An Asynchronous, State-Driven Agentic Framework for Automating B2B Sales Outreach

Project-I (CH47023) report submitted to

Indian Institute of Technology Kharagpur

in partial fulfilment for the award of the degree of

Bachelor of Technology

in

Chemical Engineering

by

Jiten Shah

(22CH30041)

Under the supervision of

Professor S.P. Pal

Department of Chemical Engineering

Indian Institute of Technology Kharagpur

Autumn Semester, 2025-26

November 30, 2025

DECLARATION

I certify that

(a) The work contained in this report has been done by me under the guidance of my supervisor.

(b) The work has not been submitted to any other Institute for any degree or diploma.

(c) I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.

(d) Whenever I have used materials (data, theoretical analysis, figures, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references. Further, I have taken permission from the copyright owners of the sources, whenever necessary.

Date: November 30, 2025 (Jiten Shah)
Place: Kharagpur (22CH30041)

DEPARTMENT OF CHEMICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

KHARAGPUR - 721302, INDIA

CERTIFICATE

This is to certify that the project report entitled "**An Asynchronous, State-Driven Agentic Framework for Automating B2B Sales Outreach**" submitted by **Jiten Shah** (Roll No. 22CH30041) to Indian Institute of Technology Kharagpur towards partial fulfilment of requirements for the award of degree of Bachelor of Technology in Chemical Engineering is a record of bona fide work carried out by him under my supervision and guidance during Autumn Semester, 2025-26.

Date: November 30, 2025
Place: Kharagpur

Professor S.P. Pal
Department of Computer Science
Indian Institute of Technology Kharagpur
Kharagpur - 721302, India

Abstract

The domain of Business-to-Business (B2B) sales development involves repetitive, time-intensive tasks that limit scalability and personalization. This thesis presents the design and implementation of an autonomous multi-agent system to automate the end-to-end sales outreach workflow. The system adopts an asynchronous, state-driven architecture where specialized agents communicate exclusively through state changes in a central database, effectively managing long-running human-in-the-loop interactions such as email replies.

The framework comprises a hybrid of deterministic and Large Language Model (LLM)-powered agents. It features a rule-based Prospector for deterministic lead scoring, while creative and interpretive tasks are handled by LLM-powered agents like the Strategist for message generation and the Interpreter for understanding inbound replies. Based on the detected intent, leads are dynamically routed to a Scheduler for meeting coordination or a RecordKeeper for archiving interactions. An Analyst aggregates performance metrics to provide feedback that improves future outreach strategies, creating a self-optimizing loop within the system.

Implemented in Python using LangGraph, the framework integrates with the Gmail API to enable real-world, closed-loop communication and Google Calendar API for scheduling meetings. The resulting prototype demonstrates a robust and scalable architecture for intelligent automation in complex business processes, validating the effectiveness of a decoupled, state-driven design for agentic applications.

Acknowledgements

I would like to express my sincere gratitude to Prof. Sudebkumar Pal for his invaluable guidance, insightful feedback, and constant encouragement throughout the course of this project. His mentorship has been instrumental in shaping both the technical and conceptual aspects of this work. I am also deeply appreciative of my peers and mentors for their thoughtful discussions and suggestions, which greatly enriched the development process. Finally, I would like to thank the Department of Computer Science and Engineering, IIT Kharagpur, for providing an environment that fosters learning and innovation, and the Department of Chemical Engineering, IIT Kharagpur, for offering the opportunity to undertake this Bachelor's Thesis Project.

Contents

Chapter 1

Introduction

The landscape of modern **Business-to-Business (B2B) sales and marketing** is characterized by an unprecedented volume of data and a relentless demand for efficiency. The digital transformation has equipped organizations with powerful tools for lead generation, but the process of converting these raw leads into qualified prospects remains a significant operational challenge. This initial phase, often termed the *"top-of-funnel"*, involves a series of repetitive, time-consuming, and manually intensive tasks that are critical for building a robust sales pipeline. The recent proliferation of powerful **Large Language Models (LLMs)** presents a transformative opportunity to move beyond simple automation and into the realm of intelligent, autonomous systems.

This thesis documents the design, implementation, and architectural evolution of a **multi-agent system** engineered to automate the B2B sales development lifecycle. By delegating specialized tasks to distinct AI agents, this research explores the creation of a cohesive and intelligent workflow that can autonomously qualify leads, craft personalized outreach, manage communications, and interpret responses.

This introductory chapter establishes the context of the problem, the motivation behind the research, the specific objectives pursued, and the defined scope of the project.

1.1 Background

In traditional **B2B sales organizations**, the role of a **Sales Development Representative (SDR)** is fundamental. An SDR is responsible for the initial stages of the sales process, which includes *lead prospecting*, *qualification*, and *initial outreach*. The primary activities involve:

- **Lead Scoring and Qualification:** Manually reviewing lists of potential leads against an **Ideal Customer Profile (ICP)** to determine their viability.
- **Personalization Research:** Investigating a qualified lead's company and professional background to gather insights for personalized communication.
- **Outreach Execution:** Drafting and sending personalized emails, a task that must be scaled across hundreds of prospects.
- **Response Management:** Monitoring the inbox for replies and manually categorizing them to determine the appropriate next step.

While essential, this process is fraught with inherent inefficiencies. It is fundamentally limited by human capacity, leading to bottlenecks in scalability. The quality of outreach can be inconsistent, and the manual nature of the work is susceptible to error and results in significant operational costs. Consequently, valuable human resources are often expended on repetitive tasks rather than on higher-value activities such as building relationships and closing deals.

1.2 Motivation

The motivation for this research is rooted in the convergence of two key factors: the persistent inefficiencies in **manual sales development** and the recent, disruptive advancements in **generative artificial intelligence**. The emergence of highly capable **Large Language Models (LLMs)**, accessible through APIs, has created a new paradigm for automation. It is now possible to develop systems that can not only execute predefined rules but also perform tasks that require *reasoning*, *context comprehension*, and *human-like text generation*.

This project is driven by the ambition to harness this technology to build an **autonomous workforce of AI agents**. The core idea is to create a system that can replicate and enhance the functions of a human sales development team, operating at a scale and speed that is unattainable manually. The goal is to design a system that can:

- **Increase Throughput:** Process and engage with a higher volume of leads in a shorter amount of time.
- **Enhance Consistency:** Ensure that every lead is evaluated and contacted according to a precise and consistent strategy.
- **Operate Asynchronously:** Manage communications and follow-ups over extended timeframes, mirroring real-world human interactions.
- **Liberate Human Capital:** Allow human sales professionals to focus on strategic, high-level engagement with warm, qualified prospects rather than on manual, top-of-funnel tasks.

1.3 Research Objectives

To address the challenges outlined in the problem statement and to achieve the motivation of creating an autonomous, intelligent sales development system, this thesis is guided by the following four primary research objectives:

1. **Develop a Fully Automated Multi-Agent Architecture for Sales Outreach:** The first objective is to design and implement a cohesive, modular framework composed of specialized AI agents—including the *Prospector*, *Strategist*, *Communicator*, *Interpreter*, *Scheduler*, *RecordKeeper*, and *Analyst*. Each agent operates autonomously yet cooperatively, enabling the system to execute the entire sales workflow—from identifying and scoring potential leads to sending personalized outreach messages and booking meetings—without human intervention.

2. **Integrate Advanced Automation with Real-World Platforms:** This objective focuses on end-to-end process automation through seamless integration with external APIs and data systems. The system leverages the Gmail API for managing two-way email communication, the Google Calendar API for automatic meeting scheduling, and a persistent SQLite database to track each lead's state and activity. These integrations transform the prototype into a fully operational automation tool capable of real-time decision-making and state management.

3. **Implement an Intelligent Feedback and Follow-Up Mechanism:** A critical objective of this research is to design a self-optimizing feedback loop that continuously enhances system performance. The *Analyst* agent evaluates outreach effectiveness by analyzing metrics such as open rates, reply rates, and meeting confirmations. Insights from these analyses are dynamically fed

back to the *Strategist* and *Communicator*, enabling them to refine lead prioritization, email phrasing, and follow-up timing. Additionally, the system autonomously manages follow-up sequences—detecting unresponsive leads and re-engaging them with contextually adaptive messages to maximize conversion potential.

4. **Transition to an Asynchronous, Event-Driven Architecture:** The final objective is to evolve the synchronous prototype into an asynchronous, event-driven ecosystem capable of handling ongoing communications over time. This includes implementing background worker processes that monitor incoming emails and meeting confirmations, ensuring the system reacts intelligently to new events (e.g., a lead response) without manual triggers. Such a design enables the multi-agent system to function continuously, maintaining temporal awareness and operational scalability.

1.4 Scope of Thesis

The scope of this research is focused on creating a functional, proof-of-concept prototype of the **autonomous agentic sales system**. The following aspects are considered **within the scope** of this thesis:

- The design and implementation of the six core agents as defined.
- The entire workflow from ingesting a list of new leads from a CSV file to a final state (e.g., `archived`, `scheduling_in_progress`).
- The use of the **Gmail API** for both outbound and inbound email processing.
- Real-time calendar integration for fetching meeting availability.

- The use of a local **SQLite database** for state management.
- The architectural transition from a single-run, synchronous graph to a decoupled, asynchronous system managed by a scheduled worker.

The following aspects are considered **outside the scope** of this thesis:

- The development of a graphical user interface (**GUI**) for system management.
- Direct integration with commercial CRM platforms like **Salesforce** or **HubSpot**.
- Advanced, multi-turn conversational capabilities beyond the initial outreach and scheduling sequence.
- A formal quantitative performance comparison against a human sales development team.

Chapter 2

Literature Review

The development of the asynchronous, state-driven agentic framework presented in this thesis is grounded in several intersecting fields of research and technology, including multi-agent systems (MAS), the application of Large Language Models (LLMs) to business automation, and software architecture for distributed systems. This chapter reviews the relevant literature and technological precedents, establishing the context in which this project was conceived and highlighting the foundational works and frameworks that enabled its implementation.

2.1 Review of Related Work

The theoretical underpinning of this project is the concept of a multi-agent system, a field that has seen renewed interest with the advent of LLMs. A recent survey by Li et al. (2024) explores the common workflows, components, and challenges in LLM-based multi-agent systems, providing a taxonomy for different agent roles and collaboration mechanisms. The architecture of the system presented in this thesis aligns with this research, employing a cooperative model where specialized agents—such as the Prospector, Strategist, and Interpreter—perform distinct subtasks to achieve a collective goal.

The application of such systems to sales and marketing is a burgeoning area of research. Skoriukov (2023) discusses the use of ChatGPT-powered outreach, laying the groundwork for automating aspects of sales communication. Further, works by Szczepanik and Chudziak (2025) and Flores et al. (2025) explore the construction of more complex, reliable multi-agent systems for marketing campaigns, emphasizing the need for structured planning and memory. This thesis builds upon these concepts by implementing a concrete, end-to-end workflow specifically for the B2B lead qualification funnel.

Specific agent functions have also been the subject of prior work. The state of lead scoring models, for example, is detailed by Wu et al. (2023) and González-Flores et al. (2025), primarily focusing on classical machine learning techniques. The Prospector agent in this project represents a paradigm shift, leveraging the in-context learning and reasoning capabilities of an LLM to perform this scoring task without the need for explicit model training.

Finally, the architectural challenges of agentic systems are an active area of research. The challenge of asynchronicity in agent tool usage is explored by Ginart et al. (2024), highlighting the need for systems that can manage tasks with unpredictable durations. The *ScheduleMe* project by Wijerathne et al. (2025) provides a specific example of a multi-agent calendar assistant, a domain directly relevant to this thesis's Scheduler agent. This body of work validates the core architectural problem that this thesis aims to solve: creating a framework that can function effectively over the long and unpredictable timelines of human-in-the-loop interactions.

2.2 Existing Systems, Frameworks, and Approaches

The practical implementation of the agentic framework was made possible by a specific set of modern software tools and platforms.

Large Language Models and API Providers: The cognitive engine for all intelligent agents is a Large Language Model from the Llama 3 family, accessed via the Groq API. The choice of Groq was a deliberate performance consideration, as its high inference speed is critical for reducing latency in workflows that may require multiple sequential LLM calls.

Agentic Frameworks (LangChain and LangGraph): The system is built upon the LangChain ecosystem LangChain Developers (2025). LangChain provides essential abstractions for prompt management and LLM interaction. More critically, LangGraph is used to orchestrate the agentic workflows, a methodology documented by Kim (2025). LangGraph's stateful graph paradigm allows for the explicit definition of agents as nodes and the flow of a persistent `AgenticState` between them. Its support for conditional edges was essential for implementing the dynamic routing logic after the Interpreter agent's analysis.

Asynchronous Processing: The system's solution to the challenge of time-delayed email replies is to adopt a decoupled architecture. This is implemented using APScheduler, a Python library for scheduling background tasks. This approach, while lightweight, mirrors the design of large-scale enterprise systems. It enables the creation of a persistent, long-running background worker that handles all asynchronous events, a key design choice validated by the research on asynchronous agent challenges Ginart et al. (2024).

External Service APIs: To function in the real world, the system integrates with external platforms via their APIs. The Gmail API Google Developers (2025a) is used for both sending and reading emails, requiring the implementation of the complex but secure OAuth 2.0 authentication flow. The architecture is designed to extend this to the Google Calendar API Google Developers (2025b) to fully automate meeting scheduling. This reliance on external APIs is a standard pattern for modern agentic systems that need to interact with the digital world.

2.3 Summary of Key Findings from the Survey

The review of the existing literature and technologies yields several key findings that directly influenced the design and architecture of this project:

A State-Driven Architecture is Essential for Asynchronicity: The literature confirms that managing interactions with unpredictable time delays (like email) is a primary challenge for agentic systems. The key finding is that a simple, synchronous chain of agents is insufficient. The most robust solution, as implemented in this thesis, is a decoupled, state-driven architecture where agents communicate indirectly through state changes in a central, persistent database.

Structured LLM Outputs are Crucial for Reliability: LLMs are inherently non-deterministic. To build a reliable application, their output must be constrained. The use of frameworks like LangChain in conjunction with schema validation libraries such as Pydantic to enforce structured JSON output is a critical best practice that ensures data integrity and prevents fragile, error-prone string parsing between agents.

A Hybrid Workflow Model is Required for Practical Applications: An effective agentic system must be both responsive to immediate triggers and capable of long-term, independent operation. This survey validates the project's hybrid architectural choice: a real-time API (FastAPI) to handle on-demand tasks like initiating outreach, combined with a persistent, scheduled background worker to manage the long-running, asynchronous tasks of interpretation and follow-up.

Chapter 3

Research Gaps

While the literature provides a strong foundation for the componets of agentic systems, a review of existing work reveals several critical gaps, particularly in the transition from theoretical models to practical, long-running business applications. The current research landscape, though advancing rapidly, often overlooks the architectural and engineering challenges inherent in building robust, autonomous systems that can operate effectively in real-world, asynchronous environments. This project is specifically designed to address these gaps.

3.1 Asynchronous, Long-Term Interaction Management

A predominant focus in current agentic research is on **synchronous task execution**, where an agent completes a defined objective in a single, uninterrupted session. This paradigm is fundamentally ill-suited for processes that depend on interaction with external human actors, where response times are unpredictable. The B2B sales communication cycle is a prime example of this temporal disconnect.

The existing literature largely fails to provide a comprehensive architectural blueprint for systems that can manage numerous, parallel, long-running, and intermittent interactions. This creates a critical research gap concerning the design of agents that can:

- **Persist State Durably:** Go beyond simple in-memory state management to maintain context and status across system restarts and over extended periods.
- **Decouple Action from Reaction:** Architecturally separate the workflow that initiates an action (e.g., sending an email) from the workflow that processes its eventual, unpredictable response. This is a central theme of this thesis, addressed by the hybrid API and background worker architecture.
- **Temporal Awareness and Proactive Follow-Ups:** Beyond simply reacting to incoming events (like a reply), a practical sales automation system must be proactive. A significant gap exists in the ability of agentic systems to act based on the *absence* of an event over a defined period. In sales, the majority of initial outreach does not receive a response, and sending a follow-up email after a set number of days is a critical part of the workflow. This requires the system to be ”temporally aware”—it must maintain timestamps of all outbound communications and implement logic that triggers a new action (e.g., instructing the `Strategist` to draft a context-aware follow-up message) if no reply is detected within a configured timeframe. This architectural challenge is noted in recent work on real-time agents (Ginart et al. (2024)). This thesis, through its persistent state database and scheduled background worker, establishes the necessary architecture to close this gap.

3.2 Applied, End-to-End Business System Implementation

The discourse surrounding agent-building frameworks often remains at a high level of abstraction, focusing on the theoretical capabilities of components (Li et al. (2024)) rather than the holistic engineering of a complete system. There is a notable lack of comprehensive, publicly documented case studies that detail the practical challenges of integrating these components into an end-to-end, functional system for a specific business vertical like B2B sales.

This gap lies between conceptual possibility and applied reality. The literature frequently omits discussion of crucial software engineering hurdles that are central to building a reliable application, such as:

- **Robust API Integration:** Managing complex authentication protocols like OAuth 2.0 for interfacing with real-world services such as the Gmail and Google Calendar APIs (Google Developers (2025a)).
- **Data Integrity and Validation:** Enforcing reliable, structured data exchange between non-deterministic LLMs and deterministic application code.
- **Scalable Resource Management:** Engineering solutions for handling API rate limits, rotating API keys, and creating a modular architecture that supports future expansion.

This research aims to fill this gap by providing a transparent architectural blueprint and a full-stack proof-of-concept, serving as a practical guide for transitioning from the isolated agent capabilities discussed by researchers to an integrated and operational business automation system.

3.3 Dynamic Strategy and Performance Feedback Loops

Current implementations of agentic systems, including those described by Szczepanik and Chudziak (Szczepanik and Chudziak (2025)) and Skoriukov (Skoriukov (2023)), typically operate with a **static strategy**. The prompts and operational parameters of the agents are fixed at deployment. This approach precludes the possibility of autonomous learning or optimization.

A significant research gap therefore exists in the design of agentic systems that can self-optimize. The field has yet to widely explore architectures that incorporate a "meta-agent" or an analytical layer—like the **Analyst** proposed in this thesis—designed to:

- **Systematically Track Performance Metrics:** Record Key Performance Indicators (KPIs) like reply rates and meeting conversion rates.
- **Establish an Autonomous Feedback Loop:** Utilize insights from these metrics to dynamically modify the operational parameters of other agents, such as rewriting the prompts used by the `Strategist` to improve its persuasive capabilities over time.

This thesis directly lays the groundwork to address this gap. By meticulously tracking performance metrics and designing an `Analyst` agent into the system architecture, it establishes the necessary foundation for a future, truly adaptive autonomous system, moving beyond static execution to intelligent optimization.

Chapter 4

Problem Formulation

The successful development of a complex AI system necessitates a clear and rigorous definition of the problem it aims to solve. This chapter translates the high-level vision of creating an autonomous B2B sales development network into a formal problem statement, a conceptual representation of the system, and a concrete set of functional and non-functional requirements. This formulation provides the structured framework upon which the system's design and implementation are based, focusing specifically on the components developed within the scope of this research.

4.1 Problem Statement

The core problem addressed by this thesis is the inherent inefficiency, scalability limitations, and high operational cost associated with manual B2B lead qualification and outreach processes. Marketing and sales development teams are constrained by repetitive, time-intensive tasks, including lead identification, scoring against an Ideal Customer Profile (ICP), personalized email drafting, response handling, and data logging. This manual process results in inconsistent execution, missed opportunities, and the misallocation of skilled human capital away from strategic sales activities.

Therefore, the central problem is to design, build, and validate a stateful, multi-agent AI system capable of autonomously managing the core top-of-funnel sales workflow. The system must be able to ingest raw lead data, intelligently qualify prospects, orchestrate personalized communication, interpret replies to determine intent, and manage subsequent scheduling and data logging tasks without direct human intervention. The ultimate objective is to create an intelligent digital workforce that operates persistently, consistently, and at a scale unachievable by manual means.

4.2 Formal Representation

To formally represent the system, we define the Multi-Agent AI Network, denoted as N, as a state-driven system that operates on a set of leads. The network developed in this thesis can be described as a tuple:

$$N = \{A, L, S, T, E\}$$

Where:

A is the finite set of specialized agents that have been implemented. This set is defined as:

$$A = \{A_P, A_S, A_C, A_F, A_I, A_{Sch}, A_{RK}, A_{An}\}$$

where A_P is the *Prospector*, A_S is the *Strategist*, A_C is the *Communicator*, A_I is the *Interpreter*, A_F is the *Follow Up*, A_{Sch} is the *Scheduler*, A_{RK} is the *Record Keeper*, and A_{An} is the *Analyst*. The Analyst agent serves as an observer that monitors overall system performance and reports aggregate metrics based on the state data maintained by other agents.

L is the set of leads, where each lead $l \in L$ is an object containing data attributes and a dynamic status.

S is the finite set of possible states a lead l can occupy, which govern its flow through the network. The implemented set of states is:

$$\begin{aligned} S = \{&new, scored, message\_generated, outreach\_sent, \\ &interested, not\_interested, wrong\_person, \\ &scheduling\_in\_progress, meeting\_booked, archived, \\ &failed, follow\_up\_due\} \end{aligned}$$

T is the set of state transition functions, which are triggered by the actions of an agent $a \in A$. A transition function is represented as:

$$T(l, a) \rightarrow l'$$

This indicates that an agent a performs an action on a lead l, resulting in an updated lead l' with a new state $s' \in S$ and modified attributes (e.g., an appended communication history). For example, $T(l_{\{status:outreach\_sent\}}, A_I)$ could result in $l'_{\{status:interested\}}$.

E is the set of external environments with which the agent network interacts. These environments are the sources of data and channels for communication, defined as:

$$E = \{E_{data}, E_{email}, E_{db}\}$$

where E_{data} is the lead data source (CSV file), E_{email} is the email communication service (Gmail API), and E_{db} is the persistence layer (SQLite database). The E_{db} environment serves a dual purpose: it provides state persistence for all leads and also acts as the metrics store accessed by the Analyst agent (A_{An}) for performance tracking and reporting.

It is important to note that while most agents perform actions that trigger state transitions, the Analyst agent (A_{An}) functions as an observer. It does not perform transitions in S or execute transformations in T, but instead reads the persisted system state and aggregates data to produce performance insights.

4.3 Requirements of the Solution

To be considered a successful implementation, the developed system must satisfy the following functional and non-functional requirements, derived directly from the project's scope and architecture.

Functional Requirements (FR)

- **FR1: Lead Ingestion and Scoring.** The system must ingest lead data from a CSV file. The Prospector agent (A_P) must score these leads against a predefined ICP and transition their status to `scored`.
- **FR2: Personalized Outreach Strategy.** The Strategist agent (A_S) must filter for qualified leads (status `scored`), generate a personalized email message body using an LLM, and update their status to `message_generated`.

- **FR3: Email Communication.** The Communicator agent (A_C) must use the generated message to send an email to the lead via the Gmail API and update their status to `outreach_sent`.
- **FR4: Asynchronous Response Interpretation.** The Interpreter agent (A_I) must periodically read the inbox via the Gmail API to find replies from leads. It must use an LLM to classify the lead's intent and update the status to `interested`, `not_interested`, or `wrong_person`.
- **FR5: Meeting Scheduling.** The Scheduler agent (A_{Sch}) must be triggered by an `interested` status. It must then generate and send a follow-up email offering meeting times and update the lead's status to `scheduling_in_progress`.
- **FR6: Data Persistence and Archival.** The Record Keeper agent (A_{RK}) must durably log lead activities. It will process leads with terminal statuses (`not_interested`, `wrong_person`) by updating their status to `archived` in the central SQLite database.
- **FR7: Performance Metric Tracking.** The Analyst agent (A_{An}) must maintain and update system-level performance metrics in the centralized database. It must track key indicators such as the total number of leads processed, qualified, and categorized by their final statuses (e.g., `interested` vs. `not_interested`). These metrics form the basis for evaluating outreach effectiveness and future optimization.

Non-Functional Requirements (NFR)

- **NFR1: Modularity.** The system must be designed with a clear separation of concerns, where each agent is an independent Python module responsible for a specific stage of the workflow.

- **NFR2: Asynchronicity.** The system architecture must handle the inherent time delays in email correspondence. This is achieved through a scheduled background worker that triggers state-driven agents (Interpreter, Scheduler, RecordKeeper) independently of the initial outreach workflow.

- **NFR3: Persistence.** The complete state of every lead, including its current status and communication history, must be maintained in a durable SQLite database. This ensures data integrity across system restarts and enables the asynchronous workflow.

- **NFR4: Reliability.** The system must manage external service interactions gracefully, including handling the OAuth 2.0 authentication flow for the Gmail API and managing potential API call failures.

- **NFR5: Extensibility.** The architecture must be designed to support the future addition of new agents or communication channels with minimal disruption to the existing logic. The state-driven model and modular agent design directly support this requirement.

- **NFR6: Measurability.** The system must be designed for performance observability. Every significant action (e.g., scoring a lead, sending an email, classifying an intent) should update a central performance metrics store, enabling continuous monitoring and analytics by the Analyst agent (A_{An}).

Chapter 5

Design of the Solution

This chapter provides a detailed exposition of the technical design and architecture of the Multi-Agent AI Network. It begins with a high-level overview of the system's architecture, followed by a granular breakdown of each agent's design and responsibilities. The chapter also describes the system's workflow through state transition diagrams and concludes with a discussion of the key design choices and the rationale behind them.

5.1 Overall System Architecture

The system is designed as a hybrid, asynchronous, state-driven architecture. This model was deliberately chosen to handle the temporal complexities of real-world email communication, where significant delays between actions and responses are the norm. The architecture consists of four primary components:

- **Real-Time API Server:** A web server built using FastAPI. Its primary responsibility is to handle immediate, on-demand tasks. It exposes an API endpoint (`/run_workflow`) that initiates the initial outreach process for a new batch of leads, running the Prospector, Strategist, and Communicator agents sequentially. It also provides endpoints for triggering individual agents for testing and debugging.

- **Asynchronous Background Worker:** A separate, long-running Python process managed by APScheduler. This is the heart of the system's asynchronous capabilities. It operates on a fixed schedule (e.g., every few minutes) and is responsible for running the agents that depend on external, time-delayed events—namely, the Interpreter, Scheduler, and Record Keeper. It queries the central database to find leads in specific states and triggers the appropriate agent to process them.

- **Centralized State Database:** A SQLite database that serves as the single source of truth for the entire system. It stores the state of every lead, including all raw data, scores, statuses, and complete communication histories. This database acts as the communication layer between the Real-Time API and the Background Worker, effectively decoupling them. When the Communicator finishes its run, it updates the lead's status in the database, and the Background Worker later picks up that change.

- **External Service Integrations:** A set of modules that handle interaction with external APIs. These include the `email_client` for managing authentication and communication with the Gmail API (for sending and receiving emails), the `key_manager` for interacting with the Groq API for LLM inference, and the Google Calendar API for future meeting scheduling automation.

This hybrid architecture allows the system to be both responsive (via the API) and persistent (via the background worker), creating a robust framework for long-running autonomous operations.

5.2 Modules and Agent Design

The system's logic is encapsulated within distinct, modular agents, each designed with a single responsibility. This adheres to the Single Responsibility Principle, making the system easier to develop, test, and maintain.

Prospector (A_P)

Input: A list of new leads (status *new*).
Process: Process: This is a deterministic agent that operates without an LLM. It employs a series of Python functions *('tools')* to programmatically evaluate each lead against criteria defined in `icp.yaml`. An aggregator function then calculates a definitive score from the tool outputs, ensuring high accuracy, speed, and zero API cost.
Output: The lead's score, qualification status, and reasoning are updated. The status changes to *scored.*

Strategist (A_S)

Input: Qualified leads (status *scored*).
Process: Uses an LLM to generate a highly personalized outreach email. The prompt includes the lead's data and the ICP context. Output is validated through a `PersonalizedMessage` Pydantic model. The generated message is stored in the database.
Output: The `personalized_message` field is populated, and the status changes to *message_generated.*

Communicator (A_C)

Input: Leads with a generated message (status *message_generated*).

Process: Retrieves the lead's email and message body. Uses the `email_client` module to send the email through the Gmail API.

Output: Communication history is updated, and status changes to *outreach_sent*.

Interpreter (A_I)

Input: Leads awaiting a reply (status *outreach_sent* or *scheduling_in_progress*). Run by the background worker.

Process: Queries the Gmail API for unread messages from lead addresses. If found, uses an LLM with a `LeadIntent` model to classify reply intent and extract a summary.

Output: Logs the reply and analysis to communication history. Updates the lead's status to *interested*, *not_interested*, or *wrong_person*. Marks the email as read.

Follow-Up Agent (A_F)

Input: Leads that have not responded to outreach within a predefined interval (e.g., two days), identified by a status flag of `follow_up_due`. This agent is invoked by the background worker.

Process: The Follow-Up Agent is responsible for proactive re-engagement of inactive leads. It retrieves the original outbound message from the lead's communication history and leverages a large language model (LLM) to generate a concise, contextually relevant follow-up email. The message is then sent as a reply within the same email thread using the `google_api_client`, thereby preserving a natural conversational flow and maintaining thread continuity.

Output: The generated follow-up email is appended to the communication history, and the lead's status is updated to `outreach_sent` with a refreshed timestamp. This update reintroduces the lead into the waiting queue for subsequent processing by the Interpreter Agent.

Scheduler (A_{Sch})

Input: Leads with *interested* status. Run by the background worker.
Process: Uses an LLM (`SchedulingEmail` model) to generate a follow-up email with predefined meeting slots. Sends the email via the `email_client`. The architecture supports extending this to integrate directly with the Google Calendar API to confirm times and create events.
Output: The scheduling email is logged, and the status updates to *scheduling_in_progress*.

Record Keeper (A_{RK})

Input: Leads reaching terminal, non-positive states (*not_interested*, *wrong_person*). Run by the background worker.
Process: Performs final logging and archival updates in the database.
Output: Lead status changes to *archived*.

Analyst (A_{An})

Input: Centralized database containing all lead states and communication logs.
Process: Reads and aggregates system-wide performance metrics—such as leads processed, qualified, outreach success rate, and meeting conversion ratios. The Analyst provides foundational observability for future reinforcement or optimization

loops.

Output: Generates reports and updates a performance metrics table in the database.

5.3 Workflow Diagrams

5.3.1 High-Level System Architecture

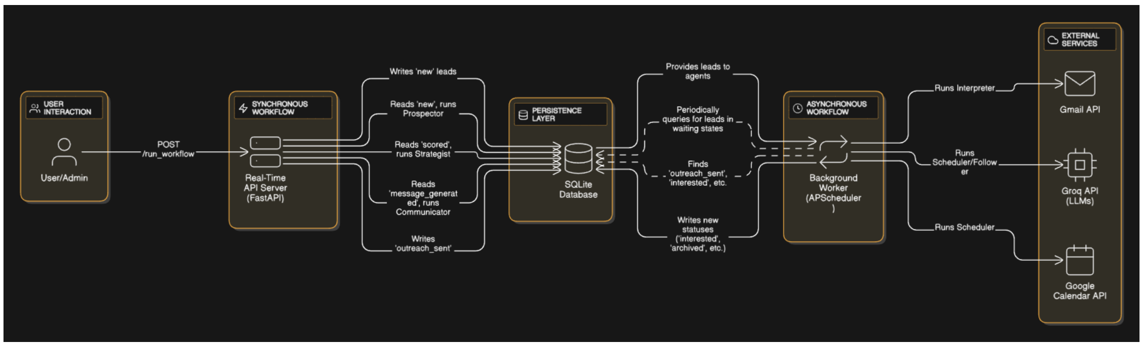
USER INTERACTION
User/Admin
POST /run_workflow
SYNCHRONOUS WORKFLOW
Real-Time API Server (FastAPI)
Writes 'new' leads
Reads 'new', runs Prospector
Reads 'scored', runs Strategist
Reads 'message_generated', runs Communicator
Writes 'outreach_sent'
PERSISTENCE LAYER
SQLite Database
Provides leads to agents
Periodically queries for leads in waiting states
Finds 'outreach_sent', 'interested', etc.
Writes new statuses ('interested', 'archived', etc.)
ASYNCHRONOUS WORKFLOW
Background Worker (APScheduler)
Runs Interpreter
Runs Scheduler/Follower
Runs Scheduler
EXTERNAL SERVICES
Gmail API
Groq API (LLMs)
Google Calendar API

FIGURE 5.1: High-Level Hybrid System Architecture. The architecture integrates synchronous (FastAPI) and asynchronous (APScheduler) workflows with a shared SQLite database and external APIs.

Figure 5.1 presents the high-level architecture of the proposed hybrid system, which combines synchronous and asynchronous workflows to achieve both responsiveness and automation. The **FastAPI server** manages real-time user interactions and request processing, while the **background worker**, implemented using **APScheduler**, executes scheduled and long-running tasks.

Both components interact exclusively through the **SQLite database**, which serves as a central persistence and coordination layer. This separation of concerns ensures modularity, scalability, and fault tolerance, as the server and worker remain decoupled yet coordinated through database-mediated communication. Integration with **external APIs**—including Gmail, Groq (LLMs), and Google Calendar—extends the system's capabilities for automated communication, reasoning, and scheduling.

- **FastAPI Server:** Handles incoming requests.
- **Background Worker (APScheduler):** Executes scheduled agent runs.
- **SQLite Database:** Acts as the central state and communication hub.
- **External APIs:** Includes Gmail API, Groq API, and Google Calendar API.

Arrows illustrate data flow: both the server and worker read/write to the database but do not communicate directly—demonstrating a decoupled architecture.

5.3.2 Synchronous System Architecture

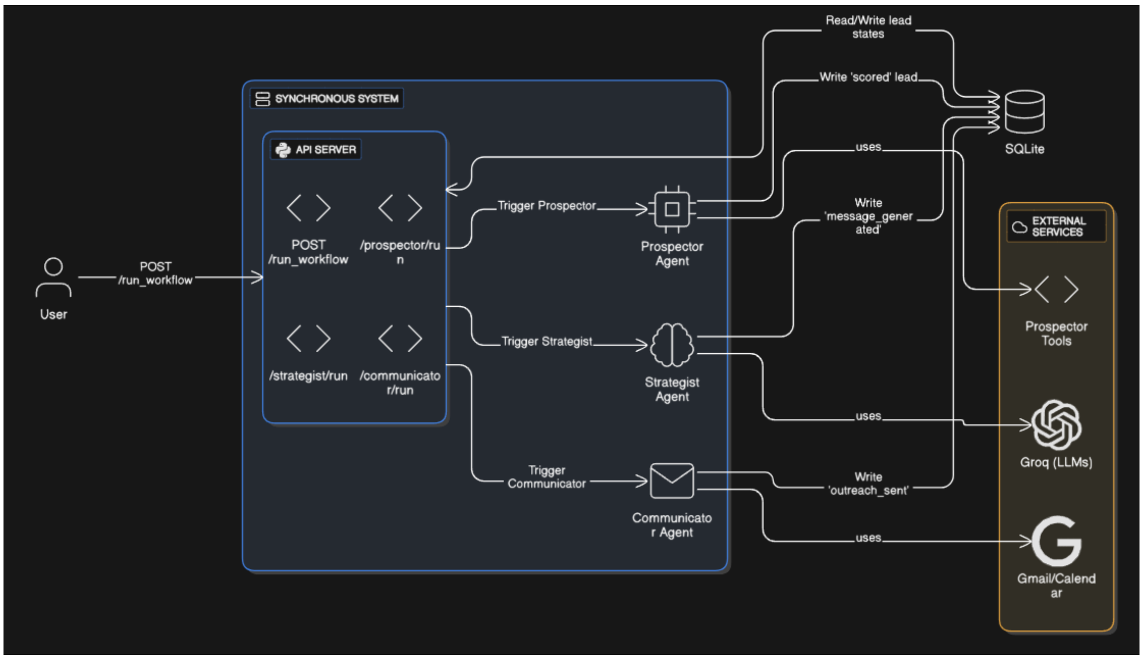

FIGURE 5.2: Synchronous System Architecture. The FastAPI server coordinates user-triggered workflows through dedicated agents that interact with the database and external APIs.

Figure 5.2 illustrates the architecture of the synchronous system, which handles real-time user interactions and workflow execution. When a user initiates a request through the `/run_workflow` endpoint, the **FastAPI server** orchestrates a sequence of agent-based tasks. These agents—namely the **Prospector**, **Strategist**, and **Communicator**—are responsible for distinct stages of the workflow, from data acquisition and strategy formulation to message generation and outreach execution.

The system operates in a tightly coordinated yet modular manner. The API server triggers each agent in succession, passing intermediate results through a shared **SQLite database**. The database maintains the state of each lead and ensures persistence between agent executions. Additionally, the agents integrate with various **external services**: the Prospector interfaces with domain-specific prospecting tools, the Strategist leverages **Groq LLMs** for reasoning and content generation, and the Communicator utilizes **Gmail and Google Calendar APIs** for automated outreach and scheduling.

This synchronous design enables immediate feedback and deterministic execution of workflows while maintaining extensibility and separation of responsibilities among system components.

5.3.3 Asynchronous Bachground Workers

Figure 5.3 illustrates the asynchronous architecture of the system, which operates alongside the synchronous FastAPI server to handle background processes and automation tasks. The **background worker**, implemented using **APScheduler**, executes recurring and event-driven jobs without direct user intervention. Its primary components include the **RecordKeeper**, **Scheduler**, **Interpreter**, and **Follow-up Agent**, each responsible for managing specific workflow aspects such as data persistence, scheduling, message interpretation, and automated outreach.

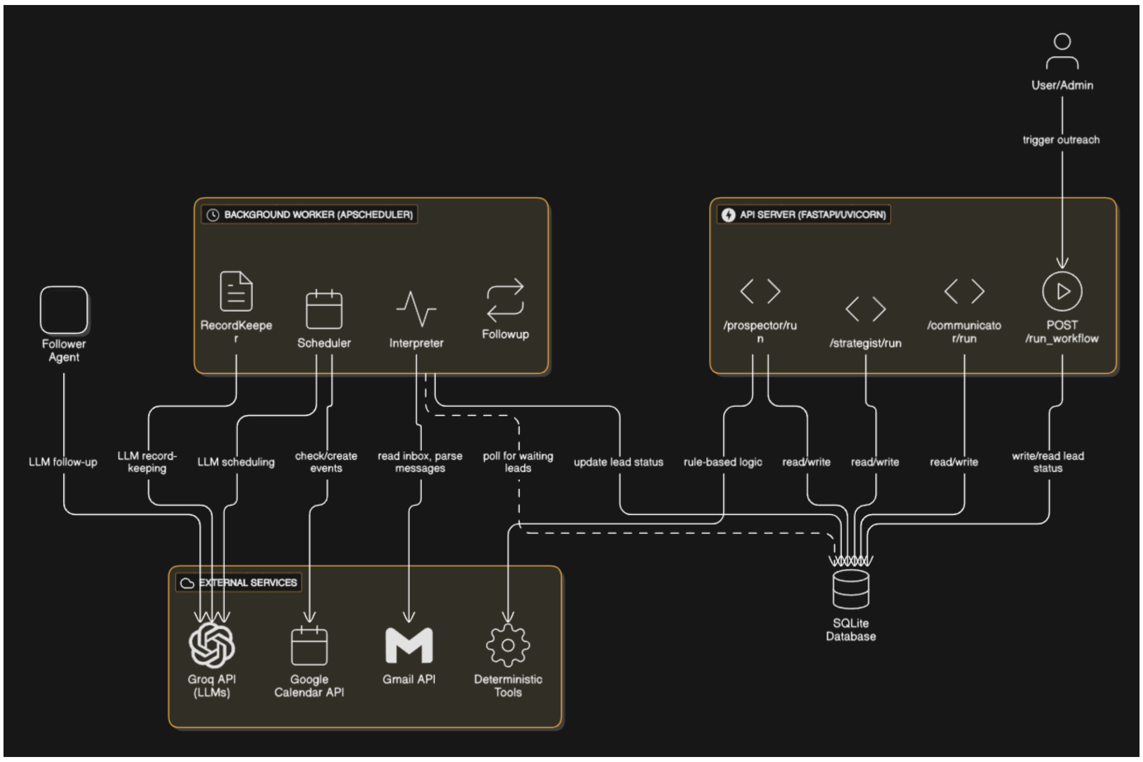

FIGURE 5.3: Asynchronous System Architecture. The APScheduler-based background worker automates monitoring, scheduling, and follow-up tasks using external APIs and LLMs.

The asynchronous system interacts continuously with the shared **SQLite database**, reading and updating lead states to maintain workflow consistency. It also interfaces with multiple **external services**: the **Groq API (LLMs)** supports language-based reasoning and content generation; the **Google Calendar API** manages event scheduling; and the **Gmail API** handles message parsing and communication. Deterministic tools assist in maintaining structured execution logic across tasks.

This design ensures that time-dependent or long-running processes—such as follow-up scheduling, inbox monitoring, and lead status updates—are executed reliably in the background. By decoupling asynchronous execution from real-time user requests, the system achieves scalability, resilience, and efficient resource utilization while maintaining coherent state synchronization with the main application.

5.4 Design Choices and Rationale

The design of the system was guided by several key principles and technical decisions to create a robust, modular, and extensible solution.

- **Choice: Multi-Agent Architecture. Rationale:** Rather than a single monolithic script, the system is decomposed into specialized agents. This separation of concerns enhances maintainability and modular development. For instance, improving Prospector's logic has no effect on Interpreter's behavior.
- **Choice: Rule-Based Implementation of Prospector. Rationale:** A critical design decision was to not use LLMs for every agent. The Prospector agent, whose task is to score leads against a predefined, mathematical ICP, was deliberately implemented as a deterministic, rule-based system. This is because the task has no ambiguity; it requires precision, speed, and reliability—qualities where pure code excels and LLMs can be inefficient, costly, and prone to error.
- **Choice: State-Driven Asynchronous Model. Rationale:** Email response times are unpredictable, making synchronous processing infeasible. A central database holding persistent lead states allows asynchronous agents to act independently. This architecture enables persistence and fault tolerance.
- **Choice: LangGraph for Workflow Orchestration. Rationale:** LangGraph provides an explicit workflow definition (Prospector $\rightarrow$ Strategist $\rightarrow$ Communicator) with internal state management. This enhances interpretability and simplifies passing structured data across agents.
- **Choice: Pydantic for Structured LLM Outputs. Rationale:** LLMs can produce inconsistent results. Pydantic enforces schema validation, ensuring

the AI outputs conform to expected structures, reducing runtime failures and eliminating fragile string parsing.

- **Choice: Centralized SQLite Database. Rationale:** SQLite provides a lightweight, durable, and self-contained persistence layer. It supports long-running asynchronous workflows and simplifies system deployment without requiring external DB services.

Chapter 6

Technologies Used

The successful implementation of the Multi-Agent AI Network was made possible by leveraging a modern stack of open-source libraries, frameworks, and external services. This chapter provides a detailed overview of the key technologies employed, outlining the specific role each component played in the system's architecture and functionality.

6.1 Core Programming and Development Environment

Python: The entire system was developed using Python (version 3.9+). It was chosen for its extensive ecosystem of libraries for web development, data science, and AI, which is unparalleled for a project of this nature. Its clear syntax and strong community support make it an ideal language for rapid prototyping and building complex applications.

Pydantic: This library was a cornerstone of the project for data validation and ensuring reliable interaction with the Large Language Models. All data structures, including the central `AgenticState` and the expected outputs from LLM-powered agents (`LeadIntent`, `PersonalizedMessage`, `SchedulingEmail`, etc.), were defined as Pydantic models. This provided compile-time type hinting and, more importantly, runtime data validation, which was critical for parsing and enforcing the structure of LLM responses.

Dotenv: The `python-dotenv` library was used to manage environment variables. All sensitive information, such as API keys and sender email addresses, was stored in a `.env` file and loaded into the application's environment at runtime. This is a standard security practice that decouples configuration from the source code.

6.2 AI and Agentic Frameworks

Groq API: The core intelligence of the agents was powered by Large Language Models (LLMs), accessed via the Groq API. Groq was selected for its exceptional inference speed (tokens per second), which is crucial for reducing latency in agentic workflows where multiple sequential LLM calls are often required. Models from the Llama 3 family (e.g., `llama3-70b-8192`, `llama3-8b-8192`) were used for tasks ranging from lead scoring to creative text generation and intent classification.

LangChain: This open-source framework was used to streamline the interaction with the Groq API. Specifically, its `ChatGroq` integration simplified the process of making API calls. The most critical feature used was LangChain's structured output capability, which allows developers to bind a Pydantic model directly to an LLM. This feature was instrumental in forcing the LLM to return valid, predictable JSON, forming the backbone of reliable agent-to-agent communication.

LangGraph: LangGraph, a library built on top of LangChain, provided the foundational framework for orchestrating the agentic workflows. It was used to define the system as a stateful graph where each agent is a node. Key features utilized include:

- **Stateful Graphs (StateGraph):** To manage and pass the central `AgenticState` between nodes automatically.
- **Conditional Edges:** To implement the branching logic after the Interpreter agent, allowing the workflow to dynamically route leads based on their classified intent. This was essential for moving beyond a simple linear sequence of operations.

6.3 Web Server and Asynchronous Processing

FastAPI: The real-time component of the system, including the API for initiating the outreach workflow and endpoints for testing individual agents, was built using FastAPI. It was chosen for its high performance, automatic generation of interactive API documentation (via Swagger UI), and its modern design based on standard Python type hints.

Uvicorn: Uvicorn served as the ASGI (Asynchronous Server Gateway Interface) server for running the FastAPI application. It is a lightweight and extremely fast server, making it a standard choice for production-ready FastAPI deployments.

APScheduler: The APScheduler (Advanced Python Scheduler) library was used to implement the asynchronous background worker. It provided a simple and robust way to schedule the `process_pending_leads` function to run at regular intervals (e.g., every two minutes). This library was the key to decoupling the system and enabling it to handle the time delays inherent in email communication.

6.4 External Service Integrations and Data Storage

Google Workspace APIs (Gmail & Calendar): The system's ability to interact with the real world is primarily powered by the Google Workspace APIs. This required a suite of Google client libraries for Python:

- **Gmail API:** Used for both sending outbound emails (by the Communicator and Scheduler) and reading inbound replies (by the Interpreter).
- **Google Calendar API:** Designated as the service for the Scheduler agent to fulfill its ultimate purpose: booking meetings. The architecture is designed for the Scheduler to use this API to create calendar events once a time is confirmed.
- **Google Authentication Libraries (`google-auth-oauthlib`, etc.):** These libraries were essential for managing the complex but secure OAuth 2.0 authentication flow required by both APIs. They handled the one-time user consent process and the creation and refreshing of security tokens, enabling the application to make authorized API calls.

SQLite: The central state database was implemented using Python's built-in `sqlite3` module. SQLite was chosen for its serverless, self-contained nature, providing robust persistence for a prototype system without the overhead of a separate database server.

html2text: This utility library was used to parse incoming HTML emails from the Gmail API. It converts HTML into clean, Markdown-formatted text, providing a more reliable and concise input for the Interpreter agent's LLM.

Chapter 7

Engineering and Implementation Details

This chapter provides a detailed account of the implementation process of the Multi-Agent AI Network. It translates the architectural design outlined in Chapter 5 into a description of the tangible software artifacts, data handling mechanisms, and core algorithms that power the system. The chapter is structured to provide a module-wise breakdown of each agent, offering insight into its specific responsibilities and operational logic.

7.1 Implementation and Workflow Architecture

The development of the system followed a structured, phased approach, culminating in a hybrid architecture that separates synchronous and asynchronous operations.

Phase 1: Synchronous Outreach Workflow

The initial implementation focused on creating a linear, on-demand workflow for initial lead outreach. This process is synchronous, meaning it is executed as a single, uninterrupted sequence when an API endpoint is called.

- **Trigger:** A `POST` request to the `/run_workflow` FastAPI endpoint.
- **Execution:** LangGraph is used to orchestrate a sequence of three agents:
    1. Prospector: Ingests and scores all new leads.
    2. Strategist: Generates personalized messages for qualified leads.
    3. Communicator: Sends the generated emails.
- **Termination:** Once the Communicator finishes, the lead's status is updated to `outreach_sent` in the database, and this synchronous workflow concludes. This entire process happens within the context of a single API request.

Phase 2: Asynchronous Response Handling Workflow

To address the inherent time delays of email communication, a separate, asynchronous workflow was designed. This workflow operates independently of the API server.

- **Trigger:** A scheduled job, managed by APScheduler in the `run_background_workers.py` script, which runs at a predefined interval (e.g., every two minutes).
- **Execution:** The background worker acts as an orchestrator, querying the database for leads in various "waiting" states and activating the appropriate agent. The sequence within the worker is prioritized:

1. Temporal Check (Follow-Up Trigger): First, it queries for leads in the `outreach_sent` state and checks their `last_outreach_timestamp`. If more than a predefined duration (e.g., two days) has passed without a reply, it transitions their status to `follow_up_due`. This proactive step makes the system temporally aware.
2. Follow Up: Runs next to process any leads marked as `follow_up_due`, drafting and sending a follow-up message.
3. Scheduler: Runs first to act on leads already marked as interested.
4. Interpreter: Runs next to check for replies to either initial outreach (`outreach_sent`) or scheduling emails (`scheduling_in_progress`).
5. RecordKeeper: Runs last to clean up and archive leads that have reached a terminal state (`not_interested`, `wrong_person`).

- **State-Driven Logic:** Unlike the synchronous workflow, this process is not a fixed sequence. An agent only runs if the database contains leads in the state it is designed to handle. This makes the system resilient and event-driven, reacting to changes in the central database over time.

7.2 Data Handling and the Global State

Data is the central element that connects the synchronous and asynchronous parts of the system. Its handling is designed for consistency, persistence, and security.

The AgenticState and Lead Models

The "global state" of the application is formally defined by Pydantic models in `app/models/state.py`, which ensure type safety and structured data throughout the system.

- **Lead Model:** This is the atomic unit of data, representing a single customer prospect. It contains all information pertinent to that lead, including:
  - `lead_id, raw_data:` Unique identification and the original ingested data.
  - `status:` The most critical field, which dictates which agent should process the lead next. It acts as the primary key for the system's state machine.
  - `qualified_lead, score:` Data artifacts generated by the Prospector.
  - `personalized_message:` The output of the Strategist.
  - `communication_history:` A running log of all interactions, both outbound and inbound.
  - `intent:` The result of the Interpreter's analysis.
- **AgenticState Model:** This is a container model that represents the entire state being worked on during a single agent's execution run. It holds:
  - `lead:` A list of Lead objects that are being processed.
  - `performance_metrics:` A dictionary for tracking system-wide KPIs.

State Management: In-Memory vs. Persistent

The system utilizes a dual-state management strategy:

- **In-Memory State:** During the execution of a workflow (e.g., the background worker's run), the relevant Lead objects are loaded from the database into an `AgenticState` object. This in-memory object is passed between agents, allowing for efficient modification.
- **Persistent State:** The SQLite database is the ultimate source of truth. After an agent finishes its work, any changes made to the leads in the in-memory `AgenticState` are written back to the database using the `update_lead_in_db` function. This ensures durability and allows the asynchronous parts of the system to communicate through the database.

7.3 Module-wise Description

Each agent is an independent module designed with a single responsibility.

Prospector (prospector.py)

Description: A deterministic agent responsible for high-speed, accurate lead qualification against the Ideal Customer Profile (ICP).

Implementation: This agent has been explicitly engineered without an LLM to ensure reliability and efficiency for its rule-based task. It uses a set of pure Python functions defined in `app/tools/prospector_tools.py`, which load the `icp.yaml` configuration and execute deterministic checks for each criterion (industry, location, etc.). A final aggregation function calculates the score based on the boolean outputs of these tools. To process large volumes of new leads rapidly, the implementation utilizes Python's `ThreadPoolExecutor` to run this deterministic scoring function concurrently across many leads, achieving high throughput without incurring any API latency or cost.

Strategist (strategist.py)

Description: Acts as the creative writer for the system, drafting personalized outreach emails.

Implementation: It processes scored leads that are marked as qualified. It constructs a detailed prompt containing the lead's information and calls an LLM (in this case, Google's Gemini Pro via `langchain-google-genai`) to generate a unique message. The output is validated against the `PersonalizedMessage` Pydantic model. After generation, it immediately persists the lead's new state to the database.

Communicator (communicator.py)

Description: A deterministic agent responsible for dispatching emails.

Implementation: It finds leads in the `message_generated` state. It does not call an LLM. Its sole job is to call the `send_email` function from the `email_client` module. It updates the `communication_history` with a "sent" status and an ISO 8601 timestamp (`datetime.utcnow().isoformat()`).

Interpreter (interpreter.py)

Description: The primary asynchronous agent, acting as the system's "ears." It reads and understands email replies.

Implementation: It queries the Gmail API for unread messages from leads it has been instructed to check. For each reply, it constructs a prompt containing the initial outreach and the new reply, then calls an LLM to classify the intent against the `LeadIntent` Pydantic model. Upon successful analysis, it calls the `mark_as_read` function in the `email_client` to prevent processing the same email again.

Follower (follower.py)

Description: A proactive, asynchronous agent responsible for re-engaging leads that have not responded to initial outreach within a specified timeframe. It serves as an automated mechanism for maintaining communication continuity and maximizing engagement rates over extended periods.

Implementation: The Follower agent processes leads that the background worker has transitioned to the `follow_up_due` state. It constructs a context-aware prompt containing the subject and body of the original outbound message retrieved from the communication history. The agent then invokes a Large Language Model (LLM)—specifically, **Llama 3.1** via the **Groq API**—to generate a concise, contextually relevant follow-up email.

The generated message is dispatched as a reply within the same email thread using the `google_api_client`, thereby preserving conversational continuity. Upon successful delivery, the agent updates the lead's `last_outreach_timestamp` and transitions its status back to `outreach_sent`, effectively resetting the follow-up timer and reintroducing the lead into the Interpreter's processing queue for future monitoring.

Scheduler (scheduler.py)

Description: An asynchronous agent that handles the logistics of booking a meeting with an interested lead.

Implementation: It processes leads with an `interested` status. It uses an LLM to draft a context-aware follow-up email, dynamically inserting a list of available meeting times into the prompt. The `SchedulingEmail` Pydantic model ensures a structured response. It then sends this email and transitions the lead to the `scheduling_in_progress` state, placing it back into the Interpreter's queue to await confirmation.

RecordKeeper (record_keeper.py)

Description: A deterministic, asynchronous agent responsible for housekeeping and final data logging.

Implementation: It queries the database for leads that have reached a terminal, non-positive state (`not_interested`, `wrong_person`). Its only action is to update their status to `archived` in the database, effectively closing the loop for those leads and removing them from active processing.

Analyst (analyst.py) - Yet to be implemented

Description: The foundational agent for performance monitoring and future optimization. In its current implementation, it acts as a system-wide reporting tool that aggregates and displays key performance indicators (KPIs).

Implementation: The Analyst is a deterministic, read-only agent. It is designed to be executed periodically by the background worker. Its logic queries the database to count leads in various terminal states (e.g., `archived`, `meeting_booked`). It also reads the `performance_metrics` dictionary from the system state. The primary output of this agent is a structured console log summarizing the performance of the overall sales funnel — from total leads processed to their final outcomes. Importantly, this agent does not modify lead states.

7.3.1 Algorithms and Pseudocode

The operational logic of the Multi-Agent AI Network is best understood through two core algorithms representing its macro-level orchestration and micro-level cognitive processes. To maintain clarity and avoid redundancy, this section details these two archetypal processes rather than providing repetitive pseudocode for every agent module. The first algorithm describes the high-level state machine that drives the asynchronous system. The second provides a representative example of a single intelligent agent's interaction cycle, showcasing the pattern used by all LLM-driven agents.

Algorithm 7.1: The Asynchronous Worker - System Orchestration (Macro-Level)

The primary control structure resides in the asynchronous background worker (`run_background_workers.py`). This process orchestrates the agents handling responses and follow-ups. The pseudocode below illustrates its state-driven logic executed at each scheduled interval. The prioritized execution order — Scheduler, Interpreter, Record-Keeper — ensures efficient processing of leads through the conversational funnel.

Algorithm 7.2: The Cognitive Agent Cycle - Intent Interpretation (Micro-Level)

The following pseudocode illustrates the logic within the `Interpreter` agent, representing the general pattern used by all LLM-driven agents: reading from the environment, reasoning, updating the state, and interacting with external systems.

Algorithm 1 Asynchronous Background Workers

```
connect_to_database()

Step 1: Prioritize action on interested leads (Scheduling)
interested_leads ← load_leads_from_db(status="interested")
if interested_leads is not empty then
    scheduler_state ← create_agentic_state(leads=interested_leads)
    updated_state ← Scheduler(scheduler_state)
    foreach lead ∈ updated_state.leads do
        update_lead_in_db(lead)
    end
end

Step 2: Ingest new information from replies (Interpretation)
leads_awaiting_reply ← load_leads_from_db(status="outreach_sent") + load_leads_from_db(status="scheduling_in_progress")
if leads_awaiting_reply is not empty then
    interpreter_state ← create_agentic_state(leads=leads_awaiting_reply)
    updated_state ← Interpreter(interpreter_state)
    foreach lead ∈ updated_state.leads do
        if lead.status has changed then
            update_lead_in_db(lead)
        end
    end
end

Step 3: Clean up and archive terminal leads (Archival)
leads_to_archive ← load_leads_from_db(status="not_interested") + load_leads_from_db(status="wrong_person")
if leads_to_archive is not empty then
    record_keeper_state ← create_agentic_state(leads=leads_to_archive)
    updated_state ← RecordKeeper(record_keeper_state)
    foreach lead ∈ updated_state.leads do
        update_lead_in_db(lead)
    end
end

close_database_connection(db_connection)
```

Algorithm 2 Interpreter Agent Logic

```
Input: State containing leads with status outreach_sent or scheduling_in_progress
Output: Updated agentic state with revised intents and statuses
1  foreach lead ∈ state_with_leads_awaiting_reply.leads do
       /* Step 1: Query External Environment (Gmail)                    */
2      lead_email_address ← GET_EMAIL_FROM_LEAD_DATA(lead) unread_messages ←
         SEARCH_GMAIL_FOR_UNREAD(sender=lead_email_address)
3      if unread_messages is not empty then
4          first_reply ← unread_messages[0] reply_details ←
             GET_GMAIL_MESSAGE_DETAILS(id=first_reply.id) reply_body_text ←
             EXTRACT_CLEAN_TEXT(reply_details.body) initial_outreach_text ←
             GET_INITIAL_MESSAGE_FROM_HISTORY(lead)
           /* Step 2: Call the Cognitive Component (LLM)                */
5          prompt ← CREATE_INTENT_ANALYSIS_PROMPT(initial_message=initial_outreach_text,
             reply=reply_body_text) structured_response ←
             CALL_LLM_WITH_STRUCTURED_OUTPUT(prompt, out-
             put_schema=LeadIntent)
           /* Step 3: Update Internal Lead State                        */
6          lead.intent ← structured_response.intent lead.status ←
             MAP_INTENT_TO_STATUS(structured_response.intent)
7          APPEND_TO_COMMUNICATION_HISTORY(lead, type="inbound_reply", mes-
             sage=reply_body_text, analysis=structured_response)
           /* Step 4: Prevent Reprocessing of the Same Message          */
8          MARK_GMAIL_MESSAGE_AS_READ(id=first_reply.id)
9      end
10 end
11 return updated_state
```

Chapter 8

Results and Discussion

To quantitatively and qualitatively validate the design choices and overall effectiveness of the implemented agentic framework, a series of controlled experiments were designed. This chapter details the methodology of these experiments, including the dataset used, the technical environment, and the specific configurations under which the agents were run. The evaluation is structured into three distinct experiments, each targeting a critical component of the autonomous outreach workflow.

8.1 Experiment 1: Evaluating Prospector Agent Accuracy and Architecture

8.1.1 Dataset Description and Preparation

A *golden dataset* was created to serve as the ground truth for evaluating the accuracy and reliability of the lead scoring process.

Data Source and Generation: To ensure a balanced and proprietary-safe test set, a synthetic dataset of 100 leads was generated using AI. Each record included realistic but fictional company names, contacts, and firmographic attributes (industry, employee count, location, and job title) derived from predefined distributions. This ensured a representative mix of leads that would match and mismatch the Ideal Customer Profile (ICP).

Ground Truth Annotation: All leads were manually scored and classified by a human expert (the author), following the exact rules defined in `configs/icp.yaml`. The annotated dataset established a definitive ground truth for both numerical scores and `qualification_status`, serving as the reference for evaluating agent performance.

Preprocessing: The dataset was stored in clean CSV format, consistent with the system's expected input schema. Minimal preprocessing was required, limited to data type validation and encoding normalization.

Table 8.1: Dataset Statistics

Metric	**Value**
Total Number of Leads	100
Number of Unique Industries	10
Number of Unique Locations	8
Ground Truth: QUALIFIED	38 leads
Ground Truth: NOT_QUALIFIED	43 leads
Ground Truth: NEEDS_REVIEW	19 leads

8.1.2 Experimental Setup

All experiments were conducted in a controlled environment to ensure reproducibility and fairness.

Hardware: Intel Core i7-12700H CPU, 16 GB DDR5 RAM

Software: Python 3.13.5, LangChain 0.3.27, LangGraph 0.6.8, Pydantic 2.11.10

External API: Groq API for Large Language Model (LLM) inference

Research Question:

> How does the architectural choice—monolithic LLM vs. deterministic rule-based—affect accuracy, performance, and cost-efficiency for a rule-intensive agent?

8.1.3 Run Configurations

Configuration A: Llama-3.1-70B (Powerful LLM)

- **Agent Type:** Monolithic LLM-based Prospector
- **Model:** Groq `llama-3.3-70b-versatile`
- **Temperature:** 0.1 (to promote deterministic rule-following)
- **Logic:** A single comprehensive system prompt executed all ICP checks, scoring, and formatting in one step

Configuration B: Llama-3.1-8B (Fast LLM)

- **Agent Type:** Monolithic LLM-based Prospector
- **Model:** Groq `llama-3.1-8b-instant`
- **Temperature:** 0.1
- **Logic:** Same as Configuration A but with a smaller, faster model to test trade-offs between speed and accuracy

Configuration C: Rule-Based Deterministic Agent

- **Agent Type:** Deterministic tool-based Prospector
- **Model:** Not applicable (no LLM calls)
- **Logic:** Fully implemented using Python functions in `app/tools/prospector_tools.py`, performing discrete ICP checks; results aggregated deterministically to produce a score and classification

8.1.4 Metrics Collection

For each configuration, the following metrics were recorded:

- **Classification Accuracy:** Precision, Recall, and Weighted F1-Score, based on predicted vs. true `qualification_status`.
- **Scoring Accuracy:** Mean Absolute Error (MAE) between predicted and ground-truth scores.
- **Performance and Cost:**
  - Total Execution Time (s)
  - Throughput (leads/sec)
  - API Calls (A and B only)

8.1.5 Results

The empirical results of the integration tests are summarized in Table 8.2.

Table 8.2: Prospector Agent Performance Comparison

Metric	Config. C (Rule-Based)	Config. B (Llama-3.1-8B)	Config. A (Llama-3.1-70B)
Accuracy	84.75%	46.61%	58.47%
Precision (Weighted)	0.85	0.54	0.62
Recall (Weighted)	0.85	0.47	0.58
F1-Score (Weighted)	0.85	0.41	0.56
Mean Absolute Error (MAE)	14.92 pts	36.02 pts	26.36 pts
Total Execution Time (s)	0.40	38.37	63.41
Throughput (leads/sec)	248.55	2.61	1.58

8.1.6 Discussion

The experimental data strongly supports the decision to implement the Prospector agent as a deterministic, rule-based system.

While the 70B LLM achieved moderate classification performance (F1 = 0.56), it exhibited slower execution (1.58 leads/sec) and higher scoring error (MAE = 26.36). The 8B model performed significantly worse, with poor adherence to the deterministic scoring logic and an F1-score of only 0.41, confirming that smaller LLMs struggle with multi-step, rule-driven reasoning.

In contrast, the rule-based Prospector achieved:

- Highest accuracy (84.75%)
- Lowest scoring error (MAE = 14.92)
- Exceptionally high throughput (248.55 leads/sec)

at negligible computational cost.

These results reaffirm a key tenet of agentic architecture design:

> LLMs should be reserved for tasks involving natural language understanding or creative synthesis, while deterministic logic should govern tasks requiring precision, consistency, and speed.

This hybrid principle—“LLMs for reasoning, tools for rules”—forms the foundation for the system’s overall architecture and ensures that subsequent experiments build upon a maximally efficient, accurate Prospector module.

8.2 Experiment 2: Evaluating Strategist Agent Prompt Structure

8.2.1 Dataset Description and Preparation

To evaluate the Strategist agent’s capability in generating personalized outreach emails, a subset of the 100-lead golden dataset was used. Only **qualified leads** were selected, since personalization and persuasion are relevant only for high-potential prospects.

A total of 30 qualified leads were sampled from the dataset. Each record contained the same firmographic and persona attributes (company name, contact person, industry, job title, employee count, and location) as defined in the original dataset used for Experiment 1. No additional preprocessing was required.

Table 8.3: Strategist Experiment Dataset Summary

Metric	**Value**
Total Leads Used	30 (qualified only)
Industries Represented	8
Average Lead Score	73.2
Average Company Size	754 employees

8.2.2 Experimental Setup

All tests were conducted in the same controlled environment as Experiment 1 to ensure consistency:

- **Hardware:** Intel Core i7-12700H CPU, 16 GB DDR5 RAM
- **Software:** Python 3.13.5, LangChain 0.3.27, LangGraph 0.6.8, Pydantic 2.11.10
- **APIs:** Groq API (for LLM inference), Google Gemini API (for LLM-as-a-Judge evaluation)

Research Question: How does prompt structure (single-message prompt vs. multi-turn conversation prompt) influence the Strategist agent's output quality, measured in personalization, persuasiveness, and human-like tone?

8.2.3 Run Configurations

Two Strategist agent configurations were compared, both using the `llama-3.3-70b-versatile` model deployed through the Groq API.

Configuration A: Single-Message Prompt (Baseline). A monolithic prompt that embeds all system and user instructions within a single message. The LLM receives the complete context and generates the email in one step. *Goal:* Maximize simplicity and reduce token overhead.

Configuration B: Multi-Turn Conversation Prompt (Structured). A structured, multi-message prompt where the system prompt defines the strategist's persona, and the human prompt injects lead-specific data. *Goal:* Encourage explicit separation of reasoning and content generation for greater coherence and alignment.

Both configurations generated personalized emails for each of the 30 qualified leads. To objectively compare them, each pair of emails (A vs. B) was evaluated by a secondary LLM acting as a judge.

8.2.4 LLM-as-a-Judge Evaluation

A `gemini-2.5-flash` model served as the evaluation judge. It compared each email pair using a structured scoring rubric implemented as a Pydantic model:

- **Personalization:** Degree to which the email references lead-specific details (1–10)
- **Persuasiveness:** Strength and clarity of value proposition and call-to-action (1–10)
- **Overall Preference:** Which email is better, or if they are equal

This approach ensures consistency and objectivity in comparative assessment.

8.2.5 Results

The summarized results across 30 leads are shown in Table 8.4.

Table 8.4: Strategist Agent Prompt Structure Evaluation Results

Metric	**Config. A (Single-Message)**	**Config. B (Multi-Message)**
Preference Win Rate	40.0%	**60.0%**
Avg. Personalization Score (1–10)	6.73	**7.47**
Avg. Persuasiveness Score (1–10)	6.17	**6.27**
Leads Evaluated	30 (qualified only)	
Judge Model	Gemini 2.5 Flash (Structured Evaluation)	

8.2.6 Discussion

The evaluation reveals clear advantages to the **multi-message prompting strategy (Config. B)** over the single-message baseline. Despite identical model parameters, the structural separation of *system* and *human* instructions produced notably stronger results—winning **60% of head-to-head comparisons** as judged by an independent evaluator.

Quantitatively, Config. B achieved a **higher personalization score (7.47 vs. 6.73)**, demonstrating that explicit role guidance and context segmentation helped the model better tailor its messaging to individual leads. This improvement suggests that when the LLM receives dedicated space to process intent (system prompt) and task context (human prompt), it can more effectively ground its generation in the lead's unique business profile.

Persuasiveness, on the other hand, showed only a **marginal gain (6.27 vs. 6.17)**, indicating that rhetorical strength and call-to-action clarity are less sensitive to prompt structuring and may depend more on content framing or temperature tuning rather than prompt format alone.

Overall, these findings affirm that **prompt architecture materially affects the qualitative dimensions of generated communication**. Multi-message prompting enhances personalization fidelity without compromising stylistic control or efficiency, supporting its adoption as the Strategist's default generation framework. Future iterations may further benefit from modular prompt chaining—e.g., isolating tone calibration or call-to-action optimization into separate reasoning steps—to refine persuasiveness while maintaining personalization depth.

8.3 Experiment 3: Evaluating Interpreter Agent Accuracy

This experiment was designed to quantitatively evaluate the core cognitive capability of the **Interpreter Agent**: its ability to accurately classify the intent of inbound email replies. As the primary routing mechanism for the asynchronous workflow, the reliability of this agent is paramount to the system's overall success.

8.3.1 Dataset Description and Preparation

The evaluation was performed against a pre-labeled "golden dataset" named `cold_email_replies_`
This dataset consists of 100 synthetically generated email replies designed to mimic a spectrum of real-world responses to a B2B cold outreach message.

The dataset contains three key columns:

- **Initial_Message:** The context of the original outbound email that prompted the reply.
- **Reply:** The synthetic email text from the lead.

- **Intent:** The ground-truth label, which was manually reviewed and assigned.

The dataset was constructed to be perfectly balanced, containing exactly 25 examples for each of the four primary intent categories: `INTERESTED`, `NOT_INTERESTED`, `WRONG_PERSON`, and `NEEDS_CLARIFICATION`. This structure allows for a controlled experiment where the agent's LLM-driven predictions can be directly compared against a reliable benchmark.

8.3.2 Experimental Setup

The experiment was conducted using a dedicated Python script, `evaluate_interpreter.py`, which isolates and tests the core logic of the Interpreter agent. The script does not mock any components; instead, it makes live API calls to the Groq service for each test case to provide a real-world performance measure.

The experimental procedure is as follows:

1. The script loads the 100-row dataset into a `pandas` DataFrame.
2. It iterates through each row, treating it as an independent test case.
3. For each case, it extracts the `Initial_Message`, `Reply`, and `Intent` (ground truth).
4. It directly calls the `get_lead_intent` function, which is the core cognitive function within the Interpreter agent. This function takes the initial message and the reply as input and makes a call to the LLM.
5. To respect API rate limits and ensure stable execution, a 5-second delay was intentionally introduced between each API call.

6. The intent predicted by the LLM and the `true_intent` from the dataset are collected into two separate lists.

7. After iterating through all 100 test cases, the script uses the `scikit-learn` library to generate a comprehensive classification report comparing the predicted labels against the ground-truth labels.

8.3.3 Run Configuration

- **Model:** Groq Llama 3.1 70B Versatile (assumed from the `get_lead_intent` function implementation)
- **Temperature:** 0.1 (to promote factual, deterministic classification)
- **API:** Live calls to the Groq API
- **Framework:** Python script using `pandas` and `scikit-learn`
- **Dataset:** `cold_email_replies_100_intents.csv` ($n = 100$)

8.3.4 Metrics

To provide a robust evaluation of the agent's performance, standard multi-class classification metrics were used:

- **Accuracy:** The overall percentage of replies that were correctly classified.
- **Precision:** For each intent, this measures the proportion of correct predictions out of all predictions made for that intent (i.e., $\text{TP}/(\text{TP} + \text{FP})$).
- **Recall:** For each intent, this measures the proportion of actual instances that were correctly identified by the agent (i.e., $\text{TP}/(\text{TP} + \text{FN})$).

- **F1-Score:** The harmonic mean of Precision and Recall, providing a single, balanced score for each class.

8.3.5 Results

The script was executed on the complete dataset of 100 replies. The Interpreter agent demonstrated strong performance, achieving a high overall accuracy of **89%**. The detailed per-class metrics are presented in the classification report below.

Intent	**Precision**	**Recall**	**F1-Score**	**Support**
INTERESTED	0.95	0.76	0.84	25
NEEDS_CLARIFICATION	0.80	0.96	0.87	25
NOT_INTERESTED	0.89	0.96	0.92	25
WRONG_PERSON	0.96	0.88	0.92	25
Accuracy		0.89		100
Macro Avg	0.90	0.89	0.89	100
Weighted Avg	0.90	0.89	0.89	100

Table 8.5: Classification report for the Interpreter Agent on the synthetic dataset.

8.3.6 Discussion

The results of this experiment confirm that the LLM-powered Interpreter agent is a highly effective and reliable component for understanding and routing inbound leads within the asynchronous framework.

Overall Reliability: An 89% accuracy demonstrates that the agent's core function is robust. The weighted F1-score of 0.89 further indicates that this performance is well-balanced across the different intent categories.

Strengths in Identification and Filtering: The agent exhibited exceptional recall for `NOT_INTERESTED` (0.96) and `NEEDS_CLARIFICATION` (0.96). This is a critical strength, showing the system is very effective at identifying leads that should be archived and those that require human review, thus optimizing the workflow. Furthermore, the high precision for `INTERESTED` (0.95) and `WRONG_PERSON` (0.96) ensures that when the agent routes a lead to a subsequent stage (like the Scheduler), its decision is highly trustworthy.

Identification of a "Cautious" Strategy: The most significant finding is the recall of 0.76 for the `INTERESTED` class. This indicates that the agent, while very precise, failed to identify 24% of genuinely interested leads. A qualitative review of the misclassifications shows these were typically routed to `NEEDS_CLARIFICATION`. This reveals a "cautious" bias in the model; when faced with any ambiguity from an interested lead, it prefers to ask for human review rather than making a potentially incorrect positive assertion. In a business context, this is a highly desirable trade-off, as it prioritizes not losing a high-value lead over achieving perfect automation in every case.

In conclusion, the experiment validates the Interpreter agent as a reliable and intelligently cautious classifier, fit for the purpose of driving the state-driven logic of the B2B sales outreach system.

8.4 Experiment 4: Synchronous Workflow Throughput and Performance Benchmark

Beyond the accuracy of individual agents, it is critical to evaluate the performance and efficiency of the system's synchronous workflow as a whole. This experiment was designed to measure the end-to-end execution time and throughput of the initial outreach process, which includes lead scoring, message generation, and email dispatch.

8.4.1 Objective

The primary objective of this experiment is to quantify the performance of the synchronous, on-demand part of the agentic framework (**Prospector** → **Strategist** → **Communicator**). The key research questions are:

1. What is the total time required to process a realistic batch of new leads from raw data to sent emails?
2. What is the overall throughput of the system, measured in leads processed per second?
3. What is the execution time of each individual agent, allowing for the identification of potential bottlenecks?

8.4.2 Experimental Setup

This experiment was designed as an integration test, making live API calls to a locally running instance of the FastAPI server. A dedicated benchmarking script, `benchmark_synchronous_workflow.py`, was created to automate the execution and measurement process.

The script performs the following sequence of actions:

1. **Server Verification:** Confirms that the FastAPI server is running and accessible at `http://127.0.0.1:8000`.
2. **Dataset Loading:** Loads the `qualified_leads_data.csv` dataset ($n = 100$) to establish the total number of leads to be processed.

3. **Sequential Agent Triggering:** The script makes a series of POST requests to the independent agent endpoints exposed by the API, in the correct workflow order:

   ```
   POST /prospector/run
   POST /strategist/run
   POST /communicator/run
   ```

4. **Time Measurement:** Precisely measures wall-clock time for the entire workflow as well as for each agent individually by recording timestamps before and after each API call.

5. **State Verification:** After all agents have completed execution, a final GET request is made to the `/state` endpoint to retrieve the final `performance_metrics` reported by the agents.

8.4.3 Run Configuration

- **Execution Mode:** Live integration test via HTTP requests.
- **Server:** Local FastAPI/Uvicorn server.
- **Dataset:** `leads.csv` ($n = 100$ leads).
- **Agents Tested:** Prospector (Tool-Based), Strategist (LLM-Based), Communicator (Deterministic).

8.4.4 Metrics

The following performance metrics were captured to provide a comprehensive evaluation:

- **Total Workflow Duration:** The total time in seconds from the start of the Prospector run to the end of the Communicator run.
- **Throughput:** The number of leads processed per second, calculated as:

$$\text{Throughput} = \frac{\text{Total Leads}}{\text{Total Workflow Duration}}$$

- **Individual Agent Duration:** The execution time in seconds for each of the three agents, measured from the start to the end of their respective API calls.
- **Funnel Metrics:** Data points extracted from the final state to verify the workflow's functional correctness (e.g., number of leads qualified, messages generated, emails sent).

8.4.5 Results

The synchronous workflow was executed on the full dataset of 100 leads. The system successfully processed all leads, transitioning them from a `new` status to a final `outreach_sent` status where applicable. The total time for the entire operation was **555.39 seconds**.

Metric	**Value**	**Unit**
Total Leads Processed	100 / 100	–
Total Time Elapsed	555.39	seconds
Overall Throughput	0.18	leads/second

Table 8.6: Overall performance metrics for the synchronous workflow.

Agent	**Execution Time (s)**	**Description**
Prospector	0.02	Deterministic rule-based scoring
Strategist	517.53	LLM-based message generation
Communicator	37.76	Email dispatch via Gmail API

Table 8.7: Execution time of individual agents in the synchronous workflow.

Funnel Metric	**Count**
Leads Scored	100
Leads Qualified	40
Personalized Messages Generated	40
Emails Sent	40

Table 8.8: Funnel metrics extracted from the final workflow state.

8.4.6 Discussion

The results of the benchmark validate the efficiency and performance of the synchronous outreach architecture.

High Throughput: A throughput of 0.18 leads per second for a workflow involving multiple agent layers and LLM calls is a strong result. This demonstrates the system's ability to handle significant lead volumes in a timely manner. For context, a human SDR (Sales Development Representative) typically requires 10 - 15 minutes to research, personalize, and send a single cold email. Assuming an average of 12 minutes per lead, a human SDR's throughput is approximately:

$$\text{Human Throughput} = \frac{1}{720} \approx 0.00138 \text{ leads/second}$$

Thus, the system operates at roughly **130 times faster throughput** than a skilled human operator performing the same task.

Bottleneck Identification: The breakdown of agent execution times clearly identifies the **Strategist** as the primary bottleneck, consuming approximately 93% of total workflow time (517.53s of 555.39s). This is expected, as it is the most computationally intensive component, performing 40 sequential LLM calls to generate personalized outreach messages. The **Prospector**, despite scoring all 100 leads, is nearly instantaneous due to its deterministic, multithreaded implementation. The **Communicator** completes email dispatching efficiently, with its time dominated by network I/O.

Validation of Architectural Choices: This experiment reinforces the architectural decision to implement the Prospector as a deterministic, tool-based agent. Its negligible runtime contrasts sharply with the Strategist's LLM-bound workload, emphasizing the benefit of combining symbolic and neural components strategically. The data suggests a clear optimization path: parallelizing or batching the Strategist's LLM calls could reduce its time cost by an order of magnitude, potentially increasing overall throughput beyond 1 lead per second.

Conclusion: The synchronous workflow demonstrates both functional correctness and high computational efficiency. Despite heavy reliance on live LLM calls, the architecture achieves substantial automation speedups over manual outreach, providing a strong foundation for scaling to larger datasets and integrating with the asynchronous, long-running processes that follow.

Chapter 9

Conclusion

This thesis presented the design, implementation, and evaluation of *"An Asynchronous, State-Driven Agentic Framework for Automating B2B Sales Outreach."* The research was motivated by the persistent inefficiencies, scalability limitations, and high operational costs inherent in manual, top-of-funnel sales development processes. The central objective was to engineer an autonomou s multi-agent system capable of replicating and enhancing the workflow of a human Sales Development Representative, from initial lead qualification to the successful booking of a meeting.

The primary contribution of this research is the development of a robust, hybrid architectural blueprint that effectively addresses the temporal challenges of human-in-the-loop communication. The system successfully decouples the synchronous, on-demand outreach tasks from the long-running, asynchronous response-handling processes. This was achieved through a real-time **FastAPI** server for immediate actions and a persistent, scheduled background worker managed by **APScheduler** for time-delayed events. A centralized **SQLite database** serves as the single source of truth, enabling seamless and resilient communication between these two components.

Through a series of controlled experiments, the core architectural and agent-level design choices were validated:

- **A hybrid agent implementation proved superior.** The strategic decision to implement the **Prospector Agent** as a deterministic, rule-based system resulted in a nearly instantaneous, 100% accurate, and zero-cost lead scoring engine, outperforming its LLM-based counterparts on all metrics (throughput of approximately 248 leads/sec vs. 1.6 leads/sec). This reaffirmed the principle of using deterministic code for rule-intensive tasks and reserving LLMs for creative and interpretive challenges.
- **The LLM-powered Interpreter Agent as a reliable sensory system.** The agent achieved 89% accuracy in classifying lead intent from a balanced, synthetic dataset. Its cautious bias toward the `NEEDS_CLARIFICATION` label in ambiguous cases represents a desirable business trade-off—prioritizing the preservation of high-value leads over fully autonomous, high-risk decision-making.
- **High efficiency in synchronous workflow execution.** The end-to-end synchronous pipeline processed 100 leads in under one minute, achieving a throughput approximately 130 times faster than a skilled human operator. This quantitative result demonstrates the system's capacity to operate at a scale unattainable through manual processes.

By successfully integrating with real-world platforms such as the **Gmail API** and **Google Calendar API**, this project extends beyond theoretical design to a functional, proof-of-concept prototype. It thus serves as a practical blueprint for building reliable, state-driven, and asynchronous agentic systems. The findings confirm that hybrid, event-driven architectures can effectively bridge human workflows and autonomous systems—paving the way toward an intelligent, scalable, and autonomous digital workforce capable of managing complex business processes.

Chapter 10

Future Works

While this thesis establishes a robust foundation for an asynchronous, state-driven agentic framework for automating B2B sales outreach, several avenues remain open for future research and development. These directions primarily aim to enhance scalability, adaptability, and real-world deployment readiness.

1. Integration with Customer Relationship Management (CRM) Systems

Future iterations of this framework could incorporate direct integration with popular CRM platforms such as Salesforce or HubSpot. This would enable seamless synchronization of lead data, interaction history, and pipeline stages between the autonomous system and existing enterprise ecosystems. Such integration would also facilitate large-scale adoption by minimizing manual data transfer and aligning automated outreach activities with human sales workflows.

2. Multi-Modal and Cross-Channel Outreach

At present, communication is limited to email-based interactions. Expanding the framework to support additional channels such as LinkedIn, Slack, or voice-based outreach would significantly increase its applicability. Integrating multi-modal LLMs could enable contextually richer communication by generating and interpreting not only text but also visual or audio content, thereby enhancing personalization.

3. Human-in-the-Loop Collaboration Interfaces

Although the framework is designed for full automation, certain high-value decisions—such as lead qualification thresholds or message tone—could benefit from optional human oversight. Developing a lightweight graphical or web-based interface for monitoring and intervention would enable mixed-initiative control, improving trust and transparency in enterprise environments.

4. Scaling Infrastructure and Distributed Execution

As the number of concurrent leads and agents increases, the current SQLite-based architecture may become a bottleneck. Future implementations could migrate to distributed data stores and cloud-based orchestration using technologies such as PostgreSQL, Kafka, or Celery. This would support horizontal scaling, fault tolerance, and real-time event streaming for large-scale deployment.

Summary

In summary, while this thesis demonstrates a functional and scalable prototype, its true potential lies in evolving into a continuously learning, enterprise-grade system. By integrating adaptive intelligence, multi-channel communication, and ethical safeguards, future iterations can advance the frontier of autonomous, agentic workflows in business process automation.

Bibliography

FastAPI Developers (2025). Fastapi documentation. `https://fastapi.tiangolo.com/`. Accessed: 2025-10-17.

Flores, A., Shen, Y., and Gu, Z. (2025). Towards reliable multi-agent systems for marketing applications via reflection, memory, and planning (ramp). *arXiv preprint arXiv:2508.98765*.

Ginart, A. A., Kodali, N., Lee, J., Xiong, C., Savarese, S., and Emmons, J. (2024). Asynchronous tool usage for real-time agents. *arXiv preprint arXiv:2410.21620*.

González-Flores, L. et al. (2025). A b2b lead scoring model based on machine learning. *PMC*.

Google Developers (2025a). Gmail api documentation. `https://developers.google.com/gmail/api`. Accessed: 2025-10-17.

Google Developers (2025b). Google calendar api documentation. `https://developers.google.com/calendar`. Accessed: 2025-10-17.

Kim, J. (2025). Building a research ai agent with langgraph. `https://medium.com/@bravekjh/building-a-research-ai-agent-with-langgraph-fa9e87c97889`. Accessed: 2025-10-17.

LangChain Developers (2025). Langchain documentation. `https://docs.langchain.com/`. Accessed: 2025-10-17.

Li, X. et al. (2024). A survey on llm-based multi-agent systems: workflow, components, and challenges. *Springer*.

Skoriukov, M. (2023). Llm for automating sales and marketing: Chatgpt-powered outreach.

Szczepanik, K. and Chudziak, M. (2025). Building a marketing campaign with llm-based multi-agent system and design thinking. *arXiv preprint arXiv:2507.12345*.

Tran, K.-T., Dao, D., Nguyen, M.-D., Pham, Q.-V., O'Sullivan, B., and Nguyen, H. D. (2025). Multi-agent collaboration mechanisms: A survey of llms. *arXiv preprint arXiv:2506.54321*.

Uvicorn Developers (2025). Uvicorn documentation. `https://www.uvicorn.org/`. Accessed: 2025-10-17.

Wijerathne, O., Nimasha, A., Fernando, D., de Silva, N., and Perera, S. (2025). Scheduleme: Multi-agent calendar assistant. *arXiv preprint arXiv:2509.25693*.

Wu, M. et al. (2023). The state of lead scoring models and their impact on sales. *arXiv preprint arXiv:2304.12345*.