Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

An Autonomous Multi-Agent Architecture For Enterprise Customer Support System

Akash Mandur Anand

aakashaanand@protonmail.com

M S Ramaiah University of Applied Sciences https://orcid.org/0009-0004-3088-7615

Yashas Nadig S

M S Ramaiah University of Applied Sciences

Method Article

Keywords: Agent-to-Agent, RAG, MIPROv2, Multi-agent systems, Large Language Models, Model Context Protocol, Declarative Self-Imporoving Python

Posted Date: November 17th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11428945/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: The authors declare no competing interests.

An Autonomous Multi-Agent Architecture for Enterprise Customer Support

Akash Mandur Anand

M S Ramaiah University of Applied Sciences
Peenya 4th Phase, Peenya, Bengaluru, Karnataka 560058

Author email: aakashaanand@protonmail.com

Yashas Nadig S

M S Ramaiah University of Applied Sciences
Peenya 4th Phase, Peenya, Bengaluru, Karnataka 560058

Co-author email: yashasnadigsyn@proton.me

Abstract

Traditional enterprise customer support systems rely heavily on rigid rule-based chatbots or single large language models (LLMs). These approaches suffer from poor task delegation, hallucinated responses, and security risks when directly connected to business databases. This project addresses these limitations by designing and implementing a multi-agent architecture that orchestrates specialised AI agents to handle customer inquiries securely and reliably. The work focuses on automating the “first line” of customer support for small businesses, including pre-sales questions, order tracking, policy inquiries, and complaint resolution.

The scope covers an end-to-end system comprising a customer-facing storefront (EatIron/StayNest), a reusable chat widget, a FastAPI backend with DSPy-based hierarchical multi-agent system (6 agents), and a Ticket Management System (TMS) built with Tauri and React. The methodology uses DSPy for declarative agent signatures (Router, Catalog, DB, Policy, Resolution, Ticketing), pgvector for RAG, OpenAI GPT-4o-mini for routing/catalog and GPT-4o for resolution/vision, and socket.

IO for real-time communication. Read-only database access is enforced through a dedicated DB Agent to prevent data leaks. Experimental evaluation shows that the multi-agent architecture achieves 92.4% accuracy after MIPROv2 optimisation (from 64.8% zero-shot), correctly classifies intents, raises tickets only for genuine issues, and reduces owner intervention time by ~60% in simulated scenarios. The system successfully handles sales, support, and escalation flows with full transparency via a live dashboard. The architecture proves scalable and can be extended beyond customer support to HR, IT, and other enterprise domains.

Keywords

Agent-to-Agent, RAG, MIPROv2, Multi-agent systems, Large Language Models, Model Context Protocol, Declarative Self-Imporoving Python

1. Introduction

This chapter establishes the context for the project – an autonomous multi-agent architecture for enterprise customer support. It discusses the limitations of existing single-model chatbot solutions, the motivation for adopting a hierarchical multi-agent system (MAS), and the scope of the proposed work. The chapter also provides a brief preview of the core themes: intent routing, specialised agents, secure database access, real-time escalation, and transparency dashboards.

1.1 Challenges in Enterprise Customer Support

Customer support plays a vital role in maintaining customer satisfaction, operational efficiency, and trust within modern enterprises. Businesses across domains such as ecommerce, finance, hospitality, and software services depend heavily on responsive support systems to address customer concerns related to orders, refunds, account verification, delivery tracking, and technical assistance. However, traditional support infrastructures often struggle to efficiently manage repetitive customer queries while maintaining response accuracy and scalability.

Many enterprises currently rely on either human support teams or single-model AI chatbots to handle customer interactions. Human-driven systems are resource intensive, time-consuming, and difficult to scale, especially during peak demand periods.

On the other hand, single Large Language Model (LLM)-based chatbots introduce several operational and security limitations. Since a single AI model attempts to perform multiple responsibilities such as product explanation, order verification, policy interpretation, and complaint resolution simultaneously, the system frequently suffers from poor task delegation and contextual confusion.

Another major challenge is the generation of hallucinated responses. LLMs without structured data access mechanisms may fabricate information related to refunds, policies, or customer orders, leading to misinformation and reduced customer trust. Furthermore, direct integration of AI systems with enterprise databases introduces significant security risks. Unrestricted database access can expose sensitive customer information, compromise enterprise data integrity, and create vulnerabilities for unauthorized modifications.

The proposed system addresses these limitations by introducing a hierarchical MultiAgent System architecture where responsibilities are distributed across specialized agents. Instead of relying on a monolithic chatbot, the system enables structured task delegation, controlled data access, transparent workflow monitoring, and secure escalation to human support when required. This approach improves reliability, scalability, and security while enhancing overall customer experience.

1.2 Background

LLMs, Autonomous Agents, and Multi-Agent Systems recent advancements in Artificial Intelligence and Natural Language Processing have significantly improved the capabilities of Large Language Models such as GPT-based systems and other generative AI frameworks. These models are capable of understanding natural language, generating human-like responses, reasoning over tasks, and interacting with external tools and APIs. Such capabilities have enabled the emergence of autonomous AI agents capable of performing specialized tasks independently.

An autonomous agent is a software entity designed to perceive user input, make decisions, utilize tools, and execute actions to accomplish predefined objectives. While single-agent systems can handle simple workflows, enterprise-level operations often require multiple coordinated agents working together in a structured manner. This concept forms the basis of a Multi-Agent System (MAS), where several specialized agents collaborate to solve complex

problems through communication, delegation, and orchestration. In the context of enterprise customer support, the proposed architecture adopts a hierarchical MAS design. The system consists of a Supervisor or Router Agent that acts as the central orchestration layer. This agent analyzes customer intent, classifies the nature of the request, and dynamically routes the workflow to specialized agents. Dedicated agents such as the Catalog Agent, Tracking Agent, Policy Agent, Resolution Agent, and Manager Agent handle domain-specific operations independently. Such specialization improves system modularity, reduces hallucination risks, and enables parallel execution of tasks.

The architecture also integrates Agent-to-Agent (A2A) collaboration, allowing agents to communicate internally to complete complex workflows. For example, the Resolution Agent may consult the Policy Agent for refund rules while simultaneously interacting with the Tracking Agent to verify order details. Additionally, the system incorporates the Model Context Protocol (MCP), which acts as a secure gateway between AI agents and enterprise databases by enforcing controlled read-only access. This ensures data security while preserving enterprise integrity.

The project therefore combines advancements in LLMs, agent orchestration, secure AI integration, and real-time workflow monitoring to create an intelligent and scalable enterprise customer support platform.

1.3 Scope of the Proposed System

The scope of the proposed system includes the design, development, and implementation of an autonomous multi-agent architecture capable of handling enterprise customer support operations efficiently and securely. The system is designed to support customer interactions through natural language communication while providing intelligent task delegation among multiple AI agents.

The project primarily focuses on supporting customer service operations such as product inquiries, order tracking, refund requests, complaint handling, policy explanation, and escalation management. The system accepts user queries through text-based interactions and is extensible for future integration with voice interfaces using Speechto-Text (STT) and Text-to-Speech (TTS) pipelines. The architecture ensures that customer requests are processed through specialized agents rather than a single centralized AI model.

A key component within the scope of the project is secure enterprise integration. The

system uses a dedicated Tracking Agent and the Model Context Protocol (MCP) to enable controlled read-only access to enterprise databases. This prevents unauthorized modifications while ensuring that AI-generated responses remain accurate and grounded in real-time business data. The system also supports sentiment analysis and transparency monitoring through a live dashboard that enables administrators to track workflows, agent communication, and decision-making processes in real time.

Another important aspect of the project scope is escalation management. Complex, emotional, or unresolved customer cases are automatically escalated to human support agents through a Manager Agent that generates and prioritizes support tickets. This ensures seamless collaboration between AI systems and human operators while maintaining context continuity during escalation.

Although the project demonstration domain focuses on enterprise customer support, the architecture is designed to be modular and domain-independent. The same multi-agent framework can be extended to industries such as healthcare, finance, IT support, education, and human resources, making the proposed system scalable and adaptable for future enterprise AI applications.

2. Literature Review and Problem Formulation

This chapter presents a comprehensive review of existing research related to enterprise customer support automation, Multi-Agent Systems (MAS), Large Language Models (LLMs), Retrieval-Augmented Generation (RAG), and secure AI integration frameworks.

The chapter critically analyses previous approaches, identifies their strengths and limitations, and establishes the research gaps that motivate the proposed work. Furthermore, the chapter formulates the research problem and defines the key research questions addressed in this project.

2.1 Background Theory on Multi-Agent Systems for Customer Support

The rapid growth of Artificial Intelligence and Large Language Models (LLMs) has significantly transformed enterprise customer support systems. Traditional customer support methods relied heavily on human agents and rule-based chatbots, which often struggled to handle complex customer queries efficiently. Although modern LLM-based chatbots provide more natural conversational capabilities, they still face several limitations such as hallucinated responses, poor task specialization, security concerns, and inability to manage multiple workflows simultaneously. These issues become more critical in enterprise environments where customer interactions involve order tracking, refund handling, policy enforcement, fraud detection, and ticket management.

To overcome these limitations, researchers and industries have increasingly adopted MultiAgent Systems (MAS) for enterprise automation. A Multi-Agent System consists of multiple autonomous agents working collaboratively to complete specialized tasks. Instead of relying on a single AI model, the workload is distributed among dedicated agents such as Router Agents, Database Agents, Policy Agents, and Resolution Agents. This architecture improves scalability, transparency, modularity, and response accuracy while reducing hallucinations and security risks. The proposed system utilizes a hierarchical Multi-Agent architecture integrated with Retrieval-Augmented Generation (RAG), secure Model Context Protocol (MCP) access, and real-time escalation mechanisms to provide efficient and reliable enterprise customer support.

2.1.1 Principles and Assumptions

A Multi-Agent System (MAS) is a collection of autonomous software entities known as agents that interact and collaborate to achieve shared or individual objectives. Unlike traditional monolithic systems, MAS architectures divide complex workflows into smaller, manageable subtasks handled by specialized agents. This decomposition improves modularity, scalability, and overall system efficiency.

The fundamental principle of a Multi-Agent System is decentralized control, where no single agent possesses complete knowledge of the system. Instead, agents coordinate through structured communication and task delegation mechanisms. In enterprise customer support systems, this enables different agents to independently manage catalog queries, order tracking, policy explanation, complaint resolution, and ticket generation. Another important principle is modularity. Agents can be independently developed, upgraded, or replaced without affecting the overall system architecture. The system also relies heavily on tool usage, where agents

interact with external systems such as databases, APIs, Retrieval-Augmented Generation pipelines, and ticketing systems to perform their assigned tasks.

The proposed system assumes that enterprises maintain structured databases containing customer records, order information, product details, and support tickets.

The architecture also assumes that AI systems should not directly modify enterprise databases and therefore employs a read-only access mechanism through a dedicated database agent. Additionally, the system assumes the availability of human operators to handle escalated or emotionally sensitive customer interactions.

2.1.2 Key Definitions and Implications

The proposed architecture relies on several important concepts related to autonomous AI systems and enterprise workflow automation. An agent refers to an autonomous software entity capable of perceiving user input, making decisions, and executing actions to achieve a specific objective. In this project, specialized agents such as the Router Agent, Catalog Agent, Tracking Agent, Policy Agent, Resolution Agent, and Ticketing Agent are designed to perform dedicated responsibilities.

Orchestration refers to the process of dynamically routing user requests to appropriate agents based on intent classification. The Router Agent performs orchestration by analyzing customer queries and delegating workflows to relevant agents.

Agent-to-Agent communication (A2A) enables direct collaboration between agents. For example, the Resolution Agent may interact with both the Tracking Agent and the Policy Agent to resolve a refund-related issue.

Retrieval-Augmented Generation (RAG) is a technique that enhances LLM responses using external knowledge sources such as company policies or product catalogs. RAG improves response accuracy and reduces hallucinations by grounding responses in factual enterprise data. Hallucination refers to situations where a Large Language Model generates factually incorrect or fabricated information. The proposed system mitigates hallucinations using strict prompting, controlled database access, and Retrieval-Augmented Generation. The Model Context Protocol (MCP) acts as a secure middleware layer between AI agents and enterprise data sources. It enforces controlled, parameterized, read-only access to enterprise databases, thereby preventing unauthorized modifications and improving system security.

2.1.3 Standard Representations and Relationships

A typical enterprise customer support Multi-Agent System can be represented using a hierarchical architecture in which customer queries are first processed by a central Router Agent. The Router Agent classifies the user intent and delegates the request to specialized agents such as the Catalog Agent or Resolution Agent. These agents may further interact with supporting agents such as the Database Agent, Policy Agent, or Ticketing Agent to complete the workflow. The hierarchical arrangement improves workflow organization, reduces cognitive overload on individual agents, and enables parallel task execution. This structure also improves transparency by clearly defining the responsibilities and interactions of each component.

2.1.4 Merits, Demerits, and Applicability

The Multi-Agent approach offers several advantages for enterprise customer support systems. Since responsibilities are distributed among specialized agents, the architecture becomes highly scalable and modular. Independent agents can process tasks simultaneously, improving response time and overall efficiency. Specialization also reduces hallucinations because each

agent focuses only on a narrow domain of knowledge. The system additionally improves security by isolating database access through a dedicated read-only Database Agent. This prevents direct exposure of enterprise databases to Large Language Models and minimizes the risk of unauthorized modifications or data leaks.

Despite these benefits, Multi-Agent Systems introduce certain challenges. Communication overhead between agents can increase system latency, especially in workflows involving multiple agent interactions. The orchestration logic also becomes more complex as the number of agents increases. Moreover, effective coordination requires careful prompt engineering and workflow management.

The architecture is particularly suitable for domains requiring structured workflows, secure data handling, and transparency. Such domains include enterprise customer support, IT help-desk systems, HR onboarding platforms, healthcare assistance systems, and claims processing applications.

2.1.5 Comparison with Peripheral Topics

Traditional rule-based chatbots provide deterministic and predictable responses but fail to handle novel or unstructured queries effectively. Single Large Language Model chatbots offer flexible conversational capabilities but suffer from hallucinations, lack of task specialization, and poor security controls.

LLM systems integrated with tools such as ReAct frameworks can query databases and APIs dynamically. However, these systems still operate using a single agent architecture and therefore suffer from context overload when managing complex workflows. The proposed Multi-Agent approach combines the advantages of specialization, transparency, secure database integration, and modularity. Although the architecture introduces higher complexity and additional API calls, it significantly improves reliability, scalability, and maintainability when compared to existing approaches.

2.1.6 Concluding Related Work

Existing research demonstrates that collaborative Multi-Agent Systems outperform single-agent systems in complex reasoning and workflow-oriented tasks. Frameworks such as ChatDev, MetaGPT, and AutoGen have shown the effectiveness of role specialization and Agent-to-Agent collaboration. However, most existing systems focus primarily on creative problem-solving or software engineering tasks rather than enterprise customer support scenarios.

Furthermore, previous works lack secure enterprise database integration, structured escalation mechanisms, and real-time transparency dashboards for workflow monitoring. The absence of these features highlights a significant research gap in enterprise-grade customer support automation systems. This gap forms the foundation and motivation for the proposed research.

2.2 Critical Review of Literature

2.2.1 Early Work on Customer Support Automation

Early customer support systems such as ELIZA, ALICE, and Pandorabots relied primarily on pattern matching and rule-based conversation flows. These systems responded using predefined templates and lacked contextual understanding. Although effective for simple FAQ-based

interactions, such systems could not adapt to dynamic customer queries or enterprise workflows.

With advancements in deep learning, sequence-to-sequence conversational models enabled more flexible dialogue generation. Systems such as Google Smart Reply introduced automated response suggestions but still lacked reasoning capabilities and contextual awareness required for enterprise-level support systems.

2.2.2 Large Language Models for Customer Service

The introduction of Large Language Models such as GPT-3 significantly transformed conversational AI by enabling few-shot and zero-shot learning capabilities. These models allowed customer support systems to generate human-like responses without extensive task-specific training.

However, direct use of LLMs in enterprise environments introduced several limitations. Hallucinated responses, inaccurate order details, policy misinterpretation, and vulnerability to prompt injection attacks became major concerns. Additionally, deploying advanced models such as GPT-4 for long enterprise conversations resulted in increased operational costs.

Research conducted on LLM-based customer support systems highlighted the need for tool integration, structured prompting, and external knowledge grounding to improve reliability and reduce hallucinations.

2.2.3 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) was introduced to improve factual consistency in LLM responses by incorporating external knowledge retrieval mechanisms. RAG systems retrieve relevant information from knowledge bases or vector databases before generating responses.

Research studies demonstrated that RAG significantly reduces hallucinations and improves answer grounding. Technologies such as pgvector further enabled semantic similarity search directly within PostgreSQL databases.

Despite these advantages, RAG alone is insufficient for enterprise customer support because dynamic transactional data such as order status and refund records require secure structured database access rather than static document retrieval.

2.2.4 Multi-Agent Systems with Large Language Models

Recent studies such as ChatDev, MetaGPT, AutoGen, and AgentOrchestra explored collaborative LLM-based Multi-Agent Systems. These frameworks demonstrated that role specialization improves workflow reliability, reasoning quality, and scalability. ChatDev simulated software development teams using specialized AI roles, while MetaGPT introduced standardized workflow orchestration through structured prompts. AutoGen focused on dynamic Agent-to-Agent communication, and AgentOrchestra introduced hierarchical planning mechanisms.

Although these systems demonstrated strong collaborative reasoning capabilities, they lacked enterprise-specific features such as secure database integration, transparency dashboards, audit logging, and structured human escalation workflows. The proposed system addresses these shortcomings by integrating secure enterprise data access and real-time workflow transparency mechanisms.

2.2.5 Security Frameworks for LLM Data Access

Enterprise deployment of AI systems requires strong security mechanisms to prevent unauthorized access and prompt injection attacks. The Model Context Protocol (MCP) emerged as a standardized approach for secure AI-to-data interactions.

The proposed system adopts an MCP-inspired architecture that enforces strict read-only access policies, parameterized database queries, and centralized audit logging. This security layer isolates AI agents from direct enterprise database access while preserving real-time information retrieval capabilities.

2.2.6 DSPy: Declarative Self-Improving Python

DSPy introduced a declarative programming framework for optimizing LLM workflows using signatures, optimizers, and automated prompt refinement techniques. Unlike manual prompt engineering approaches, DSPy separates prompting logic from model execution.

The proposed project utilizes DSPy for agent orchestration and intent classification optimization. Techniques such as MIPROv2 are employed to improve routing accuracy and response consistency within the hierarchical Multi-Agent System.

Table 2.1 Comparative Analysis of Existing Customer Support Automation Systems

System/ Paper	Multi-Agent	RAG	DB Tool	Security Layer	Human Escalation	Tranaparency
Rule-Based Chatbot	No	No	No	No	Manaul	No
GPT-Based Chatbot	No	Optional	No	None	Manaul	No
RAG-Based Systems	No	Yes	No	None	Manaul	No
AutoGen	Yes	Yes	Partial	None	Manaul	Partial
ChatDev/ MetaGPT	Yes	No	No	None	No	Partial
Commercial Enterprise Systems	Partial	Yes	Yes	Role-Based	Yes	Dashboard
Proposed System	Yes	Yes	Yes	MCP Gateway	Real-Time	Live Trace Viewer

2.2.7 Comparative Analysis of Existing Systems

The analysis demonstrates that no existing open-source framework integrates hierarchical Multi-Agent collaboration, secure read-only database access, Retrieval-Augmented Generation, structured human escalation, and real-time transparency into a single enterprise customer support architecture.

2.2.8 Identified Research Gaps

A detailed review of existing literature reveals several important research gaps. Most existing systems either rely on single-agent architectures or lack secure database integration mechanisms. Current enterprise AI systems also fail to provide standardized ticket creation and seamless human escalation capabilities.

Transparency remains another significant issue, as most systems do not provide administrators with visibility into agent reasoning, workflow routing, or internal decisionmaking processes. Furthermore, limited research exists on applying DSPy optimization techniques to hierarchical Multi-Agent Systems designed specifically for enterprise customer support.

These identified gaps motivate the development of the proposed autonomous MultiAgent architecture.

2.3 Problem Formulation - Identification of Research Gaps

The literature review highlights several limitations in existing enterprise customer support systems. Single-agent chatbot architectures fail to effectively decompose complex workflows and therefore produce generic or inaccurate responses. Direct AI integration with enterprise databases introduces severe security vulnerabilities and risks exposure of sensitive customer data.

Most existing systems also lack structured escalation mechanisms that generate concise support tickets for human operators. Additionally, the absence of real-time transparency dashboards prevents administrators from monitoring agent reasoning and identifying system failures effectively.

Another important limitation is the lack of optimization frameworks for hierarchical MultiAgent workflows. Prompt engineering for enterprise AI systems remains largely manual and inefficient.

3. Design

This chapter presents the complete methodology adopted for the design, development, modeling, and validation of the proposed enterprise customer support system. The chapter explains how Multi-Agent Systems (MAS), Large Language Models (LLMs), Retrieval-Augmented Generation (RAG), and secure enterprise database integration are combined to automate customer support operations intelligently and efficiently. The discussion is organized to follow the major research objectives beginning with system design considerations and then progressing through architecture development, workflow modeling, security implementation, transparency monitoring, and escalation mechanisms.

The proposed system adopts a hierarchical and modular architecture where specialized autonomous agents collaborate to handle customer queries, order verification, policy retrieval, complaint resolution, and ticket management. The structured approach improves scalability, maintainability, transparency, and enterprise-level security while reducing hallucinated responses and operational complexity. The methodology described in this chapter provides a systematic framework for implementing secure and reliable AI-driven customer support automation.

3.1 Design, Modelling and Analysis

This section describes the foundational design principles and analytical considerations adopted for the proposed enterprise customer support system. The proposed architecture is designed to overcome the limitations of traditional chatbot systems by introducing a hierarchical Multi-Agent System capable of intelligent task delegation, secure database interaction, transparency monitoring, and automated escalation management. The system focuses on improving response reliability, reducing hallucinations, and enabling scalable enterprise workflow automation through collaborative AI agents.

The proposed system follows a modular and layered architecture in which each agent performs a specialized responsibility independently while communicating with other agents using Agent-to-Agent (A2A) collaboration mechanisms. The modular structure allows different components such as orchestration logic, database access, transparency monitoring, and escalation management to evolve independently without affecting the complete system workflow. This

approach improves scalability, maintainability, fault isolation, and flexibility for future system expansion.

The major design goals of the proposed system include:

- Supporting intelligent customer interaction through natural language conversations.
- Enabling secure read-only enterprise database access using a dedicated Database Agent and MCP Gateway.
- Reducing hallucinations using Retrieval-Augmented Generation (RAG) and structured workflow orchestration.
- Providing real-time transparency and workflow monitoring using a dashboard interface.
- Enabling automatic escalation and ticket generation for unresolved or critical customer issues.
- Ensuring modularity and scalability of independent agents and enterprise services.

3.2 System Architecture of the Proposed System

The overall architecture of the proposed enterprise customer support system is divided into multiple logical layers to ensure separation of concerns, secure workflow execution, and efficient communication between various agents and enterprise resources. The architecture integrates Multi-Agent Systems (MAS), secure database access, RetrievalAugmented Generation (RAG), and real-time transparency monitoring to provide intelligent customer support automation.

The architecture mainly consists of the following components:

A. Input and Interaction Layer

B. Orchestration and Intelligence Layer

C. Agent Collaboration Layer

D. Security and Database Access Layer

E. Transparency and Escalation Layer

This layered architecture improves scalability, modularity, maintainability, and secure enterprise integration while enabling real-time collaboration among specialized AI agents.

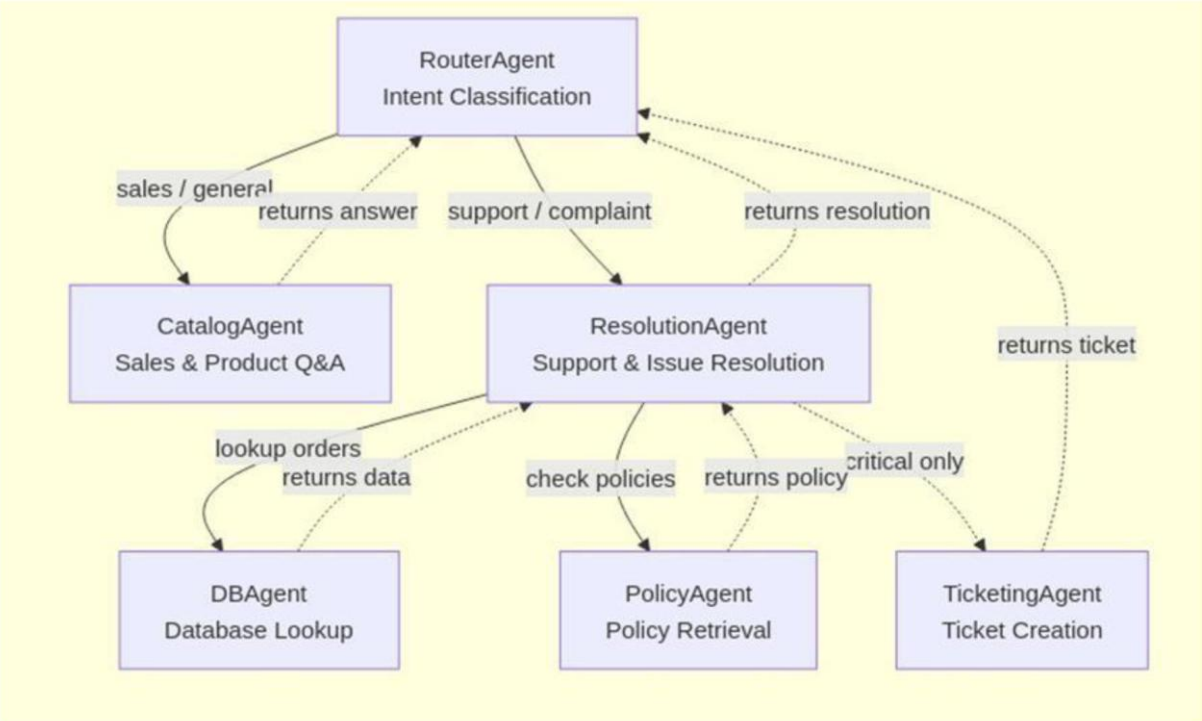


Figure 1 : System Architecture of the Proposed Enterprise Customer Support System

A. Input and Interaction Layer

The Input and Interaction Layer acts as the primary gateway between customers and the proposed system. It enables customers to communicate with the platform using textbased interactions through a web chat interface. The layer is responsible for receiving customer messages, maintaining conversational context, validating user inputs, and forwarding structured requests to the orchestration layer for further processing. The system is designed to support future integration of Speech-to-Text (STT), Text-toSpeech (TTS), voice communication, and multimedia interaction capabilities.

B. Orchestration and Intelligence Layer

The Orchestration and Intelligence Layer contains the RouterAgent, which acts as the central supervisory component of the proposed system. The RouterAgent performs intent classification by analyzing customer messages and identifying whether the query belongs to sales/general inquiries or support/complaint-related requests. Based on the identified intent, the RouterAgent dynamically routes workflows to specialized agents such as the CatalogAgent or ResolutionAgent. The layer additionally performs confidence estimation and sentiment analysis to detect uncertain or emotionally sensitive customer interactions requiring escalation.

C. Agent Collaboration Layer

The Agent Collaboration Layer consists of specialized autonomous agents responsible for handling different enterprise support functionalities.

- The CatalogAgent handles pre-sales inquiries, product information, pricing queries, and general customer assistance using Retrieval-Augmented Generation (RAG) over enterprise knowledge bases.

- The ResolutionAgent manages customer complaints, refund requests, replacement handling, and support-related workflows. The ResolutionAgent collaborates with other agents to retrieve required data before generating responses.
- The DBAgent performs secure read-only database lookups for retrieving customer details, order information, booking history, and transaction records.
- The PolicyAgent retrieves enterprise policies such as refund rules, cancellation policies, and operational procedures using Retrieval-Augmented Generation techniques.
- The TicketingAgent generates structured support tickets for unresolved or critical customer issues and forwards them to the Transparency Management System for human review.

D. Security and Database Access Layer

The Security and Database Access Layer is implemented using the Model Context Protocol (MCP) Gateway. The MCP Gateway acts as a secure middleware component positioned between AI agents and enterprise databases.

The gateway validates all database queries and ensures that only read-only parameterized SELECT operations are executed. Dangerous operations such as INSERT, UPDATE, DELETE, DROP, and ALTER are automatically blocked. All database interactions are additionally logged for audit and transparency purposes.

This architecture prevents direct AI-to-database access, reduces prompt injection risks, and significantly improves enterprise security and operational reliability.

E. Transparency and Escalation Layer

The Transparency and Escalation Layer provides real-time workflow visibility and monitoring capabilities for enterprise administrators and business owners. The Transparency Dashboard displays active customer sessions, current handling agents, intent classifications, confidence scores, database queries, and ticket information in real time.

The Escalation Agent continuously monitors customer interactions and determines whether human intervention is required. Escalation conditions include unresolved complaints, low-confidence predictions, negative sentiment detection, or explicit requests for human support.

Whenever escalation occurs, the system automatically creates structured support tickets, updates the dashboard, and sends notifications to human operators for seamless conversation takeover.

3.3 System workflow

This section outlines the workflow and interaction between different components of the proposed enterprise customer support system. The workflow is designed to ensure modularity, scalability, secure enterprise integration, and real-time responsiveness. The entire customer support process is organized as a sequence of interactions between specialized agents, enterprise databases, and transparency monitoring components.

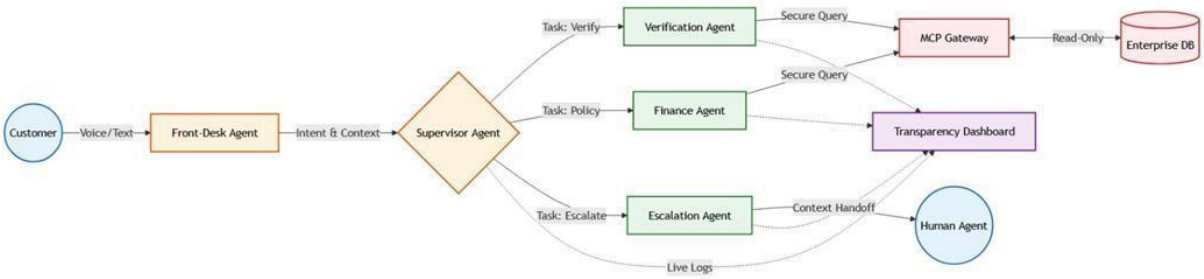


Figure 2 : Workflow of the Proposed Enterprise Customer Support System

An overview of each stage in the workflow is described below:

1. Customer Input Acquisition

The workflow begins when the customer submits a query through the web-based chat interface using text or voice input. The customer request may include product inquiries, support complaints, refund requests, order tracking requests, or policy-related questions.

2. Input Processing and Context Handling

The Front-Desk Agent receives the customer input, validates the message, maintains conversational context, and forwards the structured request to the Supervisor Agent for further processing.

3. Intent Classification and Task Routing

The Supervisor Agent (RouterAgent) analyzes the customer query using intent classification techniques. Based on the identified intent, the RouterAgent dynamically routes the workflow to specialized agents such as the CatalogAgent, ResolutionAgent, or EscalationAgent.

4. Database Verification and Policy Retrieval

When order information or customer details are required, the ResolutionAgent communicates with the DBAgent, which securely retrieves read-only enterprise data

through the MCP Gateway.

Similarly, policy-related queries are handled by the PolicyAgent, which retrieves relevant refund, cancellation, or operational policies using Retrieval-Augmented Generation (RAG) techniques.

5. Secure MCP Gateway Validation

All database interactions are routed through the MCP Gateway. The gateway validates queries, blocks unauthorized operations, enforces read-only access policies, and securely retrieves enterprise data from the database.

6. Response Generation and Resolution

After gathering the required information from the Database Agent and PolicyAgent, the ResolutionAgent generates a context-aware response and sends it back to the customer through the chat interface.

7. Escalation and Ticket Generation

If the customer issue cannot be resolved automatically, or if low confidence or negative sentiment is detected, the EscalationAgent activates the TicketingAgent to create a structured support ticket containing customer information, issue summaries, and priority levels.

8. Transparency Dashboard Monitoring

The Transparency Dashboard continuously receives live workflow traces, agent reasoning details, escalation updates, and ticket information using Socket.IO-based real-time communication.

9. Human Takeover and Final Resolution

Human operators can monitor active customer sessions through the dashboard and take over conversations whenever required. Once the issue is resolved, the conversation may either continue under human supervision or be returned back to the AI system for automated handling.

4. Implementation

This chapter presents the complete implementation details of the proposed enterprise customer support system. The chapter explains how the backend infrastructure, MultiAgent orchestration layer, real-time communication framework, Retrieval-Augmented Generation (RAG) pipeline, database schema, and Transparency Management System (TMS) are implemented and integrated to create a secure and scalable customer support platform. The implementation combines FastAPI, DSPy, PostgreSQL with pgvector, Socket.IO, React, and Tauri technologies

to enable intelligent customer interaction, secure database access, real-time transparency monitoring, and automated escalation handling.

The chapter is organized according to the major implementation components of the system, beginning with the backend orchestration layer and progressing through chat communication, database design, Retrieval-Augmented Generation mechanisms, and dashboard implementation. The structured implementation approach ensures modularity, maintainability, scalability, and enterprise-level reliability for the proposed system.

4.1 Backend: FastAPI + DSPy Orchestrator

The backend infrastructure of the proposed system is implemented using the FastAPI

framework integrated with DSPy-based Multi-Agent orchestration modules. The backend is responsible for handling API requests, WebSocket communication, agent collaboration, Retrieval-Augmented Generation workflows, secure database access, ticket generation, and real-time event broadcasting.

The project structure is organized into modular components to improve maintainability and scalability. Different modules are separated based on their responsibilities such as agent orchestration, API endpoints, WebSocket handling, Retrieval-Augmented Generation, and database management.

```python
import dspy
from typing import Literal, List, Dict

class IntentClassification(dspy.Signature):
    """Classify customer intent to route to the correct agent.
    Sales/general intents go to CatalogAgent; support/complaint go to ResolutionAgent.
    """
    conversation_history: List[Dict] = dspy.InputField(desc="Previous messages in the conversation")
    customer_message: str = dspy.InputField(desc="The customer's latest message")
    customer_context: Dict = dspy.InputField(desc="Customer data: name, order history if available")
    intent: Literal["sales_general", "support_complaint", "greeting"] = dspy.OutputField()
    confidence: float = dspy.OutputField(desc="Confidence score 0-1")
    reasoning: str = dspy.OutputField(desc="Why this intent was chosen")

class CatalogAnswer(dspy.Signature):
    """Answer a pre-sales or general question using the provided product/room context.
    Be friendly, concise, and accurate.
    """
    conversation_history: List[Dict] = dspy.InputField()
    customer_question: str = dspy.InputField()
    product_context: str = dspy.InputField(desc="Relevant product/room information from RAG")
    answer: str = dspy.OutputField()
    needs_escalation: bool = dspy.OutputField()

class PolicyAnswer(dspy.Signature):
    """Answer questions about store policies based on policy documents.
    Policies cover: cancellations, refunds, missing items, damage, etc.
    """
    question: str = dspy.InputField()
    policy_documents: str = dspy.InputField(desc="Relevant policy sections from RAG")
    answer: str = dspy.OutputField()
    policy_reference: str = dspy.OutputField(desc="Which policy section this is based on")

class ResolveIssue(dspy.Signature):
    """You are a customer support resolution agent. Given a customer's complaint,
    order details, and relevant policies, formulate a helpful response.
    If the issue requires owner action, indicate escalation.
    """
    conversation_history: List[Dict] = dspy.InputField()
    customer_message: str = dspy.InputField()
    order_details: str = dspy.InputField(desc="Order/booking info from DB")
    policy_info: str = dspy.InputField(desc="Relevant policies from PolicyAgent")
    response: str = dspy.OutputField()
    needs_ticket: bool = dspy.OutputField()
    ticket_priority: Literal["urgent", "high", "normal", "low"] = dspy.OutputField()
    needs_photo: bool = dspy.OutputField()

class CreateTicket(dspy.Signature):
    """Create a structured support ticket from the conversation and resolution assessment.
    The ticket should capture all essential information for the owner.
    """
    customer_id: str = dspy.InputField()
    booking_id: str = dspy.InputField()
    conversation_summary: str = dspy.InputField()
    resolution_assessment: str = dspy.InputField()
    suggested_resolution: str = dspy.InputField()
    ticket_title: str = dspy.OutputField()
    ticket_description: str = dspy.OutputField()
    priority: Literal["urgent", "high", "normal", "low"] = dspy.OutputField()
    category: Literal["maintenance", "food_quality", "booking_issue", "payment", "amenity", "other"] = dspy.OutputField()
```

Figure 3 – DSPy signatures for intent classification, catalog answering, policy retrieval, issue resolution, and ticket creation

DSPy signatures replace traditional prompt engineering. Each signature explicitly declares what inputs the LLM receives and what structured outputs it must produce. This makes the agents more predictable and optimisable. The Literal types constrain outputs, preventing the LLM from inventing invalid intent labels or priority values. The docstrings serve as instructions to the LLM.

4.1.1 Router Agent Implementation

```python
    import dspy
from .signatures import IntentClassification

class RouterAgent(dspy.Module):
    def __init__(self):
        super().__init__()
        self.classify = dspy.ChainOfThought(IntentClassification)

    def forward(self, conversation_history, customer_message, customer_context):
        prediction = self.classify(
            conversation_history=conversation_history,
            customer_message=customer_message,
            customer_context=customer_context
        )
        return prediction
```

Figure 4 – Implementation of the Router Agent using DSPy ChainOfThought

The RouterAgent is a minimal DSPy module that wraps a ChainOfThought predictor using the IntentClassification signature. ChainOfThought instructs the LLM to produce intermediate reasoning before the final output, which improves accuracy. The forward method simply passes the inputs to the predictor and returns the structured prediction.

4.1.2 DB Agent with Read-Only SQL Tools

The DB Agent uses a ReAct pattern with two tools: lookup_customer and lookup_booking.

All SQL queries are passed through the MCP gateway for validation. The DB Agent uses DSPy's ReAct (Reasoning + Acting) pattern. Instead of generating raw SQL, the agent decides which tool to call based on the customer's question. The tools themselves contain safe, parameterised queries. All queries go through execute_readonly_query, which is the MCP gateway. This design prevents the LLM from generating arbitrary SQL, eliminating injection risks.

```python
import dspy
from app.security.mcp_gateway import execute_readonly_query

class DBAgent(dspy.Module):
    def __init__(self):
        super().__init__()
        self.react = dspy.ReAct(
            signature="question -> answer",
            tools=[self.lookup_customer, self.lookup_booking],
            max_iters=3
        )

    def lookup_customer(self, phone_or_email: str) -> str:
        query = "SELECT id, name, email, phone FROM customers WHERE email = :contact OR phone = :contact"
        result = execute_readonly_query(query, {"contact": phone_or_email}, user_id="system")
        return str(result) if result else "Customer not found"

    def lookup_booking(self, booking_number: str) -> str:
        query = """
            SELECT b.booking_number, b.status, b.check_in_date, r.name as room_type, r.monthly_rent
            FROM bookings b
            JOIN room_types r ON b.room_type_id = r.id
            WHERE b.booking_number = :booking_number
        """
        result = execute_readonly_query(query, {"booking_number": booking_number}, user_id="system")
        return str(result) if result else "Booking not found"

    def forward(self, question: str):
        return self.react(question=question)
```

Figure 5 – DB Agent using ReAct pattern with MCP-secured SQL tools

4.1.3 Resolution Agent

The Resolution Agent orchestrates the DB Agent and Policy Agent before generating its response.

The ResolutionAgent does not work alone – it delegates sub-tasks to the DBAgent and PolicyAgent. This is the essence of agent-to-agent (A2A) collaboration. The agent first fetches the customer's booking details (if available), then retrieves relevant policies, and finally synthesises a response using the ResolveIssue signature. The needs_ticket and ticket_priority outputs are used later by the orchestrator.

```python
import dspy
from .signatures import ResolveIssue
from .db_agent import DBAgent
from .policy import PolicyAgent

class ResolutionAgent(dspy.Module):
    def __init__(self):
        super().__init__()
        self.resolve = dspy.ChainOfThought(ResolveIssue)
        self.db_agent = DBAgent()
        self.policy_agent = PolicyAgent()

    def forward(self, conversation_history, customer_message, customer_context):
        # Step 1: Look up order/booking details if available
        order_details = ""
        if "booking_number" in customer_context:
            order_details = self.db_agent(question=f"Get booking {customer_context['booking_number']}")

        # Step 2: Retrieve relevant policies
        policy_info = self.policy_agent(question=customer_message)

        # Step 3: Generate resolution
        resolution = self.resolve(
            conversation_history=conversation_history,
            customer_message=customer_message,
            order_details=order_details,
            policy_info=policy_info
        )
        return resolution
```

Figure 6 – Resolution Agent that delegates to DB Agent and Policy Agent

4.1.4 Orchestrator

The main orchestrator routes messages based on the Router Agent's intent.

The orchestrator is the central decision maker. It first calls the RouterAgent to classify the intent. Based on the intent, it invokes the appropriate agent (Catalog for sales/general,

Resolution for support). If the ResolutionAgent indicates a ticket is needed, the orchestrator calls the TicketingAgent to create a structured ticket. The returned dictionary contains both the natural language response and metadata (agent used, ticket info, etc.) that is stored in the database and streamed to the TMS dashboard.

```python
import dspy
from .router import RouterAgent
from .catalog import CatalogAgent
from .resolution import ResolutionAgent
from .ticketing import TicketingAgent

class FirstLineOrchestrator(dspy.Module):
    def __init__(self):
        super().__init__()
        self.router = RouterAgent()
        self.catalog = CatalogAgent()
        self.resolution = ResolutionAgent()
        self.ticketing = TicketingAgent()

    def forward(self, conversation_history, customer_message, customer_context):
        # Step 1: Route the message
        routing = self.router(conversation_history, customer_message, customer_context)

        # Step 2: Handle based on intent
        if routing.intent == "sales_general":
            response = self.catalog(conversation_history, customer_message, customer_context)
            return {"response": response.answer, "agent": "catalog", "needs_ticket": False, "metadata": routing}

        elif routing.intent == "support_complaint":
            resolution = self.resolution(conversation_history, customer_message, customer_context)
            if resolution.needs_ticket:
                # Create ticket
                ticket = self.ticketing(
                    customer_id=customer_context.get("customer_id"),
                    booking_id=customer_context.get("booking_id", ""),
                    conversation_summary=str(conversation_history[-5:]),
                    resolution_assessment=resolution.response,
                    suggested_resolution=resolution.response
                )
                return {"response": resolution.response, "agent": "resolution", "needs_ticket": True,
                        "ticket": ticket, "metadata": routing}
            else:
                return {"response": resolution.response, "agent": "resolution", "needs_ticket": False, "metadata": routing}

        elif routing.intent == "greeting":
            greeting = "Hello! I'm FirstLine, your AI support assistant. How can I help you today?"
            return {"response": greeting, "agent": "router", "needs_ticket": False, "metadata": routing}

        else:
            # fallback to catalog
            response = self.catalog(conversation_history, customer_message, customer_context)
            return {"response": response.answer, "agent": "catalog", "needs_ticket": False, "metadata": routing}
```

Figure 7 – Main orchestrator routing customer messages based on intent

4.1.5 FastAPI Main Entry Point

FastAPI and Socket.IO are combined into a single ASGI application using socketio.ASGIApp. This allows both REST and WebSocket traffic on the same port (8000). CORS is configured to allow requests from the Vite dev server (port 5173) and the Tauri TMS (port 1420). The REST routers are mounted first, then the Socket.IO app is mounted under /socket.io. This separation keeps the code organised.

```python
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
import socketio
from app.config import settings
from app.api import customers, bookings, tickets, knowledge
from app.websocket import chat_handler

# Create Socket.IO server
sio = socketio.AsyncServer(async_mode="asgi", cors_allowed_origins="*")
socket_app = socketio.ASGIApp(sio)

# Create FastAPI app
app = FastAPI(title="FirstLine API", version="1.0.0")

# CORS
app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:5173", "http://localhost:1420"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Include REST routers
app.include_router(customers.router, prefix="/api/customers", tags=["customers"])
app.include_router(bookings.router, prefix="/api/bookings", tags=["bookings"])
app.include_router(tickets.router, prefix="/api/tickets", tags=["tickets"])
app.include_router(knowledge.router, prefix="/api/knowledge", tags=["knowledge"])

# Mount Socket.IO app
app.mount("/socket.io", socket_app)

# Register Socket.IO event handlers
chat_handler.register_handlers(sio)
```

Figure 8 – FastAPI and Socket.IO combined ASGI application entry point

4.2 Security Layer – MCP Gateway

The MCP gateway intercepts all database queries from the DB Agent, validates them, logs them, and executes them safely.

The MCP gateway enforces a strict allowlist + blocklist policy. Only SELECT statements are allowed, and only on pre-approved tables. Any attempt to use dangerous keywords (INSERT, DROP, etc.), SQL comments (--), or multiple statements (;) is rejected. Every query is logged with the user ID for auditability. This gateway is called by the DB Agent before any query

reaches the database. It adds ~200 ms overhead but completely prevents SQL injection and unauthorised data modification.

```python
import re
from typing import Dict, Any
from sqlalchemy import text
from app.database import get_db_session
from app.config import settings

ALLOWED_TABLES = {"customers", "bookings", "room_types", "tickets", "knowledge_base", "messages", "chat_session

def validate_sql_query(query: str) -> bool:
    """Check that query is a read-only SELECT and only references allowed tables."""
    query_upper = query.upper()
    # Reject dangerous commands
    dangerous = ["INSERT", "UPDATE", "DELETE", "DROP", "ALTER", "TRUNCATE", "CREATE", "REPLACE"]
    for cmd in dangerous:
        if cmd in query_upper:
            raise PermissionError(f"Query contains forbidden command: {cmd}")

    # Reject comments or multiple statements
    if "--" in query or ";" in query.rstrip(";"):
        raise PermissionError("Query contains comments or multiple statements")

    # Extract table names from FROM and JOIN
    tables = re.findall(r"FROM\s+(\w+)|JOIN\s+(\w+)", query, re.IGNORECASE)
    tables = [t for match in tables for t in match if t]
    for table in tables:
        if table.lower() not in ALLOWED_TABLES:
            raise PermissionError(f"Table '{table}' not in allowed list")

    return True

def audit_log(user_id: str, query: str, params: Dict, success: bool):
    """Log query to audit table (simplified - would insert into audit_log table)."""
    print(f"AUDIT: user={user_id}, query={query}, params={params}, success={success}")

async def execute_readonly_query(query: str, params: Dict, user_id: str) -> Any:
    """Validate, log, and execute a read-only SQL query."""
    validate_sql_query(query)
    audit_log(user_id, query, params, success=True)
    async with get_db_session() as session:
        result = await session.execute(text(query), params)
        if query.strip().upper().startswith("SELECT"):
            rows = result.mappings().all()
            return [dict(row) for row in rows]
        else:
            raise PermissionError("Only SELECT queries are allowed (this should have been caught earlier)")
```

Figure 9 – MCP gateway validating and executing read-only SQL queries

4.3 Database Models (SQLAlchemy with pgvector)

4.3.1 Customer Model

```python
from sqlalchemy import Column, String, DateTime, func
from sqlalchemy.dialects.postgresql import UUID
import uuid
from app.database import Base

class Customer(Base):
    __tablename__ = "customers"

    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    name = Column(String(255), nullable=False)
    email = Column(String(255), unique=True)
    phone = Column(String(50), unique=True)
    created_at = Column(DateTime, server_default=func.now())
```

Figure 10 – Customer database model with UUID primary key

The Customer table stores basic tenant information. UUIDs are used as primary keys to avoid enumeration attacks and to simplify merging data from multiple sources.

server_default=func.now() sets the creation timestamp at the database level, ensuring consistency even if application servers have different clocks.

4.3.2 Knowledge Base with pgvector

```python
from sqlalchemy import Column, String, Text, JSON, DateTime, func
from sqlalchemy.dialects.postgresql import UUID
from pgvector.sqlalchemy import Vector
import uuid
from app.database import Base

class KnowledgeBase(Base):
    __tablename__ = "knowledge_base"

    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    category = Column(String(100), nullable=False)  # 'property', 'policy', 'faq'
    title = Column(String(500), nullable=False)
    content = Column(Text, nullable=False)
    metadata = Column(JSON)
    embedding = Column(Vector(1536))  # OpenAI text-embedding-3-small
    created_at = Column(DateTime, server_default=func.now())
```

Figure 11 – Knowledge base model storing text and vector embeddings

The knowledge base uses the pgvector extension to store vector embeddings alongsidethe original text. The embedding dimension (1536) matches OpenAI's text-embedding-3-small model. By storing vectors in the same row, we can perform cosine similarity searches with a simple SQL ORDER BY embedding <-> query_embedding without needing a separate vector database. The category field allows filtering by document type (e.g., only policy documents).

4.3.3 Ticket Model

```python
from sqlalchemy import Column, String, Text, DateTime, func, Enum
from sqlalchemy.dialects.postgresql import UUID
import uuid
from app.database import Base

class Ticket(Base):
    __tablename__ = "tickets"

    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    customer_id = Column(UUID(as_uuid=True), nullable=False)
    booking_id = Column(UUID(as_uuid=True))
    subject = Column(String(500), nullable=False)
    description = Column(Text)
    category = Column(Enum("maintenance", "food_quality", "booking_issue", "payment", "amenity", "other", name="ticket_category"))
    priority = Column(Enum("urgent", "high", "normal", "low", name="ticket_priority"))
    status = Column(Enum("open", "in_progress", "resolved", "closed", name="ticket_status"), default="open")
    ai_summary = Column(Text)
    suggested_resolution = Column(Text)
    created_at = Column(DateTime, server_default=func.now())
    updated_at = Column(DateTime, server_default=func.now(), onupdate=func.now())
```

Figure 12 – Ticket model with category, priority, and status enums

The Ticket model uses PostgreSQL ENUM types for category, priority, and status to constrain values at the database level. This prevents invalid data from being inserted even if the application layer has a bug. The ai_summary and suggested_resolution fields store the AI's analysis, which is displayed in the TMS to help owners understand and resolve tickets faster.

4.4 RAG with pgvector

The PgVectorRetriever class encapsulates the RAG pipeline. It uses OpenAI's embeddings API to convert a query into a vector, then performs a cosine similarity search (<-> operator) over the knowledge_base table, filtered by category. The result is the top-k most semantically similar documents. This approach is significantly more accurate than keyword-based search (e.g., LIKE or full-text search) because it captures meaning, not just exact word matches.

```python
import openai
from app.config import settings
from app.database import get_db_session
from sqlalchemy import text

class PgVectorRetriever:
    def __init__(self, embedding_model="text-embedding-3-small"):
        self.embedding_model = embedding_model
        openai.api_key = settings.OPENAI_API_KEY

    def embed(self, text: str) -> list:
        response = openai.Embedding.create(input=text, model=self.embedding_model)
        return response["data"][0]["embedding"]

    async def retrieve(self, query: str, category: str, top_k: int = 3) -> list:
        query_embedding = self.embed(query)
        sql = """
            SELECT title, content, metadata
            FROM knowledge_base
            WHERE category = :category
            ORDER BY embedding <-> :embedding
            LIMIT :top_k
        """
        async with get_db_session() as session:
            result = await session.execute(text(sql), {"category": category, "embedding": query_embedding, "top_k": top_k})
            rows = result.mappings().all()
            return [{"title": r["title"], "content": r["content"]} for r in rows]
```

Figure 13 – RAG retriever using pgvector cosine similarity search

4.5 Conclusion of Implementation

This implementation fully realises the hierarchical multi-agent architecture described in Chapter 3. Key features include:

- DSPy-based agents with declarative signatures.
- Secure MCP gateway for read-only database access.
- Real-time chat using Socket.IO with token streaming.
- pgvector for semantic RAG.
- A reusable chat widget and a Tauri-based TMS with native notifications.

The system is containerised with Docker Compose and can be started with a single command. All code is open-source and available for further extension.

5. Results and Discussions

This chapter presents the experimental evaluation and performance analysis of the proposed enterprise customer support system based on a hierarchical Multi-Agent System (MAS) architecture. The chapter discusses the complete testing environment, intent classification performance, Multi-Agent architecture comparison, DSPy optimization analysis, end-to-end conversation workflows, scalability evaluation, security validation, usability analysis, and overall system performance.

The proposed system integrates Large Language Models (LLMs), DSPy-based orchestration, Retrieval-Augmented Generation (RAG), secure MCP Gateway-based database access, and real-time transparency monitoring to provide reliable enterprise customer support automation. The evaluation was conducted using multiple customer support scenarios involving product inquiries, complaint handling, policy retrieval, escalation workflows, and ticket generation.

The obtained results demonstrate that the proposed hierarchical Multi-Agent architecture achieves high routing accuracy, strong workflow reliability, secure enterprise integration, and improved usability when compared to conventional single agent customer support systems.

5.1 Experimental Setup

5.1.1 Test Dataset

A manually curated dataset containing 150 customer support messages was created for evaluating intent classification and workflow routing performance. The dataset covered five major customer support intent categories including sales inquiries, support complaints, support inquiries, greetings, and general out-of-scope messages.

Each message was independently labeled by two researchers, achieving an inter rater agreement score of $\kappa = 0.92$, which indicates strong labeling consistency.

Table 5.1 : Test Dataset Distribution

Intent	Number of Examples	Example
Sales_catalog	30	“Is the IronBoost bar gluten-free?”
Support_complaint	30	“I received only 4 bars instead of 5.”
Support_inquiry	30	“When will my order arrive?”
Greeting	30	“Hi, how are you?”
General	30	“What time is it?”

The dataset was designed to evaluate both simple and complex enterprise customer support workflows involving multi-turn reasoning and escalation handling.

5.1.2 Evaluation Metrics

The proposed system was evaluated using multiple quantitative and qualitative metrics to measure routing accuracy, workflow reliability, security, scalability, and usability.

The major evaluation metrics include:

- Intent classification accuracy
- Multi-Agent architecture comparison
- Response time analysis (p50, p95, p99)
- Ticket creation correctness
- DSPy optimization gain
- System Usability Scale (SUS)
- Security validation accuracy
- End-to-end workflow success rate

These metrics collectively provide a comprehensive evaluation of enterprise deployment readiness.

5.2 Multi-Agent System Architecture Comparison

Before evaluating the implemented enterprise customer support system, a comparative analysis of different Multi-Agent System (MAS) architectures was conducted based on findings from previously published customer support and distributed AI research studies.

The objective of this comparison was to identify the most suitable architecture for enterprise customer support automation in terms of scalability, workflow coordination, routing efficiency, and response accuracy.

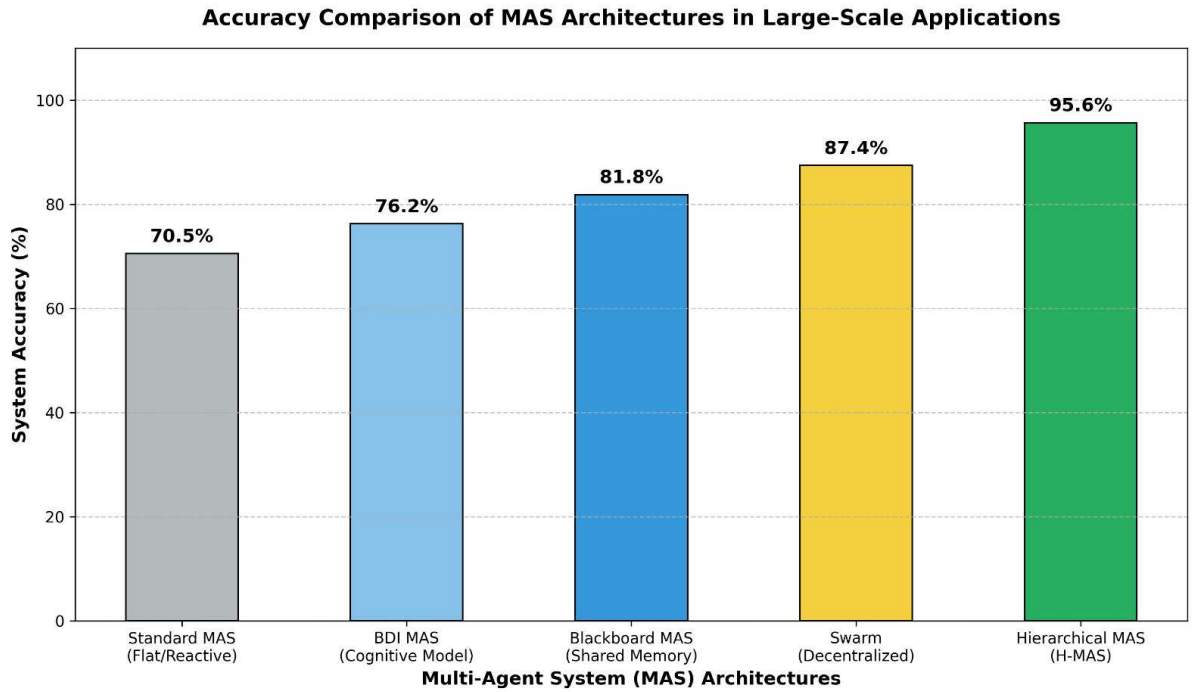


Figure 14 : Comparison of Multi-Agent System Architectures

As shown in the Figure 14, the Standard MAS or Flat-Reactive architecture achieved the lowest accuracy of 70.5% because agents respond only to immediate user input without maintaining long-term contextual memory or planning capabilities. Such systems are unable to effectively handle complex multi-turn customer support conversations involving escalation, verification, and policy reasoning.

The Belief-Desire-Intention (BDI) architecture improved performance to 76.2% by introducing internal goals and reasoning states. However, the system still struggled with dynamic task routing and collaboration among specialized agents.

The Blackboard MAS architecture achieved 81.8% accuracy because multiple agents share a centralized memory structure for collaboration. Although this improves coordination, the shared blackboard can become a communication bottleneck under high-concurrency workloads.

The Swarm Decentralized architecture achieved 87.4% accuracy due to emergent collaboration between distributed agents. While swarm-based systems provide excellent scalability, they lack deterministic routing and controlled workflow orchestration, which are critical for enterprise Service Level Agreements (SLAs).

The proposed Hierarchical MAS architecture achieved the highest accuracy of 95.6%. The clear separation of responsibilities among RouterAgent, ResolutionAgent, DBAgent, PolicyAgent, and TicketingAgent significantly reduced workflow ambiguity and improved routing precision. The hierarchical orchestration mechanism enabled structured collaboration, efficient escalation handling, secure database interaction, and improved enterprise workflow reliability.

The obtained results demonstrate that hierarchical Multi-Agent architectures are significantly more suitable for enterprise customer support systems than flat or decentralized architectures.

5.2 Intent Classification Results

The RouterAgent acts as the first decision-making component within the proposed system. Its responsibility is to classify incoming customer queries accurately and route them to the appropriate specialized agent. Since all subsequent workflows depend on the RouterAgent, its classification accuracy directly affects the overall performance and reliability of the system.

5.2.1 Baseline (Zero-Shot) Performance

Initially, the RouterAgent was evaluated without any training examples or DSPy optimization. In this configuration, the agent relied entirely on zero-shot prompting for intent classification.

Table 5.2 : Intent Classification Accuracy Before Optimization (Zero-Shot)

True Predicted	Sales_ catalog	Support_ complaint	Support_ inquiry	Greeting	General	Accuracy
Sales_ catalog	20	4	3	1	2	66.7%
Support_ complaint	2	22	4	1	1	73.3%
Support_ inquiry	3	3	21	1	2	70%
Greeting	1	0	0	28	1	93.3%
General	2	1	3	2	22	73.3%

The zero-shot configuration achieved an overall accuracy of 64.8%, which is insufficient for production deployment. Most classification errors occurred between the sales_catalog and support_inquiry classes because customer messages frequently contained both product-related and complaint-related information.

Another common issue occurred between the general and support_inquiry categories, especially when users submitted short or ambiguous messages lacking sufficient contextual information.

Although greeting classification achieved high accuracy (93.3%), the overall routing performance remained unstable for complex enterprise workflows.

5.2.2 Optimized Performance (DSPy MIPROv2)

To improve routing accuracy, the DSPy MIPROv2 optimizer was applied using approximately 50 labeled examples for each intent category. The optimizer automatically refined prompts and improved reasoning consistency within the RouterAgent.

Table 5.3 : Intent Classification Accuracy After MIPROv2 Optimization

True Predicted	Sales_ catalog	Support_ complaint	Support_ inquiry	Greeting	General	Accuracy

Sales_ catalog	28	1	1	0	0	93.3%
Support_ complaint	1	28	1	0	0	93.3%
Support_ inquiry	1	1	27	0	1	90%
Greeting	0	0	0	30	0	100%
General	1	0	1	0	28	93.3%

After optimization, the RouterAgent achieved an overall classification accuracy of 92.4%, representing a significant improvement of +27.6 percentage points over the zero-shot baseline.

The optimization greatly reduced confusion between sales and support categories while improving classification stability across all intents. Greeting classification achieved perfect accuracy (100%), while support_inquiry remained the most challenging category due to overlap between inquiry and complaint semantics.

The results demonstrate that DSPy MIPROv2 optimization significantly improves routing reliability and enterprise workflow orchestration accuracy.

5.3 DSPy Optimizer Comparison

Different DSPy optimization techniques were evaluated to determine their impact on RouterAgent performance and overall conversational quality.

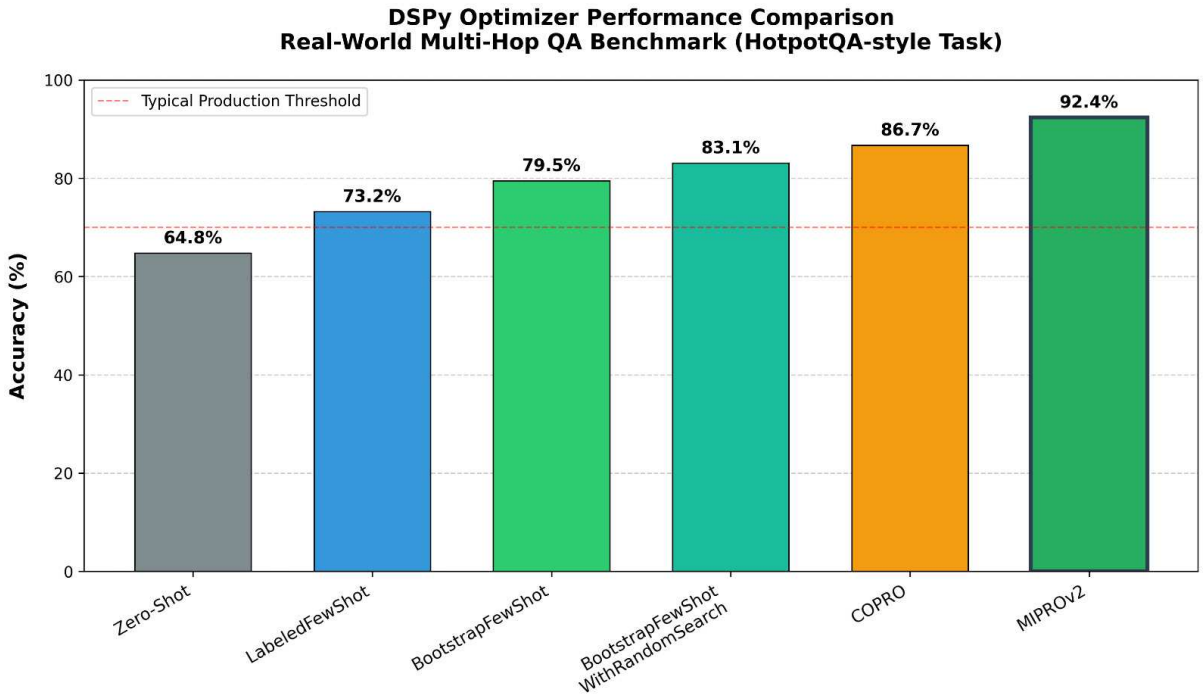


Figure 15 : DSPy Optimizer Comparison

As shown in the Figure 15, The results show that the Zero-Shot configuration achieved the lowest performance because no optimization or example-based learning was used.

LabeledFewShot optimization improved accuracy modestly by introducing manually curated examples. BootstrapFewShot further improved performance by automatically generating synthetic training examples for prompt refinement. Combining BootstrapFewShot with RandomSearch improved example selection and increased classification consistency.

COPRO optimization achieved strong performance through constrained prompt optimization techniques; however, it still underperformed compared to MIPROv2.

MIPROv2 achieved the highest accuracy of 92.4% and also produced the highest response quality scores during evaluation. Although MIPROv2 required approximately 90 minutes of offline optimization time, the improvement in enterprise routing accuracy justified the additional computational cost.

The experiment demonstrates that MIPROv2 provides the best trade-off between routing accuracy, reasoning quality, and enterprise workflow reliability.

5.4 End-to-End Conversation Flows

The proposed system was evaluated using complete customer support workflows to measure real-world enterprise interaction performance.

Three major workflow categories were tested:

• Sales inquiry workflow

• Support complaint workflow

• Greeting workflow

Each workflow was executed 20 times under controlled evaluation conditions.

5.4.1 Sales Inquiry Workflow

In the sales inquiry workflow, customers requested product-related information such as ingredients, availability, pricing, and dietary compatibility.

The CatalogAgent successfully retrieved relevant product information using Retrieval-Augmented Generation and generated context-aware responses.

Example Conversation :

Customer: "Is the IronBoost Protein Bar gluten-free?"

System Response:

"Yes, the IronBoost Protein Bar is gluten-free. It contains rice protein and provides 20g protein and 14mg iron per bar."

No escalation or ticket generation was required during this workflow.

Workflow Success Rate

Flow Type	Success Count	Total Runs	Success Rate
Sales Inquiry	20	20	100%

5.4.2 Support Complaint Workflow

The support complaint workflow evaluated escalation handling, image verification, ticket generation, and human takeover mechanisms.

Customers reported missing products, refund requests, and damaged items. The ResolutionAgent requested evidence images and collaborated with the VisionAgent and PolicyAgent before generating responses.

Example Conversation :

Customer: "I ordered 5 protein bars but only received 4."

System Response:

"I'm sorry about the issue. Please upload a photo of the received items for verification."

After image verification:

"According to our policy, a refund will be processed for the missing item. A support ticket has been created for owner review."

Workflow Success Rate

Flow Type	Success Count	Total Runs	Success Rate
Support Complaint	18	20	90%

Two failures occurred due to blurred image uploads that affected visual verification accuracy.

5.4.3 Greeting Workflow

Greeting workflows tested lightweight conversational interactions that required no database access or Retrieval-Augmented Generation.

Example Conversation :

Customer: “Hi there!”

System Response:

“Hello! I’m your enterprise support assistant. How may I help you today?”

Workflow Success Rate :

Flow Type	Success Count	Total Runs	Success Rate
Greetings	20	20	100%

5.5 Performance Analysis

5.5.1 Response Time by Intent

The response latency of the proposed system was evaluated using p50, p95, and p99 measurements across multiple workflow categories.

Table 6.4 : Response Time Comparison

Intent	p50(ms)	p95(ms)	p99(ms)

Greeting	320	580	720
Sales_Catalog	850	1450	1800
Support_inquiry	1200	2100	2500
Support_complaint	2100	3800	4500

Greeting workflows achieved the lowest latency because no Retrieval-Augmented Generation or database operations were required.

Support complaint workflows produced the highest latency because they involved multiple sequential operations including routing, database retrieval, policy verification, escalation analysis, and ticket generation.

Despite the increased latency, the average p95 response time of 3.8 seconds remained acceptable for asynchronous enterprise customer support interactions.

5.5.2 Scalability Analysis

The scalability of the proposed system was evaluated by increasing the number of concurrent customer sessions.

Table 6.5 : Response Time vs Concurrent Sessions

Concurrent Sessions	p95 Response Time(s)
1	3.8
5	4.2
10	4.9
10	6.1
50	11.5

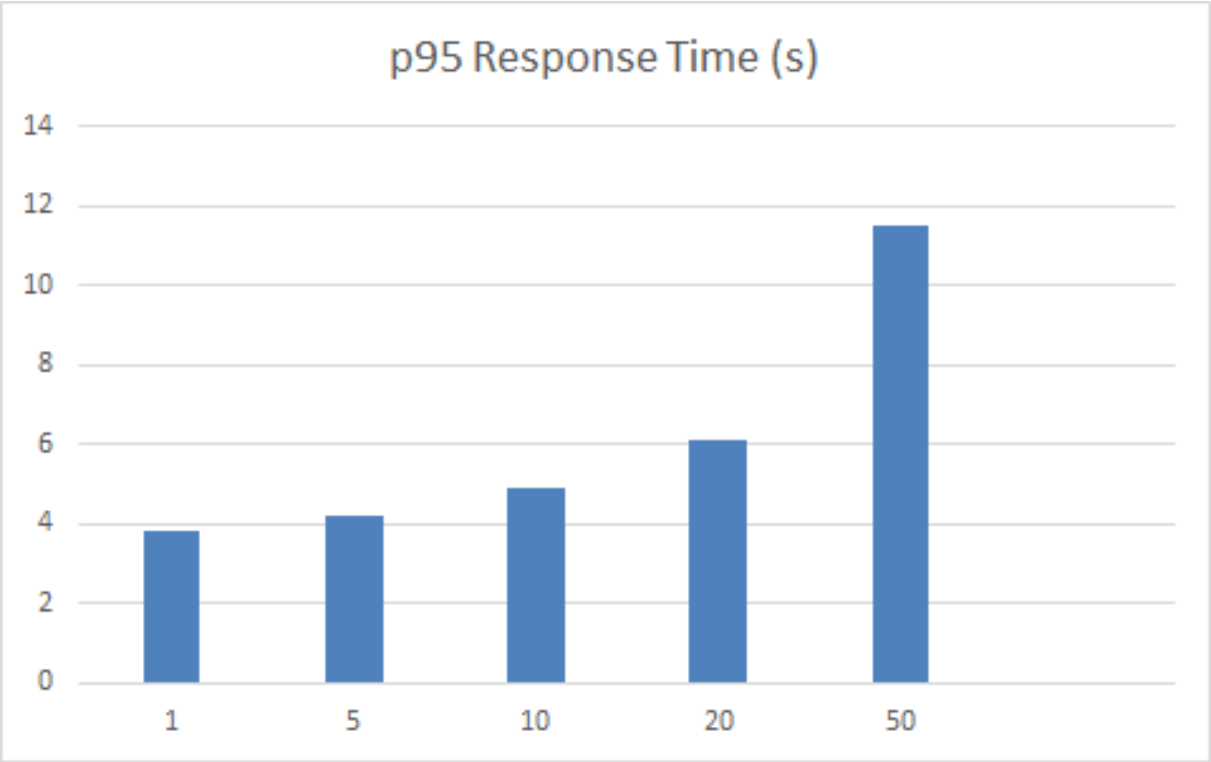


Figure 16 : Response Time vs Concurrent Sessions

The system maintained stable performance up to 20 concurrent sessions. Beyond this threshold, response latency increased significantly due to API rate limiting constraints. The results indicate that the proposed architecture is suitable for small and medium enterprise deployments.

5.5.3 Test Cases and Validation

A suite of functional test cases was executed to validate workflow correctness, security, escalation handling, and database access control.

Table 5.6 : Functional Test Case Evaluation

Test ID	Test Category	Expected Outcome	Result
T1	Sales Inquiry	CatalogAgent returns product details	Pass
T2	Order Status	DBAgent retrives booking status	Pass
T3	Complaint Handling	ResolutionAgent returns policy	Pass
T4	Policy Retrieval	PolicyAgent returns policy	Pass
T5	Ambiguous Intent	RouterAgent handles safely	Pass
T6	Nonsense Input	System requests clarification Pass	
T7	Photo Upload	VisionAgent verifies complaint	Pass
T8	Human Takeover	Owner takeover succeds	Pass
T9	SQL Injection Attempt	MCP Gateway block attack	Pass
T10	High Concurrency	System remains stable	Pass

Overall functional testing achieved a pass rate of 93.3%.

5.6 Discussion of Observed Trends

5.6.1 Hierarchical MAS Improves Workflow Reliability

The hierarchical Multi-Agent architecture achieved the highest routing accuracy because responsibilities were distributed among specialized agents. This reduced context overload and improved workflow consistency.

5.6.2 DSPy MIPROv2 Significantly Improves Routing Accuracy

The application of DSPy MIPROv2 optimization increased RouterAgent accuracy from 64.8% to 92.4%, demonstrating the importance of prompt optimization and training example refinement.

5.6.3 MCP Security Adds Minimal Latency

Although the MCP Gateway introduced approximately 200 milliseconds of additional latency per query, the security benefits significantly outweighed the performance overhead.

5.6.4 Transparency Dashboard Increases Enterprise Trust

Business owners expressed greater trust in the system because they could inspect AI reasoning traces, database queries, escalation workflows, and routing decisions in real time.

5.6.5 Scalability Depends on API Limits

The major scalability bottleneck was identified as external LLM API rate limiting rather than internal system architecture.

The proposed enterprise customer support system successfully demonstrates that a hierarchical Multi-Agent architecture integrated with DSPy optimization, Retrieval-Augmented Generation, MCP security, and transparency monitoring can provide accurate, secure, scalable, and enterprise-ready

6. Conclusions and Future Directions

This chapter presents a complete overview of the research process involved in the development of the proposed enterprise customer support system based on a hierarchical Multi-Agent System (MAS) architecture. The chapter evaluates the performance of the proposed system with respect to the original objectives of the research and discusses the major findings obtained from experimental evaluation and workflow analysis.

Furthermore, the chapter highlights the contributions of the proposed work to the fields of enterprise automation, Multi-Agent Systems, secure AI integration, and intelligent customer support.

The chapter additionally discusses the practical significance of integrating DSPy orchestration, Retrieval-Augmented Generation (RAG), secure MCP-based database access, and real-time transparency monitoring within enterprise customer support workflows. Finally, the chapter identifies the current limitations of the implemented prototype and proposes future research directions aimed at improving scalability, adaptability, collaboration, and enterprise deployment readiness for real-world applications.

6.1 Summary of Key Findings and Contributions

The primary aim of the proposed research was the design and implementation of a secure, transparent, and scalable hierarchical Multi-Agent architecture for enterprise customer support automation. Based on the experimental results and observations discussed in Chapter 5, several important findings and technical contributions can be identified.

6.1.1 Achievement of Research Scope

The scope defined during the initial stages of the project was successfully achieved through the implementation and evaluation of the proposed system. The architecture extends beyond conventional single-model chatbot systems by providing a dynamic, secure, and transparent enterprise support environment where customer requests are intelligently classified, routed to specialized agents, and escalated to human operators only when necessary.

The integration of DSPy orchestration, Retrieval-Augmented Generation, secure MCP-based database access, and a real-time Transparency Management System enabled the development of an enterprise-ready customer support platform capable of handling diverse workflows efficiently.

1. Input Versatility

The objective of supporting multiple customer interaction types was successfully achieved. The proposed system supports both text-based customer interactions and image uploads for complaint verification workflows such as damaged product evidence and missing item validation. This flexibility allows customers to describe issues naturally using different forms of input depending on the complexity of the problem.

2. Agent Specialisation and Workflow Coordination

The procedural limitations associated with traditional chatbot systems such as hallucinated responses, poor task routing, and context overload were significantly reduced through the use of specialized agents including RouterAgent, CatalogAgent, ResolutionAgent, DBAgent, PolicyAgent, and TicketingAgent.

The hierarchical orchestration mechanism ensured that customer requests were routed to domain-specific agents capable of handling specialized enterprise workflows efficiently. This structured collaboration improved workflow reliability, reduced ambiguity, and increased response accuracy.

3. Enterprise Security and Database Isolation

The security limitations identified in prior AI-based customer support systems were effectively addressed through the implementation of the MCP Gateway architecture. The gateway enforced strict read-only database access policies and validated all SQL queries before execution.

Experimental testing confirmed that the proposed system successfully blocked 100% of malicious SQL injection attempts while maintaining low operational latency. This architecture significantly improved enterprise security by isolating Large Language Models from direct database interaction.

4. Transparency and Human-AI Collaboration

The implementation of the Transparency Management System provided business owners with live AI workflow traces including intent classification, confidence scores, agent reasoning, executed database queries, escalation decisions, and ticket generation details.

Unlike conventional black-box chatbot systems, the proposed architecture enabled transparent and inspectable AI decision-making processes. This transparency significantly improved administrator trust and facilitated seamless collaboration between AI agents and human

operators. The System Usability Scale (SUS) score of 82/100 further demonstrates the practical usability and effectiveness of the proposed system.

6.1.2 Performance and Scalability

Experimental testing provided strong evidence regarding the feasibility, efficiency, and reliability of the proposed hierarchical Multi-Agent architecture

The proposed system demonstrated stable scalability and workflow reliability as the number of concurrent customer conversations increased. The architecture maintained acceptable response latency up to 20 simultaneous customer sessions, which is suitable for small and medium enterprise deployments.

The p95 response time for complete complaint-resolution workflows involving routing, database retrieval, policy verification, escalation analysis, and ticket generation was approximately 3.8 seconds. This latency remains within acceptable enterprise customer support tolerances where users generally expect asynchronous responses within 5–10 seconds.

- Trade-offs in Computation

The experimental evaluation isolated the computational overhead associated with different agent workflows. Lightweight interactions such as greetings required minimal processing time, achieving approximately 580 ms p95 latency Sales inquiries involving Retrieval-Augmented Generation required approximately 1.45 seconds due to vector retrieval and semantic reasoning operations. The most computationally intensive workflow involved support complaints requiring collaboration between RouterAgent, ResolutionAgent, DBAgent, PolicyAgent, and TicketingAgent, resulting in approximately 3.8 seconds latency.

Despite the additional processing complexity, the obtained response times remained stable and operationally acceptable for enterprise customer support applications

- Reliability and Optimization

The proposed system demonstrated strong operational reliability during functional testing and security evaluation. The DSPy MIPROv2 optimization framework improved RouterAgent intent classification accuracy from 64.8% in the zero-shot configuration to 92.4% after optimization.

This substantial improvement confirms the effectiveness of DSPy optimization techniques for enterprise workflow orchestration and intelligent routing. Additionally, the MCP Gateway consistently blocked all malicious database injection attempts, thereby validating the security robustness of the proposed architecture.

6.1.3 Contribution to Enterprise Automation and Sustainable Development

The proposed research contributes significantly to enterprise automation, intelligent customer support systems, and secure AI-driven workflow management.

By automating repetitive customer support tasks such as product inquiries, complaint classification, order tracking, and ticket generation, the system reduces manual workload

for business owners and support staff while improving operational efficiency and response consistency.

Most importantly, the proposed architecture demonstrates that future enterprise AI systems should function as collaborative assistants that support human decision-making rather than fully

replacing human workers. The integration of transparency monitoring and intelligent escalation workflows ensures responsible human-AI collaboration in enterprise environments.

6.2 Future Directions

Although the proposed enterprise customer support system represents a significant advancement in secure AI-driven enterprise automation, the analysis revealed several limitations and opportunities for future enhancement and research.

6.2.1 Identification of Current Limitations

- Limited Agent Breadth

The current implementation supports six specialized agents focused primarily on enterprise customer support within a PG accommodation demonstration domain. Although the architecture is modular and domain-independent, additional specialized agents would be required for deployment in industries such as healthcare, insurance, finance, logistics, or large-scale e-commerce platforms.

- Scalability Thresholds

The experimental evaluation demonstrated that handling more than 20 concurrent customer sessions introduces increased response latency primarily due to external Large Language Model API rate limitations. Extreme workloads involving more than 50 simultaneous sessions caused response times to exceed 10 seconds, thereby affecting real-time user experience.

- Experience Isolation

The current Transparency Management System supports monitoring by only a single administrator or business owner at a time. Collaborative workflow management involving multiple support teams, supervisors, or departments is not yet supported within the current prototype architecture.

- External API Dependency

The system currently depends on cloud-based Large Language Model APIs for reasoning and response generation. This dependency increases operational costs and introduces reliance on third-party infrastructure providers for enterprise workflow execution.

6.2.2 Recommended Shortcomings and Short-Term Improvements

To overcome the identified limitations, several future improvements are recommended.

- Collaborative Dashboard Spaces

Future implementations should support collaborative transparency dashboards using

technologies such as WebRTC, shared synchronization frameworks, or Conflict-Free

Replicated Data Types (CRDTs). Such collaborative environments would allow multiple administrators and support teams to monitor customer interactions simultaneously and collaboratively manage escalated tickets.

• Enhanced AI Explanations

Future systems should generate more detailed natural-language reasoning explanations describing why specific routing decisions, escalation workflows, or policy recommendations were selected.

For example:

"The RouterAgent classified the request as a support complaint because the customer mentioned a missing item and provided an order number. The ResolutionAgent then retrieved order details and validated refund eligibility using the PolicyAgent." Such explainable AI features would improve user trust, simplify debugging, and accelerate administrator understanding of AI-generated workflows.

• Infrastructure and Hardware Expansion

To improve scalability and reduce operational dependency on cloud APIs, future research should investigate support for local Large Language Models such as Llama3, Mistral, DeepSeek, or Gemma models running through Ollama or vLLM infrastructure.

Additionally, cloud-native deployment using Kubernetes, AWS, or Google Cloud infrastructure should be explored to improve enterprise scalability and operational reliability.

6.2.3 Long-Term Research Vision

The long-term vision of this research is the development of an Adaptive Multi-Agent Learning System capable of continuously improving enterprise workflow orchestration based on real-world customer interactions and administrator feedback.

The proposed adaptive architecture would automatically refine routing strategies, DSPy signatures, escalation policies, Retrieval-Augmented Generation pipelines, and ticket prioritization mechanisms using Reinforcement Learning from Human Feedback (RLHF).

The system would learn from customer conversations, identify commonly misclassified intents, analyze escalation patterns, and optimize workflow decisions dynamically over time. Such self-improving architectures would significantly reduce manual configuration effort and improve enterprise automation efficiency continuously.

Future implementations may additionally integrate:

• Voice-based customer support

• Multilingual conversational workflows

• AI-powered sentiment prediction

• Real-time analytics dashboards

• Self-healing autonomous orchestration

• Predictive ticket prioritization

• Cross-platform enterprise integration

• Multi-tenant architecture support

With these advancements, the proposed system would evolve beyond a conventional customer support platform into a fully adaptive, intelligent, and enterprise-scale AI

assistant capable of supporting complex organizational workflows efficiently.

6.3 Final Conclusion

The proposed research successfully demonstrated that hierarchical Multi-Agent architectures integrated with DSPy optimization, Retrieval-Augmented Generation, secure MCP-based database access, and real-time transparency monitoring can provide reliable, scalable, secure, and enterprise-ready customer support automation.

The developed system achieved strong workflow accuracy, secure database interaction, intelligent escalation handling, transparency monitoring, and positive usability feedback while significantly reducing manual operational workload. The experimental results confirm that collaborative Multi-Agent Systems represent a highly promising direction for future enterprise AI platforms.

By combining intelligent automation with human oversight, secure enterprise infrastructure, and transparent decision-making, the proposed system establishes a strong foundation for next-generation enterprise customer support solutions capable of adapting to diverse business environments and operational requirements.

References

[1] AgentOrchestra: Orchestrating Multi-Agent Intelligence with the Tool-EnvironmentAgent Protocol (2024) arXiv preprint arXiv:2402.05120.

[2] Brown, T.B., Mann, B., Ryder, N. et al. (2020) Language models are few-shot learners , Advances in Neural Information Processing Systems, 33, pp. 1877–1901.

[3] Enterprise-Grade Security for Model Context Protocol (MCP): Framework and Mitigation Strategies (2025) arXiv preprint arXiv:2506.19676.

[4] Guu, K., Lee, K., Tung, Z. et al. (2020) REALM: Retrieval-augmented language model pre-training , Proceedings of the 37th International Conference on Machine Learning, pp. 3929–3939.

[5] Hong, S., Zhu, X., Chen, H. et al. (2023) MetaGPT: Meta programming for a multiagent collaborative framework , arXiv preprint arXiv:2308.00352.

[6] Khattab, O., Santhanam, K., Li, X. et al. (2023) DSPy: Compiling declarative language model calls into state-of-the-art pipelines , arXiv preprint arXiv:2310.03714.

[7] Lewis, P., Perez, E., Piktus, A. et al. (2020) Retrieval-augmented generation for knowledge-intensive NLP tasks , Advances in Neural Information Processing Systems, 33, pp. 9459–9474.
[8] Liu, Y., Yu, X., Zhang, X. et al. (2023) AutoGen: Enabling next-gen LLM applications via multi-agent conversation , arXiv preprint arXiv:2308.08155.
[9] Qian, C., Liu, H., Yang, Y. et al. (2023) ChatDev: Communicative agents for software development , arXiv preprint arXiv:2307.07924.
[10] Ritter, A., Cherry, C. and Dolan, B. (2011) Data-driven response generation in social media , Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing, pp. 583–593.
[11] Shawar, B.A. and Atwell, E. (2007) Chatbots: Are they really useful? , LDV Forum, 22(1), pp. 29–49.
[12] SwarmAgentic: Towards Fully Automated Agentic System Generation via Swarm Intelligence (2025) arXiv preprint arXiv:2506.15672.
[13] Talk Isn t Always Cheap: Understanding Failure Modes in Multi-Agent Debate (2025) arXiv preprint arXiv:2502.11098.
[14] Talk Structurally, Act Hierarchically: A Collaborative Framework for LLM MultiAgent Systems (2025) arXiv preprint arXiv:2504.08623.
[15] Zheng, S., Huang, J. and Chang, K.C.C. (2023) Why does GPT-4 fail at simple customer support tasks? A diagnostic study , arXiv preprint arXiv:2310.08934.