Project Final Report

Lakshan S. N. – IT21800900

Supervisor: Prof. Nuwan Kodagoda
Co-supervisor: Dr. Lakmini Abeywardhana

BSc (Hons) in Information Technology Specializing
in Software Engineering

Sri Lanka Institute of Information Technology
Faculty of Computing
Department of Software Engineering

August 2025

GENERATIVE AI-BASED CHATBOT FOR EMPLOYEE AND CUSTOMER SUPPORT AUTOMATION

Lakshan S. N.

IT21800900

BSc (Hons) in Information Technology Specializing in Software Engineering

Department of Software Engineering

Sri Lanka Institute of Information Technology

August 2025

1. Declaration

I declare that this is my own work, and this dissertation does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any other university or Institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text. Also, I hereby grant to Sri Lanka Institute of Information Technology the non-exclusive right to reproduce and distribute my dissertation in whole or part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as article or books).

Signature: Date:

Signature of the Supervisor: Date:

2. Abstract

Multi-agent system (MAS) frameworks have attracted growing attention in recent years, yet most continue to face significant limitations that restrict their use beyond basic demonstrations. Common challenges include translating high-level goals into concrete and executable tasks, managing interdependent subtasks, coordinating agents with varied capabilities, and ensuring reliable performance across distributed environments. Failures in one agent can cascade through the system, undermining overall stability. Security adds further complexity, as many frameworks rely on third-party services that process sensitive data, raising concerns about confidentiality and trust.

This report introduces a new architecture intended to address these limitations in a systematic way. The design integrates quantized local language models with modular Model Context Protocol (MCP) servers to provide a foundation for privacy-preserving, reliable, and extensible multi-agent systems. By running language models locally in quantized form, external exposure is minimized, reducing risks associated with data leakage or vendor dependency. At the same time, modular MCP servers supply structured interfaces that agents can use to coordinate, communicate, and exchange information without tightly coupled dependencies.

The orchestration layer plays a central role in this architecture. It provides mechanisms for structured task decomposition, dependency tracking, and robust fault handling. Agents operating under this framework can pursue objectives collaboratively while retaining flexibility in execution strategies. Failures are isolated, dependencies are managed explicitly, and coordination remains transparent, even in distributed deployments.

The purpose of this work is to demonstrate that multi-agent systems can achieve both reliability and confidentiality without sacrificing extensibility. By combining the efficiency of quantized models with the modularity of MCP servers, the proposed framework delivers a practical foundation for advancing agentic systems beyond the constraints of existing approaches, while ensuring that security and control remain central to design.

Key Words: *multi-agent systems, orchestration, distributed execution, privacy-preserving computation, quantized local models.*

3. Acknowledgement

I wish to express my sincere gratitude to my supervisor, **Prof. Nuwan Kodagoda**, and co-supervisor, **Dr. Lakmini Abeywardhana**, for their invaluable guidance and continuous support throughout this research. I am also grateful for the opportunity to work on a real-world industry project with **LOLC**, which they facilitated.

Table of Contents

4. List of Figures

5. List of Tables

6. List of Abbreviations

IT - Information Technology
AI - Artificial Intelligence
LOLC - Lanka ORIX Leasing Company
MAS – Multi Agentic System
LLM - Large Language Model
RAG - Retrieval-Augmented Generation
NLTK - Natural Language Toolkit
NER - Named entity recognition
UI - User Interface
API - Application programming interface

1. Introduction

Multi-agent system (MAS) frameworks have attracted growing attention in recent years, yet most continue to face significant limitations that restrict their use beyond basic demonstrations. Common challenges include translating high-level goals into concrete and executable tasks, managing interdependent subtasks, coordinating agents with varied capabilities, and ensuring reliable performance across distributed environments. Failures in one agent can cascade through the system, undermining overall stability. Security adds further complexity, as many frameworks rely on third-party services that process sensitive data, raising concerns about confidentiality and trust.

This report introduces a new architecture intended to address these limitations in a systematic way. The design integrates quantised local language models with modular Model Context Protocol (MCP) servers to provide a foundation for privacy-preserving, reliable, and extensible multi-agent systems. By running language models locally in quantized form, external exposure is minimised, reducing risks associated with data leakage or vendor dependency. At the same time, modular MCP servers supply structured interfaces that agents can use to coordinate, communicate, and exchange information without tightly coupled dependencies.

The orchestration layer plays a central role in this architecture. It provides mechanisms for structured task decomposition, dependency tracking, and robust fault handling. Agents operating under this framework can pursue objectives collaboratively while retaining flexibility in execution strategies. Failures are isolated, dependencies are managed explicitly, and coordination remains transparent, even in distributed deployments.

The purpose of this work is to demonstrate that multi-agent systems can achieve both reliability and confidentiality without sacrificing extensibility. By combining the efficiency of quantized models with the modularity of MCP servers, the proposed framework delivers a practical foundation for advancing agentic systems beyond the constraints of existing approaches, while ensuring that security and control remain central to design.

6.1 Background Literature

Over the last ten years, the development of artificial intelligence has moved steadily toward more agentic and embodied forms of development. This direction started with the landmark introduction of the transformer architecture in Google's 2017 paper, titled Attention Is All You Need [1], which replaced the previous recurrent models with an attention-based mechanism that allowed large-scale language modelling over long contexts. As a result, the transformer model contributed to the development of large language models, which now define the space of generative AI with their ability to generate coherent text, images, and other types of media.

Alongside these services, the growth of open-source models has expanded the field. Projects such as GPT-OSS[2], Mistral[3], and Meta's LLaMA[4] provide researchers and developers with alternatives that can be studied and modified . Open-source availability has allowed faster experimentation and broader transparency compared to closed systems. Platforms like Hugging Face support this ecosystem by offering model repositories, datasets, and tools that encourage reproducibility and community-driven improvements. This open ecosystem also encourages collaboration between academic groups, startups, and independent developers, widening access to advanced AI beyond a handful of major corporations.

Another significant development is the ability to run models directly on local machines. Tools like Ollama make this possible by providing simple command-line interfaces and REST APIs to manage local inference [5]. This approach offers more privacy and control, since data does not need to leave the device. For many users, local inference also reduces dependency on commercial providers, ensuring that applications remain usable even if access to cloud services is limited. Local execution also lowers operational costs, as running models on personal or organizational hardware can remove recurring subscription fees.

The advance of quantization techniques has made this more practical. By reducing the precision of model weights, quantized models require less memory and compute power while maintaining acceptable accuracy [6]. This technical shift enables mid- to high-tier consumer computers, equipped with either CPUs or GPUs, to run models that previously required specialized infrastructure. In practice, this means that personal laptops, desktops, and even edge devices are increasingly able to handle tasks that were once reserved for large data centers.

Together, these trends suggest a future where closed platforms, open-source projects, and local inference coexist. Rather than one model of access dominating, the ecosystem is likely to remain diverse: some users will rely on commercial providers for scale and stability, while others will turn to open or local systems for transparency, customization, and independence. This coexistence shapes how AI is accessed and how it is controlled and governed across different contexts.

Recent years have seen the rise of experimental AI agents that extend beyond text generation and move closer to autonomous assistance. Google has introduced projects such as Jules, which links directly to GitHub repositories in order to identify and fix bugs [7]. This shows how an agent can interact with live

codebases, perform reasoning over software projects, and make targeted changes. While still experimental, systems like this hint at how agentic AI could integrate into established developer workflows.

In parallel, companies have begun shaping business models around coding agents. Microsoft's Copilot, available inside Visual Studio Code and other platforms, provides real-time code suggestions and automation [8]. It demonstrates a commercial path where generative capabilities are embedded into everyday tools, lowering development effort and streamlining productivity. Copilot remains tied to subscription pricing, reflecting a shift where AI assistance is treated as a professional service.
Other experimental platforms, including Bolt and Cursor, explore how LLM-driven loops can be used to create software more independently [9], [10]. These agents do not simply suggest snippets but aim to generate, test, and refine larger portions of code. They employ different systematic approaches to reach a similar goal: software production with less direct human input. Paid tiers remain common here, showing a balance between experimentation and commercial sustainability.

Together, these examples illustrate the transition into an agentic world. Current systems are still limited in autonomy and reliability, but they mark a clear step forward from traditional generative tools. Each approach bug-fixing agents, in-editor copilots, or autonomous software builders offers a different vision of how agents may evolve, signaling the varied directions of research and industry interest [11].

In the current exploration of agentic AI, several frameworks have been proposed to manage the complexity of multi-agent systems. One of the most visible examples is AutoGen by Microsoft [12]. Its architecture revolves around a conversation-centric message bus, where agents communicate through structured exchanges rather than ad-hoc calls. This approach highlights dialogue as the organizing principle. It allows flexible orchestration of multiple agents while keeping their interactions transparent and modular. Such a design benefits research and enterprise use by making coordination easier to monitor and control.
Another evolving framework is CrewAI[13], which emphasizes collaboration through specialized roles . In this system, each agent is assigned a position—such as researcher, planner, or executor and the architecture is built to manage how these roles coordinate. The advantage of this role-based setup is clarity: tasks can be distributed according to capability, and workflows remain predictable. This systematic division is especially useful for projects that require sustained cooperation between different types of reasoning or action.

By contrast, LangGraph takes a graph-based approach [14]. Instead of treating agents as conversational peers or role-bound workers, LangGraph represents them as nodes within a structured graph. The graph defines how information flows, how decisions branch, and how processes converge. This structure offers fine control over dependencies and execution order. It is particularly suited for tasks where outcomes must follow complex paths, as it captures both state and logic in a way that is easier to visualize and extend.
These frameworks illustrate distinct architectural philosophies: conversation as a bus, roles as a structure, and graphs as a workflow map. Each provides its own strengths for coordination, transparency, or control. In contrast, the research presented here explores a design built without reliance on existing frameworks. The intention is to understand the principles of agentic orchestration more directly and to shape an architecture tailored to specific needs, rather than adapting to the constraints of a pre-defined system.

6.2 Research Gap

	Customization	*Use Case*	*LLM Types*	*inter-service communication*
Proposed System	Very High	Multi-step Workflow For the Given Goal	Very Flexible	Yes
Auto GPT	Low	Open-Ended Conversational Flows	Extensive	No
Crew AI	Low	Collaborative Tasks	Extensive	No

Table 1 - Comparison Chart Between Proposed System With Auto GPT and Crew AI

The current landscape of experimental agentic AI systems has made significant progress but also shows several limitations that restrict their use in more complex or practical situations. Research prototypes like AutoGPT and CrewAI represent early steps toward autonomous workflows. These systems show that large language models can be configured as agents capable of following open-ended instructions, reasoning through multiple steps, and interacting with external tasks. However, both remain proof-of-concept tools that lack the depth of customization, control, and communication needed for reliable use in structured environments.

AutoGPT is widely known as one of the first efforts to create an autonomous system based on large language models. Its design focuses on open-ended conversational exchanges, where the model interprets broad instructions and produces iterative responses to achieve loosely defined goals. While this approach effectively showcases autonomous interaction, it is less suitable for environments that require generating and following precise workflows. Customization in AutoGPT is limited, with minimal ability to adapt the system to specific organizational needs or domain-specific tasks. Its support for large language models is broad, as it can connect to various back-end providers, but this flexibility does not extend to designing workflows or making structural changes. Importantly, AutoGPT lacks built-in mechanisms for interservice communication, meaning it cannot directly interact with external systems, knowledge bases, or APIs without additional custom development by the user.

CrewAI adopts a different approach by emphasizing collaborative tasks performed by multiple interacting agents. The idea of agent collaboration is valuable, especially when tasks can be broken into subtasks assigned to specialized roles. However, in practice, CrewAI faces similar limitations to AutoGPT. Its design remains fairly rigid, offering limited customization in how workflows are created and executed. Although the system can show coordination among agents, it doesn't provide ways to customize workflows to meet a user's specific goals. Its deep LLM integration is still mainly conversational and task-focused, lacking more advanced programmatic abstraction. Like AutoGPT, CrewAI does not support interservice communication, which limits its effectiveness in scenarios requiring interaction with databases, APIs, or external knowledge systems.

Taken together, AutoGPT and CrewAI illustrate the state of experimental agentic AI research: promising, but constrained. Both provide insight into how large language models can be used as autonomous agents,

yet neither system resolves the practical challenges of goal alignment, customizability, and integration. In effect, these limitations highlight the research gap that this work seeks to address.

The proposed system addresses these gaps through a multi-agent architecture deliberately built from scratch to simplify unnecessary complexity while enabling extensive customization. Instead of depending on static conversational flows or loosely defined collaborations, the system allows users to give goal-oriented instructions, which are then converted internally into a multi-step workflow via task decomposition. This approach ensures that each input is not only processed but also expanded into a structured plan, with subtasks assigned to agents specialized for those functions. By automating workflow creation this way, the system avoids the fragility of open-ended conversational loops and instead offers purpose-driven execution aligned with the specific needs.

In terms of customization, the proposed system is designed to be very high compared to AutoGPT and CrewAI. Users and developers can configure how tasks are decomposed, assign roles to agents, and tailor workflows according to the unique needs of a project or domain. This level of adaptability is possible because the system is not simply a wrapper around an existing large language model but rather a full framework that integrates agents, tools, and workflows under a unified structure. By abstracting the underlying complexity, it allows non-expert users to benefit from powerful agentic capabilities while still granting technical users the flexibility to extend and modify system behavior.

Flexibility in LLM usage is another key improvement. Instead of tying the architecture to a single provider or only offering conversational interfaces, the system uses predefined templates that enable easy integration with popular large language models. This method allows developers to choose the best model for each task without needing to redesign workflows or fully change infrastructures. In practice, this flexibility lowers reliance on one vendor and makes the system more adaptable to the fast-changing AI landscape, where new models and updates are frequently released.

Perhaps the most significant gap addressed is the inclusion of interservice communication. Where AutoGPT and CrewAI mainly operate independently, the proposed system is intentionally designed to connect with external services via attached APIs. For example, an agent within the system can be pre-created to interface with a knowledge base API, enabling agents to retrieve, validate, or update information during execution. This capacity to interact with other systems broadens the platform's utility beyond standalone reasoning, transforming it into a connected environment where multiple sources of data and services can work together. As a result, the system advances closer to practical deployment scenarios where AI agents must not only reason but also interact dynamically with organizational resources and external platforms.

The comparison between AutoGPT, CrewAI, and the proposed system highlights the unique contribution of this research. AutoGPT and CrewAI show that agentic behavior can be built around large language models, but they fall short in customization, workflow alignment, integration flexibility, and interservice communication. The proposed system addresses these specific shortcomings by combining a multi-agent framework, goal-driven task decomposition, flexible LLM templates, and built-in API communication. Together, these features create a platform that is not only experimentally validated but also practically adaptable, bridging the gap between research prototypes and applied systems.

This research gap can therefore be summarized along four dimensions. First, in customization, the proposed system offers high configurability, whereas existing systems provide limited options. Second, in use case, the system directly supports structured workflows aligned with specific goals, rather than open-ended or collaborative discussions. Third, in handling LLMs, the system achieves flexibility through programmatic templates, while existing prototypes only offer general access. Finally, in interservice communication, the system includes built-in API integration, a feature missing in AutoGPT and CrewAI. By combining these four advancements, the system paves a way for agentic AI research that moves beyond proof-of-concept demonstrations toward a more generalizable and extensible framework.

6.3 Research Problem

The evolution of AI from generative systems toward agentic and embodied forms introduces new technical challenges that remain unresolved in research and application. Existing prototypes such as AutoGPT and CrewAI show the possibility of autonomous workflows. Yet, their limited customization, lack of structured orchestration, and absence of interservice communication highlight gaps that still need to be addressed. The proposed multi-agentic system aims to close these gaps, but doing so requires addressing a series of core research problems. These problems span from coordination and orchestration of multiple agents to execution reliability, error handling, and architecture design that ensure both extensibility and security.

6.3.1 How to effectively coordinate multiple AI agents with different capabilities

A fundamental challenge in building a multi-agentic system is coordination. Unlike single-agent approaches, a multi-agent architecture must manage agents with different capabilities, responsibilities, and performance characteristics. This raises the problem of how to assign tasks appropriately so that each agent's strengths are used efficiently while avoiding overlap or conflict[15].

In practice, coordination involves questions such as: How should agent tasks be scheduled? How should communication occur between agents and the central orchestrator? How can redundancy be reduced without losing flexibility? Coordination also extends to resource usage, since multiple agents may need access to the same tools or APIs at the same time.

A related issue is the design of the orchestrator. The orchestrator acts as the central unit that allocates tasks, supervises workflows, and manages interactions between agents. However, its role raises additional questions: Should orchestration be centralized or distributed? How much autonomy should individual agents have compared to the orchestrator? Balancing these design decisions requires a careful examination of both architectural models and software engineering practices.

6.3.2 How to provide reliable execution in a distributed environment

Once coordination is established, the next research problem is ensuring reliable execution. Distributed environments introduce uncertainty: tasks may fail, APIs may become unavailable, or external services may return inconsistent results. An agentic system must be robust enough to continue functioning even when parts of the environment degrade.

Reliability requires answers to several subproblems: How should agent tasks be planned so that dependencies are clear and recoverable? How should execution logs be maintained to allow rollback or re-execution? What strategies can ensure that results are reproducible even when using probabilistic models? These questions point toward the need for strong workflow management practices, where the orchestrator supervises not only which tasks are assigned but also how they are executed and validated.

This problem also raises the issue of architecture selection. For example, should the system follow a microservices model, where each agent is encapsulated and independently deployable, or a monolithic model, where coordination is tightly coupled? Each choice carries trade-offs for reliability, scalability, and debugging. The problem therefore involves not only building agents but also designing an architecture that minimizes points of failure while supporting flexibility.

6.3.3 How to handle failures and dependencies between agentic tasks

In any complex workflow, some tasks will depend on the successful completion of others. When one task fails, the downstream effects can disrupt the entire workflow. A third research problem is therefore how to handle failures and manage dependencies in agentic environments[16].

Key questions emerge: How should dependencies between tasks be represented in the system? How should the orchestrator respond when a dependent task fails? Should the system retry, reassign the task to another agent, or escalate the error to the user? These are not trivial questions, since failure-handling strategies affect both reliability and efficiency.

This problem also connects to the issue of software engineering design patterns. Mechanisms such as retry policies, circuit breakers, or compensation workflows—common in distributed systems—must be adapted to the specific needs of agentic AI. A system that cannot gracefully handle failures risks producing incomplete results or misleading outputs, undermining user trust. Managing dependencies carefully is therefore not only a technical requirement but also a foundational aspect of creating a system that can be applied in real-world scenarios.

6.3.4 How the orchestrator decomposes high-level goals into executable tasks

Perhaps the most defining challenge of agentic AI is goal decomposition. Unlike traditional software, where tasks are explicitly defined by developers, an agentic system is expected to take a high-level goal provided by a user and transform it into a structured workflow of executable subtasks. This requires not only natural language understanding but also planning, sequencing, and mapping subtasks to available agents and tools[17].

The difficulty lies in designing an orchestrator that can perform this decomposition reliably. How should goals be analyzed to identify the minimal set of tasks required? How can workflows avoid redundancy or unnecessary steps? How should outputs be validated to ensure that the original goal has been met? These questions define the orchestration problem at the core of the system.

Additional subproblems extend from this. Task planning raises design issues about the architecture of the orchestrator itself: should it follow a hierarchical planning model, a rule-based model, or a learning-based model that adapts over time? Software engineering concepts such as modularity, abstraction, and separation of concerns must be applied carefully to keep the system maintainable. Beyond design, further questions arise about implementation choices: Which programming language provides the best trade-off between performance and developer productivity? How should APIs be integrated so that interservice communication is smooth yet secure? How can prompt injection and other security vulnerabilities be avoided?

Finally, the orchestrator must manage infrastructure handoff: moving from planning to execution across distributed services without losing context or control. This makes decomposition not just a cognitive task but a systems engineering problem.

6.4 Research Objectives

The primary objective of this research is to design and implement a flexible orchestration system for AI agents that can effectively coordinate multiple autonomous entities to accomplish complex objectives. This orchestration system acts as the central management layer, responsible for the allocation, monitoring, and successful completion of tasks in a distributed environment. By building such a system, the research aims to address the limitations of existing frameworks, which often lack modularity, fail to manage interdependencies effectively, and struggle with fault tolerance. The objective therefore focuses on establishing reliability, adaptability, and scalability within multi agent orchestration while ensuring that each agent can operate independently but still contribute to an overall coordinated workflow.

A core component of this objective is to enable the Orchestrator to translate high level goals into a sequence of executable tasks. Goals are often abstract and need to be refined into actionable subtasks that can be executed by specialized agents. The proposed system must provide a systematic decomposition process that identifies dependencies, assigns tasks to the most appropriate agents, and ensures correct sequencing. This decomposition not only improves efficiency but also prevents bottlenecks and conflicts that could otherwise arise in distributed operations. By formalizing goal to task decomposition, the orchestration layer transforms abstract intentions into structured execution plans, which ensures that collaboration among agents is both meaningful and efficient.

Another objective is to build a monitoring and core API framework that supports orchestration operations across agents. Monitoring involves real time tracking of agent activity, task progress, and system status, which is essential for detecting failures, handling exceptions, and ensuring reliability. The core API provides standardized communication interfaces, which guarantees that agents developed with different internal designs can still interact seamlessly with the orchestration system. This API driven design enhances modularity and encourages flexibility by enabling developers to integrate various types of agents, whether general purpose or highly specialized, into the overall workflow. The monitoring and API mechanisms therefore serve as the backbone of the Orchestrator, ensuring transparency, control, and adaptability in multi agent environments.

A further objective is to design robust task assignment and dependency management techniques. Once goals have been decomposed into subtasks, these tasks must be distributed among agents in a way that balances workloads and respects interdependencies. The Orchestrator must therefore manage a dynamic task queue, track execution status, and reassign tasks when agents fail or unexpected conditions occur. This ensures continuity of operations even in the presence of faults. In addition, the system must maintain a dynamic feedback loop, constantly updating dependent tasks and providing status reports to initiating processes. The outcome is a resilient orchestration framework that adapts to changing conditions, reduces the risk of cascading failures, and maintains steady progress toward overall goals.

In addition to task assignment, the objectives also emphasize enabling inter service communication between agents. For example, a knowledge base agent should be capable of interfacing with the system autonomously, while still maintaining effective coordination with other agents such as a reasoning agent or a planning agent. By supporting modular inter service communication, the system ensures that specialized agents can contribute expertise independently but still interact meaningfully within a larger workflow. This capability strengthens collaboration and opens the possibility for complex workflows to emerge from the interaction of autonomous but connected components.

Another specific objective is to ensure programmatic flexibility in integrating different types of large language models and AI agents. The Orchestrator should allow developers to select from a variety of models, either open source or commercial, by relying on predefined templates and abstracted interfaces. This flexibility is essential for adapting to rapid changes in the AI landscape, where new models frequently emerge and require evaluation. Furthermore, the system should be able to accommodate quantized models or locally deployed inference engines, ensuring that execution can remain private and efficient without reliance on external providers.

A further dimension of the objectives is security and reliability. Since multi agent orchestration inherently involves external communication, the system must ensure that agents cannot be exploited through adversarial prompts or malicious API interactions. This requires implementing measures such as prompt validation, controlled interfaces, and sandboxed execution environments. The goal is not only to preserve the integrity of the system but also to prevent risks such as data leakage, injection attacks, or unintended execution. Security therefore becomes a foundational element of the Orchestrator's design rather than an afterthought.

Finally, the objectives address infrastructure management and deployment considerations. The Orchestrator must be designed with portability in mind, ensuring that it can operate in different environments ranging from cloud-based clusters to local machines. Managing handoff between environments, handling scaling issues, and maintaining efficient resource allocation all form part of this broader infrastructure-related goal. By following sound software engineering practices such as modular architecture, design patterns, and separation of concerns, the system will be structured to support maintainability and future extension. The combined effect of these objectives is developing a system that advances the state of research in agentic AI by offering a reliable, adaptable, and secure orchestration framew

7. Methodology

7.1 Methodology

For the overall system design, we decided to adopt a microservice architecture, as it provides scalability, modularity, and fault isolation while supporting the integration of heterogeneous components. The system follows the API Gateway pattern, which enables a single entry point for all client interactions while routing requests internally to the appropriate microservices. This architectural decision simplifies communication, secures external access, and ensures that each component can evolve independently without disrupting the rest of the system. Figure X illustrates the overall architecture and shows the interconnections between the services, where the API Gateway acts as the mediator between the end user and the multiple services.

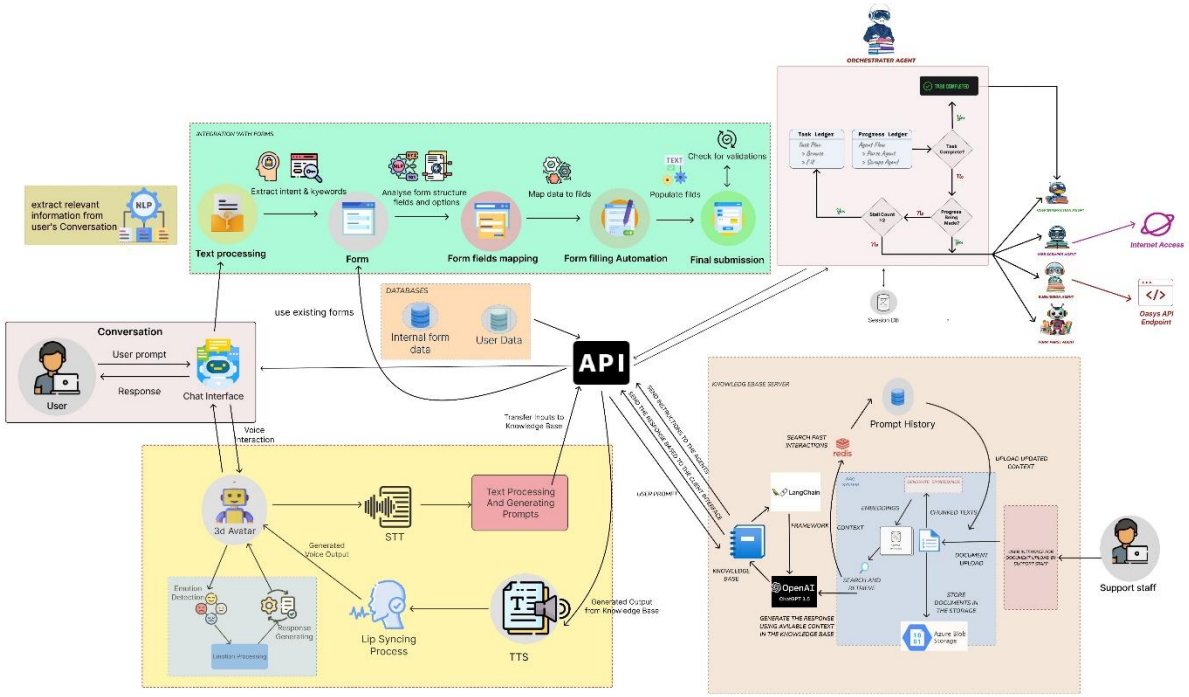

Figure 1 - Microservice Architecture with API Gateway

The system integrates several distinct components, each encapsulated as an independent microservice. These services interact with one another through lightweight communication protocols, ensuring that the architecture remains loosely coupled and highly flexible. A central feature of the design is the Multi-Agentic System (MAS) Component, which is connected to the end user through natural conversation. This component is supported by three other microservices that provide additional capabilities: the Form Integration Component, the Knowledgebase Component, and the 3D Avatar Component. Each plays a

specific role in enabling the orchestration of tasks, ensuring that the system operates seamlessly to achieve complex goals.

The Form Integration Component is responsible for extracting relevant information from the user's conversation. When a user engages in dialogue, the system must not only capture intent but also identify structured data that can be used in subsequent tasks. For example, when a user provides instructions, requirements, or parameters, this component transforms the unstructured text into structured data representations suitable for downstream processing. This design improves efficiency, avoids misinterpretation, and creates consistency in how user-provided information is handled across the system.

The Knowledgebase Component serves as a retrieval-augmented generation (RAG) based system that maintains a structured knowledge repository alongside conversation history. This allows the system to augment responses with factual information, preserve context across interactions, and provide continuity in long-running conversations. The knowledgebase acts as a memory layer, ensuring that the system does not rely solely on ephemeral states but instead leverages a persistent store of information. This persistence is essential in multi-agent workflows, where agents need to recall previous decisions, access stored knowledge, and align their outputs with the evolving goals of the user.

Another significant microservice is the 3D Avatar Component, which adds a layer of interactivity by enabling speech-driven communication and visual embodiment. This service integrates text-to-speech conversion with lipsyncing capabilities, allowing the avatar to animate responses in a natural and engaging manner. By linking conversational outputs from the MAS Component to the avatar, the system delivers a more immersive experience, where responses are not only textual but also auditory and visual. This integration makes interactions more human-like, bridging the gap between technical workflows and natural communication.

Together, these components form an interconnected ecosystem, each contributing specialized functionality while remaining modular and replaceable. The MAS Component stands at the center of this ecosystem, orchestrating multi-agent interactions and managing the decomposition of user goals into executable workflows. Unlike other services, the MAS is not narrowly scoped but instead functions as a dynamic control layer. It coordinates the activities of multiple agents, determines how tasks should be decomposed, and ensures that the right agent is assigned to the right task at the right time. This orchestration capability enables the system to go beyond simple responses and instead execute structured, multi-step workflows aligned with user goals.

In the design, the MAS also abstracts away underlying complexities, ensuring that users only need to specify high-level objectives without worrying about low-level task planning. For example, when a user specifies a complex requirement, the MAS decomposes it into subtasks, distributes them to specialized agents, monitors execution, and integrates the results into a coherent final output. This ability to transform abstract objectives into actionable workflows is the central innovation of the MAS Component. It enables not only efficiency but also adaptability, as the system can flexibly incorporate new tools, integrate additional APIs, and evolve its workflows without breaking its architecture.

The final part of this methodology focuses on the MAS Component in detail. While the supporting microservices provide input extraction, memory, and expressive interaction, the MAS provides the intelligence necessary for orchestrating the multi-agent process. It ensures coordination, maintains reliability in execution, and supports inter-service communication, such as querying the Knowledgebase API or invoking a task through the Form Integration service. By combining orchestration with modular microservices, the system is capable of handling distributed workflows in a structured yet flexible manner. This positions the MAS not only as a coordination layer but also as the backbone of a scalable multi-agentic architecture that aligns with the objectives of this research.

7.1.1 System Architecture

The architecture of the proposed system is structured into four primary layers, which are organized hierarchically to ensure clarity, modularity, and maintainability. The design follows a layered architectural style, where each layer has distinct responsibilities but also interacts seamlessly with the others through well-defined communication channels. Figure X presents an overview of the system's architecture, where the four layers—Client, API, Orchestration, and Core System Components—are arranged to demonstrate how user interaction flows through the system.

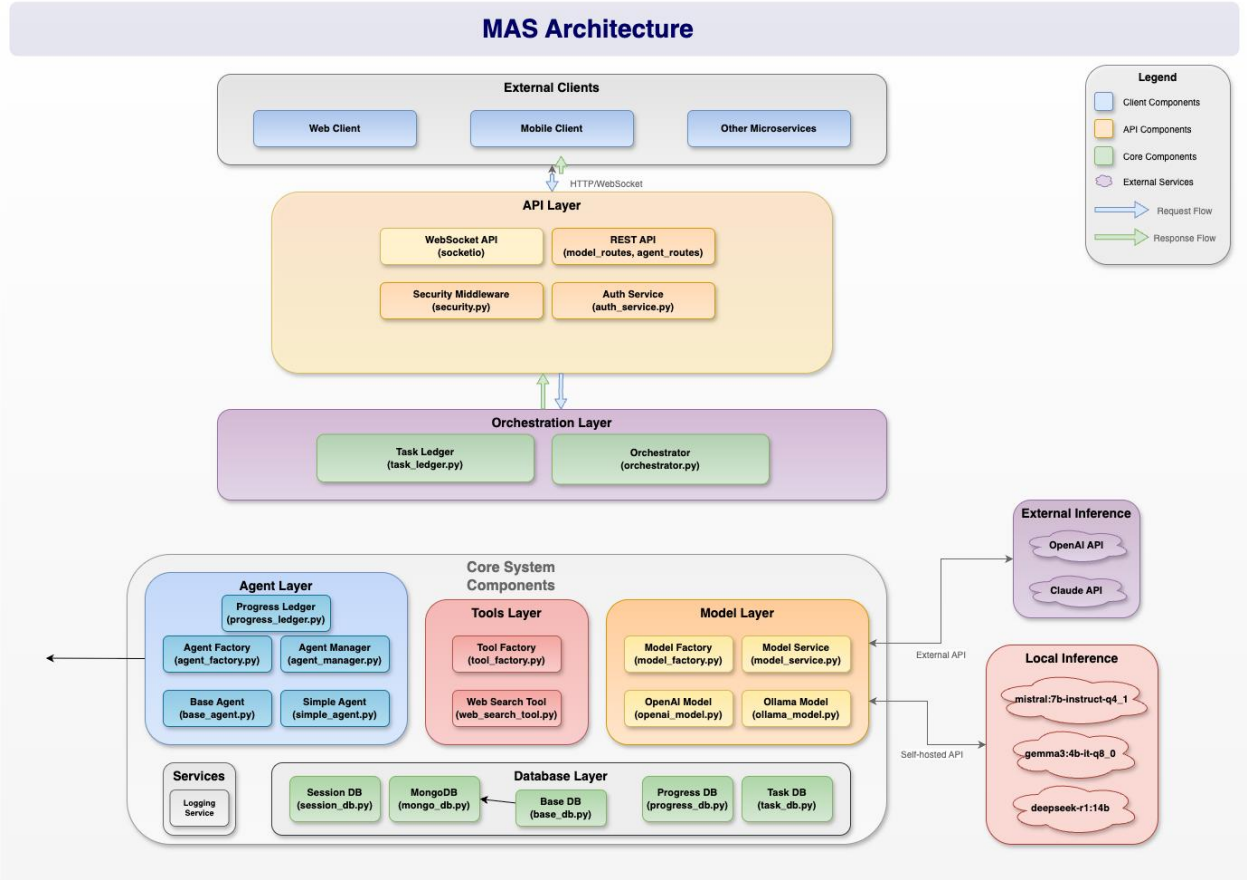

Figure 2 –Architectural Design Diagram of Multi Agent System Component

At the top of the architecture lies the **Client Layer**, which represents all possible points of interaction with the system. This layer primarily consists of end users who initiate requests or interact with the system through conversational interfaces. Additionally, it includes other microservices that can interact with the system programmatically via the API Gateway. By incorporating both human and machine clients, the system is not limited to direct user interactions but is also capable of integration with external services and applications. The Client Layer thus establishes the entry point into the system and highlights its openness for expansion and integration.

The **API Layer** is positioned beneath the Client Layer and acts as the system's secure entry channel. This layer exposes REST APIs that serve as a unified interface for all inbound requests, regardless of whether they originate from users or external services. A significant feature of the API Layer is its inclusion of authentication and security middleware. These mechanisms protect the system from unauthorized access, enforce access control policies, and ensure the integrity of transmitted data. By centralizing request handling at the API Layer, the architecture enforces a consistent interaction model while decoupling clients from the internal complexities of the system.

The **Orchestration Layer** follows as the central coordination point within the architecture. This layer contains two essential elements: the Orchestrator and the Task Ledger. The Orchestrator is responsible for managing workflows, distributing tasks to the appropriate components, and ensuring that complex requests are broken down into manageable units of work. In support of this, the Task Ledger functions as a persistent tracking mechanism that records task assignments, progress, and results. Together, these two elements provide a reliable orchestration framework, ensuring that tasks are not only executed efficiently but also monitored for accountability and fault recovery. This makes the Orchestration Layer a critical bridge between incoming requests and the underlying system functionalities.

At the foundation of the architecture is the Core System Components Layer, which encapsulates the system's internal intelligence and functionality. Within this layer reside several submodules, including Agents, Tools, Models, Services, and Databases. Each submodule has a distinct role yet contributes to the unified functioning of the system. The Agents serve as autonomous decision-making entities, responsible for processing instructions and engaging in multi-agent collaboration. The Tools represent functional extensions that agents can invoke to perform specific tasks, such as querying external systems or transforming data. Models provide the computational intelligence, with large language models (LLMs) accessed through Ollama integrations to support reasoning, dialogue, and knowledge augmentation. Services encapsulate reusable business logic, while Databases provide persistence for structured and unstructured data. Collectively, these components form the operational core of the system, ensuring that requests are executed with precision, memory, and scalability.

By structuring the system across these four layers, the architecture achieves a balance between modularity and integration. Each layer abstracts complexity from the one above it while exposing the necessary interfaces for communication. The Client interacts only with the API, the API delegates to the Orchestration Layer, and the Orchestration Layer coordinates with the Core Components to execute tasks. This hierarchy allows for a clean separation of concerns, supporting scalability and maintainability while also making it easier to expand individual layers without disrupting the entire system. For example, additional microservices can be added at the Client Layer without affecting the Orchestration Layer, or new tools and models can be integrated into the Core System Components without requiring redesign at the higher levels.

In summary, the layered system architecture provides both structural clarity and operational efficiency. It ensures that interactions flow logically from end users and external services down to the system's core intelligence, while each layer contributes specialized functionality. The following subsections will describe each component of the Core System Layer in greater detail, with emphasis on the orchestration and MAS-related design choices that differentiate this system from conventional approaches.

7.1.2 AgentFactory

The role of the AgentFactory is to provide a consistent and flexible way of creating new agent instances without scattering object construction logic across the system. Instead of hardcoding class instantiation in multiple places, the factory centralizes the responsibility and ensures that every agent is initialized with the correct configuration, references, and metadata. From an object-oriented perspective, the factory encapsulates the complexity of object creation while exposing a simple interface for clients. This approach follows the Factory Method pattern, where the creation of objects is delegated to a specialized component, allowing the rest of the codebase to remain decoupled from the specifics of which class is being created.

Internally, the AgentFactory maintains a registry of agent types, usually implemented as a dictionary that maps type identifiers to the corresponding class constructors. When a request comes in for an agent of a certain type, the factory looks up the class, initializes it with the required parameters such as database references or capability sets, and returns the ready-to-use instance. This registry-based design also makes the system extensible. Adding a new type of agent does not require rewriting existing logic but only registering the new agent class with the factory. This aligns with the Open/Closed Principle from SOLID design, since the factory is open for extension but closed for modification.

In practice, the AgentFactory also provides utilities to list all available agent types and sometimes to unregister or replace existing types. This becomes important in experimental or research contexts where agents evolve rapidly. The centralization of agent creation not only reduces code duplication but also provides a single point of control for logging, validation, and dependency injection during construction. In a multi-agent framework, this consistency is crucial for maintaining predictable behavior across a wide range of agent implementations.

7.1.3 AgentManager

While the AgentFactory is concerned with creation, the AgentManager is responsible for the overall lifecycle of agents and the orchestration of their tasks. It acts as a supervisory component that maintains a pool of active agents, delegates tasks to the appropriate instances, monitors execution, and records status updates. In an object-oriented design, the manager functions as a higher-level controller that coordinates many lower-level worker objects. This is an application of the Mediator pattern, where communication and coordination between agents are funneled through a single authority rather than being handled peer-to-peer.

The AgentManager keeps track of which agents are active, what tasks they are currently executing, and what resources they are using. This allows it to make informed decisions when assigning new tasks, avoiding overload on a single agent and balancing work across the system. Each assignment involves not only passing a task object to the agent but also registering the task with the system's databases and ensuring that progress can be monitored. If an agent fails or becomes unresponsive, the manager detects the issue and either retries the task with another agent or logs the failure for review. This fault-tolerant

supervision is particularly important in distributed and asynchronous environments where partial failure is common.

Design-wise, the AgentManager often collaborates closely with the Observer pattern. Agents emit status updates—such as “task started,” “in progress,” or “completed”—and the manager listens for these signals to update central logs or trigger further actions. In addition, the manager exposes higher-level APIs that the orchestrator or workflow components use to request work from agents. This separation of roles maintains a clean abstraction boundary: lower-level execution details stay within agents, while system-wide coordination is concentrated in the manager. By doing so, the design avoids tight coupling and preserves flexibility for future expansion.

7.1.4 ProgressLedger

The ProgressLedger is the persistent record keeper of agent activity. While the manager provides live coordination, the ledger ensures that every action is tracked and available for review, debugging, and workflow reconstruction. Conceptually, it is similar to an accounting book: each step taken by an agent is logged with time stamps, identifiers, status codes, and, when necessary, error traces. This allows developers and operators to trace the history of a task from assignment to completion, even if the system crashes or agents are restarted.

From an object-oriented standpoint, the ledger is designed as a dedicated service class that interacts with the database layer. It provides well-defined methods to log progress, retrieve historical records, summarize workflow status, and search for specific events. By encapsulating these behaviors, the ledger hides database details from other parts of the system, thereby respecting the Single Responsibility Principle. Other classes do not need to know how progress data is stored or queried; they only call the ledger’s methods to interact with the records.

In terms of design patterns, the ProgressLedger echoes the Memento pattern, since it captures and preserves the state of ongoing processes for future inspection. At the same time, it resembles a specialized logger, but with richer semantics that are specific to workflows and agent tasks. Instead of raw log messages, the ledger stores structured events that can later be aggregated into summaries, such as how many tasks succeeded, how many failed, and how long each stage of the workflow took. This structured persistence is what allows the system to produce meaningful analytics and to recover gracefully after disruptions.

The collaboration between the AgentManager and the ProgressLedger forms the backbone of accountability in the system. The manager ensures that work is delegated and supervised in real time, while the ledger guarantees that all events are captured for long-term reliability. Together, they provide the infrastructure needed for a scalable, multi-agent framework where transparency and traceability are just as important as raw execution speed

7.1.5 Tools

Within the system, tools represent the functional instruments that extend the abilities of agents. While agents embody the decision-making and execution framework, tools are the actionable entities that agents call upon to solve specific problems. To ensure consistency and extensibility, the design follows an object-oriented approach where an abstract Tool base class defines the common attributes and methods shared across all implementations. This base establishes metadata, parameter validation, and the standardized asynchronous execution cycle. By enforcing this contract, every tool can be plugged into the larger system without friction, creating a reliable mechanism for reusing operations.

A practical example is the CalculatorTool, a concrete implementation built to handle mathematical operations. It covers addition, subtraction, multiplication, division, as well as more advanced functions like trigonometric evaluation, logarithmic calculations, factorials, and power operations. Each request passes through parameter validation, ensuring inputs are correct before the actual computation. This careful handling prevents cascading errors, especially when these results become inputs for subsequent tasks managed by agents.

Another key implementation is the WebSearchTool. This tool demonstrates the system's ability to interact with the external world by issuing asynchronous HTTP requests, parsing responses, and extracting relevant content. Built on top of asynchronous libraries, it balances responsiveness with efficiency, making it possible for agents to pull real-time information from the web. The tool is not limited to surface-level search. It also extracts metadata, filters irrelevant content, and presents structured output that agents can interpret directly.

Overseeing these implementations is the ToolFactory, which manages dynamic registration, discovery, and instantiation of available tools. The factory pattern makes it possible to extend the system without altering its core logic, as new tools can be registered at runtime. Through this structure, the orchestration layer gains a unified interface for invoking diverse operations, while maintaining clean separation of concerns.

Together, these components ensure that tools remain interchangeable, testable, and easy to extend, all while fitting into the broader workflow governed by agents and orchestrators.

7.1.6 Models

Where tools act as precise instruments, models form the cognitive core of the system. They are responsible for generating text, embedding knowledge, and engaging in conversational tasks. To unify interaction with different providers, the system adopts a shared abstraction: the BaseModel. This abstract class defines a clear contract for methods such as text generation, chat-based conversation, embedding creation, and streaming outputs. It also includes capability reporting and availability checks, ensuring that each model can be queried for what it supports before being assigned tasks.

One of the concrete implementations is the OpenAIModel, which integrates with the OpenAI API. It manages retries, handles streaming responses, and ensures graceful fallback when errors occur. Its main strength lies in providing stable, high-quality language understanding and generation, which is crucial for orchestrators when breaking down high-level goals into executable subtasks.

For cases where local inference is preferred, the OllamaModel is used. It connects with local models through the Ollama API, allowing private and offline deployment of large language models. This design reduces dependency on external services and gives more control over data privacy. Complementing this is the GroqModel, which provides access to fast inference with open-source models, making it useful for time-sensitive or resource-constrained scenarios.

To manage these varied backends, the system includes a ModelFactory and a ModelService. The factory handles instantiation, lifecycle management, and registry of available models. It ensures that new models can be added without breaking compatibility with existing components. On top of this, the ModelService provides a unified layer for executing model operations, whether generating text, creating embeddings, or handling real-time streaming. By abstracting away the complexity of different providers, it simplifies orchestration while offering flexibility in choosing the right model for each task.

This layered design not only increases modularity but also supports future extensibility. As new model providers emerge, they can be integrated by implementing the BaseModel interface and registering through the factory, leaving the rest of the system untouched.

7.1.7 Database

The database component of the system is built on an object-oriented foundation that separates abstraction from concrete implementation. At the highest level, the BaseDB class serves as the abstract blueprint for all data connectors. It defines the common interface that every database implementation must follow, such as create, read, update, and delete operations, along with utility functions for query handling and connection management. By defining this interface in a strict way, the design enforces consistency across different storage classes, while also making it possible to swap or extend the underlying technology without rewriting business logic. The use of an abstract base class here follows the Template Method design pattern, where the general structure is fixed but subclasses provide the details.

The primary implementation in this system is the MongoDB connector, which acts as the concrete subclass of BaseDB. It manages connection pooling, error handling, and asynchronous operations to support the system's high concurrency needs. MongoDB was chosen because its document-oriented nature is well suited to dynamic agent workflows, where tasks, progress logs, and conversation histories often vary in structure. The MongoDB class wraps complex operations in methods that handle retries and validation, ensuring reliability even under transient failures. This approach applies the Adapter pattern, since the MongoDB driver is wrapped in a uniform interface that aligns with the BaseDB contract.

On top of the general MongoDB class, the system defines several specialized subclasses that deal with specific domains of data. TaskDatabase is responsible for managing task lifecycle information. It stores new tasks, retrieves pending ones, updates status as they progress through agents, and links tasks back to the workflows they belong to. This separation ensures that task management logic is encapsulated in one place, making it easier to evolve independently of other concerns. The ProgressDB class focuses on recording execution updates, agent activity, and workflow summaries. It acts as the persistence layer for the progress ledger, allowing real-time monitoring and historical analysis. By centralizing progress data, the system ensures accountability and provides the foundation for audit and debugging tools.

A third specialized subclass, SessionDB, maintains user session data. It records when a session starts, tracks activity, and manages expiration or cleanup. This ensures that the system can tie interactions back to specific users and workflows while avoiding uncontrolled growth of unused data. Session management also supports features such as resuming conversations or linking an ongoing workflow to a returning user. Each of these subclasses relies on the same BaseDB principles, but they refine the interface for their domain, demonstrating the principle of specialization through inheritance.

The database layer as a whole provides a stable backbone to the system. By combining abstraction, specialization, and encapsulation, it isolates persistence concerns from orchestration and agent logic. This structure follows classic object-oriented design while remaining flexible enough to incorporate new databases or alternative storage mechanisms if needed. In practice, the database classes embody a mixture of Template Method, Adapter, and Single Responsibility patterns, making them a well-structured part of the system's foundation.

7.1.8 Orchestrator

The Orchestrator is the central coordinating class of the system, serving as the brain that bridges high-level goals with concrete execution. Its main function is to receive abstract objectives from users or external services and transform them into smaller, executable tasks. This decomposition process is crucial: complex goals such as “analyze market data and prepare a summary report” cannot be handled by a single agent or tool in one step. Instead, the Orchestrator consults available model services, often large language models (LLMs), to generate subtasks that are logically ordered, validated, and normalized before execution.

From an object-oriented design perspective, the Orchestrator implements the Mediator pattern. Rather than having agents communicate directly with each other or with databases, the Orchestrator mediates all interactions. This ensures that each component remains loosely coupled, preserving modularity and easing maintainability. For example, if a new agent type is introduced, it does not need to directly coordinate with other components; instead, it plugs into the Orchestrator’s delegation flow. This pattern reduces interdependencies and allows for a scalable architecture.

Internally, the Orchestrator collaborates with the AgentManager to assign tasks to the right agent. Assignment is not a static process. The Orchestrator takes into account agent capabilities, workload, and availability. If a failure occurs, the Orchestrator dynamically reassigns the task to a fallback agent or retries execution based on system policies. This resilience reflects the Strategy pattern, where different delegation strategies can be applied depending on the task type, error condition, or dependency constraints. Additionally, it enforces reliability by monitoring execution states and making adjustments on the fly.

Another key role of the Orchestrator is queue management. Tasks are not executed in isolation; they often have dependencies or must follow a particular sequence. The Orchestrator ensures that dependencies are respected and that the execution flow proceeds in a way that aligns with user intent. This task scheduling capability allows for concurrent workflows, where multiple goals can be managed in parallel without conflict.

Overall, the Orchestrator is not merely a dispatcher. It embodies intelligence, resilience, and adaptability. Its careful use of object-oriented principles and design patterns ensures that the system can manage complex workflows, handle failures gracefully, and continuously adapt as new components (agents, tools, or models) are added.

7.1.9 TaskLedger

Complementing the Orchestrator's strategic role, the TaskLedger serves as the authoritative source of truth for all task-related state. Every task that is created, assigned, executed, or completed must pass through the TaskLedger's lifecycle management system. This ensures that no task is lost, duplicated, or left in an undefined state. In practice, this class functions as the system's state machine, guiding tasks through stages such as created → assigned → in-progress → completed/failed.

From an object-oriented design standpoint, the TaskLedger provides strong encapsulation of lifecycle management. Agents and services do not directly manipulate task states; instead, they request updates through well-defined interfaces. This maintains integrity and prevents inconsistent task transitions. For example, a task cannot jump directly from "created" to "completed" without going through the required intermediate states, because the ledger enforces strict rules for progression.

The TaskLedger also plays an essential role in auditability and monitoring. By recording every transition and update, it provides a transparent record of what happened, when it happened, and which agent was responsible. This log is not just operational but also strategic: it allows for debugging failures, analyzing performance bottlenecks, and even predicting workflow completion times. Through its persistent tracking, the TaskLedger ensures that workflows remain reliable and observable, even in distributed or asynchronous environments.

Design-wise, the TaskLedger embodies aspects of the Observer pattern. While it is the primary store of task state, other components such as the ProgressLedger or monitoring dashboards can observe its changes to trigger further actions. For instance, when a task moves into the "failed" state, the Orchestrator may be notified to reassign it, while the ProgressLedger updates user-facing progress summaries. This interplay of state tracking and notification turns the TaskLedger into the backbone of reactive system behavior.

In summary, the TaskLedger provides the necessary accountability that balances the Orchestrator's flexibility. While the Orchestrator makes dynamic decisions about task assignment and execution, the TaskLedger enforces order, consistency, and transparency. Together, they create a system where creativity and control coexist: the Orchestrator allows tasks to evolve dynamically, while the TaskLedger ensures they are always tracked, reliable, and auditable.

7.1.10 Tools

The Tools module provides the system with concrete capabilities that agents can invoke to perform specialized operations, while remaining decoupled from orchestration logic. At its core, tools encapsulate discrete functionalities as independent components, making them reusable and testable. The architecture of the Tools module exemplifies the Command pattern, where each tool represents an executable command with clearly defined inputs, outputs, and validation rules.

The foundation is the Tool (ABC), an abstract base class that defines the interface for all tools. It requires methods for metadata retrieval, parameter validation, and asynchronous execution. By enforcing these methods, the system guarantees consistency across all tools, enabling the Orchestrator and AgentManager to invoke them polymorphically. The Tool ABC also supports context management, allowing tools to acquire and release resources safely during execution, which is particularly relevant for operations that involve external APIs or file systems.

Concrete tool classes extend this base. The CalculatorTool provides mathematical operations, including arithmetic, trigonometry, logarithms, and factorials. Its responsibilities are strictly computational, reflecting the Single Responsibility Principle, while orchestration and logging are delegated elsewhere. The WebSearchTool performs asynchronous web search, content extraction, and metadata parsing using libraries such as aiohttp and BeautifulSoup. It is designed to handle errors gracefully and sanitize inputs, protecting the system from invalid or malicious queries.

Central to managing these tools is the ToolFactory, which registers, instantiates, and manages tool classes dynamically. Implementing the Factory Method pattern, the ToolFactory allows new tools to be added without modifying downstream code, adhering to the Open/Closed Principle. This flexibility enables the system to evolve its capabilities rapidly while maintaining a stable interface for agents and orchestrators.

In addition to individual tools, the system includes the MCP Server (Multi-Component Processor Server), which acts as a centralized runtime environment for executing complex tool workflows. The MCP Server coordinates multiple tools in parallel, manages asynchronous execution queues, and exposes a network interface for remote invocation. From a design perspective, the MCP Server applies the Facade pattern, presenting a simplified interface to agents and orchestrators while internally managing task scheduling, resource allocation, and execution monitoring. By centralizing tool execution, the MCP Server ensures consistent logging, error handling, and load balancing, supporting both local and distributed workflows.

Together, the Tools module and MCP Server provide a robust infrastructure for executing actions in the system. Agents can invoke a single tool or an orchestrated sequence through the MCP Server, while factories and abstract base classes ensure that adding, updating, or replacing tools remains straightforward. This architecture balances modularity, extensibility, and reliability, forming a critical foundation for multi-agent coordination and task execution.

7.1.11 Models

The Models module provides the intelligence backbone of the system, abstracting different AI model integrations under a unified framework. While agents and tools define workflows and operations, models are the computational engines that handle reasoning, natural language processing, and generation tasks.

At its core is the BaseModel (ABC), which defines a consistent interface across all model implementations. It specifies common operations such as text generation, embedding retrieval, and response validation. This ensures polymorphism: whether the system uses an OpenAIModel, OllamaModel, or GroqModel, higher-level components like the Orchestrator can invoke them uniformly without concerning themselves with model-specific APIs.

Concrete classes like OpenAIModel, OllamaModel, and GroqModel act as adapters, encapsulating vendor-specific logic. This aligns with the Adapter pattern, allowing the system to integrate heterogeneous model APIs under a common interface. Each class is also responsible for handling rate limits, authentication, and error responses unique to its provider. This separation of concerns keeps higher layers clean and avoids vendor lock-in.

The ModelFactory provides centralized instantiation and registry management. It ensures that models are created based on configuration parameters, environment variables, or runtime requests. Following the Factory Method pattern, it allows new models to be integrated without refactoring existing logic. For instance, if a new LLM provider is adopted, a corresponding class can be added and registered with the factory seamlessly.

Finally, the ModelService provides a Facade over all models, exposing unified operations for downstream consumers. Instead of agents dealing with multiple APIs, the ModelService offers high-level functions such as “generate_response,” “retrieve_embeddings,” or “classify_text,” which internally delegate to the correct model instance. This greatly simplifies orchestration and reduces coupling.

In essence, the Models module transforms a fragmented landscape of third-party AI providers into a coherent subsystem. Through abstract base classes, adapters, factories, and service facades, it achieves extensibility, maintainability, and consistency all critical for scaling an AI-driven multi-agent framework.

7.1.12 LoggingService

The LoggingService is a centralized component responsible for capturing, formatting, and distributing log data across the system. Its primary role is to provide visibility into operations, errors, and workflow progress while supporting monitoring and debugging. By abstracting logging from business logic, it ensures that agents, orchestrators, tools, and models can focus on their primary responsibilities, while consistent operational information is maintained.

At the base, the LoggingService follows object-oriented principles, encapsulating all functionality in a single cohesive class. It defines methods for structured logging, component-specific log formatting, and severity-based output. This approach supports the Single Responsibility Principle, as the class deals exclusively with logging concerns without mixing execution logic. Each log entry includes metadata such as component name, timestamp, task or workflow identifiers, and severity level. By standardizing metadata, downstream monitoring tools or dashboards can aggregate and analyze logs efficiently.

The service also incorporates extensibility through design patterns. By using the Strategy pattern, LoggingService allows different output formats or backends to be selected dynamically. For example, logs can be routed to console output during development, to files for persistent storage, or to remote monitoring platforms in production. Similarly, color-coded or structured JSON formats can be applied depending on the audience or destination. In distributed workflows, LoggingService works in tandem with the MCP server and TaskLedger, ensuring that agent progress and task states are consistently recorded and accessible.

Additionally, LoggingService supports hierarchical logging, where subcomponents inherit contextual information from their parent components. This design makes it possible to track a specific agent, workflow, or tool operation across multiple layers without redundancy. By combining encapsulation, strategy selection, and hierarchical metadata, the LoggingService provides both flexibility and reliability, establishing a stable foundation for observability in the multi-agent system.

7.1.13 AuthService

The AuthService manages authentication and authorization for the system, ensuring that only valid clients, agents, or users can access resources. Its primary function is to enforce security across API endpoints, tools, and orchestrator interactions, maintaining the integrity and privacy of operations.

Implemented as a dedicated class, AuthService encapsulates all authentication logic, including API key validation, token generation, password hashing, and token revocation. This follows the Single Responsibility Principle, isolating security concerns from execution, orchestration, or storage logic. Methods for token management include creation of signed JWTs, verification of token validity, and refresh handling, supporting both short-lived sessions and persistent workflows. The service also handles API key validation for external integrations, enabling secure inter-service communication with tools, models, or knowledge base components.

From a design perspective, AuthService incorporates the Facade pattern, providing a simplified interface for complex security operations. Components interacting with the system do not need to handle cryptographic details or validation protocols directly. Internally, the class may apply additional design patterns, such as the Decorator pattern for dynamically adding validation or logging behavior to authentication functions, and the Adapter pattern for integrating third-party libraries like bcrypt or JWT libraries.

In distributed or multi-agent workflows, AuthService interacts with session management in the database layer, ensuring that user sessions are tracked, validated, and cleaned up appropriately. It also works closely with middleware in the API layer to enforce endpoint-level security policies consistently. By centralizing authentication, authorization, and token lifecycle management, AuthService ensures that the multi-agent orchestration system operates securely, maintaining integrity without introducing unnecessary complexity into business logic.

7.2 Commercialization Aspects of the Product

7.2.1 Motivation Market and (limiting users) & Users

The target market of the proposed IT support system is medium to large-scale organizations requiring constant IT services, which include financial institutions, government agencies, and enterprises like LOLC. In these organizations, IT administrators, helpdesk and the technical staff involved in the incident management and system monitoring will use the system. The system will facilitate smooth business operations with the lowest downtime by identifying the pain points that IT teams experience when they respond to incidents or solve technical problems. In the long term, the solution can also be incorporated in the small and medium-sized businesses (SMB) that need reliable and affordable IT support solutions.

7.2.2 Value Proposition

The system has a solid value offering by offering cost-effectiveness, scalability, and customization with regard to the domain. Compared to generic IT support tools, this product has been customized to fit the specific needs of LOLC and other organizations of a similar nature; therefore, this makes it very flexible to manipulate to internal operations. Its internal knowledge base, caching, and active monitoring will save the time consumed in solving problems and avoiding repetitive incidents. Consequently, the system allows organizations to reduce downtimes, reduce operational expenses, and customer dissatisfaction, making the system more appealing as opposed to the off-the-shelf IT support tools.

7.2.3 Market Research and Demand

The modern world is witnessing an increase in the demand of intelligent IT support systems in the various industries, in particular, as businesses are taking on the digital transformation approach. The support systems used by many companies are manual or old-school ticketing systems that are inefficient and ineffective. Within the financial services sector, a relatively small IT outage can, in fact, have a huge impact on operations and image impact. Upon identifying the requirements of LOLC and comparing with some of the available IT support systems like Zendesk and ServiceNow, we realized the lack of a tailorable, cost-effective, and locally customizable solution to IT support needs. This confirms the possible demand for our solution and in the enterprise market on a broader scale, including LOLC.

7.2.4 Revenue Model

The system can be commercialized as a Software-as-a-Service (SaaS) i.e., organization may pay subscription fee based upon the number of users or incidents managed. Smaller businesses may be offered a freemium system where free features are offered but advanced analytics, reporting, and integrations will be charged. Higher-end enterprises can choose to pay an annually renewable licence fee that will come with technical support and system maintenance. Such a flexible pricing policy will allow discounts to small firms and maximize revenue collection among the enterprises.

7.2.5 The development and deployment

The development and deployment cost The commercialization plan must also look at the cost it will use to develop the system, implement it and to maintain it into the future. The development cost is incurred through software engineering, testing and integrating with the general 3rd party APIs such as Redis and ChromaDB. Deployment will require cloud infrastructure provider, monitoring tools as a way to support high availability/scalability of deployment. Maintenance costs would comprise of updating, bug fixing, and the provision of support services to the customers. The cost-based ideas on a user-friendly interface, avoiding excessive reliance on proprietary technologies, and wherever feasible, using open-source technologies ensure the cost of ownership is within a reasonable range, thereby, ensuring a competitive solution price in the market

Growth Potential and Scalability

The system is scalable and can be applied by organizations of different sizes as it was designed to be scalable. For the purpose of testing, it is feasible to initially implement it as a pilot program within LOLC. Once proven to be successful, the system can be scaled up to include more branches or even other industries like government, healthcare, and education. To make sure that the solution can be applied anywhere in the world, future improvement can include mobile app connectivity, predictive analytics driven by artificial intelligence, and multilingual support. This gives long-term sustainability and flexibility to the ever-changing requirements of the industry.

7.3 Testing And Implementation

Testing and implementation represent a critical stage of this research since they validate the design decisions and ensure that the multi-agentic orchestration system functions as intended. Given that the current work is focused on a prototype rather than a production-level system, testing activities were primarily carried out under the scope of prototype testing. This allowed the team to concentrate on identifying functional correctness, ensuring proper communication among microservices, and validating that the orchestration component operates reliably within the defined architecture. The goal was not only to confirm that each part of the system works in isolation, but also to observe how the different modules behave when integrated into a single workflow.

The first level of testing was unit testing, which focused on validating the internal logic of individual modules. Each microservice, such as the Multi-Agent System (MAS) component, the Form Integration component, the Knowledgebase component, and the Avatar component, was tested independently to ensure that it could execute its assigned responsibilities. For example, the MAS component was tested to confirm that it could decompose goals into tasks and record assignments in the task ledger, while the Knowledgebase component was checked to verify that relevant information could be retrieved from stored data and fed back to the orchestration layer. These unit tests acted as the foundation, providing confidence that each building block of the prototype was functioning correctly.

Following this, integration testing was conducted to ensure that the different services could interact effectively with one another. Since the architecture relies on an API Gateway to coordinate requests and responses between the client, orchestration, and core components, integration testing was essential for verifying smooth data flow. For example, when a user initiated a query, the system needed to ensure that the request traveled through the API Gateway, reached the orchestration layer, was processed by the appropriate agents, and returned as a response. Particular attention was paid to the communication between the orchestrator, the task ledger, and the MAS component, since this interaction is central to the system's reliability. Integration testing highlighted potential bottlenecks and confirmed that agents could coordinate tasks without conflicts.

System-level testing was then applied to validate the prototype as a whole. This stage involved running end-to-end workflows where users interacted with the system through conversations, triggering responses from the MAS, drawing knowledge from the Knowledgebase, and receiving outputs from the Avatar component. By simulating realistic scenarios, the team was able to confirm that the system delivered coherent results across multiple services. For instance, a request from a user to generate structured knowledge would activate both the Form Integration and Knowledgebase components, while the MAS oversaw coordination to ensure the tasks followed the proper order. System testing ensured that the orchestration layer did not fail under multi-component workloads and that outputs remained consistent.

In addition to system testing, prototype testing also included user acceptance testing (UAT). Since the system was designed to provide a conversational experience with a 3D avatar, it was important to evaluate how end users perceived the prototype in practice. A small group of testers

interacted with the system to validate conversational flow, task accuracy, and response clarity. Feedback from this stage was valuable in identifying usability challenges and fine-tuning aspects such as the speed of agent responses or the accuracy of retrieved information. While the testing was limited in scope, it provided meaningful insights into how the system would perform in real-world scenarios.

The implementation strategy was guided by a phased rollout approach, gradually integrating the components instead of deploying the system all at once. The core system components, including the orchestration layer and task ledger, were implemented first since they serve as the backbone of the architecture. Once these were functional, the MAS was connected, followed by the integration of additional microservices such as the Form Integration and Knowledgebase modules. Finally, the 3D Avatar component was added to complete the conversational experience. This incremental approach made it easier to identify issues early and correct them without affecting the rest of the system.

For deployment, the prototype was containerized to ensure modularity and reproducibility across environments. Each microservice was packaged and managed separately, while the API Gateway provided a unified entry point secured with authentication middleware. The orchestration layer communicated with core system components and also connected to external large language models through Ollama, ensuring flexibility for processing complex requests. Monitoring tools were also included to record logs, track execution times, and capture errors, which supported ongoing improvements during prototype testing.

In summary, the testing and implementation process validated that the system can function as a cohesive prototype, with each service working independently as well as in collaboration under orchestration. By applying unit, integration, system, and user acceptance testing, supported by prototype testing practices, the research ensured that the design objectives were met. The phased implementation strategy further reduced risks, making it easier to manage complexity and ensure a reliable orchestration process. These efforts provide a strong foundation for future iterations, where the prototype can evolve into a more scalable and production-ready platform.

8. Results and Discussion

8.1 Results

The evaluation of the proposed system was conducted primarily through prototype testing, which was used to validate the feasibility of the architecture and its ability to handle orchestration, task decomposition, and multi-agent collaboration in a microservice-based environment. The early results confirmed that the orchestration layer was capable of breaking down high-level user requests into smaller, manageable subtasks and assigning them across different agents, while the Task Ledger successfully tracked the progress of these assignments. The Multi-Agentic System coordinated agents effectively, enabling them to call tools, access APIs, and query the knowledgebase where necessary. Security and authentication layers within the API gateway also functioned as expected, ensuring proper request validation. In addition, the integration of a 3D avatar component provided an engaging user interface, while the knowledgebase component, enhanced through retrieval-augmented generation, maintained contextual awareness and improved conversational continuity compared to stateless interactions.

Despite these encouraging results, prototype testing revealed a number of important challenges. One key finding was that multi-agent collaboration introduced noticeable latency, as agents often required multiple exchanges before converging on a solution. The extent of this delay depended significantly on the type of language model in use. Smaller local models accessed through Ollama provided faster responses but were prone to hallucinations and inaccuracies, while larger commercial models delivered higher-quality reasoning at the expense of greater computational cost and budget requirements. This highlighted a fundamental tradeoff: the higher the model capability, the more accurate and reliable the results, but the system becomes more resource-intensive and expensive to operate. Furthermore, multi-agent reasoning sometimes fell into repetitive or common thinking patterns, where agents pursued similar approaches without diversity in problem-solving. In such cases, human feedback was necessary to intervene and guide the orchestrator toward a more effective strategy, showing the importance of human-in-the-loop mechanisms in systems that rely on multi-agent coordination.

From a usability perspective, initial testing showed that users found the conversational flow natural and intuitive. The avatar increased engagement, though occasional delays occurred when synchronizing lip movement with speech output during high-latency responses. The knowledgebase retrieval performed well within the defined domain but struggled when applied outside its trained dataset, reflecting the limitations of retrieval-augmented generation when not properly tuned. Scalability tests also showed that the architecture could handle multiple workflows at once in its prototype form, though performance overheads became more visible as agent concurrency increased. Components such as the avatar rendering and knowledgebase queries added further overhead, which limited responsiveness under heavier workloads.

Overall, the results demonstrate that the proposed four-layer architecture is a viable and modular approach for implementing multi-agent systems. Its strengths lie in modularity, extensibility, and task transparency, as each component can be independently deployed or replaced, while the orchestrator and Task Ledger provide visibility into how user goals are decomposed and tracked. At the same time, the prototype highlighted limitations related to performance tradeoffs,

scalability, and the dependence on human oversight. The system achieved its core goal of proving that orchestration and multi-agent collaboration are possible in this architecture, but the practical challenges underline the importance of optimizing orchestration efficiency, carefully selecting models based on budget and performance needs, and incorporating stronger mechanisms for feedback and error correction.

In conclusion, the testing phase provided valuable insights into both the feasibility and the limitations of the system. While the results validate the practicality of the design and its adaptability to different contexts, they also emphasize that multi-agent collaboration is not yet seamless. Effective use requires balancing accuracy, cost, and speed, supported by human feedback to correct reasoning loops. These findings highlight promising directions for future work, including refining orchestration strategies, improving model selection frameworks, and addressing the performance overhead of components such as avatars and retrieval-based knowledgebases. By addressing these aspects, the system can be further developed into a more scalable, reliable, and user-friendly solution.

8.2 Research Findings

The research carried out in this study has generated several important findings that contribute to the broader understanding of multi-agent system design, orchestration, and implementation in distributed environments. One of the most significant findings is that effective coordination among agents requires not only a well-structured orchestration layer but also a supporting mechanism such as a task ledger to track dependencies, failures, and execution progress. Without such tracking, the orchestration process becomes opaque and error-prone, leading to inefficient communication between agents and an increased likelihood of tasks being duplicated or abandoned. This demonstrates that visibility and accountability are essential elements of any architecture that attempts to scale agent collaboration beyond simple interactions.

Another key finding relates to the role of model selection in shaping the system's effectiveness. During prototype testing, it became clear that the choice of language model directly affects the tradeoff between accuracy, reliability, and cost. Larger commercial models demonstrated better reasoning, lower hallucination rates, and improved reliability in decomposing high-level goals into executable subtasks. However, their usage significantly increased infrastructure costs, making them less feasible for resource-constrained deployments. In contrast, smaller local models were faster and less expensive but suffered from frequent hallucinations, which undermined the reliability of multi-agent workflows. This highlights the critical need for hybrid approaches, where different types of models are strategically deployed depending on task complexity and available resources. It also reinforces the idea that agentic systems are not entirely autonomous; rather, their performance is shaped by conscious architectural choices and budgetary constraints.

A further research finding is that human feedback remains an indispensable part of the multi-agent collaboration process. Although agents are capable of executing reasoning loops and collaborating on subtasks, they often fall into repetitive or common thinking patterns, which limit diversity in their problem-solving strategies. This finding suggests that multi-agent systems cannot yet be relied upon to consistently generate innovative or optimal solutions without human-in-the-loop supervision. Human intervention serves not only to correct errors and redirect the orchestration process but also to improve long-term system performance when feedback mechanisms are integrated into the orchestration logic. In this sense, human oversight acts as both a safeguard against failure and a driver for continuous system refinement.

Security and reliability also emerged as central themes in the findings. Since the system architecture involved multiple components communicating through APIs, concerns such as prompt injection, insecure handoffs, and unauthorized access became apparent. The use of an API gateway pattern, with integrated authentication and security middleware, provided a strong safeguard against many of these risks, but it also increased complexity within the deployment environment. This illustrates that building multi-agent systems in production environments cannot be achieved without prioritizing security as a first-class concern, rather than treating it as an afterthought. Preserving trust between agents, orchestrators, and external APIs requires both architectural discipline and robust engineering practices.

The modular microservice architecture used in this study also revealed insights about scalability and extensibility. By separating the Multi-Agentic System, Form Integration, Knowledgebase, and 3D Avatar components, the system achieved a level of modularity that allowed independent development and deployment. This decoupling proved beneficial for testing and future maintenance, as each service could be iterated on without disrupting the entire workflow. However, the findings also showed that inter-service communication adds latency, especially when orchestration requires agents to coordinate with knowledgebases and external tools simultaneously. This indicates that while microservices provide clear benefits for maintainability, their use in agentic systems requires careful optimization to avoid performance degradation under high-concurrency workloads.

Finally, the research highlighted the importance of orchestration strategies in decomposing high-level goals into executable subtasks. The orchestrator proved capable of breaking down complex instructions, but the findings suggest that different orchestration algorithms vary widely in effectiveness. Rule-based strategies provided predictability but lacked flexibility, while LLM-driven orchestration offered adaptability but introduced inconsistency. This opens up a pathway for future exploration into hybrid orchestration strategies that combine deterministic structures with adaptive reasoning, potentially leading to more balanced and efficient agent collaboration.

Taken together, these findings suggest that multi-agent systems are technically feasible within microservice-based architectures but remain far from plug-and-play solutions. Their effectiveness depends on careful orchestration, conscious model selection, strong security practices, and human oversight. The study contributes to the field by providing a structured approach to building such systems, identifying tradeoffs in accuracy, cost, and scalability, and

highlighting areas where further innovation is required. These findings not only validate the prototype but also provide actionable insights for practitioners and researchers aiming to advance multi-agent system deployment in real-world scenarios.

8.3 Discussion

The results obtained from the prototype implementation and testing provide important insights into the feasibility and challenges of designing a multi-agent orchestration system. The discussion here interprets those results in light of the project objectives and the broader literature on agent collaboration, orchestration, and distributed AI systems.

One of the central observations is that orchestration plays a decisive role in the effectiveness of multi-agent collaboration. While the prototype confirmed that an orchestrator coupled with a task ledger can reliably track assignments and progress, it also became evident that the system's efficiency is highly dependent on how tasks are decomposed and distributed. In scenarios where subtasks were straightforward, the orchestrator handled distribution seamlessly. However, in more complex workflows, bottlenecks emerged as agents waited for missing dependencies or misinterpreted incomplete subtasks. This highlights the importance of refining orchestration strategies, possibly combining deterministic planning with adaptive reasoning to reduce these coordination inefficiencies.

Another significant discussion point arises from the tradeoff between performance and cost in model selection. Larger models demonstrated higher reliability and accuracy, reducing hallucination and improving overall task execution. Yet, the reliance on large models makes the system less affordable and less suitable for deployments with limited resources. On the other hand, lightweight local models offered cost-effectiveness and faster inference but suffered from frequent hallucinations, which in turn required corrective feedback loops. This tension underscores the need for hybrid strategies—deploying local models for routine or low-risk tasks while reserving more powerful models for complex reasoning. Such a design not only balances cost and performance but also aligns with real-world scenarios where organizations must optimize limited budgets.

The testing process also revealed that despite advances in agentic reasoning, human feedback continues to play a critical role in ensuring correctness and creativity. Multi-agent collaboration often led to repetitive thinking patterns, with agents converging on similar solutions rather than exploring diverse alternatives. This behavior limits the system's potential for innovation unless human supervision is integrated into the loop. The findings suggest that effective multi-agent systems should be designed as human-AI collaborative ecosystems rather than fully autonomous systems. Human intervention provides necessary corrections, injects creativity, and ensures accountability, especially when agents operate in high-stakes or ambiguous contexts.

Security and reliability emerged as recurring concerns during implementation. Given the distributed nature of the system, where multiple layers—client, API, orchestration, and core system components—interact with each other, vulnerabilities such as prompt injection, insecure communications, and unauthorized access posed significant risks. The use of an API gateway with authentication and middleware proved valuable in mitigating these threats, but it also introduced additional complexity to the architecture. The discussion here reinforces that robust security mechanisms must be treated as integral design considerations from the start, particularly when working with LLMs and microservice-based environments that may expose multiple communication endpoints.

Another point of discussion relates to scalability and system extensibility. The microservice-based architecture allowed components such as the knowledgebase, multi-agent system, and 3D avatar to be developed and tested independently. This modularity provided clear benefits in terms of maintainability and rapid prototyping. However, the distributed design introduced additional latency during inter-service communication, especially when orchestration required frequent back-and-forth exchanges between agents and the knowledgebase. This latency did not prevent the prototype from functioning but raises concerns about real-world scalability when the number of agents and tasks increases. Optimizing communication protocols and caching strategies may help alleviate these delays in future iterations.

Furthermore, the discussion highlights the balance between rule-based and adaptive orchestration approaches. Rule-based orchestration ensured predictability and reduced the risk of task misallocation, but it lacked the flexibility needed for complex, evolving scenarios. In contrast, orchestration driven by language models provided adaptability but at the cost of occasional inconsistency. This points to the necessity of hybrid orchestration strategies, which can combine the robustness of deterministic planning with the creativity and adaptability of LLM reasoning.

Finally, the discussion emphasizes that while the prototype validated the conceptual feasibility of multi-agent orchestration systems, their practical deployment remains challenging. Issues such as cost constraints, model reliability, orchestration strategies, and the necessity of human oversight indicate that fully autonomous multi-agent systems are not yet ready for widespread real-world application. Instead, they should be approached as augmentative systems that enhance human capabilities while requiring continuous supervision and refinement.

In summary, the discussion underscores that the system developed in this study demonstrates strong potential for advancing multi-agent collaboration within microservice architectures. However, it also reveals critical challenges that must be addressed for such systems to mature, including model selection tradeoffs, orchestration optimization, latency management, and integration of human feedback. These insights pave the way for further research and refinement, ensuring that future implementations can balance scalability, reliability, cost, and creativity in a more sustainable manner.

9. Summary of Each Student's Contribution

Lakshan S. N.

Served as the Team Lead and Project Manager, coordinating the project workflow through Jira task assignments and reviews. Took responsibility for resolving merge conflicts, managing the automation pipeline, and overseeing deployment and infrastructure. Designed and architected the Multi-Agent System (MAS), ensuring its effective integration with other components of the system.

10. Conclusion

This research explored the design and implementation of a Multi-Agent Orchestration System (MAS) integrated within a microservice-based architecture. The study demonstrated that multi-agent collaboration, while powerful, presents unique challenges such as task coordination, dependency management, reliability in distributed environments, and the prevention of failures during execution. By addressing these through an orchestrator-led design, the system ensures that high-level goals are decomposed into manageable tasks, properly assigned, monitored, and executed in a scalable manner.

The research findings highlight that the performance of multi-agent systems is highly dependent on resource allocation and model selection. More advanced models provide higher accuracy and reliability, whereas lightweight local models are cost-effective but more prone to hallucinations and require careful intervention through human feedback. This trade-off underscores the importance of balancing budget, efficiency, and accuracy in real-world deployments.

Additionally, the research emphasized the role of security, communication, and modularity in achieving sustainable orchestration. With an API gateway layer, secure communication with external services, and integration with components such as knowledge bases, form processors, and 3D avatar interfaces, the MAS demonstrated flexibility in adapting to different use cases. The project also showed that modular architecture supports reusability, scalability, and the integration of new technologies without disrupting the existing system.

Ultimately, this research contributes to the understanding of agentic AI orchestration, offering insights into best practices for architecture, task decomposition, monitoring, and inter-agent communication. While limitations such as dependency on model quality and the need for human-in-the-loop adjustments remain, the system provides a strong foundation for future exploration. With further refinement and scaling, this orchestration approach can significantly enhance complex workflows across domains, enabling more reliable, collaborative, and intelligent multi-agent system

11. References

[1] A. Vaswani *et al.*, “Attention Is All You Need,” Aug. 02, 2023, *arXiv*: arXiv:1706.03762. doi: 10.48550/arXiv.1706.03762.

[2] “Open models by OpenAI.” Accessed: Aug. 28, 2025. [Online]. Available: https://openai.com/open-models/

[3] “Frontier AI LLMs, assistants, agents, services | Mistral AI.” Accessed: Aug. 28, 2025. [Online]. Available: https://mistral.ai/

[4] H. Touvron *et al.*, “LLaMA: Open and Efficient Foundation Language Models,” Feb. 27, 2023, *arXiv*: arXiv:2302.13971. doi: 10.48550/arXiv.2302.13971.

[5] *ollama/ollama*. (Aug. 28, 2025). Go. Ollama. Accessed: Aug. 28, 2025. [Online]. Available: https://github.com/ollama/ollama

[6] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort, “A White Paper on Neural Network Quantization,” June 15, 2021, *arXiv*: arXiv:2106.08295. doi: 10.48550/arXiv.2106.08295.

[7] “Jules - An Asynchronous Coding Agent.” Accessed: Aug. 28, 2025. [Online]. Available: https://jules.google

[8] “microsoft/vscode-copilot-chat: Copilot Chat extension for VS Code.” Accessed: Aug. 28, 2025. [Online]. Available: https://github.com/microsoft/vscode-copilot-chat

[9] “Bolt.new,” bolt.new. Accessed: Aug. 28, 2025. [Online]. Available: https://bolt.new/

[10] “Cursor - The AI Code Editor.” Accessed: Aug. 28, 2025. [Online]. Available: https://cursor.com/

[11] “Overview Leaderboard | LMArena.” Accessed: Aug. 28, 2025. [Online]. Available: https://lmarena.ai/leaderboard

[12] Q. Wu *et al.*, “AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation,” Oct. 03, 2023, *arXiv*: arXiv:2308.08155. doi: 10.48550/arXiv.2308.08155.

[13] L. Zhang, Z. Ji, and B. Chen, “CREW: Facilitating Human-AI Teaming Research,” Jan. 01, 2025, *arXiv*: arXiv:2408.00170. doi: 10.48550/arXiv.2408.00170.

[14] J. Wang and Z. Duan, “Agent AI with LangGraph: A Modular Framework for Enhancing Machine Translation Using Large Language Models,” Dec. 05, 2024, *arXiv*: arXiv:2412.03801. doi: 10.48550/arXiv.2412.03801.

[15] S. Han, Q. Zhang, Y. Yao, W. Jin, and Z. Xu, “LLM Multi-Agent Systems: Challenges and Open Problems,” May 12, 2025, *arXiv*: arXiv:2402.03578. doi: 10.48550/arXiv.2402.03578.

[16] S. Raza, R. Sapkota, M. Karkee, and C. Emmanouilidis, “TRiSM for Agentic AI: A Review of Trust, Risk, and Security Management in LLM-based Agentic Multi-Agent Systems,” July 09, 2025, *arXiv*: arXiv:2506.04133. doi: 10.48550/arXiv.2506.04133.

[17] W. Zhang *et al.*, “AgentOrchestra: A Hierarchical Multi-Agent Framework for General-Purpose Task Solving,” Aug. 13, 2025, *arXiv*: arXiv:2506.12508. doi: 10.48550/arXiv.2506.12508.

12. Appendices

12.1 Exponential Back Off Algorithm

```python
async def _make_request_with_retry(self, method: str, url: str, json_data: Dict[str, Any]) -> Dict[str, Any]:
    """Make HTTP request with exponential backoff retry logic."""
    attempt = 0
    last_error = None

    while attempt <= self.max_retries:
        try:
            return await self._make_request(method, url, json_data)
        except asyncio.TimeoutError as e:
            last_error = f"Request timeout (attempt {attempt + 1}/{self.max_retries})"
            logging_service.log_warning(last_error)
        except Exception as e:
            last_error = str(e)
            logging_service.log_warning(f"Request failed (attempt {attempt + 1}/{self.max_retries}): {last_error}")

        # Exponential backoff only if not the last attempt
        if attempt < self.max_retries:
            await asyncio.sleep(self.backoff_factor ** attempt)
        attempt += 1

    raise Exception(f"Max retries exceeded. Last error: {last_error}")
```

Figure 3 - Implementation Of Exponential back Off

LLM Temperature Tuning Methodology for Retry Mechanisms

```python
while retry_count <= max_retries:
        try:
            # If this is a retry attempt, log it
            if retry_count > 0:
                logging_service.log_warning(f"Retrying task creation (attempt {retry_count}/{max_retries})")

            # Prepare prompt for LLM to generate tasks
            prompt = self._create_task_planning_prompt(goal, context)

            # Each retry uses slightly different temperature for more varied outputs
            temperature = 0.7 + (retry_count * 0.1)  # Increase temperature slightly with each retry
```

Figure 4 - implementation of LLM temprature tuning

12.2 Design Architure Development

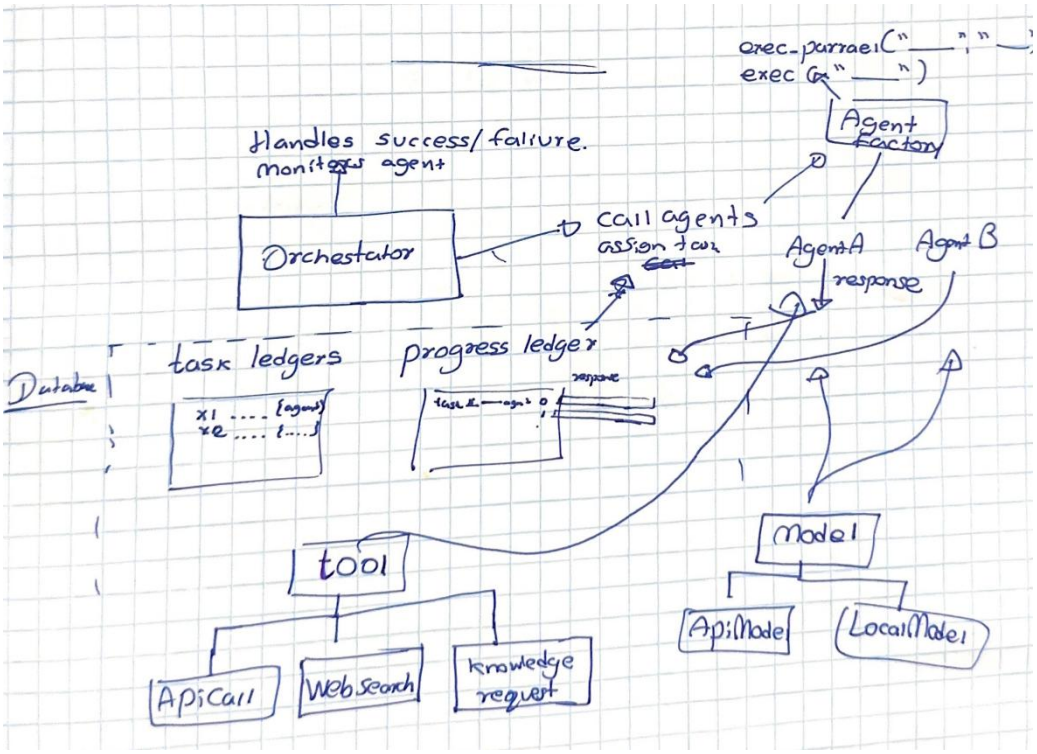

Figure 5 - Modeling Agentic System

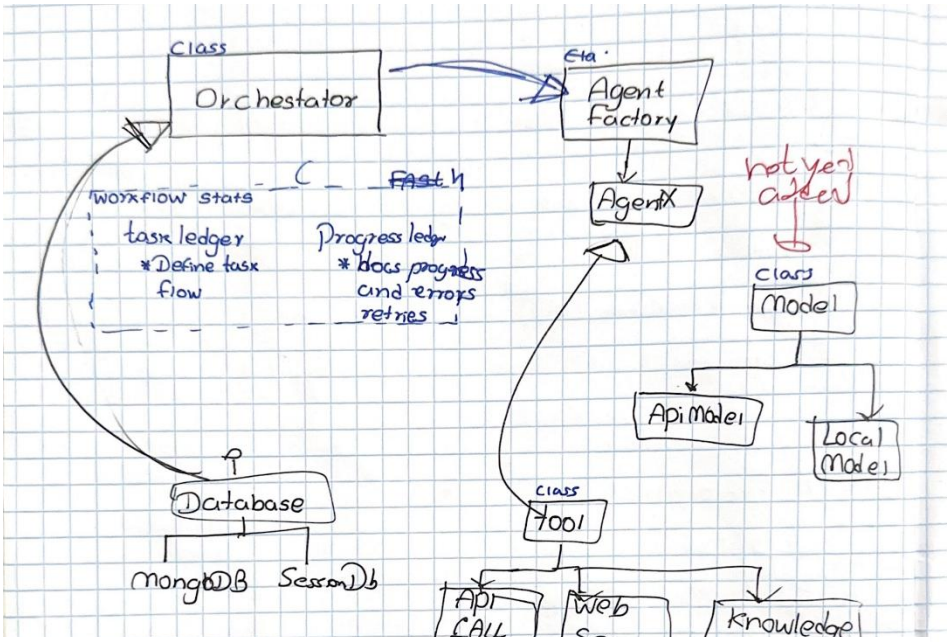

Figure 6 - Modeling Inter Agent Communication

Designing AI Agents Flow From Scratch

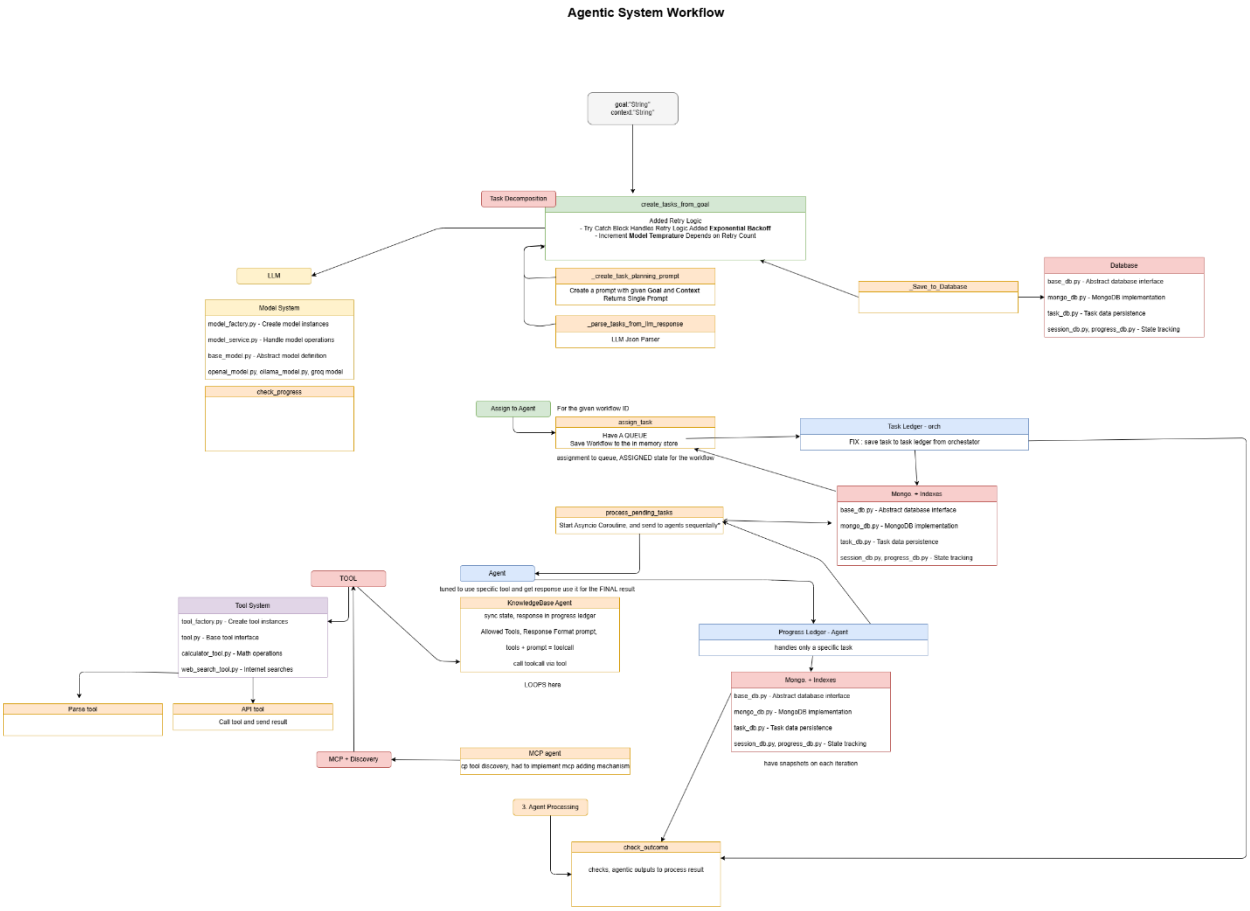

Figure 7 – Inter Class Communication and collaboration