carried out on automated unit test generation using the TestEval benchmark. Five dependent variables are evaluated (success rate, error handling, latency, cost, and agent communication turns) running each architecture independently on 30 tasks, all using Claude Sonnet 4, for 270 task executions in total. The Sequential pattern recorded the highest success rate (92.2%), the lowest latency variance (295.56 s^2), and a competitive cost (0.089 USD per task). The Single-Agent Baseline reached a moderate success rate (81.1%) at the lowest cost (0.083 USD per task), while the Hierarchical pattern recorded the lowest success rate (54.4%), the highest median latency (142.37 s), and the highest mean cost (0.308 USD per task). The hierarchical pattern suffered from supervisor information bottleneck according to the qualitative results confirmed by 100% of the failing tasks of the architecture. These results come from a single LLM, a single benchmark, and non-optimized prompts per architecture, which means replication is needed before these findings are treated as general architectural principles.

Keywords: LLMs, Multi-agent AI systems, Software unit test generation, Architecture patterns in multi-agent systems, Agentic AI.

Acknowledgements

We would like to thank our supervisor Prof. Robert Feldt for his guidance, feedback, and continued support throughout the course of the thesis. We would also like to thank the examiner Prof. Christian Berger for the constructive feedback provided during the review process, which helped us reflect more critically on the validity and generalizability of our findings. We gratefully acknowledge LayerLogic for providing access to Amazon Web Services (AWS) credits. The computational resources made available through LayerLogic's support were essential for running the multi-agent pipeline experiments and collecting the token usage measurements reported in this study. Finally, we express our gratitude to Chalmers University of Technology and The University of Gothenburg for providing the academic environment and resources that made this research possible.

Contents

List of Figures

List of Tables

1

Introduction

1.1 Problem Description

Large Language Models (LLMs) have become an essential part of modern intelligent software systems due to their capacity to plan and reason at levels similar to those of humans [16]. With the development of more recent models like the GPT models from OpenAI, Gemini from Google and Claude from Anthropic, the reasoning capacity of these models has increased substantially. This allowed these models to be integrated into software systems as the core element of reasoning and decision-making, allowing the system to autonomously plan and manage complex tasks [28][33]. Although single AI agents demonstrate significant problem-solving capacity, they face limitations in planning long-term tasks that require maintaining a large amount of context across multiple reasoning processes. They are also prone to hallucinations that provide plausible but incorrect outputs. This can lead to cascading inaccuracies throughout the reasoning chain, resulting in unreliable outputs [16]. These limitations shifted the paradigm towards Multi-agent systems (MAS) or agentic AI, where agents break down into smaller and more specialized agents. Worker agents interact to break down tasks, distribute responsibilities and handle complex decision-making processes [17]. This breakthrough shifted the AI landscape to a new era of task-oriented frameworks powered by capable LLMs. AI agents are software systems that have a cognitive ability (the LLM model powers the system) to plan and make decisions. In addition, these systems include tools that allow them to take actions in the real world or digital environments [38].

Multi-agent systems rely heavily on their structural design, often referred to as architectural patterns, and interaction models to solve and manage complex long-thinking tasks effectively. The literature has shown a number of distinct coordination patterns. The hierarchical pattern assigns a central orchestrator agent to break down tasks and delegate sub-tasks to a specialized worker agents at multiple levels of delegation [30]. Sequential patterns transfer task output from one agent to the next in a linear fashion [28]. Other patterns include a decentralized pattern or mesh-based design, where agents participate in peer-to-peer debates or voting mechanisms to reach a desired outcome [16][28]. Frameworks such as MetaGPT implement role-based sequential coordination using Standard Operating Procedures (SOPs) for software engineering tasks by applying the "Waterfall" approach, effectively managing the

interactions between agents [17]. AgentCoder implements a three-agent sequential pipeline, achieving 77.4% pass@1 on code generation benchmarks to improve code quality [19]. Although multi-agent frameworks are becoming more widespread, there is still a lack of systematic empirical data available for choosing the most suitable architectural design for a specific software engineering task. The current agent coordination decisions for the performance, reliability, cost trade-offs are often made without empirical data, such as deciding between hierarchical and sequential patterns [28]. This lack of empirical evidence represents an essential gap in the research field, specifically for software engineering tasks such as automated test generation, where reliability and cost consistency are critical for practical consideration.

1.2 Purpose of the Study

This study aims to investigate and compare three architectural patterns that are used in multi-agent LLM-based systems. With a specific focus on their impact on software engineering tasks, especially automated software testing and test case generation. The purpose of the study is to identify and categorize common architectural patterns, such as single-agent, sequential and hierarchical architectures. In addition, analyze the trade-offs associated with these patterns, such as performance, reliability, cost, and efficiency, as identified in existing multi-agent systems studies [16][39][7]. Through an experimental evaluation, the study investigates how different multi-agent architectures behave in creating software test cases relative to simple single-agent baseline techniques. The study aims to determine whether and when multi-agent designs yield measurable benefits over the single-agent techniques by providing the simplest baseline model and then gradually adding the architectural complexity.

Furthermore, the study will provide a generalizable and empirically supported architectural model for designing LLM-based multi-agent systems in software engineering contexts. The model is supported by standardized evaluation criteria and real experimental data, helping future researchers replicate, extend, and validate architectural choices, as well as make well-informed design decisions.

1.3 Significance of the Study

This study addresses a critical gap in the understanding of architectural design in LLM-based multi-agent systems, through a systematic comparison of various architectural patterns in standardized performance and reliability metrics [16] [28]. Although several frameworks such as MetaGPT and AgentCoder have shown an impressive result on specific tasks [17][19], these evaluations focus on the performance of individual frameworks rather than a systematic comparison of the core architectural patterns under specified experimental conditions. In addition, this study provides statistical measures of variability in three repeated experimental runs, such as variance, standard deviation, coefficient of variation, and 95% confidence intervals. This allows for easy evaluation of the average performance as well

as the consistency of each architectural pattern's behavior across tasks.

From a research perspective, the study creates a clear experimental baseline and evaluation framework that makes it possible for future studies to extend, validate, and compare results across different multi-agent system architectures. Evaluation metrics, experimental design, and benchmark tasks are documented to enable future researchers to extend, validate, and compare results across different models, benchmarks, and task types.

From a practical perspective, the findings provide researchers and practitioners quantitative and qualitative data to help them choose architectural patterns for LLM-based multi-agent systems in software engineering tasks. Today's architectural choices, including the decision between hierarchical and sequential coordination, are frequently made without actual data on their cost, performance, and reliability trade-offs[28]. The trade-off analysis offers specific data on the latency, cost, reliability, and error handling metrics of three architectural patterns. This information may help developers of automated test generation pipelines and related software engineering applications make better decisions.

It should be noted that the results of this investigation are specific to the TestEval benchmark tasks and the evaluated conditions. Therefore, these limitations constrain the generalizability of the findings and more research would be needed to generalize them to other types of tasks or deployment situations.

1.4 Research Questions

RQ1: What are the current architectural patterns for multi-agent LLM systems reported in academic and grey literature?
Goal: The goal of RQ1 is to identify and catalog the architectural patterns currently used in LLM-based multi-agent systems through a systematic review of the existing literature. This question establishes the theoretical foundation of the study by describing the many possible coordination patterns, together with their structural features, main advantages, and any disadvantages. The results of RQ1 guide the choice of patterns for the quantitative experiment in RQ2 and offer practitioners and academics creating multi-agent systems for software engineering tasks a systematic resource.

RQ2: How do different architectural patterns compare in terms of performance (latency/cost) and reliability (success rate/error handling)?
Goal: This research question aims to empirically evaluate three architectural patterns, namely sequential, hierarchical, and single-agent baseline, on five measurable metrics: error handling, latency, cost, success rate and agent communication turns. A controlled quantitative experiment that evaluates each pattern on the same set of automated test creation tasks under the same conditions is used to answer this question. The results of RQ2 enable the investigation of coordination techniques for software engineering activities by offering quantitative proof of the performance and

reliability trade-offs connected to each architectural pattern.

RQ3: How suitable are different multi-agent LLM architectural patterns for software engineering tasks, and to what extent can the resulting insights be generalized to other application domains?
Goal: The goal of RQ3 is to analyze the generalization of observed patterns beyond the particular benchmark examined and to evaluate the quantitative results of RQ2 in connection with software engineering task requirements. This question is addressed through qualitative analysis of agent interaction traces across 22 sample tasks for all three experiment runs per task, by looking at how coordination mechanisms occur during the automated test generation. Across the three architectural patterns we found six behavioral categories with a significant architecture-specific distribution. These categories investigate qualitative insights into the reason behind the quantitative differences observed in the RQ2 as well as the conditions in which the pattern is suitable for automated test generation tasks.

1.5 Contribution

This thesis contributes (i) a taxonomy of architectural patterns for multi-agent LLM systems, structured along control flow, role specialisation, and information flow, from which three patterns are selected for empirical comparison; (ii) a controlled empirical comparison of the Single-Agent, Sequential, and Hierarchical patterns on 30 TestEval tasks using Claude Sonnet 4. The experiment produced quantifying results of the cost, latency, and reliability trade-offs of each pattern; (iii) a qualitative account, derived from directed content analysis of 66 interaction traces, of the behavioral mechanisms that explain the observed differences, including the supervisor information bottleneck; and (iv) a set of practical architecture-selection guidelines together with a publicly archived replication package [2].

1.6 Scope

The scope of the study is limited to investigating unit test generation as a task in software engineering prompting a single model (claude sonnet 4), on one benchmark (TestEval), using three different patterns (Single-Agent, Sequential and Hierarchical) without any prompt optimization. Other LLMs and SE tasks are excluded from the scope, such as bug repair, requirements engineering or code review. The results are meant as a controlled reference point, not general architecture principles.

1.7 Thesis Outline

The following table presents an overview of the sections and the structure of the study.

Chapter	Relevance
Background	Provides an overview of the terms used in the study.
Related Work	Reviews the related work and addresses the gap that the study aims to fill.
Methods	describes the methods used to conduct the study.
Results	Gives detailed results of the literature review and the experiment.
Discussion	Presents a discussion of the results and considers practical implications.
Conclusion	Presents the conclusion drawn from the results and the discussion.

2

Background

This section establishes the theoretical foundation of the study. It provides an explanation of the transition from single AI agent systems to agentic AI and multi-agent systems **(MAS)**. By categorizing different architectural patterns proposed in the literature and their application within the software engineering domain.

2.1 From Generative AI to Agentic AI

Generative AI represents a subfield of artificial intelligence that can understand and respond to text, images, audio, video, or any type of data input by interacting with prompts using generative models. The GenAI paradigm contains many different types of models. Foundational Models (FM) and Large Language Models (LLM) are two distinct categories of models. Both types are trained on extensive datasets to perform general interpretations of various input data and to produce content such as text or images as an output. Recent research has seen evolutionary advancement in LLM-based technologies like GitHub Copilot and ChatGPT. These techniques show exceptional performance in producing human-like content, pushing the productivity of their users to a higher limit [31].

Since 2020, Large Language Models (LLMs) have been progressing remarkably in content-generating capabilities, producing large-scale models such as GPT-3, PaLM, and LLaMA consisting of billions of parameters. These models have achieved a high level of cognitive ability, enabling them to perform reasoning and planning. The researchers took these models one step further by applying reasoning, reinforcement learning from human feedback, and prompt engineering. These techniques guide the models towards more goal-oriented objectives with more secure interactions. Leading to the growth of multi-modal models such as GPT-4, combined with both vision and language capabilities. These models eventually became the core of AI agents, empowering these agents with robust tools, contextual comprehension, strategic planning, and self-reflection capabilities. Consequently, LLM-based agents transformed from passive content generators into autonomous, self-planning, and context-aware systems [34].

2.2 Anatomy of Single LLM-based AI Agents

Atomic single-unit agents constitute the core component of MAS. Understanding the construction of atomic agents facilitates a more natural grasp of MAS functionality and collaboration. An agent is technically represented as a tuple {m,o,e,x,y}, including model (m), objective (o), environment (e), perception (x), and action (y) [38].

- The Model (LLM): Is the primary cognitive engine of the agent, and it enables the agent to think beyond the confines of rules and scripts. Having an understanding of the situation, coming up with plans, and making decisions are all under its purview [8, 33].
- The Memory: Because memory is crucial for multi-turn interactions, agents utilize both short-term (context window) and long-term (vector database) memory to maintain state across a workflow. These two different kinds of memory types help the model to comprehend the current environment, capture the context, plan and reason according to the objectives provided to the agent [21, 34].
- The Perception & The Action: The ability to perceive inputs from the environment (e.g., code repositories, user requirements) and execute actions via external tools (e.g., Python interpreters, compilers) [8].

2.3 Taxonomy of Architectural Patterns in Multi-Agent Systems

The efficiency of multi-agent systems is significantly related to their architectural topology. These systems are constructed using established software architecture patterns, which define their control structures and communication protocols within the MAS and among the agents. The literature review identified various patterns that serve different use cases. However, three patterns stood out for us, and this section will discuss how these patterns enhance the resilience of MAS and what potential trade-offs engineers encounter when implementing them.

2.3.1 Sequential and Chain Architectures

This typology resembles the "waterfall" method in software engineering; it follows a strict order of phases (requirements -> design -> coding -> testing). As a **static architecture** in which the output of one agent serves as the input of the next agent, creating a pipeline of sequential agents that respond to the results of the preceding agent's operations [38]. Agents get assigned roles to play and equipped with tools to perform actions depending on the set of skills that they got provided with. Where each agent has enough autonomy to react in its environment depending on the task that the agent gets assigned.

2.3.1.1 The Role of Standard Operating Procedures (SOPs)

The defining characteristic of this pattern is the encoding of Standard Operating Procedures (SOPs) into the agent workflow. This approach restricts the information flow between agents, limiting any "idle chatter" or hallucinations that occur, and forces agents to follow a strict order [7]. Different frameworks fulfill distinct roles within this pattern; however, the top-down approach continues to be the baseline of this typology. In the MetaGPT framework, the chain starts with a requirement specialist agent that generates a comprehensive set of requirements, which are then passed on to a system designer. This architecture thoroughly designs the system to comply with the requirements provided in the preceding phase. The system designer produces architecture diagrams and data flow charts to be delivered to an engineer agent later. These artifacts will serve as a baseline for the engineer to code, test, and validate while developing the system [17]. AgentMesh similarly implements the same approach: a planner generates a list of sub-tasks, where a coder starts implementing these sub-tasks sequentially. Finally, a reviewer validates the system to match what the planner originally created [21]. These frameworks effectively "hardcode" the software development life cycle into their workflow, thereby improving the outcomes of the sequential approach of MAS.

2.3.1.2 Artifact-Centric Communication

Artifacts-centric communication has initiated a transition in the modern sequential systems from conversational dialogues to structured shared deliverables. In MetaGPT, agents communicate using artifacts, flowcharts, and interface specifications instead of unstructured dialogue. This structured communication minimizes ambiguity and hallucinations of large language models (LLMs) by limiting their output space [17]. A shared memory referred to as a "message pool" retains agents' published artifacts such as a PRD or a Python class file. This design enables downstream agents to subscribe to the specific artifacts relevant to their roles, hence facilitating the linear handoff without information loss.

2.3.1.3 The Linear "Chain of Thought" at System Level

This design can be considered as a system-level implementation of the Chain-of-Thought (CoT) prompting technique. It decomposes complex engineering tasks into roles for different agents, converting vague user requirements (e.g., "Write a Snake game") into smaller, simpler steps. In AgentMesh, for example, the Planner agent generates a list of executable sub-tasks (e.g., "Design Data Structures" and "Implement Logic"). This decomposition decreases the cognitive burden on the coder agent and allows following agents to concentrate on their respective subtasks. Thus, it will minimize ambiguity within the system [21].

2.3.1.4 Trade-offs: Efficiency vs. Error Propagation

While this pattern is effective in terms of token utilization and latency due to the limitation of circular debates and minimization of redundant communication between agents. Furthermore, it provides a consistent workflow. The project state's

transparency (e.g., "We are currently at the testing stage") guarantees that progress can be easily tracked at every level. However, the primary vulnerability of sequential chains is error propagation. A single undetected hallucinated requirement at the top level can typically trigger a cascading failure throughout the system. Resulting in fundamental flaws once they pass the initial agents. Furthermore, fixed sequential chains lack flexibility. The system cannot implement dynamic changes once the sequence starts, leading to complexities if a task requires a domain expert not predefined in the chain to be recruited.

2.3.2 Hierarchical (Centralized) Architectures

Hierarchical and centralized architectures organize agents into hierarchical layers of authority. Within these structures, agents are assigned specific roles with distinct responsibilities. This design converts the system from a linear chain to a supervised ecosystem. Centralized hierarchical systems consist of a "Supervisor" who manages the decision-making and task orchestration of subordinate "Worker" agents throughout their life cycle. This typology is well suited to multi-stage software engineering tasks that require high reasoning and domain expertise to implement requirements concurrently [5, 33, 38].

2.3.2.1 The Orchestrator-Worker Paradigm and Role Specialization

The core element of a hierarchical system is the central orchestrator referred to as the "Supervisor". This agent receives user inputs and then transforms them into decomposable goals. Where these goals are broken down into smaller tasks delegated to specialized "Worker" agents [5]. Frameworks typically implement hierarchical systems via persona assignments. In this approach, different agents are responsible for different components of the system. For example, an agent-as-a-planner formulates an overarching strategy, while an agent-as-an-engineer allocates tasks to different sub-agents to distribute the workload. Furthermore, an agent-as-a-worker executes domain-specific logic (e.g., coding or testing) [28]. In systems such as ChatDev, a "virtual CEO" or meta-agent assigns sub-tasks to departmental agents and integrates their outputs into a cohesive final goal. Before sending the result back to the top-level supervisor agent, each sub-agent usually executes its own loop of perception, reasoning, and action [5]. Hierarchical systems also depend on structured communication protocols and information flow. These systems often exhibit a bidirectional information flow, employing a top-down approach strategy to convey sub-goals and global constraints from "Manager" agents down to "Worker" agents. The status reports and outcomes are transmitted from "Worker" agents up to "Manager" agents in a bottom-up manner. This prevents top-level agents from becoming overwhelmed by managing micro-level execution details, allowing them to retain awareness of the overall state [33].

2.3.2.2 Trade-offs: Global Coherence vs. Bottlenecks

Hierarchical systems are globally consistent and highly scalable given that each sub-agent has a specific role, is clearly responsible for its work, and can only do their job

inside rigid constraints in the hierarchy tree. Moreover, the "Supervisor" provides an additional layer of safeguard, allowing the orchestrator to modify and adapt the plans to resolve conflicts or failures as they arise [5, 28]. However, the primary feature of these systems also becomes the main constraint if the "Supervisor" misinterprets the original prompt. This constraint leads to hallucinations that may occur in the task decomposition, affecting the entire downstream workflow. Furthermore, deep hierarchical layers must exchange summaries and instructions among themselves across the network. Consequently, it results in significant communication overhead and higher token expenses [30, 38].

2.3.3 Decentralized (Mesh) Architectures

Decentralized architectures allow agents to communicate directly over communication channels without a central orchestrator, forming a mesh network. These communication channels show up as two unique dynamic techniques in which agents engage in debate or cast votes when opposing viewpoints arise. Through these processes, strive to find a consensus and formulate solutions based on optimal outcomes [7, 38]. With these systems demonstrating high fault tolerance and creativity, they often have a lot of trouble with high communication volumes and struggle to reach a final conclusion due to the absence of consensus mechanisms. Agents may engage in infinite cycles of conversation without human intervention, resulting in excessive token usage and prolonged sessions of "ReAct" events that limit their performance [38].

2.3.4 Summary of Differences

Feature	**Sequential (Chain)**	**Hierarchical (Tree)**	**Decentralized (Mesh)**
Control	Fixed/Static Flow	Centralized / Top-Down	Distributed / Peer-to-Peer
Info Flow	A $\rightarrow$ B $\rightarrow$ C	Hub $\leftrightarrow$ Spoke	Any $\leftrightarrow$ Any
Primary Risk	Error Propagation	Bottleneck / Single Point of Failure	Infinite Loops / Consensus Failure
Best For	SOP-based tasks (e.g., Documentation)	Complex Task Orchestration	Creative/Open-ended Reasoning

Table 2.1: Comparison of Three Architectural Patterns in LLM-Based Multi-Agent Systems

Table 2.1 summarizes the key structural differences between the three primary coordination patterns identified in the literature. The sequential pattern is suitable for structured, SOP-based tasks because it uses a fixed linear information flow in which each agent delivers its output directly to the next. However, it is sensitive to error

transfer when an upstream agent generates incorrect output. The hierarchical pattern centralizes control through a top-down delegation structure, where a supervisor coordinates all inter-agent communication and enabling complex task coordination. But it introduces a single point of failure at the supervisor level. The decentralized pattern shares control among agents in a peer-to-peer mesh, enabling creative or open-ended reasoning tasks and enabling flexible communication between any agents. However, it carries the potential of infinite delegation loops without proper termination conditions or majority failure. The performance and reliability trade-offs investigated in the experiment described in the results section in Phase 2 are directly shaped by these changes in structure in control and information flow.

3
Related Work

In recent years, research on multi-agent LLM-based systems has quickly grown with the increasing use of the general AI-based model to perform reasoning, planning, and complex decision-making. Early LLM-based systems mostly used single-agent architectures, in which a single model instance handles task interpretation, reasoning, and execution via prompt-based interactions. These systems have inherent limitations in terms of scalability, dependability, long-term thinking, and complex task coordination while being effective for small tasks. Therefore, recent research has switched to multi-agent LLM systems, where many agents collaborate to break down tasks, distribute responsibilities, and efficiently manage the process of decision-making.

3.1 Baseline Single Agent

Determining what a single LLM-based agent can and cannot perform on its own is an important requirement for realistic evaluation of multi-agent interaction. This has been formalized by Sapkota et al. [33], who provides a theoretical taxonomy that differentiates between reactive AI agents and agentic AI [33]. This difference is essential for our research since it guarantees that the evaluated designs reflect true multi-agent coordination patterns and defines the baseline capabilities of a single LLM agent.

A single LLM agent can perform a range of tasks on its own without any assistance, which provides a foundational baseline. This approach is often suggested in early LLM applications, depending on a single process for simple and direct control. Studies such as Guo et al. and Yan recommend single-agent setups as a starting point before multi-agent upgrades [16][43]. They highlight their applicability in simple tasks such as code creation or text summarization. Independent LLMs (e.g., GPT models) perform isolated analysis but lack the distributed capabilities of multi-agent systems. This pattern is highly efficient in low-complexity scenarios with minimal costs, as there are no communication delays or coordination costs [16][25]. However, it trades scalability and resilience, as a single point of failure might halt the progress in dynamic environments.

Furthermore, Yao et al. provide a strong foundation for agent capabilities through the ReAct pattern [44]. This study shows how a single agent might combine rea-

soning and action to improve performance within complex tasks. ReAct suggests that well-designed single-agent systems may manage complex sustained tasks without the cost of multi-agent coordination by combining task-specific action with a chain-of-thought reasoning.

Practically, a few studies show that single-agent systems may perform better than multi-agent structures in simple tasks, while collaboration overhead in multi-agent systems can introduce errors or inefficiencies [25][26][24]. For example, Li et al. show that MAAD's multi-agent framework performs better than MetaGPT's single-agent structure in architectural design quality. However, single-agent systems still show significant effectiveness in tasks that support individual reasoning[36][23].

LLM-based agents used for software engineering tasks like code generation, maintenance, and testing are surveyed by Liu et al. and Dong et al. [28][9]. They generally find that single-agent systems perform well on well-defined, short-horizon tasks but break down on challenges requiring iterative improvement, long context, or parallel subtask execution. The limitation of a single agent motivates the architecture improvement, which is reviewed in the next section.

3.2 Architecture Patterns in Multi-agent System

To get beyond single-agent limitations, architectural patterns have developed, and they are documented in a growing body of research. For foundation-model-based agents, Liu et al. offer the most comprehensive pattern collection currently available, providing designs for memory management, role specialization, and task division [28]. Moore focused on hierarchical patterns, differentiating between orchestrator-worker systems, layered structures, and hybrid structures, and adapting these to real coordination mechanisms [30]. Bandara adds industrial engineering guidance to these theoretical catalogs, while Fourney et al. suggest Magentic-One for production-grade designs. Magentic-One is a good example of a hierarchical/orchestrator-based pattern that emphasizes modularity and domain-wide generalizability [4][10]. Hong et al. present the MetaGPT framework with the implementation of a hierarchical and role specialization pattern based on software engineering tasks. The framework uses (SOPs) to reduce logic mistakes in complex tasks and achieve 81.7% pass@1 on HumanEval in code generation benchmarks [17]. As an alternative to hierarchical orchestration, AutoGen is an open-source framework that allows developers to build LLM applications using multiple conversable agents and conversation programming [42]. This framework is widely used in both industrial and academic applications and allows for adaptable agent building. In their review of improvement and difficulties in LLM-based multi-agent systems, Guo et al. demonstrated coordination and communication for collaborative tasks while pointing out gaps in common pattern evaluations [16].

A generalizable software architecture model for LLM-based multi-agent systems (MAS) is the goal of our thesis. Yan et al. build on this by taking a communication-centric approach to interaction protocols and showing common patterns across the

existing literature [43]. Liu et al. extend it to software engineering, pointing out trade-offs between scalability and flexibility [25].

There is a framework for building dynamic multi-agent systems with a specific flow of control graphs provided by LangGraph [22]. LangGraph enables the implementation of many different kinds of design coordination, such as sequential pipelines and hierarchical decentralization. Because of this method, the developers can describe complex agent interaction while keeping the control and accessibility. AgentCoder implements a three-agent sequential pipeline (programmer, test designer, and test executor), where they achieve 77.4% pass@1 on humaneval[19]. While a linear model reduces coordination cost, there is an opportunity that errors will spread along the pipeline. AgentMesh shows collaborative network patterns for code review tasks, where agents share information in order to improve defect detection that increases the token usage [21]. While MAAD and NOMAAD apply knowledge-driven sequential patterns for automated architecture design and UML generation [24][13]. Role-based coordination can be used in practical development workflow as shown by Riberiro et al., who use multi-agent techniques to present effective software development [32].

Considering the growth of multi-agent frameworks, comprehensive analysis of architecture patterns using common metrics remains a limitation. Guo et al. investigated three coordinating frameworks (centralized orchestration, decentralized negotiation, and market-based allocation). Furthermore, they discovered that each framework has different latency, cost, and fault tolerance specifications [16]. Yan et al. suggest that inner-agent communication structure is the essential motivator of system-level performance [43]. This has been confirmed by Leong et al., showing that dynamic communication topology decentralization improves task-completion efficiency over static structures [23].

3.3 Architecture Trade-offs

There are several sources of empirical evidence about trade-offs. Jia et al. use a dynamic environment acting module with an error-feedback mechanism. They demonstrate that feedback-driven multi-agent architectures with error-correction mechanisms achieve 93-96% success rates on power system simulation tasks, substantially outperforming single-agent baselines with less than 30% success rates. While maintaining rapid execution (30 seconds per task). Implying that effective and reliable coordination could be improved[20]. The AnyMac framework shows the dynamic next-agent prediction enables flexible trade-offs between accuracy and token cost. Showing that the efficiency variation reduces token usage by five times with only a small decrease in accuracy, whereas the standard version achieves the best accuracy at a higher token cost [39]. This highlights a basic conflict, while architectural complexity might lead to performance improvements, the connection between complexity and outcomes is not linear.

Tasks in software engineering provide an example of the trade-off patterns, whereas

Yang et al. examine how design choices impact quality characteristics like maintainability and scalability [7]. However, reliability benefits in test generation tasks such as unit-test coverage are those shown by Garlapati et al. and Tomic et al., who also find that LLM model choice affects multi-agent GUI test generation performance [11] [37]. These findings suggest that architecture pattern selection must be determined by a task characteristic such as iterative improvement, parallel sub-task execution, or long-term reasoning.

3.4 Research Gaps

The reviewed literature has discussed a wide range of architecture patterns in multi-agent systems, where it evaluates specific frameworks such as MetaGPT with a baseline. Focusing more on the general performance evaluation results than on which architectural characteristics matter for which specific type of task. There is a lack of a systematic plan for describing why different architectural decisions result in specific trade-offs, as well as how to choose which design is best suited for each task. Furthermore, the literature does not investigate in-depth analysis of the explanation of the models regarding the reasons behind particular design decisions. These decisions lead to architectural properties such as control, flexibility, and the loop structure and how these characteristics interact to deliver the observed trade-offs in the system's effectiveness. Additionally, there is no generalizable and reusable framework that allows different users to successfully match task characteristics to the most suitable architecture decisions in software engineering fields.

4

Methods

This chapter outlines the research strategy and methods used to systematically evaluate and analyze the architectural patterns of multi-agent large language model (LLM) systems automated unit test generation.

4.1 Research Design Overview

The methodology is a mixed-method approach, in which a primary empirical study is combined with a secondary literature review. Research approaches are classified in the ABC Framework of Software Engineering Research [35] based on their trade-offs between the generalization over actors (A), the realism of the setting (B) and the accuracy of measuring behavior (C). According to the ABC paper, different research questions demand different approaches, each with its own advantages and disadvantages.

Since this thesis seeks to quantitatively assess and measure the exact performance trade-offs (token cost, latency) and reliability characteristics (success rate, test coverage and error handling) across architecture patterns, it adopts the *laboratory experiment* technique to answer the primary empirical question. The experiment is conducted in a controlled environment specifically set up for the study, seeking to get the most accurate measurement of behavior. By evaluating these multi-agents on standardized benchmarks such as TestEval, rather than live production code, the experiment intentionally eliminates contextual realism to enhance the control over the independent variables (the architectures) and accurately measure the dependent variables (the trade-offs) without confounding real-world factors.

The study is organized in two phases: Phase 1 produces a taxonomy of architectural patterns (RQ1) through a structured literature review, and Phase 2 conducts a controlled laboratory experiment that addresses RQ2 (quantitative comparison) and RQ3 (qualitative analysis of agent interaction traces).

4.2 Phase 1: Formal Theory

This phase highlights knowledge fragmentation issues in software engineering research, as indicated by Glass et al., [14], which uses a formal theory creation through

narrative literature reviews. Glass et al. observed that progress in theory is limited when previous work is not systematically consolidated. The process of synthesis is important in the growing field of LLM-based multi-agent systems, where architectural knowledge exists across multiple academic publications and different industry implementations. This phase offers a constructive taxonomy of architecture patterns for LLM-based systems and determines their design dimensions and performance trade-offs. This phase provides theoretical foundations by analyzing documented architectures through a structured literature review to identify gathered patterns that are recurring trends to address RQ1.

4.2.1 Data Collection

The research data has been collected from academic digital libraries such as Google Scholar, ArXiv, IEEE Xplore, ACM Digital Library, and ResearchGate. Combined with some high-quality gray literature, defined here as technical reports and engineering blogs from established AI organizations (e.g., OpenAI, Microsoft Research, LangChain, and Anthropic), excluding personal opinion blogs [12]. To ensure feasibility, the search is constrained to publications on a tight time span (2023–2026), when LLM-based multi-agent systems become a distinct research topic. For the search strategy, the search strings use keywords such as "LLM Multi-Agent Architecture", "Agent Design Patterns", and "Generative AI Orchestration". The following criteria are used as follows: (1) Describe multi-agent architectures based on LLMs. (2) Explain the design and communication method decisions. (3) Describe the trade-offs and performance characteristics. For each included source, the following data items were extracted: architecture pattern category, agent roles and responsibilities, inter-agent collaboration mechanism, task decomposition strategy, and reported performance characteristics.

4.2.2 Data Analysis

Thematic analysis is used to combine the extracted data into a taxonomy of architecture patterns [6].

4.2.3 Coding Process and Validation

Before coding, pattern definitions were created by combining descriptions from the reviewed literature, using only definitions that were similar across various publications. This minimized the possibility of biased identification by defining each pattern description in common with the public's understanding rather than individual opinion. Each of the 31 publications was independently examined and summarized by two researchers who documented the architectural characteristics and coordinating techniques mentioned in each source. Each paper was assigned a pattern label only if it described using that pattern in its methodology, not only in the background or related work section. In order to represent their true architectural structures, the papers that used different coordinating mechanisms were presented with multiple pattern labels. After initial labeling, a paper was reread to ensure that assigned patterns showed up in the system's actual implementation rather than just in its

description. It is acknowledged that the coding was performed by a single researcher without the assistance of a second independent coder. This highlighted that there is no inter-rater reliability, which represents a limitation of our analysis. To mitigate this limitation, the pattern definitions were simply documented before the coding started and a second review process was performed to ensure consistency between all 31 articles.

The analysis involves multiple steps as follows:

1. **Familiarization with the data:** Reading the extracted information to gain a more comprehensive understanding of the architectural context including the most common structures and how design decisions are documented. During the initial review of the 31 selected studies, across systems like MetaGPT, AgentCoder, and Magentic-One, recurrent coordinating techniques including Sequential/Pipeline workflows, Centralized/Supervisor orchestration, and Hierarchical Multi-layer delegation were often observed.

2. **Initial coding:** Each source was carefully processed to determine its architecture properties. Codes captured coordination strategies such as centralized decision-making and the structural features of sequential transitions. For example, Magentic-One was coded as "Centralized/Supervisor" because a central orchestrator assigns tasks to worker agents. Whereas MetaGPT was coded as "Sequential/Pipeline" because tasks pass through structured phases (Requirements → Design → Code → Test).

3. **Pattern identification:** Identifying the architecture patterns and grouping similar architectures in order to find patterns. For example, a hierarchical structure is demonstrated by grouping the patterns that implement supervisor-agent, worker delegation, or centralized coordination strategy. For example, The "Feedback-driven/Self-optimise" pattern was applied to systems that used iterative corrective loops and refinement cycles. While the "Tool-augmented agent" design was applied to systems that used external APIs, code interpreters, or search tools.

4. **Pattern refinement:** Candidate patterns are compared back against the original source to ensure consistency and uniqueness. For example, in initial analysis we grouped Hierarchical Multi-layer and Centralized/Supervisor architectures together because both rely on orchestrator-based coordination. However, more analysis discovered that centralized/supervisor systems rely on a single orchestration layer, whereas hierarchical structures include several delegation levels (high-level supervisors, mid-level coordinators, and worker agents). As a result, distinct patterns were developed from them.

5. **Reviewing and expanding the patterns' categories:** Each retained pattern is characterized along three structural dimensions: control mechanism (process-driven vs. agent-autonomy), loop structure (single path, iterative refinement, nested loops), and coordination type (centralized, decentralized, hi-

erarchical). For example, Blackboard/Shared-memory was classified as decentralized coordination with an asynchronous communication, Sequential/Pipeline as process-driven with single-path execution, and Feedback-driven/Self-optimize as iterative refinement because of repeated evaluation loops.

6. **Creating the final taxonomy:** This analysis taxonomy is finalized by identifying each pattern's structural characteristics, common agent roles, the communication flows between agents, and reported trade-offs. The final taxonomy identified 10 architecture patterns.

The resulting taxonomy is presented in Chapter 5, Section 5.1, and is the basis on which the three patterns evaluated in Phase 2 were selected.

4.3 Phase 2: Empirical Evaluation

This phase adopts a laboratory experiment approach to evaluate specified architectural patterns on software engineering test generation activities [35]. This controlled setting enables specific evaluation for performance measurements such as success rate, token usage, and latency, as well as distinguishing between architecture impacts with quantitative comparisons.

4.3.1 Pattern Selection and Implementation

We selected a single-agent baseline to function as our independent variable, alongside two separate multi-agent systems based on the taxonomy developed in Phase 1 and various architectures identified in the literature review.

1. **Single-Agent Baseline (Control):** This technique uses a single large language model to implement a standard prompt or chain-of-thought method [28]. The agent is responsible for processing the requirements and interpreting the code to generate tests in a single iteration, without any peer review or any role responsibilities. This method functions as the control group to assess the foundational agentic trade-off.

2. **Sequential (Chain) Architecture:** This architecture applies a linear artifact-centric workflow to execute (SOPs) topology inspired by frameworks like MetaGPT [17]. The information is passed through specialized roles such as passing requirements from a Planner to a Coder. Agents mostly communicate through organized artifacts rather than unstructured discourse.

3. **Hierarchical (Centralized) Architecture:** This approach follows a top-down methodology, a *Supervisor* agent decomposes the task and assigns subtasks to specialized worker sub-agents (Analyst, Writer). Information flows top-down with the allocated tasks, while status reports or execution results are communicated in a bottom-up direction [38, 30].

4.3.2 Rationale of Pattern Choice

The selection of the three specific configurations—Single-Agent Baseline, Sequential, and Hierarchical is intentional and principled and it is based on how widely they are used in Software Engineering (SE) applications and how they theoretically show the range of multi-agent balance.

4.3.2.1 Prevalence in Software Engineering Tasks

Recent systematic reviews of LLM-based multi-agent systems have identified common patterns that dominate the scene when it comes to automated software development. Software code is the primary artifact generated by LLM-based systems, representing nearly 48% of all executed tasks, while the Role-Based Cooperation design pattern is the most commonly utilized, employed by 46.8% of these systems [7]. By implementing these topologies, our testbed will include various ways of role-based cooperation. The (SOPs) and waterfall workflows are implemented by the sequential linear pattern. While the *Supervisor* role, where a manager communicates sub-tasks to sub-agents in a top-down manner who in return report back to the supervisor in a bottom-up fashion, represents an Agile workflow in a hierarchical pattern. Including the single-agent baseline allows us to quantify the benefit—if any—of adding role-based coordination over a single LLM.

4.3.2.2 Coverage of the Control and Information Flow Spectrum

Selecting these patterns as they in turn represent the entire theoretical spectrum of control distribution and information flow within multi-agent architectures:

- **Single-Agent (None):** No coordination, serves as the essential control group required to quantify the "Agentic Trade-off" determining whether the increased latency and token cost of MAS are justified by improvements in functional correctness.
- **Sequential (Linear):** A linear dependencies (Single-path plan generation) is represented in this spectrum. Communication flows in one direction where it is purely artifact-centric minimizing the risk of error propagation and context loss.
- **Hierarchical (Centralized):** Represents top-down control and vertical information flow, where a manager delegates tasks to workers.

By strictly isolating the Single-agent, Sequential and Hierarchical patterns, this study ensures that any measured differences in cost, latency, and reliability are directly attributable to the system's structural collaboration mechanics.

4.3.3 Experiment Setup

This phase is designed as a *Laboratory Experiment* deliberately structured in a contrived setting. It enhances the accuracy of assessing the agents' behavior and perfor-

mance by eliminating the natural environmental context under controlled conditions. To evaluate the system, we use TestEval benchmarks, a dataset of handwritten programming tasks consisting of specification descriptions and automated unit tests [41][40]. Large Language Models typically generate high-quality code. However, they may also generate insecure or unexpected codes. Therefore, we conducted this experiment in an isolated Local Environment (sandbox) to safely compile and execute the resulting Python code and test cases. The execution yields either deterministic feedback (i.e., whether the code passes all tests or produces syntax/runtime error messages) that is either fed back to the agents for iterative refinement or logged as the final results.

4.3.3.1 Foundation Model

All agents across all three architectural configurations use the same underlying foundation model: **Claude Sonnet 4** [3], accessed through the AWS Bedrock Runtime API in the eu-central-1 region via the boto3 SDK. Routing all LLM calls through a single Bedrock client ensures that every agent in every configuration interacts with the same model endpoint under identical request parameters. Holding the foundation model constant ensures that any observed differences in performance and reliability between architectures can be attributed to the coordination mechanism itself rather than to differences in the cognitive or reasoning capabilities of the underlying model. This control is essential for internal validity: a comparison in which different patterns used different models would conflate architectural and model effects. Furthermore, Claude Sonnet 4 is a mid-priced general-purpose LLM with strong reported performance on code-generation benchmarks, which makes it representative of the class of models that practitioners would realistically deploy in a multi-agent setting. It also supports the structured prompting and conversation-history primitives required by the role-based agents in the sequential and hierarchical configurations. A consequence of using a single model is that the findings reflect the interaction between Claude Sonnet 4 and each architectural pattern; they do not, on their own, generalize to other foundation models. This limitation is discussed in Chapter 7 (Threats to Validity).

The exact inference parameters used for every LLM call, held constant across all agents and architectures. Non-included parameters are held at their default values.

Parameter	**Value**	**Explanation**
Model ID	eu.anthropic.claude-sonnet-4-20250514-v1:0	Bedrock identifier for the Claude Sonnet 4 endpoint.
Temperature	0.2	Low sampling randomness; preserves enough variability for the three-run protocol to capture LLM non-determinism and give the model a small space for creativity since code generation and unit testing is a creative activity.

Parameter	Value	Explanation
maxTokens	4096	Upper bound on tokens generated per response; sized to fit full test suites without truncation.

Table 4.1: Inference parameters held constant across all agents and architectures.

4.3.3.2 Benchmark

The experiment evaluates the systems on the **TestEval** benchmark [41, 40], a curated suite of 210 LeetCode [1] programming problems filtered by cyclomatic complexity >= 10 to bias the dataset toward non-trivial control flow. Each task is a self-contained Python Solution class accompanied by a function description; correctness is checked by executing generated unit tests via pytest with coverage measurement via pytest-cov, without external Docker containers or network dependencies. TestEval was chosen based on three rationales: First, the primary objective of the selected tasks is test-generation research. This benchmark evaluates the test artifacts our architectures are designed to produce. Second, the cyclomatic complexity is high (>= 10), which ensures that the Planner and Analyst are exercised against non-trivial, non-linear problems, eliminating architectural-pattern differences as a confounding factor. Finally, TestEval's task format (a Python class, a function signature, a description) is self-contained and reproducible, allowing pass/fail outcomes to be attributed unambiguously to the generated artifacts.

4.3.4 Experimental Variables

To strengthen our evaluation of the laboratory experiment, we structured our assessment around clearly defined independent, dependent, and controlled variables.

4.3.4.1 Independent Variable

The primary independent variable manipulated during the experiment is the architectural pattern that controls collaboration of the agent. We assess three distinct configurations that demonstrate the range of multi-agent topologies:

- *Single-Agent Baseline (Control):* A standard single LLM setup without role delegation.
- *Sequential (Chain) Architecture:* A linear artifact-centric workflow (e.g., Planner → Coder → Tester).
- *Hierarchical (Centralized) Architecture:* A top-down structure with a "supervisor" that orchestrates subordinate workers.

4.3.4.2 Dependent Variables

These variables demonstrate how the system works and how much it costs to coordinate, which can be used to figure out the trade-offs between agentic factors that

are linked to known software quality attributes:

- *Success Rate/Pass@1 (Reliability):* Proportion of tasks where generated tests execute without errors and correctly validate the target function.
- *Token Consumption (Cost):* Total input and output tokens aggregated across all agents.
- *Latency (Efficiency):* End-to-end time from task initiation to test generation completion.
- *Agent Communication Turns (Coordination Overhead):* Number of inter-agent message exchanges.
- *Error Types (Robustness):* Classification of execution failures (e.g., syntax errors, delegation leaks, timeouts).

4.3.4.3 Controlled Variables

To isolate the dependent variables from any external factors to ensure that the results are strictly the outcome of the architectural patterns, the following controlled variables are held as constants across all experimental runs:

- *Underlying Foundation Model:* The same LLM model (e.g., Claude Sonnet 4) is used to power all agents across architectures. The model provides a uniform baseline of cognitive and reasoning capabilities.
- *LLM Hyper-parameters:* Generation parameters, strictly the *Temperature* (e.g., set to 0.2 for deterministic code generation) and *Top-P*, remain fixed to prevent output variance.
- *Benchmark Input Tasks:* All architectural configurations are evaluated against the exact same set of coding tasks from the TestEval dataset.
- *Maximum Iteration/Retry Limits:* To prevent infinite loops and ensure a fair comparison of latency and token usage, a maximum number of self-correction loops is applied (e.g., a maximum of three iterations) utilizing debugging and reflection retries.

4.3.5 Data Collection

To accurately quantify the data collected and assess the trade-offs of each architectural pattern. The experiment consisted of 3 independent runs, each run executing the TestEval dataset of 30 programming tasks across all the three architecture patterns. In total, this resulted in 270 execution records with 90 records for single-agent, 90 for sequential, and 90 for hierarchical patterns that can be found in our GitHub repository [2]. The experiment is intended to collect and capture specific metrics and interaction logs automatically for each executed, as follows:

- **Functional Correctness (Reliability):** We collect pass/fail execution results from the local sandbox and metrics for each running task in the dataset. For the TestEval data set, we calculate the unbiased **Pass@1** metric, which evaluates the functional accuracy of the code generated on the first execution attempt.

- **Test Coverage (Reliability):** For each of the execution of the tasks, the sandbox environment recorded line coverage (`coverage_line_coverage`), branch coverage (`coverage_branch_coverage`), and the number of tests generated versus passed (`coverage_tests_generated`, `coverage_tests_passed`). These metrics assess whether the generated tests meaningfully exercise the target code beyond more execution success.

- **Resource Utilization (Cost):** We track the exact number of input (prompt) and output (completion) tokens consumed per task across all agents in the network. These metrics will provide an estimate of the direct API cost for tokens and computational overhead while running these architectures.

- **Time Behavior (Latency):** We logged execution timestamps to measure end-to-end task completion time and pre-agent response latency, which reflect how fast or slow different agents behave in different architectural patterns. Also it highlighting the timing characteristic introduced by these patterns.

- **Interaction Logs:** The system captures the complete message history, including intermediate artifacts, tool calls, error traces, and the number of agent turns (messages exchanged) required to reach a solution. These logs form the basis for the qualitative analysis of failure patterns and architectural behaviors.

All were stored in JSONl format, each line representing a single task execution record containing the aggregated metrics and message history for that run.

4.3.6 Data Analysis

The data collected are analyzed using a mixed-methods approach to evaluate the performance and reliability trade-offs related to the research questions [35]. This study employs an explanatory sequential design in which a quantitative analysis was first conducted to establish performance differences, followed by qualitative analysis of the interaction logs to explain and contextualize the observed patterns. This mixed approach involves:

4.3.6.1 Quantitative Analysis

This analysis is to answer RQ2, to determine if there are variations in performance, cost, and reliability between architectural patterns. The analysis was performed in Jupyter Notebook using Python (Pandas, Scipy, Seaborn) on the JSONL dataset, which was loaded into pandas Data-Frame (`raw_df`) for computation. To ensure consistency and eliminate human measurement errors, the evaluation metrics were automatically collected and recorded during the execution of the Python evaluation

scripts. The quantitative analysis proceeded as follows:

1. **Descriptive Statistics:** We determined the mean, standard deviation, and variance for each architectural design for each dependent variable (token usage, execution time, latency, Pass@1, line coverage, branch coverage). The standard deviation was first computed for each of the three independent runs in order to account for LLM output non-determinism. Then the Standard deviation averaged across runs to produce a consistent per-pattern estimate. The square of this averaged standard deviation was used to calculate the variance.

2. **Coefficient of Variation (CV):** To enable comparison across metrics on different scales, we calculated the normalized coefficient of variation for each pattern and metric:

$$CV = \frac{\sigma}{\mu} \times 100 \tag{4.1}$$

   Where σ is the averaged standard deviation and μ is the mean across all runs. This makes cross-metric and cross-pattern comparisons easier by offering a unified measure of relative variability.

3. **Confidence Intervals:** The 95% confidence intervals were computed using the t-distribution ($\alpha = 0.05$) for success rate and test failure rate metrics, where sample means are approximately normally distributed given n = 90 observations per pattern.

4. **Percentile Analysis:** Since latency distributions were found to be right-skewed, median and inter-quartile range(IQR) were used as the key indicators of central tendency and spread instead of mean and standard deviation. To describe the entire latency distribution, including tail behavior, the first quartile (Q1), third quartile (Q3), 95th percentile (p95), and 99th percentile (p99) were computed.

5. **Test Failure Rate:** In order to provide a graduated measure of reliability beyond the binary succeed rate, a test failure rate metric was calculated as the number of tests failed to tests created per task.

$$\frac{\{coverage\_tests\_generated\} - \{coverage\_tests\_passed\}}{\{coverage\_tests\_generated\}} \tag{4.2}$$

6. **Distribution Analysis:** The distribution plots for each metics across architecture patterns were generated used Kernel Density Estimation(KDE). These plots visualized the shape and spread of each measure which highlight the variation in skewed behavior between single-agent and multi-agent systems.

7. **Comparative Analysis:** To compare the mean token usage, total execution time and latency, and one-attempt (Pass@1) success rates among the three independent architectural patterns.

4.3.6.2 Qualitative Analysis

The analysis for answering RQ3. To recognize the reasons behind patterns' different performance characteristics and evaluate generalizability beyond test generation. We conducted an examination to interpret agent interaction logs by comprehending the logs and examining the interactions of sub-agents, we seeked to detect certain patterns of failures inherent to specific architectures. This qualitative insight was used to construct a practical guideline on selecting the optimal architectural pattern based on specific software engineering task constraints.

4.3.6.2.1 Direct Content Analysis The qualitative analysis was conducted using one of three methods mentioned by Hsieh and Shannon [18]. The *Direct Content Analysis* is an approach to qualitatively analyze content in research. Researchers derive an initial coding frame based on prior research and literature and then applies and refines it against the data.

4.3.6.2.2 Unit of Analysis and Sampling We sampled 22 tasks for each architecture and our interaction trace was: one task -> one architecture -> one run, comprising the full message history, intermediate artifacts, tool calls, error trace, and final outcome for a single task execution. For each of the 22 sampled tasks, the interaction trace from Run 1 of each architecture was used as the primary coding source, yielding 22 x 3 = 66 coded cases. Run 1 was chosen as the primary source rather than coding all three runs per task for tractability and because the cross-run consistency analysis in RQ2 demonstrated that, where architectures behave consistently, a single run is representative.

4.3.6.2.3 A Prior Coding Frame Before coding began, an initial coding table comprising various codes and categories was developed based on the Phase 1 literature review. This table served as the coder's primary reference framework, allowing the coder to systematically examine each trace and assess whether a category was applicable. Each category was applied using a structural definition based on observable features of the trace.

4.3.6.2.4 Coding Procedure The 66 cases were coded in a five-sheet spreadsheet that linked each case to its trace excerpts, applied codes, and evidence. The coding procedure for each case followed four steps:

1. **Load and segment the trace:** Collect agent turns, communicated artifacts, retries, and the final outcome.

2. **Apply category definitions:** For each trace, we applied the (initially seven) categories, evaluated the definition of each category against the trace, and marked its presence or absence.

3. **Record evidence and notes:** For each categorized trace, we recorded evidence, notes, and a brief justification of the assigned code.

Category	Operational definition (structural features in the trace)	Literature provenance
Iterative Self-Correction	The agent examines the output, detects any failures, and then attempts to trace the algorithm's logic by a set of retries.	Self-reflection / self-refine patterns [28, 19]
Plan-Guided Generation	A dedicated plan generated with expected outputs and code paths before any code generation happens.	Single-path plan generator pattern [28, 21]
Productive Specialization	Role separation of agents produces remarkably better reasoning, where one agent's reasoning improves another's generated tests.	Role-based cooperation [28, 17, 7]
Supervisor Information Bottleneck	Where a supervisor mediates all communication between agents, the supervisor fails to reliably transform this communication between internal agents, leading to repeated unproductive re-delegation.	Anticipated cost of centralized orchestration in HMAS [30, 7]
Assertion Hallucination	The agent generates plausible plans, reasoning and values but they do not correspond to the actual behavior of the function under test.	incorrect-output behavior observed in LLM code generation [25, 33]
Error Propagation	A downstream agent consumes a previous error produced by an earlier agent in the loop without correction, affecting the final output.	Error-propagation risk in pipeline architectures [28, 38]
Coordination Overhead*	Unnecessary communications between inter-agents exchanging information that is unrelated to the task at hand, traceable, and independent of other failure modes.	Coordination cost in HMAS [7, 30, 38]

* Discarded during frame refinement after empirical frequency was found to be zero across all 66 coded cases.

Table 4.2: *A prior* coding frame for direct content analysis (RQ3)

4. **Iterate:** Where ambiguity arose, we discussed and refined the category definitions until we reached a shared understanding of their meaning and application.

4.4 Use of Generative AI Tools

In accordance with Chalmers' regulations on the use of AI tools in thesis work, we disclose the following. Generative AI tools were used during the preparation of this thesis as follows: ChatGPT / Claude was used to suggest language and grammar improvements on author-written drafts and Claude Code assisted with portions of the experimental pipeline, the evaluation scripts implemented in Jupyter notebooks and analysis code, which the authors reviewed and tested. All AI-generated suggestions were critically reviewed, edited, and verified by the authors, who take full responsibility for the content, the correctness of all results, and the conclusions drawn. Where AI-generated content was incorporated directly, it is cited accordingly.

5

Results

This section outlines our results based on a systematic review of the literature and controlled experiments designed to address the research questions stated in the introduction. We started by defining the classification of architectural patterns used in multi-agent systems, followed by presenting the empirical data and trade-offs related to the selected architectures implemented in the experiment, along with their effects on software engineering activities such as test generation.

5.1 Phase 1: Architectural Pattern Taxonomy (RQ1)

Rank	Pattern Name	Description	Key Benefits	Primary Trade-offs & Risks
1	**Role-based /SOP-driven** [17] [24] [4] [26] [21]	Agents follow Standard Operating Procedures (SOPs) and are assigned specific professional tasks, such as Product Manager, Architect, and Engineer. Roles include responsibility, expected output, and communication guidelines.	SOPs limit uncertainty by using structured, verified final results based on human team processes and leveraging domain expertise. The clear division of responsibility improves transparency and quality.	Flexibility and quality of the output to new tasks is limited by fixed role constraints and Agents might not possess the deep understanding of the topic required for this role.

Rank	Pattern Name	Description	Key Benefits	Primary Trade-off & Risks
2	**Sequential /Pipeline (Waterfall)** [**17**] [**26**] [**19**] [**4**] [**32**]	Each agent's output becomes the input of the subsequent agent in a linear order. Tasks go through multiple phases (Requirements → Design → Code → Test). A common pattern in multi-agent software engineering frameworks, such as MetaGPT.	Clear task progression, ease of implementation, and reasoning Before moving on, each step may be independently verified. Accountability is built into every step.	Latency is the sum of the latencies of all stages, which means nothing runs in parallel and the system pays the full latency cost of each step. Early errors propagate and worsen over time, making the system inflexible to changing or repetitive task demands.
3	**Centralized /Supervisor** [**28**] [**16**] [**30**] [**10**] [**42**] [**22**]	The orchestrator(supervisor) agent breaks down tasks, allocates sub-tasks to specialized worker agents, and collects outcomes. This primary agent coordinates all activities.	Responsibility and command are clearly defined throughout every step. Also, having a single point of monitoring and logging makes debugging easier and one planner ensures global task consistency.	A single point of failure means that if the orchestrator breaks down, the entire system stops working. Consequently, this creates a scalability bottleneck, as the supervisor can become overloaded. Additionally, flexibility is limited when issues affect the supervisor's structure.

Rank	Pattern Name	Description	Key Benefits	Primary Trade-off & Risks
4	**Human-in-the-loop [28] [26] [33] [23] [42]**	The agent process includes human monitoring at specific phases. Before moving forward, humans approve, correct, or modify agent decisions before next stage start. It is essential for critical software engineering and decision-making processes.	Keeps people accountable for important decisions, detects agent mistakes before they affect subsequent tasks and increases confidence in AI systems by providing clear control.	Only when a human reviewer is available can the pipeline move forward, which limits the system's practical scalability.
5	**Retrieval-Augmented Generation (RAG) [28] [9] [42] [24] [22]**	Before providing responses, agents retrieve essential information by requesting an external knowledge base (such as a model store, document data set, or graph database). Builds the LLM on current relevant information outside of the set of training data.	It reduces perceptions by depending on validated outside information and it is feasible to update knowledge without retraining the model.	Increases the delay caused by context loading and the retrieving stage and its difficult to mange the Large knowledge bases.
6	**Tool-augmented agent [10] [44] [28] [42] [27]**	Agents contain a catalog of external tools (file systems, online search, code interpreters, and APIs) that they may use on their own to do tasks. For each sub-tasks, the agent assesses its requirements and selects the most appropriate tool from the catalog to complete it.	This increases agent capabilities beyond the creation of core LLMs and enables verified, grounded activities, such as executing code.	Unlimited tool access provides a security risk (e.g., file deletion, online browsing). Errors in tool selection might result in unsafe or incorrect actions.

Rank	Pattern Name	Description	Key Benefits	Primary Trade-off & Risks
7	**Hierarchical Multi-layer** [29] [30] [16] [21] [22]	Multiple stages structure in which low-level worker agents are directed by mid-level coordinators under the supervision of high-level supervisors. Tactical and operational choices are delegated down, while strategic decisions are taken at the top.	Scalable where every layer's parallelism lowers total latency, distinguishing between strategic and operational thinking and Fault limitation were one sub-tree fails, it doesn't affect the others.	Can lead to rigid and nonflexible structures; performance mostly relies on the quality of the role definitions.
8	**Feedback-driven /Self-optimize** [19] [28] [42] [11] [23]	Agents generate outputs, get feedback (from people, tests, execution, or other agents), and then iteratively improve their outputs. Which is focuses on improving quality and perceptions.	Regular increasing the quality of the output through iterative modifications and can be totally automated using test cases as a feedback messages.	If it did not have a specific condition, there is a possible that will not terminate. Also it have high cost computation because of many repetitive executions of the model.
9	**Blackboard /Shared memory** [24] [38] [16] [22] [21]	A central shared data store, often called a blackboard, keeps track of the current problem state and any intermediate results. Agents can read from and write to this shared space, but they don't need direct point-to-point communication with one another.	It helps with decouple the agents, since each agent needs to understand the blackboard interface. It support the asynchronous operation where allow agents to processed as soon as the needed data become available.	It can lead to conflict if the multiple agents try to write at the same time. And the blackboard may become a bottleneck or even a single point of failure.

Rank	Pattern Name	Description	Key Benefits	Primary Trade-off & Risks
10	**Knowledge-driven** [**24**] [**28**] [**13**] [**16**] [**9**]	Agents are integrated with domain-specific knowledge graphs, ontologies, or structured knowledge bases that assist in decision-making, breaking down tasks, and designing architectures.	Domain knowledge offers systematic and principled guidance for agent decisions, reducing ambiguity and enhancing the consistency of architectural results.	Creating these knowledge bases takes a lot of time and is different for each domain. knowledge can quickly become outdated, so the bases must be updated regularly as the domain changes.

Table 5.1: Top 10 Architectural Patterns in LLM-Based Multi-Agent Systems

Systematic thematic analysis of 31 publications published between 2023 and 2026 identified 10 architecture patterns for multi-agent systems. Out of the 10 patterns, we chose sequential and hierarchical as two multi-agent architectures to compare and chose the single-agent as baseline for our experiment comparison.

1- Single-agent: It is not one of the 10 patterns, it is a baseline which is very essential for the control condition for the experiment. By comparing the multi-agent architectures with single-agent baseline. Some of the studies such as AgentCoder, AutoGen and Dong et al., demonstrated that the higher coordination cost can be justified by better performance [19, 42, 9]. It would be hard to evaluate the advantages of multi-agent interaction experimentally without this baseline.

2- Sequential pattern: The reason for choosing sequential patterns is because the most common implementation in software engineering tasks in multi-agent systems such as MetaGPT, AgentMesh, AgentCoder, and NOMAD [17, 21, 19, 13]. It is simple to execute in a controlled experimental setting because of its clear task boundaries and agents passing tasks in a specific order. Its also an example of the simplest kind of multi-agent coordination, where each agent manages a separate phase in sequence.

3- Hierarchical pattern: Hierarchical was selected because it represents a fundamentally different coordination strategy than Sequential, while ranking seventh by citation count in the literature review. Papers such as Moore at al., Magentic-One, and Dong et al. show that hierarchical coordination introduces centralized task distribution, multiple control levels, and a clear separation between strategic and operational responsibilities[30, 10, 9]. In this structure, worker agents carry out des-

ignated sub-tasks while a supervisor agent has decision-making ability. This means that the information must pass upward and downward through the supervisor rather than following directly between workers. This results in what are known as "coordination gaps", which are places in the workflow where context or reasoning generated by one worker agent cannot be immediately accessed by another without first being processed and sent again by the supervisor. In construct, the sequential pattern avoids the need for an intermediary control layer by transferring each agent's entire output straight to the following agent in the chain.

From the list presented in the table, 8 of the architectures were excluded from the experiment comparison. Feedback-driven, Blackboard, and Tool-augmented was excluded Because they operate as agent-level capabilities integrated into coordination architectures rather than as separate coordination. They are beneficial in domains that need a dynamic state-share or external tool integration. Feedback-driven is more beneficial for iterative quality improvement activities, such as code debugging, refactoring, and content refining, when executable feedback is available [28, 19, 42, 20]. Blackboard is beneficial for high-complex operations, such as scientific research pipelines and architectural design, that need a stable shared state among multiple agents [20, 38, 22]. Tool-augmented is benefit With generating languages, tasks like search engine optimization, code execution, API integration, and file management require real-world-based activities [10, 44, 42].

While the Centralized/supervisor was excluded because it has the same coordination structure with Hierarchical. The reason of choosing hierarchical pattern over centralized/supervisor because a centralized supervisor is a particular instances of a hierarchical structure with just one level, whereas hierarchical is the more general situation[16, 30, 42]. Therefore, testing hierarchical models implicitly covers centralized supervisors, but not the other way around .

Where Human-in-the-loop brings human variability such as variation in response time, skill level, and consistency, which cannot be limit to standardized across runs. Moreover, human-in-the-loop patterns are better suitable for high-risk domains, such as medical diagnostic support, legal review, or critical infrastructure management where human responsibility is necessary before decisions are finalized [28, 26, 42].

Since RAG, Role-based/SOP-driven and Knowledge-driven architectures are data-retrieval methods for individual agents rather than system level coordination patterns, it was excluded. RAG is more beneficial for knowledge-intensive areas like question answering and document summarization, where agents need domain-specific retrieval to reason well [28, 9, 42]. Role-based patterns perform as a role specialization mechanism rather than a coordination strategy. It benefits for large and complex multi-roles tasks outside the scope of self-contained test generation while Role-based patterns showed a strong results in general software engineering workflows [4, 26, 17]. While Knowledge-driven is more beneficial for domains where agents are using integrated domain knowledge such as knowledge graphs or roles defined rather instead of just on what the LLM gained during training, decision-making and break down tasks [28, 13, 24].

5.2 Phase 2: Quantitative Comparison (RQ2)

This section shows the quantitative result of our experiment and highlights the trade-offs between the evaluated architectural patterns with a combined analysis into reliability, performance, and variability.

pattern	metric	metric_label	n_runs	avg_std	variance
hierarchical	agent_turns	Agent Turns	3	2.344719	5.497706e+00
	end_to_end_latency_s	Latency (s)	3	57.306895	3.284080e+03
	estimated_cost_usd	Estimated Cost (USD)	3	0.124674	1.554360e-02
	llm_calls	LLM Calls	3	2.344719	5.497706e+00
	total_input_tokens	Input Tokens	3	13006.231549	1.691621e+08
	total_output_tokens	Output Tokens	3	6058.472676	3.670509e+07
sequential	agent_turns	Agent Turns	3	1.125770	1.267358e+00
	end_to_end_latency_s	Latency (s)	3	16.393479	2.687462e+02
	estimated_cost_usd	Estimated Cost (USD)	3	0.036480	1.330819e-03
	llm_calls	LLM Calls	3	1.125770	1.267358e+00
	total_input_tokens	Input Tokens	3	3149.418534	9.918837e+06
	total_output_tokens	Output Tokens	3	1829.570359	3.347328e+06
single_agent	agent_turns	Agent Turns	3	1.183549	1.400789e+00
	end_to_end_latency_s	Latency (s)	3	24.156983	5.835598e+02
	estimated_cost_usd	Estimated Cost (USD)	3	0.051699	2.672838e-03
	llm_calls	LLM Calls	3	1.183549	1.400789e+00
	total_input_tokens	Input Tokens	3	3880.319547	1.505688e+07
	total_output_tokens	Output Tokens	3	2683.420088	7.200743e+06

Figure 5.1: Table of Calculation of Average of Standard Deviation and Variance

In figure 5.1 shows the calculation result where(n_runs = 3)indicates that all three runs were loaded correctly for each combination of patterns/metrics where the completeness of the data was verified. The (avg_std) is the average standard deviation of the three runs, indicating the degree to which the results of individual tasks vary within a single run. The variance is simply (avg_std) on a different scale of the same spread, where it tells us whether the performance is consistent or noisy.

5.2.1 Reliability (success rate/error handling)

5.2.1.1 Success Rate (pass@1)

The **success rate** is the average result across all runs, referred to **(coverage_passed)** in our data, which helps us to give an estimate of the average performance of each

architecture patterns. To visualize the success rate to obtain a true success rate, we applied 95% of the confidence interval(CI). CI represents the uncertainty in the estimations, not just the spread, which helps us to be more confident with the measured success rate. The reason behind this is that our data is Binary (success/failure) and Bounded (0-1), and the CI stays within valid limits and correctly reflects the statistical uncertainty [15]. Instead of showing the distribution of individual binary results, reflect what a reliability comparison should provide, this provides error bars that indicate uncertainty in the mean estimate (usually ±3 to 8%). The axis label of a mean ± 95% CI becomes correct, and the bars will remain well within 0 to 100%.

In the **success rate by architecture** Figure in 5.2 the sequential recorded the highest mean success rate 92.2%, followed by a single-agent 81.1%. Hierarchical, on the other hand, showed a lower mean success rate 54.4%. The figure 5.2 also presents the success rate for each run to check the performance stability across three runs. The sequential increases from around 87% in Run 1 to a high of approximately 97% in Run 2 before slightly declining to around 94% in Run 3. The single-Agent has a slightly rising trend from about 77% to around 90%. On the other hand, the Hierarchical pattern decreases around to 50% in Run 3 after being approximately constant between Run 1 and Run 2 at 57%. This late-stage indicates that the hierarchical pattern achieved the lowest success rate across all three runs and showed a decrease in the performance through the last run compared to other initial runs.

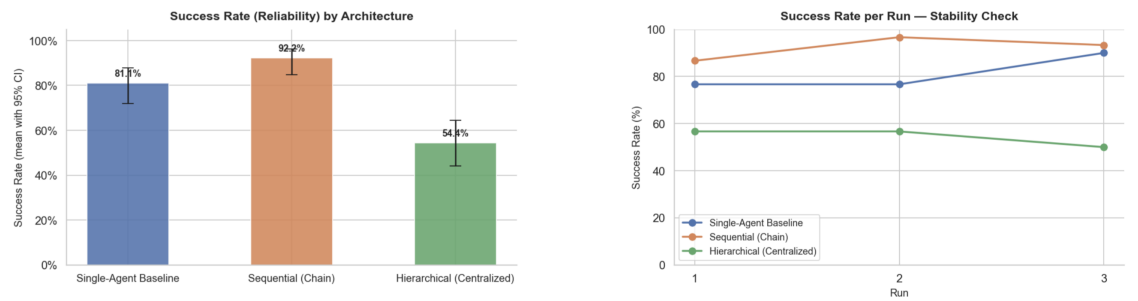

Figure 5.2: Success Rate by Each Architecture and Stability for Each Run

Figure 5.3 presents the success rate for each architectural pattern along with line and branch coverage. Sequential achieves a 91.6% line coverage, 92.2% pass rate, and 90.6% branch coverage. The single-agent achieved 81.1%, 80.6%, and 79.7% for the same metrics. Hierarchical achieved around 54% for all metrics. The three measures showed very similar values for each architecture pattern. Sequential displays 92.2%, 91.6% and 90.6% and single-agent displays 81.1%, 80.6% and 79.7%. While the hierarchical displays almost 54% for all three measures. This indicates that architectures with higher success rates also recorded a high level of test coverage(line and branch).

For showing the **distribution of the variance(stability)**by using a KDE (Kernel Density Estimation) plots per metric with no fitting, it shows that the performance fluctuations across runs and which architecture patterns consist. If the plot shows

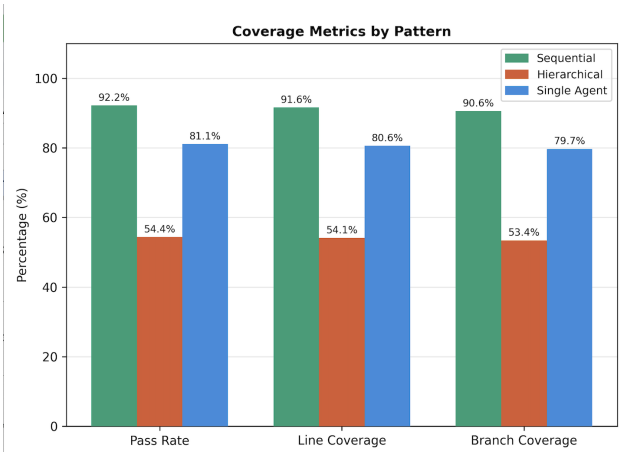

Figure 5.3: Coverage Metrics (Line and Branch Coverage) by Each Architecture Pattern

one pattern with low variance means a stable system and the high variance means an unstable and inconsistent system. The Spread of the KDE shows the patterns with narrow peak means predictable behavior while the wide spread means inconsistent run. In Figure 5.12 Success Rate Reliability (A) the hierarchical (Centralized) architecture shows the maximum variance, as demonstrated by the right-skewed density curves. This shows that while hierarchical systems can perform well in some situations, they are much less predictable and more susceptible to task variability.

5.2.1.2 Error Handling

Error handling is essential to ensure that execution failures do not bias the analysis results. To capture this, we define an error handling metric based on the difference between **tests_generated** and **tests_passed** in Figure 5.4. This gap quantifies the proportion of the generated tests that fail during execution, providing insight into each architecture's robustness in edge cases and partial failures. By evaluating how well each system can validate its own outputs, this measure allows for a more complex evaluation of reliability than binary success metrics. The information Table in Figure **??** provides a qualitative result, as it demonstrates hierarchical failure scenarios that are structurally unachievable in the other two patterns. The label of **delegation_leak** is 11 for hierarchical, since it indicates that its coordination produces a new type of output damage.

The table in figure 5.4 shows the per-run breakdown of test generation and failure rates across the three experimental runs. Hierarchical records the highest average failure rate 6.41% and shows an increasing trend across runs (6.14%, 6.41%, and 6.68%). Sequential achieves the lowest average failure rate 0.46%, remaining close to zero across all runs. Single-Agent generates the largest number of tests on average (793) while maintaining a moderate failure rate of 1.89%.

The test failure rate for each architecture patterns shown in Figure 5.5 highlights a differences between the architecture patterns, with 28.54% for hierarchical compared

Error type	Hierarchical	Sequential	Single-Agent
`api_throttle`	0	3	0
`delegation_leak`	11	0	0
`no_output`	49	83	73
`no_tests_ran`	2	1	0
`partial_failure`	15	2	15
`syntax_error`	12	1	0
`other`	1	0	2
Total	90	90	90

Table 5.2: Error type counts by architecture across all 90 executions per pattern.

Architecture	Run	Tasks (n)	Tests Generated	Tests Passed	Tests Failed	Test Failure Rate (%)
Hierarchical (Centralized)	**1**	30	391	367	24	6.14
	2	30	312	292	20	6.41
	3	30	389	363	26	6.68
Sequential (Chain)	**1**	30	368	366	2	0.54
	2	30	442	442	0	0.00
	3	30	470	466	4	0.85
Single-Agent Baseline	**1**	30	801	779	22	2.75
	2	30	795	785	10	1.26
	3	30	783	770	13	1.66

Figure 5.4: Test Failure Rate And Stability of Error Handling Across Runs

with 1.41% for sequential and 2.83% for single-agent. These differences cannot be described only by the complexity of the task since all the three patterns were evaluated the same 30 tasks across three runs. The higher failure rate observed in the Hierarchical pattern may be related to the additional coordination and communication required between sub-agents, which introduces more stages in the problem-solving process before the tests that are generated get evaluated. According to this pattern, the hierarchical architecture had more challenges than the other two structures in maintaining accuracy throughout the execution process.

To summarize the findings of the reliability analysis, the three architectural patterns revealed differences in both the success rate and the test failure rate. Sequential recorded the highest mean success rate 92.2%, the highest line coverage 91.6% and branch coverage 90.6%, and the lowest test failure rate 1.41%, with stable perfor-

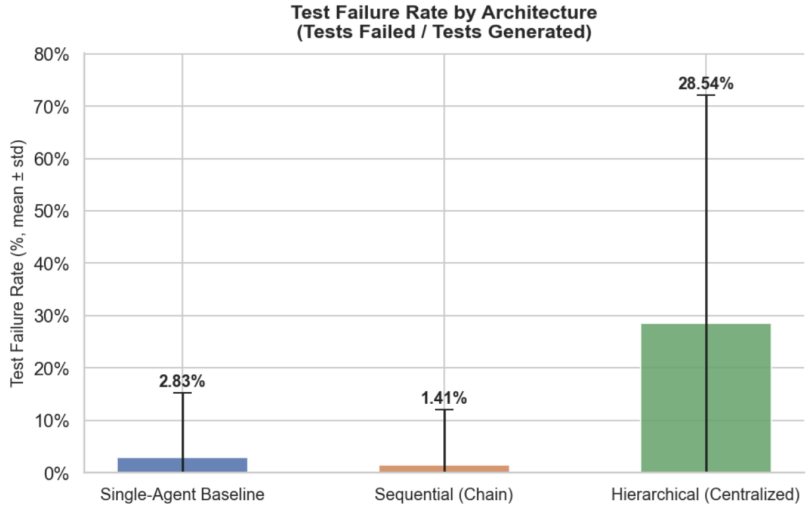

Figure 5.5: Test Failure Rate by each Architecture

mance across all three experimental runs. Single-agent recorded a mean success rate of 81.1%, line coverage of 80.6%, branch coverage of 79.7%, and a test failure rate of 2.83%. Hierarchical recorded the lowest mean success rate 54.4%, the lowest line and branch coverage approximately 54%, and the highest test failure rate 28.54%, with greater variability across runs compared to the other two patterns. The alignment between success rate and coverage indicates that the architectural choice affects both correctness probability and the depth of test generation. The test failure rate analysis indicated that hierarchical produced a higher proportion of self-failing tests than sequential and single-agent across the 30 TestEval tasks evaluated over three experimental runs.

5.2.2 Performance (latency/cost)

5.2.2.1 Latency

Coefficient of Variation (CV) shown in Figure 5.6 indicates differences in consistency across architectures. The single-agent is the most inconsistent pattern across the majority of resource metrics, with the CV values higher than sequential 38–45% and Hierarchical 38–42%. This includes 60.6% for Latency, 71.0% for Input Tokens, 61.1% for output tokens and 62.8% for Cost. This observation demonstrates that while hierarchical uses more resources overall, its variability is comparable with Sequential. Agent Turns and LLM Calls are the most consistent metrics across all architectures; Hierarchical has the lowest CV 24.5%, indicating that its coordination results in a more predictable shift structure even if that structure is costly.

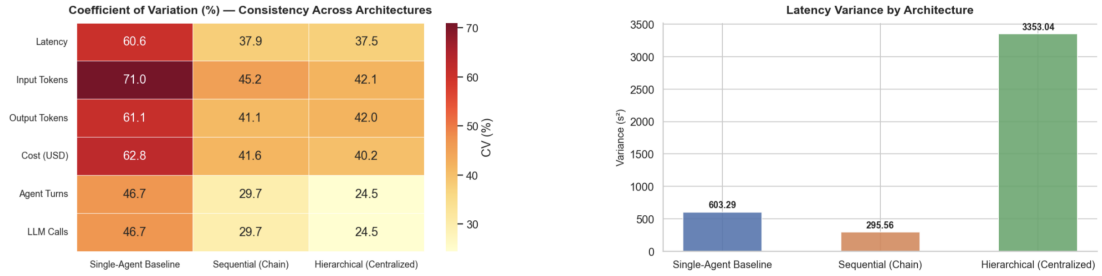

Figure 5.6: The Coefficient of Variation (CV and The latency variance for each architecture pattern)

The latency variance values in Figure 5.6 show that sequential recorded the lowest latency variance of 295.56 s^2 among the three patterns, followed by single-agent 603.29 s^2 and hierarchical 3353.04 s^2. The higher variance observed in hierarchical results indicates greater variability in execution time across tasks compared to the other two patterns.

The table in Figure 5.7 presents the first quartile (Q1) and the third quartile (Q3) for each architecture. Q1 represents the 25th percentile, which means that 25% of the recorded latency values fall below this value. Q3 represents the 75th percentile, which means that 75% of latency values fall below this value. The Q1 value of hierarchical 114.72s exceeded the Q3 values of single-agent 59.47s and sequential 54.83s, indicating that the typical latency ranges of these patterns do not overlap under the experimental conditions of this study.

Architecture	N Tasks	Median (s)	Q1 (s)	Q3 (s)	IQR (s)	p95 (s)	p99 (s)	Min (s)	Max (s)
Single-Agent Baseline	90.0	35.05	22.20	59.47	37.27	79.90	101.79	9.88	143.95
Sequential (Chain)	90.0	41.52	35.09	54.83	19.75	73.10	90.20	2.40	107.65
Hierarchical (Centralized)	90.0	142.37	114.72	185.22	70.50	281.41	298.11	64.53	299.74

Figure 5.7: Summary Calculation of the Median Latency by Architecture

Figure 5.8 shows the median latency and inter-quartile range (IQR) for each architecture. The single-agent baseline recorded a median latency of 35.05s (IQR: 22.20s–59.47s, range = 37.27s), while the sequential recorded a slightly higher median of 41.52s (IQR: 35.09s–54.83s, range = 19.75s). Sequential recorded the narrowest IQR with a width of 19.75s, approximately half that of the single-agent 37.27s, indicating lower variability spread of latency values across tasks. The hierarchical pattern recorded a median latency of 142.37s (IQR: 114.72s–185.22s, range = 70.50s), representing a 3.4× increase over sequential and a 4.1× increase over single-agent.

In the 95th percentile (p95), 95% of tasks completed faster than this value. The hierarchical recorded a p95 latency of 281.41s, compared with 73.10s for the sequential, representing a 3.8× difference between the two patterns at the tail of the latency distribution. The hierarchical p99 latency 298.11s was close to its observed maximum value of 299.74s, with a difference of only 1.63s.

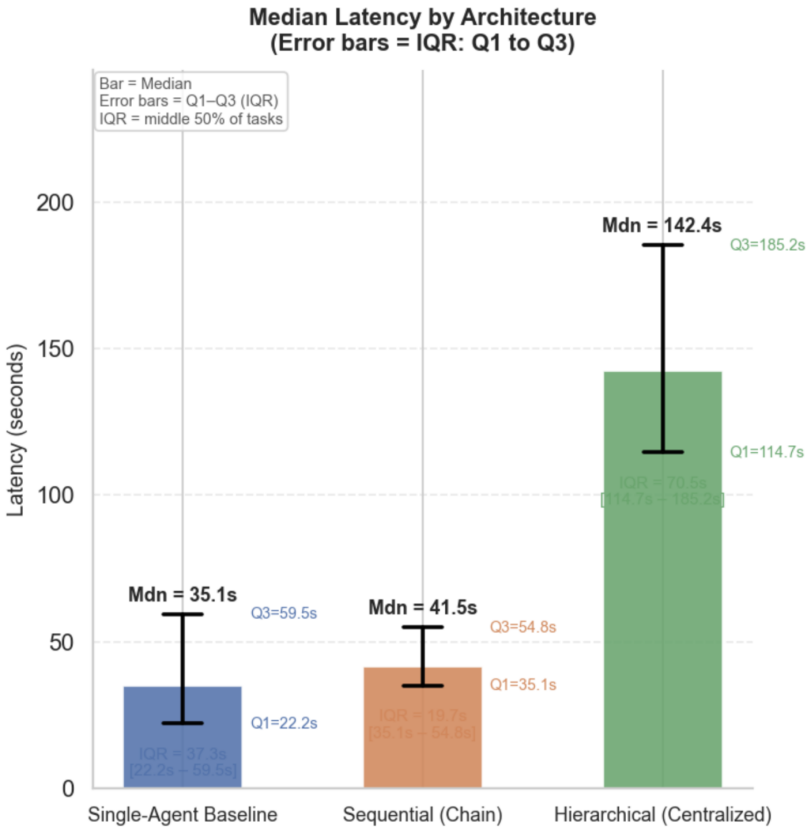

Figure 5.8: Median Latency by Architecture

The agent communication turns recorded in Figures 5.6 and 5.9 provide additional context for the latency differences observed across patterns. The CV heatmap in Figure 5.6 shows that Agent Turns and LLM Calls recorded the lowest CV values across all architectures. Hierarchical recorded the lowest CV for these two metrics 24.5%, followed by sequential 29.7% and single-agent 46.7%, indicating that the number of turns per task was more consistent within each pattern compared to other resource metrics such as tokens and cost.

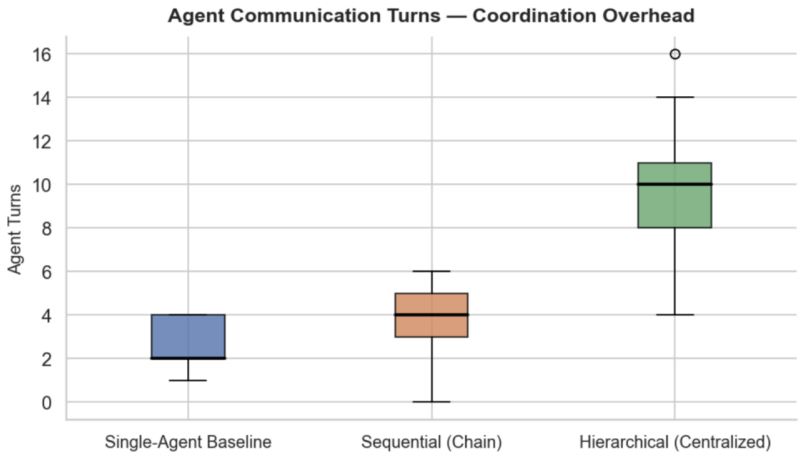

Figure 5.9: Agent Communication Turns by Architecture

The box plot in Figure 5.9 shows the distribution of agent turns per task. The single-agent baseline recorded a median of approximately 3 turns (IQR: 1–4), while sequential recorded a median of approximately 4 turns (IQR: 3–5). Both patterns remained within a bounded range with a maximum of 6 turns. The hierarchical pattern recorded a median of 10 turns (IQR: 8–11), with a whisker extending to 14 and an outlier at 16 turns. Since each agent turn requires an independent LLM call, the higher number of turns in the hierarchical pattern count corresponds to a greater number of model invocations per task, contributing to the higher latency and token consumption values observed in this study.

The distribution of variance using the density plot that shows how variability is distributed across task executions, with the summary of calculation measures of variance. These plots provide a more detailed view by highlighting the changes in distribution shape, spread, and overlap across the architectural patterns. The distribution of task-level latency variance for all three patterns is shown in the latency-Efficiency plot (D) in 5.12. The sequential pattern occupies the leftmost distribution, indicating lower variance values across tasks. The single-agent shows a wider distribution centred around 600 s^2, suggesting greater variation in execution time between tasks. The hierarchical pattern distribution shows a wider and flat curve shape on the right side of the plot covered around 2000-4000 s^2, with a mean variance of nearly 3300 s^2. A clear gap between 1500 and 2000s^2 can be observed in figure 5.12, separating the hierarchical distribution from those of the single-agent and sequential patterns. This gap indicates that the distributions do not overlap in this range. Furthermore, the flatter shape of the hierarchical distribution suggests a wider spread of variance values across tasks. This pattern is consistent with the orchestrator using different numbers of delegation loops depending on task complexity.

5.2.2.2 Cost

While sequential $0.0051 and single-agent $0.0073 are both steady, hierarchical recorded a standard deviation of $0.0103 across runs is shown in 5.10. This indicates that hierarchical cost varied across the three experimental runs. The table also shows that the standard deviation of sequential $0.0051 is smaller than that of single-agent $0.0073. Sequential was not only less expensive than hierarchical, but it also showed less cost variance between runs than the baseline.

Architecture	Run1	Run2	Run3	Mean	Std	Stability
Hierarchical (Centralized)	0.3050	0.2987	0.3189	0.3075	0.0103	Variable
Sequential (Chain)	0.0829	0.0928	0.0901	0.0886	0.0051	Stable
Single-Agent Baseline	0.0851	0.0884	0.0744	0.0826	0.0073	Stable

Figure 5.10: Summary of Mean Cost and Cost Stability Across Runs per Each Architecture

The mean cost shown in Figure 5.11 indicates that the single-agent and sequential patterns have very similar costs, with single-agent costing around $0.083 and sequential costing around $0.089 per task. In contrast, hierarchical recorded higher cost around $0.308 for each task. The token consumption plot on the right of Figure 5.11 explains the reasons behind the high cost of hierarchical. The plot shows that the input tokens exceed output tokens across all patterns as reflected by the larger values of input tokens. This indicates that processing task context require more input tokens than generating responses. This reflects the cost associated with analyzing and communicating task-related data, where each agent must access the entire task context before producing a response. Token consumption is approximately comparable for single-agent and sequential agents (5,000-7,000 input) and (4,000 output). This observation supporting the similar costs shown in the left-side plot. Although sequential introduces additional coordination stages, but it clearly does not increase the amount of context shared between agents.

Hierarchical input token consumption (31,000) is almost five times that the single-agent and sequential patterns. The output tokens (14,000) are also around 3.5 times higher. This occurs because the orchestrator (the root agent), when receiving a task must send independent sub-task descriptions to each sub-agent at every level of the hierarchy. The largest variability is observed in hierarchical input tokens consumption, which shows substantial differences across tasks, as seen in Figure 5.11. As noted in the latency subsection and shown in Figure 5.9, the hierarchical pattern recorded a median of 10 agent turns per task compared to 4 for sequential and 3 for single-agent. Each additional turn generates an independent LLM call with its own token consumption, meaning the cost per task accumulates directly

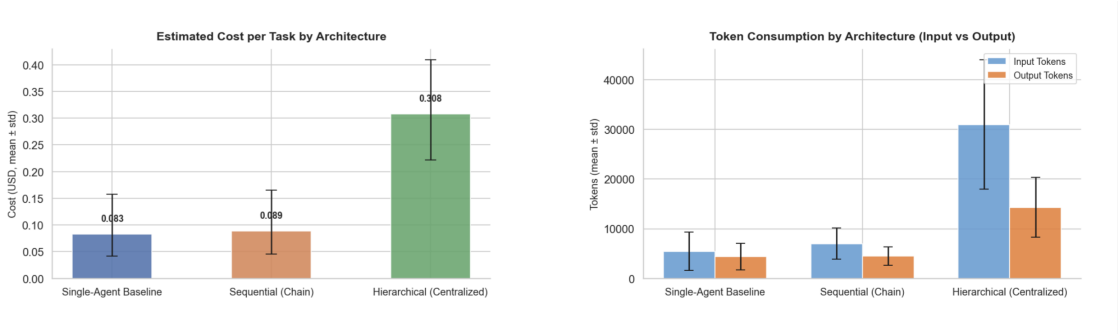

Figure 5.11: Cost and Token Consumption(input vs. output) by each architecture pattern

with the number of turns. This is reflected in the mean cost of $0.308 for hierarchical compared to $0.089 for sequential, a pattern consistent with hierarchical executing approximately 2.5× more turns per task than sequential.

Since cost is derived from token usage, the estimated cost variance distribution shown in the Estimated Cost plot (F) in Figure 5.12, reflects the findings on token consumption. Sequential and single-agent both show distributions concentrated near zero on the variance axis, consistent with their mean costs of approximately $0.083 and $0.089 average. The hierarchical distribution is separated from the other two with a wide and flat curve that centred around 0.015-0.016 USD and spanning around 0.010–0.022 USD on the variance axis. The variance distribution of the input token consumption is shown in Input Token Consumption (B) plot in Figure 5.12. The plot shows that sequential and single-agent overlap almost, with both distributions peaking near zero on the ×10 scale, indicating lower variance in input token consumption across tasks. The hierarchical distribution is separated from the other two distributions, with a mean variance of 1.7×10 and a long, flat tail extending to around 3×10. The close overlap between sequential and single-agent input token distributions demonstrates similar input token consumption across tasks. In this architecture, each agent in the sequential chain receives only the information required for its task rather than the entire interaction history. The variance distribution of output token consumption is shown in the Output Token Consumption (C) plot in Figure 5.12. Sequential distributions around 2-3×10 variance, single-agent distributions around 7×10, and hierarchical distributions around 3.5×10. Unlike the input token distribution, where sequential and single-agent overlap almost completely, the output token distributions show a wider spread for single-agent than for sequential. The wider single-agent's distribution indicates greater variation in output length across tasks. The output token variance of hierarchical is approximately 5 times higher than that of single-agent and 10 times higher than that of sequential agent. This indicates that the response lengths of hierarchical sub-agents vary depending on the tasks assigned by the orchestrator.

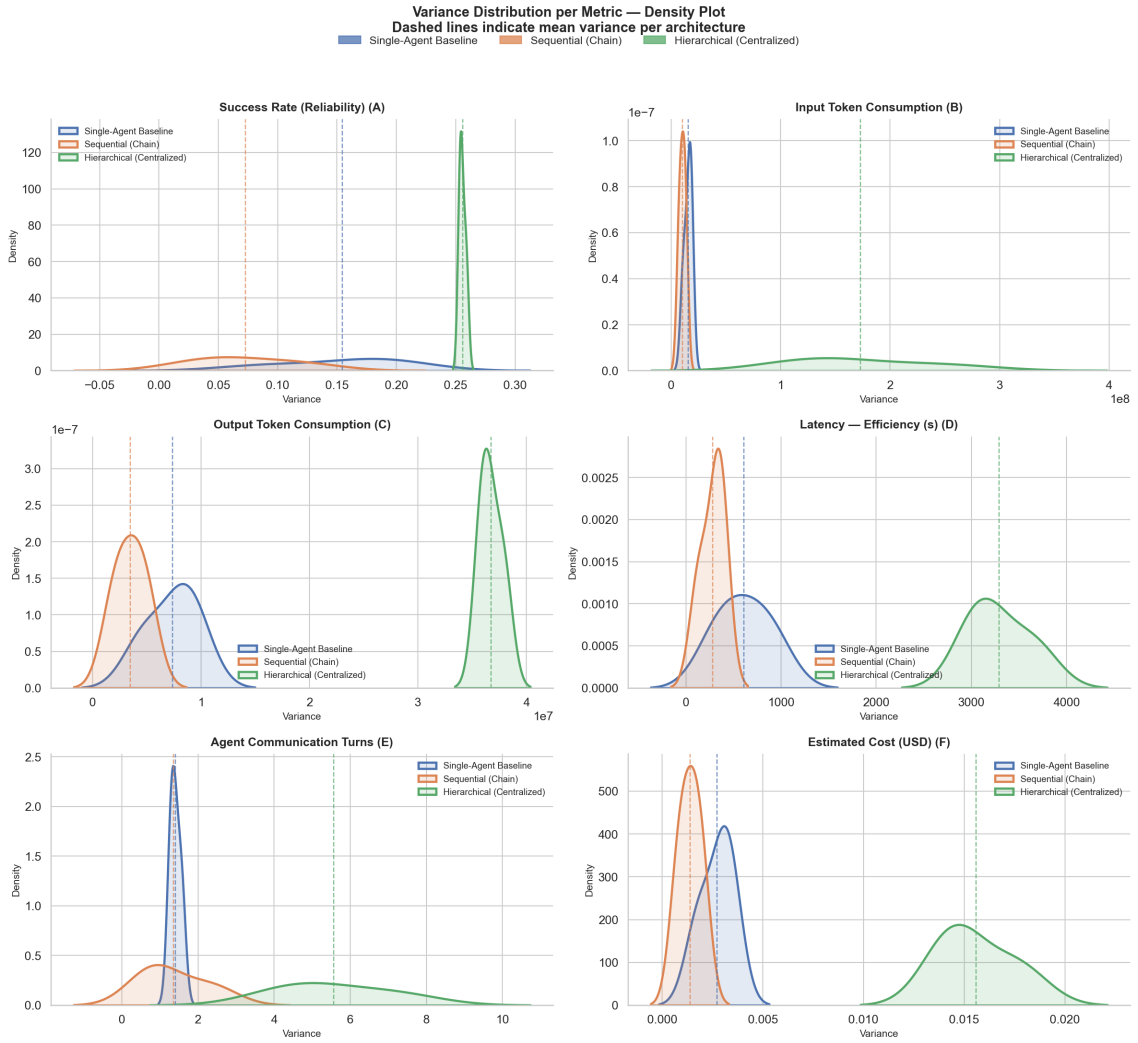

Figure 5.12: The Variance Distribution Across all Metrics of The Three Architecture Patterns

In summary, the performance analysis across latency and cost metrics showed differences across the three architectural patterns. Sequential and single-agent recorded comparable mean costs of $0.089 and $0.083 per task respectively, with standard deviations across the three runs of 0.0051 and 0.0073. Hierarchical recorded a mean cost of $0.308 per task, which is approximately 3.5× higher than sequential with standard deviation of 0.0103. The higher cost observed in hierarchical was associated with greater input token consumption (31,000) compared with (5,000 and 7,000) for sequential and single-agent. As well as a higher median agent-turn count of 10 turns compared with 4 for sequential and 3 for single-agent. These differences were observed for all three experimental runs and the 30 TestEval tasks that were evaluated in this investigation.

5.3 Phase 2: Qualitative Analysis (RQ3)

The experiment was repeated three times per architecture to account for the non-deterministic nature of LLM output generation. Even under identical inputs and controlled hyper-parameters, LLMs exhibit inherent output variance by design. Three independent runs allow us to compute a stable mean and measure run-to-run variability across all dependent variables, reducing the risk of drawing conclusions from a single unrepresentative execution.

5.3.1 Overview of the coded categories

After conducting a direct content analysis of agent interactions logs in the 22 sampled tasks, six behavioral categories emerged. Table 5.3 presents these categories with their definition and the frequencies observed in the three architecture patterns.

Category	Definition	Single-Agent	Sequential	Hierarchical
Iterative Self-Correction	Agent detects incorrect outputs after execution failure and revises them across retry attempts by tracing algorithm logic	16	0	0
Plan-Guided Generation	Dedicated planning phase produces structured test specifications with explicit expected values and code-path targets before code generation	0	3	0
Productive Specialization	Role separation produces observably better reasoning — one agent's analytical output meaningfully improves another's generated tests	0	18	12
Supervisor Information Bottleneck	Supervisor mediates all inter-agent communication but fails to transfer analytical reasoning between workers, causing unproductive re-delegation cycles	0	0	10

Category	Definition	Single-Agent	Sequential	Hierarchical
Assertion Hallucination	Agents generate test assertions with incorrect expected values, reflecting failure to reason about algorithm behavior before committing to expected outputs	6	0	0
Error Propagation	An error from one agent passes uncorrected to downstream agents, causing cascading failures in the final output	0	1	0

Table 5.3: Behavioral categories and frequencies across architectures (n=66 coded cases: 22 tasks × 3 architectures).

The categories show a strong architecture-specific distribution: Iterative Self Correction and Assertion Hallucination occur exclusively in the single-agent baseline; Plan-Guided Generation and Error Propagation appear only in the sequential pattern; the Supervisor Information Bottleneck is unique to the hierarchical pattern; Productive Specialization is the only category shared between the two multi-agent patterns.

5.3.1.1 Single-Agent Behavioral Profile

The single-agent baseline produces all tests in a single LLM call without intermediate verification. When the initial output contains errors — most commonly incorrect expected values in test assertions — the agent enters a retry loop in which it traces the logic of the algorithm and corrects specific failure assertions. This pattern, Iterative Self-Correction, was the dominant behavioral mode (16 of 22 cases).

Six of the 22 sampled single-agent cases were coded as Assertion Hallucination — failures which the agent exhausted the retry budget without converging. In Task 97 (isInterleave), the single-agent assumed isInterleave("aaa","bbb","ababab")=False when it is in fact True. Across three retries, the agent traced through the dynamic-programming table and corrected this specific assertion, but introduced new incorrect assertions for other test cases in the process. Run 1 happened to converge by the third retry (25/25 tests passed); Runs 2 and 3 did not (22/23 and 23/24 passing, respectively). The cross-run inconsistency (1/3 passes) indicates that single-agent success on algorithmically complex tasks depends on whether the stochastic retry process converges within the budget.

Across the full dataset, the single-agent achieved an 81.1% pass rate (73/90), with

67% cross-run consistency (20 of 30 tasks producing the same outcome in all three runs).

5.3.1.2 Sequential Behavioral Profile

The sequential architecture (Planner → Coder → Tester) showed a fundamentally different behavioral pattern. The most consequential feature was Productive Specialization: In 87 of 90 runs across the entire dataset, the planner agent produced a structured test specification that included explicit input values, expected outputs, and the specific code paths each test case targeted.

This upfront analytical reasoning functioned as a proactive verification. Rather than generating assertions first and correcting them after failure (as the single-agent does), the sequential planner reasoned about algorithm behavior before any test code was written. For Task 97, the planner produced entries such as: "Edge case — length mismatch. Input: s1='abc', s^2='def', s3='abcdefg'. Expected: returns False. Targets: early return condition m + n != len(s3)." The coder agent received pre-verified expected values, eliminating the assertion errors that dominated the single-agent's retry cycles. Task 97 passed on the first attempt with zero retries and 7,733 tokens — less than half the single-agent's 16,611 tokens for the same task.

The sequential architecture achieved the highest pass rate in the experiment (92.2%, 83/90), the highest cross-run consistency (83%, 25 of 30 tasks consistent across runs), and the lowest mean retry count (1.0, vs. 1.5 for single-agent and 2.1 for hierarchical). Token consumption (mean 11,507 per task) was only 17% higher than the single-agent despite involving three agent roles.

5.3.1.3 Hierarchical Behavioral Profile

The hierarchical architecture (Supervisor → Analyst/Writer workers) exhibited the most complex interaction patterns and the most pronounced failure modes. 12 of 22 sampled hierarchical cases were coded as Productive Specialization suhc as cases where the analyst-writer separation produced correct tests within the supervisor's revision budget. The remaining 10 cases were coded as a recurring pattern we labeled as the Supervisor Information Bottleneck, which appeared to be the most common failure mode in the sampled hierarchical traces.

In every coded bottleneck case, the trace exhibited the same structural pattern. The supervisor delegated an analytical task to the analyst, received a substantively correct algorithm trace, and then delegated a writing task to the writer. The writer produced tests with incorrect assertions. The supervisor responded by re-delegating to the analyst not to the writer requesting another analysis of the same failing test cases. This cycle was repeated, with the analyst producing essentially the same correct trace each time, while the writer's errors persisted because the analyst's reasoning never reached the writer's context directly. All inter-agent communication was mediated by the supervisor.

Task 97 (isInterleave) illustrates the mechanism. The analyst correctly identified

that concatenation-style inputs such as "abcdef" for isInterleave("abc","def",...) constitute valid interleavings. The writer, which never received the analyst's trace output directly, continued generating assertions assuming, such inputs were invalid. The supervisor re-delegated to the analyst three times, each time asking for re-analysis of the same failing cases. The pattern consumed 43,957 tokens (5.7× the sequential's 7,733 on the same task) without achieving a correct result. This failure was consistent across all three runs (0/3 passes), confirming an architecture inherent phenomena rather than purely stochastic explanation, though our design cannot establish this conclusively. The same supervisor bottleneck pattern appeared in tasks 126, 417, 457, 524, 420, and other hierarchical failures.

The hierarchical architecture achieved the lowest pass rate in the experiment (54.4%, 49/90), the lowest cross run consistency (47%, 14 of 30 tasks consistent across runs) and consumed substantially more tokens per task than either alternative (mean 45,281 - 4.6× single-agent, 3.9× sequential). The pass rate at maximum retries (3) was 18% (9 of 50), substantially lower than the single-agent's 39% and the sequential's 43% at the same retry count, indicating that the hierarchical's failure mode is more resistant to retry-based correction than the failure modes of the other patterns.

In summary, qualitative analysis of 22 selected behaviours across the three architectural patterns identified six behavioural categories with architecture-specific distributions. Iterative self-correction and assertion hallucination were characteristic of the single-agent baseline. Plan-guided generation and productive specialization were characteristics of the sequential pattern. The supervisor information bottleneck and productive specialization were key characteristics of the hierarchical pattern. These categories provide a qualitative explanation for the quantitative differences identified in RQ2 and are discussed more in the discussion section. Considering suitability of each pattern for software engineering tasks and the extent to which these findings can be generalized beyond the benchmark examined.

6

Discussion

This chapter discusses the empirical and the qualitative findings of the experiment in relation to the literature on multi-agent LLM systems. *All the findings derived from a single LLMs (Claude Sonnet 4 [3]) and a single benchmark family (TestEval [40, 41]), which limits the basis for generalization; replication between LLMs and benchmarks is required before treating these findings as established architectural principles.*

6.1 An interpretation of the architectural pattern taxonomy findings of RQ1

The answer to RQ1 was addressed through a comprehensive thematic analysis of 31 articles published between 2023 and 2026, which identified 10 common architecture patterns in multi-agent LLM-based systems. The patterns identified include more complex multilayer systems such as hierarchical and blackboard/shared-memory architectures, as well as simpler linear coordination structures such as sequential pipelines. This result aligns with the findings of Guo et al., who identified centralized orchestration, decentralized negotiation, and market-based allocation as distinct coordination frameworks in multi-agent systems [16]. In this context, sequential and hierarchical patterns belong to centralized orchestration, whereas more collaborative or feedback-driven patterns reflect aspects of decentralized coordination. The same applies to the findings of Liu et al., who also provide a comprehensive catalog of agent design patterns covering memory management, role specialization, and task decomposition. The patterns were ranked based on the number of publications (out of the 31 reviewed studies) in which each pattern was identified. The role-based/SOP pattern ranked first, as it appeared in the highest number of studies. This is associated with the widespread adoption of frameworks such as MetaGPT, which clearly use role specialization as a core coordination mechanism and combine role-based and sequential coordination [17]. The sequential pattern ranked second, reflecting its common use in software engineering tasks, as demonstrated by AgentCoder, which achieved 77.4% pass@1 using a three-agent sequential pipeline [19]. Furthermore, the MAAD and NOMAD frameworks also applied sequential coordination for automated architectural workflows [24][13]. The hierarchical pattern ranked seventh, as it was identified in fewer of the 31 reviewed publications. This

lower ranking reflects its higher implementation complexity, as discussed by Moore and Bandara [30][4]. The remaining architecture patterns were excluded from the quantitative comparison. Feedback-driven, blackboard, and tool-augmented patterns were not included because they are typically used as auxiliary mechanisms within architectures rather than as independent coordination patterns [28]. The centralized supervisor pattern was excluded due to its structural similarity to hierarchical coordination, where tasks are delegated to specialized sub-agents [30][16]. Furthermore, human-in-the-loop was excluded because human variability cannot be standardized across experimental runs, such as response time and skill level [42]. RAG and role-based patterns were also excluded because they operate at the individual agent level rather than at the system coordination level [28].

The sequential and hierarchical patterns were selected for quantitative comparison in Phase 2 based on their distinct coordination strategies, where sequential coordination assigns work to agents in a linear pipeline, whereas hierarchical coordination distributes tasks across multiple levels of delegation [30][10]. This comparison was important for investigating whether coordination complexity is associated with differences in performance and reliability outcomes. The results in Phase 2 showed that the two patterns produced different outcomes across all four metrics: latency, cost, success rate, and test failure rate. The single-agent baseline was included as a control condition to provide a reference point for evaluating multi-agent coordination. This comparison enables analysis of how coordination strategy affects system behavior, task performance, and operational complexity. Guo et al. and AgentCoder both recommend including a single-agent baseline before evaluating multi-agent architectures [16][19].

6.2 An interpretation of the quantitative findings of RQ2

The results section of Phase 2 quantitative comparison addresses RQ2 by comparing three architectural patterns (single-agent, sequential, and hierarchical) across four metrics: latency, cost, success rate, and error handling. The sequential architectural pattern showed the most advantageous results across all four metrics, while the hierarchical pattern showed the least advantageous outcomes under the same experimental setup.

6.2.1 Performance - Cost and Latency

Based on cost and latency, the hierarchical pattern was substantially slower and more costly than the other patterns. The hierarchical pattern recorded a median latency of 142.37s and a mean cost of $0.308 per task, compared with the sequential pattern, which recorded 41.52s and $0.089, and the single-agent pattern, which recorded 35.05s and $0.083. The hierarchical pattern works by having a root agent divide tasks into smaller sub-tasks and assign each one to different sub-agents at every level of the hierarchy. Each delegation round generates an independent LLM request that

accumulates context from previous rounds. The average agent communication turn count for the hierarchical pattern was 10 turns per task, compared with 4 turns for the sequential pattern and 3 turns for the single-agent pattern. Each additional turn caused an extra LLM request and greater token consumption, which could be a contributing factor to the higher latency and cost observed in the hierarchical system. This is consistent with Guo et al., who found that centralized patterns have different latency and cost characteristics compared with simple coordination patterns [16].

In contrast, the sequential pattern showed that increasing the coordination layer between agents does not necessarily result in increases in cost or latency. The sequential mean cost of $0.089 is very close to the cost of the single-agent pattern, $0.083, with a difference of only $0.006, and its median latency of 41.52s remains close to the single-agent value of 35.05s. This suggests that the sequential coordination chain effectively shares context between agents without adding substantial cost, unlike the hierarchical pattern, where context accumulates across multiple layers of delegation. This finding aligns with the AgentCoder study, which showed that sequential patterns reduce coordination costs compared with more complex architectural patterns [19]. A similar finding was reported by the AnyMac framework, which showed that reducing the coordination process also reduced token usage with very little effect on accuracy [39].

The variance distribution plot showed that the cost and latency distributions of the hierarchical pattern did not overlap with those of the sequential and single-agent patterns. This indicates that even single-agent tasks with the highest variability had lower latency and cost values than hierarchical tasks with the lowest variability. This observation was consistent across all 90 tasks rather than being limited to a specific subset of tasks. Yan et al. reported that the internal communication structure of an agent directly impacts system-level performance [10]. This aligns with our findings, where the sequential pattern uses a linear chain of communication and the hierarchical pattern uses multiple levels of delegation, resulting in clear differences in latency and cost. The cost stability analysis also found that the hierarchical pattern (std = 0.0103) was the only pattern whose average cost varied noticeably across runs, compared with the sequential pattern (std = 0.0051) and the single-agent pattern (std = 0.0073), suggesting that hierarchical costs were less stable across all three runs.

6.2.2 Reliability - Success Rate and Error Handling

The performance findings are supported by the reliability results. The hierarchical pattern had the lowest success rate at 54.4%, the lowest line coverage at 54.1%, the lowest branch coverage at 53.4%, and the highest test failure rate at 28.54%. In contrast, the sequential pattern achieved the highest success rate at 92.2%, the highest line coverage at 91.6%, the highest branch coverage at 90.6%, and the lowest test failure rate at 1.41%. The single-agent pattern achieved moderate results across all reliability metrics, with a success rate of 81.1%, line coverage of 80.6%, branch coverage of 79.7%, and a test failure rate of 2.83%.

The consistency between the coverage metrics and success rates across all three patterns confirms that the observed variations are not due to random execution outcomes but instead reflect genuine differences in test generation capability. This cross-metric consistency strengthens the conclusion that the hierarchical pattern's low success rate is caused by a structural weakness in producing thorough test suites rather than by random failures. The 37.8 percentage-point gap in success rate between the sequential and hierarchical patterns, combined with the 20-fold difference in test failure rate, suggests that the coordination mechanism used by the hierarchical pattern introduces reliability risks associated with the architecture itself rather than with task difficulty.

The error-type analysis provided additional insight into the characteristics of hierarchical errors. Specifically, 11 cases were identified as delegation-leak errors, meaning that the system exposed its internal coordination instructions in the generated test file output. This type of error was not observed in either the sequential or single-agent patterns because it is unique to the hierarchical coordination mechanism. These findings suggest that the boundary between internal coordination and external output requires additional controls in hierarchical implementations, as it directly affects the outcome of test generation. The hierarchical pattern also exhibited the highest test failure rate, with 28.5% of tests failing, compared with 1.2% for the sequential pattern and 2.8% for the single-agent pattern. This result indicates that the hierarchical pattern not only fails more tasks but also sometimes produces internally inconsistent outputs that do not align with its own generated solutions. One possible explanation is that individual sub-agents may not have access to the complete problem context, making it more difficult to generate test cases that align with the corresponding solution.

6.2.3 Trade-off Analysis

The findings of this study show that the three architectural patterns exhibit different trade-offs. The hierarchical pattern shows the highest values across all measured metrics, including latency, token consumption, and cost, while also recording the lowest success rate at 54.4% and the highest test failure rate at 28.54%. This combination indicates that the hierarchical pattern does not provide a favorable performance on any of the four metrics evaluated. The trade-off analysis differs for the sequential pattern. Its mean cost of $0.089 is slightly higher than that of the single-agent pattern at $0.083, and its mean latency of 41.52s is higher than the 35.05s recorded by the single-agent pattern. This indicates a small increase in latency and cost associated with sequential coordination, reflecting a measurable difference in efficiency compared with the single-agent baseline. However, the sequential pattern recorded an 11.1 percentage-point higher success rate than the single-agent pattern, with only a small cost gap of $0.006 per task. This result aligns with previous research showing that single-agent systems can perform comparably to multi-agent systems on simple and well-defined tasks, where multi-agent coordination may lead to inefficiencies rather than improved outcomes [43]. Moreover, Yan et al. and the findings presented in Magentic-One show that single-agent methods have lower latency and cost for simple operations, while multi-agent coordination becomes more

suitable for tasks that require structured delegation or shared processes [43][10]. Since all three patterns were evaluated on the same 30 TestEval tasks, the observed differences in success rate across architectural patterns cannot be explained by differences in task difficulty. However, this was not specifically investigated in this study, and task complexity may still have had varying effects on the performance of the different architecture patterns.

6.2.4 Practical Implications

From a practical perspective, the cost differences between the patterns become more apparent at scale. For 1,000 tasks, the sequential pattern would cost $88.60, while the single-agent pattern would cost $82.63, resulting in a difference of $5.97. The hierarchical pattern would cost $307.53 for 1,000 tasks, which is substantially higher than the sequential pattern. This cost difference may be a relevant consideration for automated test generation pipelines that need to manage a large number of tasks. Based on the results of this study, which evaluated 30 TestEval tasks in three runs, the sequential pattern performed better than the single-agent and hierarchical patterns in metrics of latency, cost, and reliability. The hierarchical pattern incurred the highest cost per task while successfully generating fewer test cases compared to both the single-agent and sequential patterns. The hierarchical pattern is designed for multi-agent systems that handle large and complex tasks requiring decomposition into smaller, independent sub-tasks, as described by Moore and Bandara [30][4]. However, the TestEval tasks used in this study consist of self-contained algorithmic programming problems that require access to the complete problem context during both the test and solution generation processes. When the hierarchical pattern divides a task among multiple sub-agents, each sub-agent receives only a portion of the original context. This may lead to the generation of test cases that do not fully capture all of the problem requirements. In contrast, the sequential pattern passes the entire output of one agent to the next without decomposing the task, which may help explain why its cost and success rate are very similar to those of the single-agent baseline. In industry, the hierarchical pattern remains applicable despite its lower performance in our study because it is better suited to tasks with a larger scope that require decomposition across specialized roles, such as testing large modular software systems [30][4]. These characteristics differ from the TestEval tasks that are investigated in this study. For example, role specialization within the MetaGPT framework achieved an 81.7% pass@1 score on code generation benchmarks, as demonstrated by Hong et al. [17]. This suggests that specialization can produce results that differ from those observed in our findings. In the MetaGPT framework, each agent was assigned a specialized professional role, such as product manager, architect, engineer, or QA specialist.

6.3 An interpretation of the qualitative findings of RQ3

The quantitative differences observed in RQ2 emerged from six behavioral categories that were identified and explained in the qualitative analysis. Three categories are discussed in this section because of their direct relationship to existing claims in the literature.

6.3.1 The Iterative Self-Correction

This pattern, observed in the single-agent baseline, is one of the two most common patterns for SE tasks, as [7] mentioned. While our findings align with the literature's largely positive characterization of self-reflection, we claim that this phenomenon is effective for small, isolated errors, achieving a 100% pass rate at 0–2 retries. However, it has a hard limit when agents fundamentally misunderstand the algorithm under test. The retry mechanism cannot recover from such failures, achieving only a 39% pass rate at 3 retries across the full dataset. This finding aligns with the observation by [21] that **LLMs require proactive verification supported by the architecture; otherwise, they may produce plausible-sounding but incorrect outputs, and this limitation cannot be overcome by post-hoc correction alone.**

6.3.2 The Plan-Guided Generation

This pattern, observed in the sequential architecture, provides empirical evidence for what [7] describe as the Single-Path Plan Generator design pattern. Our research revealed that, across the complete dataset, 87 out of 90 sequential runs contained a planner-generated specification with well-defined expected values and code-path objectives, suggesting that this organized planning behavior and reasoning are extremely common characteristics of the architecture's functionality. Tran et al. [38] note that pre-generation reasoning improves output quality, aligning with our theoretical expectations and reinforcing our empirical findings that **role-based protocols with clearly defined and specialized responsibilities produce superior results.**

6.3.3 The Supervisor Information Bottleneck

This limitation is the most novel finding from the qualitative analysis and the one with the strongest claim to a contribution beyond the existing literature. [34][7] and Section 2.3.2 anticipated that hierarchical multi-agent systems would face coordination and communication overhead, framing these primarily as cost concerns. However, our empirical research indicates that this prediction may be extended: supervisor mediation can lead to both *increased costs* and *information loss*. The analyst produced high-quality outputs and accurately reasoned about the algorithmic tasks, whereas the writer often produced inaccurate outputs. The supervisor's response to such outputs involved re-delegating tasks to the analyst rather than

conveying the analyst's current rationale to the writer. No direct information flow between workers was observed in any of the 41 hierarchical failures in the complete dataset. Furthermore, in the traces we examined, the supervisor did not forward the analyst's reasoning to the writer, we interpret this as a plausible mechanism for the observed failures. **This indicates that the hierarchical structure was inefficient in this context with the sampled cases, as the supervisor consistently failed to communicate the analytical results of one worker to the worker who required them.**

6.4 Generalizability beyond software engineering

Three categories in our findings could be considered architecture-inherent and potentially generalizable to different application domains. Meanwhile, the remaining three are more task-specific and require caution when extrapolating the results.

In this study, the **supervisor information bottleneck** appeared whenever the hierarchical pattern was used, which suggests it may be a property of centralized orchestration rather than a task-specific effect. Whether it generalizes to other domains and models remains an open question requiring replication (see Chapter 7). This aligns with the observations of [38] [34] that hierarchical role-based systems lack flexibility in information sharing among worker agents without proper management of inter-dependencies. The implication for any domain in which agents collaborate on integrated outputs is that the architecture cannot rely solely on the supervisor to coordinate communication between workers. Established communication mechanisms, such as communication channels or shared workspaces that enable direct worker-to-worker communication, should be implemented to avoid information loss.

The clear advantage of **upfront reasoning**, which enabled the sequential architecture to successfully complete 87 of 90 tasks, may be used for structured activities where a predetermined outcome can be specified, such as financial reporting, medical report writing, and legal document drafting. However, empirical validation in these domains is necessary before this claim can be confirmed. [7] found that 42 of the 94 surveyed articles used SOP-style planning workflows for SE tasks, suggesting that this approach is widely adopted by software engineers. Our findings confirm its value by suggesting that this pattern may generalize to domains with derivable specifications, such as legal document drafting, medical report generation, and financial reporting. However, information flow within an architecture is likely more domain-independent, resulting in a **cost-quality trade-off** that is not dependent on the specific task. Practitioners cannot assume that outcomes will improve simply by implementing multi-agent systems through the addition of more agents and coordination layers. Instead, outcomes depend on the compatibility between inherited architectural characteristics, such as information flow, and the requirements of the task. Nevertheless, empirical validation is required for each architecture and task type to fully answer this question.

6.5 Industrial Alignment and Robustness of Findings

The dominance of sequential coordination in this study is consistent with architectural choices observed in widely adopted agentic coding harnesses. The popularity of sequential and single-agent coordination patterns in this study is in line with the results of the literature evaluation, which found that Sequential was used in 20 out of 31 publications and ranked second among the 10 patterns. Sequential pipeline coordination produces good results for structured code generation tasks, as shown by previous studies such as AgentCoder [19]. Moreover, frameworks such as MAAD and NOMAD applied sequential coordination for automated software engineering workflows [24][13]. The observation that sequential coordination is more naturally aligned with structured, artifact-centric software engineering activities under baseline conditions is further supported by the absence of hierarchical coordination as a dominant pattern in these SE-focused frameworks. When translating these findings into industrial practice, it is important to distinguish between findings that are robust to experimental conditions and those that are sensitive to the specific parameters of this study. Because each delegation round creates a separate LLM request by design, the structural cost overhead related to hierarchical coordination is driven by further delegation rounds and accumulated context, which remain architectural features whether or not quick optimization is applied. On the other hand, the study's absolute success-rate metrics, which are 54.4% for hierarchical and 92.2% for Sequential, represent unoptimized baseline setups and should be understood as lower-bound estimates rather than fixed performance limits. Under various deployment scenarios, architecture-specific prompt engineering, role definition, and coordination improvements may reduce or increase these gaps. Therefore, rather than focusing on the specific numerical results, which are limited by the experimental constraints outlined in Chapter 7, practitioners adopting these findings should focus on the structural trade-offs presented.

6.6 The Role of Human Developers in Agentic Workflows

While the primary goal of this research is to evaluate these architectures as autonomous test-generation pipelines, an assessment of their industrial applicability and the role of human developers must also be considered. Our results have direct implications for how human-agent collaboration should be structured. However, the points at which human intervention occurs differ substantially across the architectural patterns. The single-agent pattern generates tests without any intermediate stage where developers can intervene to inspect or correct the generated plan. In contrast, the sequential pattern allows developers to review and edit the generated structured plan before any code is written, potentially improving reasoning about the algorithm and correcting misunderstandings made by the planner agent. For the hierarchical pattern, the natural intervention point is the supervisor role, which

coordinates communication between agents. In this scenario, the developer may delegate tasks to different agents and resolve disagreements between them. This role is consistent with our findings regarding the supervisor bottleneck, where human judgment remains more reliable than LLM-based coordination. This shift in the developer's role from test author to plan reviewer introduces a high-leverage decision point at which developers audit generated artifacts for correctness against business and security requirements that LLMs cannot yet reliably verify. Moving developers up the abstraction ladder changes how they interact with tools such as IDEs and CI pipelines, emphasizing the verification and evaluation of business and security specifications rather than the direct creation of tests.

This repositioning introduces new design considerations that existing tools should leverage to maximize the benefits of agentic systems. Developers should not rely solely on final outputs but should also be given access to intermediate artifacts such as plans, analyses, and draft tests. Tools built around the sequential pattern, in particular, should treat the planner's output as a primary user-facing artifact rather than an internal step. Similarly, tools built around hierarchical orchestration should consider whether the supervisor role is best performed by an LLM or whether a human-in-the-loop supervisor would be more effective. Although these design implications extend beyond the formal scope of this thesis, they follow directly from the empirical findings and help translate the research into practical applications.

6.7 A set of practical architecture selection guidelines

The findings support a set of decision heuristics for selecting an architectural pattern, table 6.1 presents these heuristics as a function of task properties. These heuristics are grounded in a single model and benchmark and should be treated as a starting point for hypotheses, not validated recommendations.

Task Property	**Recommended Pattern**
Task is short and bounded; retry-based correction is acceptable	**Single-agent**
Output specification can be derived before generation	**Sequential**
Workers operate on independent subtasks not requiring integration	**Hierarchical** acceptable
Token budget is a primary constraint	**Single-agent** or **Sequential**
Cross-run reliability is required (production systems)	**Sequential** (highest consistency at 83%)

Table 6.1: Practical architecture selection guidelines based on task properties.

These guidelines refine the general design considerations outlined in the Background (Section 2.3) by grounding them in observed agent behavior rather than theoretical properties alone. We suggest that practitioners should avoid using complicated architecture patterns to solve simple tasks and instead adopt the simplest architecture sufficient for the task. Use single-agents for short bounded tasks that require minimal upfront reasoning and planning. Use the sequential pattern for structured heavy planning tasks that require it. To avoid the supervisor bottleneck in the hierarchical architecture, use the pattern when working with tasks that require worker independence or if workers have direct communication channels to mitigate the supervisor's coordination limitation. This recommendation is consistent with [7] finding that role-based cooperation is the most-adopted pattern, yet it is important to understand that **it is not the role separation that produces value, it is the structured information flow that the role separation enables.**

7

Threats to Validity

Several methodological limitations bound the conclusions drawn above and should be considered when interpreting the findings.

7.1 Internal Validity

All three architecture patterns used the same model(Claude Sonnet 4) without role specialist between agents. This is a good decision to ensure that any observed differences between patterns cannot be explained when one pattern has access to more than one model. However, there are still threats that exist when the same model may interact differently with each coordination mechanism. For example, it may respond more effectively to linear sequential task structures than to multi-level hierarchical task delegation. The findings represent the relationship between Claude Sonnet 4 and each of architecture pattern and may not generalize to the implementation using different models.

The LLM output variability when running the experiment three runs is represent an internal validity. Large language models (LLM) are non-deterministic by design, the same input prompt may produce different output across repeated executions, due to sampling temperature and generation characteristics. This variability may affect metrics including success rate, token consumption, latency, and error handling between runs. This was taken into account by evaluating each architectural pattern over three experimental runs rather than just one, and the findings were presented as average between runs. In contrast, the range of output variability associated with LLM-based systems may not be sufficiently captured by three runs. Because a large number of runs would provide a more stable estimation of the performance metrics of each pattern.

In this study, none of the architecture patterns were implemented with a specific prompt optimization. This decision was made to ensure fair comparison of the patterns. Because when applying optimize just one pattern while leaving the others not optimize, would introduce a false advantage that affect the fairness of the experiment. All three conditions were therefore evaluated under standardized, Un-optimized prompt configurations.

However, this means that the results reflect non-optimized baseline implementations for all three patterns, rather than reflecting the full capability under optimized setups configuration. The outcome of the study performance metrics include the success rate, latency, token consumption, and cost may not reflect how each pattern perform under full optimize conditions. This recognizes as a risk to internal validity because the performance gaps we observed may reflect how sensitive each pattern is to prompt quality rather than architectural design alone. To address these threads in future work, each pattern should be evaluated under both non-optimized and optimized prompt conditions with equal optimization effort given to each pattern. This would allow researchers to distinguish architecture-driven performance differences from prompt-sensitivity effects, and to establish whether the hierarchical under performance observed in this study remains under optimized conditions or is an outcome of the standardized prompt design.

Qualitative analysis was conducted as direct content analysis [18], where categories and codes were pre-specified from a prior reviewed literature and utilized through structural characteristics of the agents traces. While this is a recognized qualitative method, the categories did not fully emerge from the data, which differs from inductive grounded-theory approaches. We had to eliminate for instance the *Coordination overhead* category since no clear evidence in the agent traces had an independent occurrence pointing to this category. This mitigation validates that the pre-coded categories were reviewed and revised in light of evidence. However, the boundary between *Plan-Guided Generation* and *Productive Specialization* in the sequential architecture remains a matter of coding judgment rather than clean structural distinction, which we acknowledge openly. Additionally, coding was conducted by a single researcher, no inter-rater agreement measure was computed, in contrast to [7] procedure, where three authors coded the same data and resolved disagreements through the consensus.

7.2 External Validity

Our findings derived from a single LLM (Claude Sonnet 4), a single benchmark family (TestEval), and a single SE task type (unit test generation). The identification of inherited patterns in the architectures such as *Supervisor information bottleneck* should be replicated with other LLMs to empirically verify the generalizability findings stated in discussion section (6.4). The temperature setting (0,2) was held constant; to reduce the impact of sampling randomness and guarantee that observed variances were mostly due to architectural patterns rather than variations in model generation across all tests. However, the model behavior may differ at higher temperatures settings, particularly in the single-agent architecture where more output variety may have an impact on retry-based self-correction. Another validity issue that must be recognized is the **benchmark contamination**; the TestEval benchmark tasks fundamentally consist of LeetCode [1, 41] algorithms that are widely available on the public internet. These algorithms and their solutions are likely included in the training data of LLMs, suggesting that agents are more prone to memorizing the solutions instead of engaging in reasoning about the tasks. Ana-

lyzing generated unit tests for problems the model has memorized the functional specification and potential solutions is qualitatively different from algorithmic reasoning to generate tests for novel functions. Consequently, the absolute pass rates reported in this study (81.1% single-agent, 92.2% sequential, 54.4% hierarchical) may differ if the architectures were to run on unseen tasks introducing likely different results when they rely more on reasoning rather than memorization. Replication on novel functions or on private code repositories is required before treating the absolute or relative performance figures as representative of real-world test generation tasks.

Furthermore, the study evaluated each architecture under standardized, non optimized prompt conditions which can also be framed as an external validity threat. Companies in industry usually spend money on architecture-specific prompt engineering and configuration optimization services, the observed performance characteristics, especially the performance gaps, observed for hierarchical coordination might not accurately represent the maximum performance expected in optimized production deployments. This limits the direct application of these findings to business environments.

7.3 Construct Validity

The **test failure rate** metrics measure the amount of test failed to test generated, which measures the internal consistency between the agents generated solution and its generated tests rather than measures the error tolerance or error recovery the capabilities directly. A low test failure rate shows that the agents test are consistent with its own solution, however it doesn't show if the agents test can recover effectively from edge cases during execution, handle unforeseen failures, or resume unsuccessful runs. This is an example of a construct validity limitation due to metrics **error handling** may suggest a more comprehensive capability than what is actually measured. Although the test failure rate is directly measurable from the provided data that we collect and covers an important aspect of error handling specific to test generation tasks. However, these metrics (test failure rate) do not capture the execution level error recovery behavior, but reflect an important aspect of error handling quality in the context of automated test generation. The agent that generates tests that do not match its solution, means it was failed to provide logical inconsistency across all its output, which leads to error handling failure at the output level. In the future, can solve this limitation by implementing a pipeline that can capture the error handling behavior during the execution level, to get more comprehensive evaluation of error handling capability.

The implementation of detailed software quality criteria as a single quantitative measurement introduces a potential threat. Reliability was limited to a binary pass/fail result, which does not account for test quality or degrees of incomplete accuracy beside execution that is successful. Token consumption from API cost was used to estimate the cost, which does not account for other deployment expenses, including infrastructure and human control. Wall-clock time was used to calculate end-to-end

latency, which is an external-factor threat such as API response fluctuation at the time of measurement

Agent communication turns were considered as the same, whatever the length or content of the message was. This could potentially overestimate or underestimate the true coordination load every turn. To mitigate these threats, all dependent variable measurements were automatically gathered through the experimental pipeline without human interaction. Agent turns were automatically recorded, token consumption and latency were provided by the LLM API, and the success rate has been evaluated empirically by TestEval's test execution mechanism. Because no human judgment was used in the measurement or categorization of results, this removes the possibility of individual bias in data extraction.

8

Conclusion

This chapter concludes the thesis by summarizing what was done and what was found, articulating the contributions to the field, reflecting on the role of human developers in agentic workflows, and outlining directions for future work.

8.1 Summary of Work and Findings

This thesis investigated the architectural patterns and trade-offs of multi-agent LLM-based systems through a mixed-methods empirical study. The investigation was structured around three research questions, each addressed by a distinct phase of work.

8.1.1 RQ1: Architectural Pattern Taxonomy

In the first phase, we conducted a literature review on multi-agent systems to identify common patterns reported in research and practice. Three patterns stood out for us; the first was the single-agent baseline. It served as our reference point since it had the simplest features and the most basic architecture. Second, the sequential pipeline (Planner -> Coder -> Tester), a structured pattern representing procedural workflows. Finally, the hierarchical architecture with a supervisor orchestrating Analyst and Writer workers. These selected patterns covered wide feature variation of multi-agent design space such as control flow (top-down versus pipelined), role specialization (none, role-based, or supervisor-mediated), and information flow (single-context, artifact-handoff, or supervisor-mediated). Chapter 5 (table 5.1) presented this taxonomy in a detailed table.

8.1.2 RQ2: Quantitative Comparison

In the second phase, a controlled experiment was conducted, running all three patterns against 30 TestEval tasks to generate unit tests. Each architecture was run three times independently, producing 90 records per pattern, resulting in substantially different measured outcomes. The sequential pipeline achieved the highest pass rate at 92.2% (83 of 90) and the highest cross run consistency at 83% (25 of 30 tasks producing the same outcome across runs). The single-agent baseline achieved an 81.1% pass rate (73 of 90) with 67% cross-run consistency. The hier-

archical architecture achieved only 54.4% pass rate (49 of 90) with 47% cross-run consistency. The cost-quality relationship was strongly asymmetric: sequential's 1.2× token cost relative to single-agent produced a 11.1 percentage-point pass-rate gain, while hierarchical's 4.6× token cost produced a 26.7 percentage-point pass-rate loss. Task difficulty did not differentially affect the architectures: pass rates were nearly identical between medium and hard tasks within each pattern, indicating that the observed differences are driven by architectural properties rather than task complexity.

8.1.3 RQ3: Qualitative Analysis

In the final phase, we applied a direct content analysis to the agent interaction logs. We selected 22 of the 30 tasks (66 coded cases) to perform our qualitative analysis. Six behavioral categories emerged from the analysis described in Chapter 5 (table 5.3). *Iterative Self-Correction* was the dominant category in the single-agent baseline with 16 cases, a reactive correction mechanism that succeeds for small errors but fails when the agent fundamentally misunderstands the algorithm under test. The sequential architecture was the most successful pattern in terms of pass rate, three cases were categorized as *Plan-Guided Generation*, and 18 cases as *Productive specialization*. The planner agent reasoned carefully by pre-generating and pre-verifying expected values so that subsequent agents could translate the structured reasoning into correct tests. The hierarchical pattern was successful in 12 cases only, and the 10 remaining cases failed due to *A Supervisor Information Bottleneck*. The supervisor mediated all inter-agent communication and failed to transfer artifacts between agent efficiently, causing repeated unproductive re-delegation cycles. Two further categories *Assertion Hallucination* (6 cases) and *Error Propagation* (1 case) captured failure modes that, while present, were less dominant than the architecture-specific patterns.

8.2 Contributions

This thesis makes three principal contributions to the literature on multi-agent LLM systems for software engineering.

The first contribution is the empirical comparison of three architectural patterns under controlled conditions on one task type using one LLM model and a single benchmark. Although prior literature has studied various multi-agent design patterns [7] and proposed individual frameworks such as AgentMesh and MetaGPT, few studies have conducted controlled experiments running identical tasks under the same conditions across multiple architectural patterns. Our experiment provided a comparison running identical tasks with the same LLM (Claude Sonnet 4), utilizing a consistent temperature setting and conducting three independent runs per architecture, yielding a total of 270 recordings for analysis. The records and their corresponding metrics resulted in empirical reference point which on subsequent studies can be built on, replicate, or contradict.

The second contribution is the identification of the Supervisor Information Bottleneck pattern. We observed and characterized a pattern that extends a prior theoretical work, [7, 34] presented this as a coordination and communication overhead framed as an efficiency limitation. The hierarchical pattern consumed 4.6x tokens compared to the single-agent, achieving a 26.7 percentage points lower pass rate indicating that the supervisor recycle runs resulted in higher token consumption trying to mediate the communication between agents unsuccessfully.

Finally, the architecture guideline table was grounded in observed agent behavior. Section 6.4 synthesizes the findings into a decision table mapping task properties to recommended architectural patterns. These guidelines extend the more general design considerations in the prior literature by grounding them in observed agent behavior rather than purely theoretical properties, providing practitioners with a concrete starting point for architectural decisions.

8.3 Future Work

Two extensions of the empirical work follow most directly from the findings.

8.3.1 Replication with a second LLM

In this study, we used the same LLM model(Claude Sonnet 4) a mid-priced US-developed model, for evaluating all the three architecture patterns, without a role specialization across agents. While implementing the same model for all three patterns, we can ensure that any observation differences in performance and reliability are related to the architecture patterns itself rather than differences in models capabilities. This means that our findings reflect the relationships between the Claude model and the coordination mechanism for each of the three architecture patterns, though it remains unknown whether these differences hold across other LLMs. In the future, researchers can replicate this study using another LLM model such as Mistral Large, GPT-4o, or Llama 3, including the same tasks, prompt configurations, and evaluation measures. First, check if the Supervisor Information Bottleneck seen in hierarchical coordination is unique to Claude Sonnet 4's response to supervisor-mediated delegation or whether it is present in other models as well. Second, check whether the patterns observed for the single-agent baseline, sequential, and hierarchical configurations are specific to Claude Sonnet 4 or consistent across the model landscape.

8.3.2 Extending to other software engineering tasks

The evaluation in this study focuses on automated test case generation. Performance differences between single-agent, sequential, and hierarchical patterns may not apply to other SE tasks. Such as requirements engineering as studied in the MAAD and NOMAD frameworks[24][13]. And architecture design, bug repair, and code review. These tasks differ from one another in terms of inter-agent interdependence and the complexity of reasoning. Future researchers could replicate this study across a

wider range of SE tasks. In order to establish whether the findings generalize beyond generation-oriented tasks, and to determine which coordination architecture is most suitable for each phase of the software development life-cycle.

Bibliography

[1] LeetCode. https://leetcode.com/.

[2] Mohammed Agha and Ayah Miqdad. An empirical comparison of multi-agent LLM architectural patterns for automated unit test generation. https://doi.org/10.5281/zenodo.20309242, May 2026. Replication package (source code, prompts, results, dashboard).

[3] Anthropic. Claude sonnet 4. https://www.anthropic.com/news/claude-4, 2025. Large language model.

[4] Eranga Bandara, Ross Gore, Peter Foytik, Sachin Shetty, Ravi Mukkamala, Abdul Rahman, Xueping Liang, Safdar H. Bouk, Amin Hass, Sachini Rajapakse, Ng Wee Keong, Kasun De Zoysa, Aruna Withanage, and Nilaan Loganathan. A practical guide for designing, developing, and deploying production-grade agentic AI workflows. https://doi.org/10.48550/arXiv.2512.08769, 2025.

[5] Ajay Bandi, Bhavani Kongari, Roshini Naguru, Sahitya Pasnoor, and Sri Vidya Vilipala. The rise of agentic AI: A review of definitions, frameworks, architectures, applications, evaluation metrics, and challenges. *Future Internet*, 17(9):404, 2025. https://doi.org/10.3390/fi17090404.

[6] Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3(2):77–101, 2006. https://doi.org/10.1191/1478088706qp063oa.

[7] Yangxiao Cai, Ruiyin Li, Peng Liang, Mojtaba Shahin, and Zengyang Li. Designing LLM-based multi-agent systems for software engineering tasks: Quality attributes, design patterns and rationale. https://doi.org/10.48550/arXiv.2511.08475, 2025.

[8] Zehang Deng, Yongjian Guo, Changzhou Han, Wanlun Ma, Junwu Xiong, Sheng Wen, and Yang Xiang. AI agents under threat: A survey of key security challenges and future pathways. *ACM Computing Surveys*, 57(7), February 2025. https://doi.org/10.1145/3716628.

[9] Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and

Ge Li. A survey on code generation with LLM-based agents. https://doi.org/10.48550/arXiv.2508.00083, 2025.

[10] Adam Fourney, Gagan Bansal, Hussein Mozannar, Cheng Tan, Eduardo Salinas, Erkang Zhu, Friederike Niedtner, Grace Proebsting, Griffin Bassman, Jack Gerrits, Jacob Alber, Peter Chang, Ricky Loynd, Robert West, Victor Dibia, Ahmed Awadallah, Ece Kamar, Rafah Hosn, and Saleema Amershi. Magentic-one: A generalist multi-agent system for solving complex tasks. https://doi.org/10.48550/arXiv.2411.04468, 2024.

[11] Anusha Garlapati, M N V Satya Sai Muni Parmesh, Savitha, and Jaisri S. AI-powered multi-agent framework for automated unit test case generation: Enhancing software quality through LLMs. https://doi.org/10.1109/GCAT62922.2024.10923987, 2024.

[12] Vahid Garousi, Michael Felderer, and Mika V. Mäntylä. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology*, 106:101–121, 2019. https://doi.org/10.48550/arXiv.1707.02553.

[13] Polydoros Giannouris and Sophia Ananiadou. NOMAD: A multi-agent LLM system for UML class diagram generation from natural language requirements. https://doi.org/10.48550/arXiv.2511.22409, 2025.

[14] Robert L. Glass, Iris Vessey, and Venkataraman Ramesh. Research in software engineering: an analysis of the literature. *Information and Software Technology*, 44(8):491–506, 2002. https://doi.org/10.1016/S0950-5849(02)00049-6.

[15] Sander Greenland, Stephen J. Senn, Kenneth J. Rothman, John B. Carlin, Charles Poole, Steven N. Goodman, and Douglas G. Altman. Statistical tests, p values, confidence intervals, and power: a guide to misinterpretations. *European Journal of Epidemiology*, 2016. https://pmc.ncbi.nlm.nih.gov/articles/PMC4877414/.

[16] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges. https://doi.org/10.48550/arXiv.2402.01680, 2024.

[17] Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. Metagpt: Meta programming for a multi-agent collaborative framework. https://doi.org/10.48550/arXiv.2308.00352, 2024.

[18] Hsiu-Fang Hsieh and Sarah E. Shannon. Three approaches to qualitative content analysis. *Qualitative Health Research*, 15(9):1277–1288, 2005. https://doi.org/10.1177/1049732305276687.

[19] Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. https://doi.org/10.48550/arXiv.2312.13010, 2024.

[20] Mengshuo Jia, Zeyu Cui, and Gabriela Hug. Enhancing LLMs for power system simulations: A feedback-driven multi-agent framework. *IEEE Transactions on Smart Grid*, 16(6), 2025. https://doi.org/10.1109/TSG.2025.3589114.

[21] Sourena Khanzadeh. AgentMesh: A cooperative multi-agent generative AI framework for software development automation. https://doi.org/10.48550/arXiv.2507.19902, 2025.

[22] LangChain Inc. LangGraph: Build resilient language agents as graphs. https://github.com/langchain-ai/langgraph, 2024. Open-source framework for building stateful multi-agent LLM applications.

[23] Hui Yi Leong, Yuheng Li, Yuqing Wu, Wenwen Ouyang, Wei Zhu, Jiechao Gao, and Wei Han. AMAS: Adaptively determining communication topology for LLM-based multi-agent system. https://doi.org/10.48550/arXiv.2510.01617, 2025.

[24] Ruiyin Li, Yiran Zhang, Xiyu Zhou, Peng Liang, Weisong Sun, Jifeng Xuan, Zhi Jin, and Yang Liu. MAAD: Automate software architecture design through knowledge-driven multi-agent collaboration. https://doi.org/10.48550/arXiv.2507.21382, 2025.

[25] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large language model-based agents for software engineering: A survey. https://doi.org/10.48550/arXiv.2409.02977, 2024.

[26] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. LLM-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. *ACM Transactions on Software Engineering and Methodology*, 34(124):1–30, 2025. https://doi.org/10.1145/3712003.

[27] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. AgentBench: Evaluating LLMs as agents. https://doi.org/10.48550/arXiv.2308.03688, 2023.

[28] Yue Liu, Sin Kit Lo, Qinghua Lu, Liming Zhu, Dehai Zhao, Xiwei Xu, Stefan Harrer, and Jon Whittle. Agent design pattern catalogue: A collection of architectural patterns for foundation model based agents. https://doi.org/10.48550/arXiv.2405.10467, 2024.

[29] Zoran Milosevic and Fethi Rabhi. Architecting agentic communities using de-

sign patterns: A framework grounded in ODP enterprise language formalism. https://doi.org/10.48550/arXiv.2601.03624, 2026.

[30] David J. Moore. A taxonomy of hierarchical multi-agent systems: Design patterns, coordination mechanisms, and industrial applications. https://doi.org/10.48550/arXiv.2508.12683, 2025.

[31] Anh Nguyen-Duc, Beatriz Cabrero-Daniel, Adam Przybylek, Chetan Arora, Dron Khanna, Tomas Herda, Usman Rafiq, Jorge Melegati, Eduardo Guerra, Kai-Kristian Kemell, Mika Saari, Zheying Zhang, Huy Le, Tho Quan, and Pekka Abrahamsson. Generative artificial intelligence for software engineering—a research agenda. *Software: Practice and Experience*, 55(11):1806–1843, 2025. https://doi.org/10.1002/spe.70005.

[32] Juliana Gomes Ribeiro, Joel Machado Pires, Jonas Oliveira Pereira, Frederico Araujo Durão, and Célia Ghedini Ralha. Software development using a multi-agent approach. https://doi.org/10.5753/wesaac.2025.37549, 2025.

[33] Ranjan Sapkota, Konstantinos I. Roumeliotis, and Manoj Karkee. Ai agents vs. agentic ai: A conceptual taxonomy, applications and challenges. *Information Fusion*, 126:103599, February 2026. https://doi.org/10.1016/j.inffus.2025.103599.

[34] Nadia Sibai, Yara Ahmed, Serry Sibaee, Sawsan AlHalawani, Adel Ammar, and Wadii Boulila. The path ahead for agentic ai: Challenges and opportunities. https://doi.org/10.48550/arXiv.2601.02749, 2026.

[35] Klaas-Jan Stol and Brian Fitzgerald. The ABC of software engineering research. *ACM Transactions on Software Engineering and Methodology*, 27(3), September 2018. https://doi.org/10.1145/3241743.

[36] Chuanneng Sun, Songjun Huang, and Dario Pompili. LLM-based multi-agent decision-making: Challenges and future directions. *IEEE Access*, 2025. https://doi.org/10.1109/LRA.2025.3562371.

[37] Stevan Tomic, Emil Alégroth, and Maycel Isaac. Evaluation of the choice of LLM in a multi-agent solution for GUI-test generation. https://doi.org/10.1109/ICST62969.2025.10989038, 2025.

[38] Khanh-Tung Tran, Dung Dao, Minh-Duong Nguyen, Quoc-Viet Pham, Barry O'Sullivan, and Hoang D. Nguyen. Multi-agent collaboration mechanisms: A survey of llms. https://doi.org/10.48550/arXiv.2501.06322, 2025.

[39] Song Wang, Zhen Tan, Zihan Chen, Shuang Zhou, Tianlong Chen, and Jundong Li. AnyMAC: Cascading flexible multi-agent collaboration via next-agent prediction. https://doi.org/10.48550/arXiv.2506.17784, 2025.

[40] Wenhan Wang et al. TestEval: Benchmark for LLM-based test case genera-

tion. https://github.com/LLM4SoftwareTesting/TestEval, 2024. GitHub repository.

[41] Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. TestEval: Benchmarking large language models for test case generation. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 3547–3562, 2025. https://doi.org/10.48550/arXiv.2406.04531.

[42] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversations. https://doi.org/10.48550/arXiv.2308.08155, 2024.

[43] Bingyu Yan, Zhibo Zhou, Litian Zhang, Lian Zhang, Ziyi Zhou, Dezhuang Miao, Zhoujun Li, Chaozhuo Li, and Xiaoming Zhang. Beyond self-talk: A communication-centric survey of LLM-based multi-agent systems. https://doi.org/10.48550/arXiv.2502.14321, 2025.

[44] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. https://doi.org/10.48550/arXiv.2210.03629, 2023.

A

Appendix