POLITECNICO DI TORINO

Master's Degree in Computer Engineering - Artificial Intelligence and Data Analytics

Master's Degree Thesis

Design and Implementation of a Custom Multi-Agent Platform for LLM-based AWS Cloud Orchestration

Supervisors

Prof. Paolo GARZA

Ing. Mariagrazia CARDILE

Candidate

Sergio LAMPIDECCHIA

10/2025

Summary

This thesis investigates how Large Language Models (LLMs) can be combined with agent-based architectures to overcome their limitations in managing long-term tasks, preserving context, and interacting with external tools.
Developed in collaboration with Data Reply, the project focuses on building a Multi-Agent System (MAS) where LLM-based agents collaborate in a structured workflow.
The system is organized around three core agents: a planner, which interprets natural language instructions and decomposes them into actionable steps; an extractor, responsible for identifying and structuring the relevant parameters and contextual information; and an executor, which interacts with AWS cloud services.
Platform's main goal is to autonomously interpret natural language instructions and manage AWS cloud services, deploying, configuring, and monitoring resources, thus supporting developers in the software development lifecycle.

Table of Contents

List of Tables

List of Figures

Acronyms

AI

artificial intelligence

Chapter 1

Introduction

In recent years, Large Language Models (LLMs) have brought significant progress in Artificial Intelligence, particularly in the fields of natural language processing. Beyond simply recognizing and interpreting text, these models can generate outputs that are both coherent and contextually appropriate, enabling them to perform a wide range of tasks.

However, while powerful, a single LLM often struggles with managing long-term tasks, maintaining context across extended interactions, or interacting directly with external systems and APIs. To overcome these limitations, developers have started combining LLMs with agent-based architectures, where the LLM is combined with dedicated components (agents) capable of planning, orchestrating actions, invoking external tools, and persisting knowledge over time.

Each agent is assigned a specific role or area of expertise and collaborates with other agents to accomplish more complex tasks. To achieve this, agents can retain relevant information through memory, adapt their decisions based on the surrounding context, and interface with external tools and services, such as APIs, file systems, or cloud platforms, to perform concrete actions in the real world.

This thesis project has been carried out in collaboration with **Data Reply**, a company within the Reply group specialized in Artificial Intelligence, Big Data and Quantum Computing. The work is part of a broader research effort on the application of Generative AI and autonomous agents to the software development lifecycle (SDLC). It aims to explore how intelligent agents based on LLMs can support developers by automating infrastructure setup, launching and managing AWS cloud services, producing documentation, and monitoring cloud resources.

This project is inspired by the idea of building a Multi-Agent System (MAS)

where several LLM-based agents collaborate to achieve complex objectives. Rather than operating as a simple chatbot, the system is designed to behave more like an autonomous assistant that can understand high-level goals expressed in natural language and coordinate a set of actions to fulfill them.
More specifically, the core objective is to design a system capable of interpreting natural language instructions and autonomously deploying, configuring, and managing AWS cloud services.

To achieve this, the project is designed around a set of guiding principles.
The coordination of these agents is generally managed by a **supervisor agent**, which orchestrates the overall workflow by deciding which agents should be activated, how tasks should be delegated, and which reasoning strategies are most suitable in a given context.
To support more coherent and adaptive behavior, the system incorporates memory mechanisms at two levels: **short-term memory**, which maintains the state of the current session and ongoing tasks, and **long-term memory**, which preserves relevant knowledge for reuse across different stages or sessions.
Finally, the system is evaluated systematically through a **test suite**, designed to assess its ability to plan effectively, coordinate collaboration among agents, and accomplish tasks in a robust and interpretable manner.

The thesis is organized as follows:

1. **Foundations**: Introduces the theoretical background, including Large Language Models, intelligent agents (memory, planning, reasoning, learning), multi-agent systems (MAS), cloud computing, and Software Development Lifecycle (SDLC).

2. **System Design**: Outlines the architecture and design principles of the proposed multi-agent system, describing communication flows, memory management, reasoning strategies, and human-in-the-loop supervision.

3. **System Implementation**: Details the development stack, integration of tools, and the mechanisms for execution, error handling, and system recovery.

4. **Evaluation and Testing**: Presents the benchmarking methodology, the experimental setup, and results, followed by an analysis of system performance and identified limitations.

5. **Conclusion and Future Work**: Summarizes the key contributions of the thesis and highlights promising directions for future research and development.

Chapter 2

Foundations

2.1 Large Language Models (LLMs)

Large Language Models (LLMs) are a class of advanced deep learning models designed to process, understand, and generate natural language in a coherent and contextually relevant way.
Built on transformer-based neural networks [1] and trained on vast corpora of textual data, these models are capable of capturing complex linguistic patterns, semantic relationships, and world knowledge.
Thanks to their scalability and generalization capabilities, LLMs have set new benchmarks in a wide range of Natural Language Processing (NLP) tasks, including text classification, sentiment analysis, machine translation, summarization, question answering, conversational agents, and even code generation.

2.1.1 Transformer Basics

The vast majority of modern LLMs are built upon the **Transformer** architecture. Unlike previous sequence models such as Recurrent Neural Network (RNNs) and Long Short-Term Memory networks (LSTMs), which processed data sequentially and often struggled to model long-range dependencies, Transformers operate entirely in parallel, enabling both greater efficiency and richer contextual understanding.
At the heart of the Transformer is the **self-attention** mechanism, which allows each token in a sequence to direct attend to all other tokens simultaneously, regardless of their relative position. This parallelism not only enables more efficient training on modern hardware, but also allows the model to capture richer contextual relationships between words.
The core building block of a Transformer model is composed of multiple stacked layers, each integrating two core components:

- **Multi-Head Self-Attention**: This mechanism allows the model to consider multiple representation subspaces simultaneously by computing attention weights in parallel across multiple "heads". It allows the model to assess the relevance of each token with respect to the entire input sequence.
- **Feedforward Neural Networks**: A position-wise fully connected layer applied after the attention step to refine and transform the intermediate representations. This component helps the model to learn complex, non-linear transformations beyond the patterns captured by attention.

In addition to these components, each block also incorporates **residual connections**, **layer normalization** and **position encodings**. These elements are critical for stabilizing training, improving convergence speed, and enabling the training of very deep architectures.

Variants of the Transformer

Depending on the task, LLMs can rely on different Transformer architectures, which are generally grouped into three main categories:

- **Encoder-Decoder**: This architecture is suitable for sequence-to-sequence tasks such as machine translation, abstractive summarization, and dialogue generation. The **encoder** processes the input sequence to build a rich contextual representation that captures its meaning, while the **decoder** generates the target sequence step by step, attending both to the encoder's output and to its own past predictions.
- **Encoder-Only**: These models are primarly designed for understanding and representation tasks where the goal is to extract meaning from the input rather than generate new text. By capturing bidirectional context, they are especially effective for applications such as classification, sentiment analysis, and named entity recognition. Well-known examples include BERT and its derivatives, which leverage deep bidirectional self-attention to produce high-quality contextual embeddings.
- **Decoder-Only**: These models are specifically optimized for generative tasks such as language modeling, code generation, and conversational agents. They operate in an autoregressive fashion, generating text one token at a time while conditioning each prediction solely on the tokens that precede it. This is achieved through the use of *causal* (masked) self-attention, which prevents the model from accessing future information during training or inference. GPT and its successors are the most prominent examples of this family, demonstrating state-of-the-art performance in open-ended text generation.

Transformer Layer Components

The internal architecture of Transformer-based LLMs is modular, comprising multiple stacked layers. Each layer is built from specialized subcomponents that collaboratively encode, process, and generate natural language. Their primary role is to transform raw input tokens into rich contextual representations, enabling the model to capture dependencies and reason effectively over text.
The following section outlines the core elements typically found within a Transformer layer:

- **Embedding and Positional Encoding**: Input tokens are first mapped into dense vector embeddings that capture their semantic meaning. Since the Transformer has no inherent notion of sequence order, positional encodings are added to the embeddings, allowing the model to incorporate information about token positions.

- **Self-Attention and Multi-Head Attention**: Self-attention enables each token to dynamically focus on other tokens within the sequence, assigning weights based on contextual relevance. Multi-head attention extends this by computing several attention patterns in parallel, with each head capturing different types of dependencies (e.g., syntactic vs. semantic). This mechanism allows the model to represent both local and global relationships.

- **Feedforward Network**: Following the attention layers, each token representation is processed independently through a feedforward neural network, usually composed of two linear transformations separated by a non-linear activation (ReLU or GELU). This step enriches the representation and increases the model's expressiveness.

- **Add & Norm Layers**: Residual connections combined with layer normalization are applied around both the attention and feedforward blocks. These operations stabilize training, preserve information across layers, and ensure gradient flow even in very deep models.

- **Masked Self-Attention in Decoders**: In decoder architectures (e.g., GPT), the self-attention mechanism applies a causal mask that blocks access to future tokens. This enforces autoregressive generation, ensuring that each token is predicted solely based on past context.

- **Cross-Attention (Encoder–Decoder Models)**: In sequence-to-sequence tasks such as translation, the decoder also includes a cross-attention layer. Here, the decoder attends to the encoder's output, allowing it to condition token generation on the full input sequence.

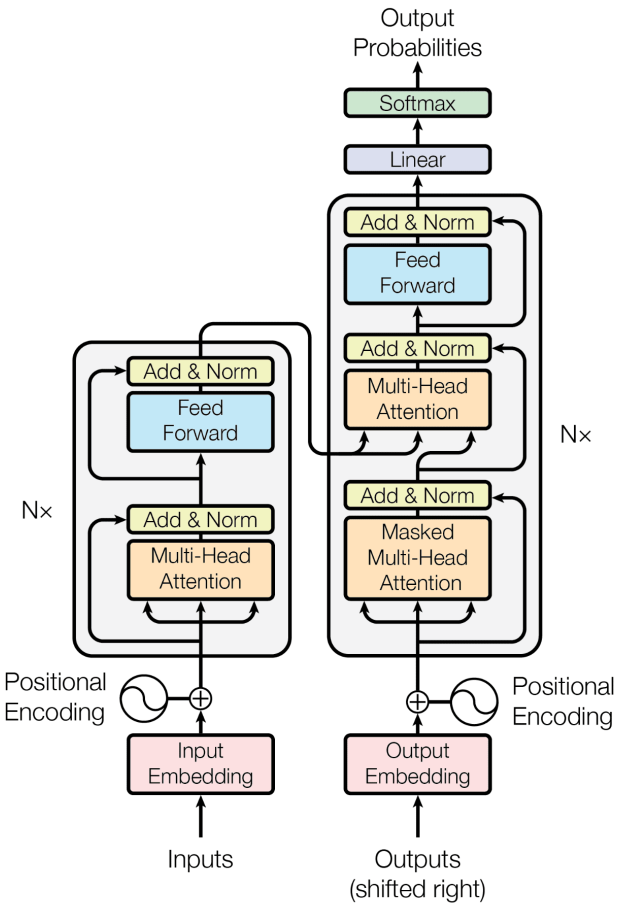

Figure 2.1: Transformer architecture.

2.2 Intelligent Agents with LLMs

Intelligent agents are autonomous systems capable of perceiving their environment, reasoning about the information they acquire, and take actions aimed at achieving specific goals.
Historically, they have been a central topic in Artificial Intelligence, often realized through symbolic reasoning, decision trees, or domain-specific machine learning techniques.
The integration of modern Large Language Models (LLMs) has significantly extended the capabilities of these agents, enabling the development of generalist agents capable of perception, reasoning, and action across a wide range of domains [2].

By embedding an LLM at their core, intelligent agents gain the ability to understand and respond to natural language instructions with remarkable flexibility, making interactions with them feel intuitive and human-like. They can interpret unstructured information, such as free-form text or conversational input, and distill it into relevant, actionable knowledge.
Finally, their capacity to interface with digital tools, APIs, and external systems enables them to extend their capabilities far beyond standalone reasoning, effectively bridging human intent with a wide range of computational resources [3].

2.2.1 The LOMAR Framework

To support structured development and analysis of intelligent agents powered by LLMs, the **LOMAR** framework provides a conceptual blueprint that captures their essential components and operational principles.

LOMAR stands for *Language Model, Objective, Memory, Actions, Rethink*, and offers a systematic way to reason about how such agents can interpret goals, maintain context, interact with external tools, and refine their behavior over time.

- **LLM**: The core reasoning engine. It interprets user input, formulates plans, and generates context-aware responses based on pre-trained knowledge and context-specific information.

- **Objective**: The agent's driving goal or task. It defines the problem space and drives the decomposition of tasks, determining the direction of the planning and execution strategies.

- **Memory**: Allows the agent to retain and retrieve relevant information from prior interactions. Memory is essential for preserving context, maintaining coherence in multi-step reasoning, supporting and personalization or long-term adaptation.

- **Actions**: Constitute the set of operations that the agent can perform to interact with its environment. These may include invoking external tools, calling APIs, executing code, manipulating data, or accessing cloud services. Through actions, the agent translates intentions into real-world effects.

- **Rethink**: A reflective component that enables the agent to evaluate its previous decisions and outputs, adapt its strategies, and review decisions using internal assessments, environmental feedback, or human input.

Agent Roles

The effectiveness and realism of an intelligent agent depend not only on its functional capabilities, but also on its **role profile**, which defines the agent behavior, interaction style, and reasoning patterns.
Depending on the application, an agent may adopt roles, such as teacher, developer, assistant, or domain expert. A role profile generally encompasses several dimensions:

- **Basic Information**: Demographic and background attributes such as age, gender, professional expertise, language skills, or domain specialization. Although not strictly necessary for functionality, these elements help align the agent's behavior with user expectations, especially in simulated or instructional settings.

- **Psychological Traits**: Personality characteristics influence how an agent communicates and makes decisions. Attributes such as assertiveness, empathy, or analytical thinking affect its interaction style and perceived trustworthiness. Communication preferences, such as formality, tone, or degree of verbosity, can also be configured to better suit the target user. In addition, agents may be designed with specific cognitive style.

- **Social Context**: Agents rarely operate in isolation. Their roles relative to users (e.g., mentor, peer, or subordinate) and other agents (e.g., team leader or collaborator) influence how they behave in multi-agent environments. Socially aware agents may exhibit cooperation, negotiation, or even conflict resolution behaviors, depending on the structure and dynamics of the system they are part of.

Memory and Context Management

In LLM-based agents, memory plays a crucial role in maintaining continuity, coherence, and context-awareness across multi-step tasks.
Unlike traditional stateless pipelines, agent-based orchestration systems benefit from memory mechanisms that allow agents to recall prior inputs, reuse previously generated outputs, and adapt their behavior dynamically throughout the session.
The memory structure adopted in this system is inspired by cognitive psychology, where memory is typically divided into distinct layers based on function and duration.
Similarly, the agent's internal memory can be categorized into different levels, each serving a specific purpose during the orchestration process.

- **Short-Term Memory (STM)**: Covers the contextual information available within a single session (e.g., current conversation). STM allows the agent to understand follow-up instructions, avoid repeating actions, and adapt to ongoing context.
- **Long-Term Memory (LTM)**: Refers to persistent knowledge across multiple sessions. It includes specific task histories and outcomes (episodic memories), as well as general knowledge, learned strategies, or reusable configurations (semantic memory).
- **Hybrid Memory**: Combines STM and LTM to offer access to both recent and persistent knowledge. This design supports dynamic adaptation, allows cross-session reasoning, and enables more intelligent decision-making in complex workflows.

Memory Formats

The structure and format of memory also play a crucial role in how information is stored, retrieved, and applied.
Different formats are employed depending on the type of data and its intended purpose:

- **Natural Language Text**: Human-readable format commonly used for storing conversational history, user instructions, or logs in human-readable form.
- **Vector Embeddings**: High-dimensional representations that capture semantic meaning, enabling similarity-based retrieval and contextual matching.
- **Structured database**: Relational or document-based formats that organize information using predefined schemas, allowing precise and efficient access to specific fields or entities.
- **Hierarchical structures**: Formats such as JSON or YAML allow the storage of nested, context-rich information, useful for representing decision paths or agent states.
- **Multi-format systems**: Hybrid approaches that integrate different formats (e.g., text, vectors, and structured elements) to balance both interpretability with computational efficiency.

Memory Operations

In intelligent agents, memory revolves around two fundamental operations: **reading** and **writing**. These operations enable the agent to retrieve past experiences

while continuosly integrating new information into its internal state.

Reading allows the agent to recall previously stored information in order to guide current decisions, predict outcomes or interpret user instructions. The effectiveness of this operation depends on several factors such as the *recency* of the information, its *relevance* to the current context, and its overall *importance* for the agent's objectives.

Writing, on the other hand, refers to storing new information or the updating existing entries. To remain efficient, the agent should minimize redundancy by refining or merging previously stored knowledge rather than duplicating it. Writing also requires managing memory limitations, which may involve deciding whether to retain, summarize, or discard specific pieces of information. In some cases, this process extends to reflexive updates, where the agent records evaluations of its own performance in order to refine future decision-making.

Planning and Reasoning

In an intelligent agent system, planning is the cognitive process that transforms a user's high-level intent into a coherent and executable sequence of actions. It acts as the bridge between natural language input and system behavior, allowing the agent to interpret complex instructions, decompose them into atomic operations, and determine the appropriate execution order.
Effective planning is essential for orchestrating multi-step tasks, resolving resources dependencies, and ensuring that actions are both feasible and contextually consistent.
In this system, the planning process is closely integrated with the reasoning capabilities of the underlying language model. The model segments the prompt, assigns meaningful descriptions to each subtask, and identifies the relationships among them. Two main strategies are used to support this capability:

Planning without Feedback

- **Single-Path Reasoning**: Executes a predefined linear sequence of steps. It prioritizes efficiency but lacks adaptability in dynamics scenarios.
- **Multi-Path Reasoning**: Evaluates multiple alternative paths at each decision point, enabling more flexible and informed choices.
- **External Planner**: Transforms high-level goals into formats compatible with external planners, such as symbolic reasoning engines or automated planning tools.

Planning with Feedback

- **Environmental Feedback**: Adapts the agent strategy based on real-time responses or environmental conditions.
- **Human Feedback**: Integrates user interventions such as corrections, clarifications, or direct supervision to iteratively improve behavior and outcomes.
- **Model Feedback**: Performs internal self-assessments to refine planning and detect inconsistencies or errors over time.

Advanced Planning Methods

- **Chain-of-Thought (CoT)**: Breaks reasoning into a sequence of natural language steps to improve interpretability and task performance, particularly in arithmetic and logic tasks.
- **Program-of-Thought (PoT)**: Represents reasoning through executable code (e.g., Python scripts), leveraging the precision and modularity of programming to handle complex tasks.
- **Tree-of-Thought (ToT)**: Builds a tree of potential reasoning paths, allowing the agent to explore, evaluate, and prune alternatives based on intermediate decisions or utility scores.
- **Self-Ask**: Allows the model to generate and answer its own clarifying questions before reaching a conclusion, mirroring human-like curiosity and doubt resolution.
- **Graph-of-Thought (GoT)**: Structures reasoning as a graph, modeling dependencies, casual relationships, and iterative refinement loops among ideas, concepts, or subgoals.

Actions Execution

Once the planning and reasoning stages have been completed, the system proceeds to the execution phase.
This stage is responsible for turning the agent's internal decisions into concrete operations that interact with the external environment. These actions are typically goal-oriented and reflect the agent's ability to operationalize its understanding in real environments, whether digital or physical.
Agent actions usually fall into one of the following categories:

1. **Task Execution**: Performing well-defined tasks, such as writing documents, answering queries, generating code, translating text, or solving problems.

2. **Communication**: Exchanging information with other agents and users, which may involve negotiation, clarification, or collaborative task management.

3. **Exploration and Learning**: Interacting with the environment to acquire new knowledge, test hypotheses, and refine internal models.

The form of action execution largely depends on the degree of integration with external systems or resources.
The system supports multiple execution modes, including:

- **Tool and API Integration**: The agent may invoke external tools or API (e.g., HuggingGPT, WebGPT, Gorilla, ToolFormer, and API-Bank) to perform specialized tasks, which allow the agent to delegate reasoning steps, perform real-time data retrieval, or trigger remote procedure.

- **Database Access**: Actions may involve querying structured (e.g., SQL databases) or semi-structured (e.g., knowledge graphs, document stores) sources. This enables the agent to validate information, retrieve contextual data, or answer fact-based queries reliably.

- **Standalone Reasoning**: In some scenarios, the agent relies solely on its internal capabilities without accessing any external resource. This mode is useful when actions must be fast, lightweight, or completely self-contained.

Learning

Learning defines how an agent adapts to its environment and improves its performance over time. It represents the mechanism through which the agent evolves beyond static programming, gradually refining its strategies and behaviors based on experience.
Different learning paradigms provide complementary ways for agents to improve, depending on the nature of the task, the context in which they operate, and the type of data available.

Learning without Fine-Tuning

This paradigm doesn't involve changing the model's internal parameters but instead leverages prompt conditioning and in-context learning.

- **Prompt Engineering**: The model learns from the data provided in the prompt, relying on examples or task descriptions provided in real time [4].

- **Zero-Shot**: The model is provided with only the task description, and it generates an answer without any specific examples. This approach assumes that the model has implicitly learned how to perform the task prior to training.
- **Few-Shot**: The prompt includes a series of demonstrations consisting of input-output pairs that represent the desired behavior. This helps the model better understand the user's intent and what kind of output is expected.
- **Instruction Prompting**: The task is described through natural language instructions rather than examples. This often requires being detailed and explicit to guide the model's behavior effectively.

Learning with Fine-Tuning

Here the model's internal weights are updated to better align with domain-specific tasks or user preferences. This is typically achieved through:

- **Supervised Fine-Tuning**: Training on labeled datasets to specialize in particular tasks (e.g., medical QA, legal reasoning).
- **Reinforcement Learning from Human Feedback (RLHF)**: The agent learns optimal behaviors through iterative feedback loops with human evaluators or reward models.

2.3 Multi-Agent Systems (MAS)

Multi-Agent Systems (MAS) are computational frameworks composed of multiple autonomous agents operating in a shared environment. Each agent is capable of making decisions, perceiving changes in its surroundings, and performing actions either independently or in coordination with others.
Depending on the context, agents may pursue their own goals or collaborate to achieve a share objective.
MAS architectures enhance scalability, fault tolerance, and task specialization, making them suitable for complex, distributed scenarios.

Architectures and Communication

The structure of a Multi-Agent System (MAS) plays a crucial role in determining how agents coordinate, share information, and divide responsibilities.
Different architectural models define how agents interact and how control flows through the system. The choice of architecture depends on the complexity of the

tasks, the level of autonomy required, and the scale of the system [5]. The most common MAS architectures include:

- **Flat Architecture**: In this model, all agents operate at the same level, and communicate using a peer-to-peer protocol. No single agent holds global control, which promotes decentralization, flexibility, and rapid interaction. This architecture is well-suited for small to medium-sized systems that require agility and distributed decision-making.
- **Hierarchical Architecture**: Agents are organized in multiple levels, forming a top-down control structure. Higher-level agents (often referred to as supervisors or coordinators) manage and delegate tasks to lower-level agents. This model supports clear responsibility distribution and is particularly effective in complex systems that require centralized planning with distributed execution.
- **Team-Based Architecture**: Agents are grouped into specialized teams, each responsible for a specific subtask. Within each team, agents collaborate closely, while coordination between teams ensures alignment with global goals. This structure encourages specialization, modularity, and dynamic role allocation.
- **Hybrid Architecture**: Combines elements of the above models to support flexible and adaptive coordination strategies. For example, a system may use a hierarchical structure for task assignment while enabling peer-to-peer communication within teams. Hybrid architectures are particularly useful in dynamic environments where agent roles, communication patterns, and workloads change over time.

Agent Relationships

In a Multi-Agent System (MAS), agents interact with each other in different ways based on the alignment or divergence of their individual goals.
These relationships significantly influences system behavior, coordination mechanisms, and overall performance.
The most common types of agent relationships are:

- **Cooperative**: Agents collaborate toward a shared goal, contributing their capabilities to maximize collective success. Cooperation may occur through unconditional assistance or more deliberative forms, where agents critically evaluate proposals before reaching consensus. These systems often rely on trust, mutual support, and coordinated strategies to maximize global performance.
- **Competitive**: Agents pursue independent or even conflicting goals, often competing for limited resources, attention, or rewards. Competition is typical

in adversarial environments, game-theoretic environments, or market-based simulations, where agents aim to outperform one another.

- **Mixed**: In many real-world scenarios, cooperation and competition coexist. Agents may form temporary alliances or collaborate on shared subgoals while still pursuing individual interests. This setting is typical in multi-agent games, negotiation frameworks, and economic simulations where strategic behavior and alliance formation play a key role.

Communication Strategies

Communication is a key component of agent collaboration. The strategy chosen to manage communication affects how quickly agents converge on solutions, how they interpret shared information, and how they resolve conflicts [5].
The system supports several communication modes:

- **One-by-One**: Agents communicate sequentially, preserving message order and context. This approach supports structured reasoning and is particularly effective for tasks such as collaborative problem-solving or code generation.

- **Simultaneous-Talk**: All agents communicate simultaneously without waiting for other, promoting the rapid exchange of diverse ideas. This strategy is useful for tasks requiring brainstorming, voting, or aggregating multiple viewpoints.

- **Simultaneous-Talk with Summarizer**: In this variation, a dedicated summarizer agent reviews all inputs after each round and produces a consolidated message. This helps reduce redundancy, align contributions, and guide the discussion forward with improved clarity.

Evaluation Approaches

Assessing the performance of a Multi-Agent System involves measuring its effectiveness, reliability, and alignment with expected outcomes.
Several evaluation criteria are commonly used:

- **Success Rate**: Measures how often agents successfully complete the assigned task. This is particularly useful in structured tasks such as planning, negotiation, or gameplay.

- **Operational Cost Analysis**: Evaluates the computational and resource cost of running the system, including token usage in LLMs and API calls. This is essential for assessing scalability and sustainability

- **Simulation Coherence**: Checks whether the overall agent behaviors appears realistic and internally consistent with respect to the environment and objectives.
- **Validity Testing**: Verifies that agent outcomes reflect known real-world dynamics, especially in simulations involving decision-making or policy outcomes.
- **Information Propagation**: Analyzes how information spreads among agents over time, identifying knowledge flow and the emergence of shared beliefs.
- **Diffusion Speed**: Measures how quickly a message, signal, or idea spreads throughout the system.
- **Source Influence**: Quantifies the actual impact of individual agents or entities on the system's final outcomes, highlighting leadership or bias.
- **Sensitivity Analysis**: Studies how small changes in input (e.g., prompt, context, initial configuration) affect overall system behavior.
- **Robustness and Stability**: Assesses whether the system behaves consistently across repeated runs or under minor variations in setup.
- **Applicability**: Evaluates whether the system is useful in practice, with respect to the needs of specific stakeholders such as researchers, policymakers, or commercial users.

2.4 Cloud Computing

Cloud computing refers to the delivery of computing services, such as servers, storage, databases, networking, software over the Internet, commonly referred as "the cloud". This paradigm enables organizations and individuals to access scalable and flexible IT resources without the need to purchase or maintain physical infrastructure directly. Services are typically offered on a pay-as-you-go basis, with cloud providers managing the underlying infrastructure transparently.

One of the most significant advantages of cloud computing is its **cost efficiency**. By leveraging economies of scale, providers are able to reduce the unit cost of computing resources, and users avoid the large upfront capital expenditures associated with on-premises infrastructure. Instead, they only pay for the resources they actually consume.

Equally important is the **scalability** and **elasticity** of cloud platforms. Resources

can be dynamically scaled up or down in response to fluctuations in demand, eliminating the inefficiencies of overprovisioning and allowing organizations to adapt their infrastructure in real time. This flexibility is complemented by the **speed** and **agility** of the cloud: new resources can be provisioned within minutes, greatly accelerating the development and deployment of applications.

Cloud computing also enhances **productivity**. By outsourcing the complexity of maintaining servers and data centers, organizations can concentrate on their core business functions rather than infrastructure management. At the same time, **reliability** is strengthened through mechanisms such as data redundancy, automated backup, and disaster recovery, which together ensure higher levels of system availability and fault tolerance.

Cloud Service Models

Cloud computing is typically organized into three main service models, which differ in the level of abstraction provided and in the division of management responsibilities between the provider and the user:

- **Infrastructure as a Service (IaaS)**: Delivers fundamental computing resources such as virtual machines, storage, and networking components. Users retain control over the operating system, middleware, and applications, while the cloud provider manages the underlying physical infrastructure. This model offers maximum flexibility but requires higher technical expertise from the user.

- **Platform as a Service (PaaS)**: Provides a managed environment for application development and deployment. The cloud provider handles infrastructure, runtime, and platform services, allowing developers to concentrate on writing code and building functionality. This model accelerates development cycles and reduces operational overhead.

- **Software as a Service (SaaS)**: Offers complete, ready-to-use applications delivered over the Internet. Users access the software through a web browser or API, while the provider manages all aspects of the service, including infrastructure, updates, and security. This model is ideal for end-users who prioritize ease of access and minimal management effort.

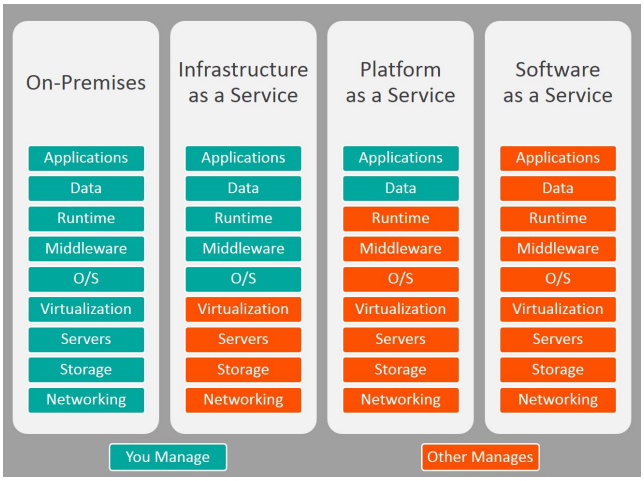

Figure 2.2: Comparison between On-Premises and Cloud service models (IaaS, PaaS, SaaS).

Cloud Deployment Models

Cloud resources can be delivered through different deployment models, each tailored to specific organizational needs and trade-offs between scalability, control, and cost.

In the **public cloud**, services are hosted on shared infrastructure and made available to users over the Internet. The infrastructure is fully owned, operated, and maintained by third-party providers, who pool resources across multiple customers. This model is particularly attractive for its high scalability, flexibility, and cost-efficiency, since organizations only pay for what they use and do not need to invest in hardware or ongoing maintenance.

By contrast, the **private cloud** is designed for exclusive use by a single organization. The infrastructure can be hosted on-premises or managed by an external provider but remains dedicated to one entity. This model offers stronger control over data, enhanced security, and compliance with regulatory requirements, making it suitable for industries that handle sensitive information such as healthcare, banking, or government services. However, these advantages come at the expense of reduced scalability and higher costs compared to public cloud solutions.

The **hybrid cloud** combines elements of both public and private environments, enabling organizations to distribute workloads strategically. For example, sensitive data may remain in the private cloud for security and compliance reasons, while less critical workloads are offloaded to the public cloud to benefit from elasticity and cost savings. This approach provides a balance between control and scalability,

while also supporting greater operational flexibility and resilience.

2.5 Software Development Life Cycle (SDLC)

The Software Development Life Cycle (SDLC) is a structured framework that defines the stages involved in the creation, deployment, and maintenance of software systems.
It provides a theoretical and methodological foundation for software engineering, ensuring that development activities are systematic, repeatable, and aligned with both user and business needs.
The SDLC is typically divided into the following key phases:

1. **Planning**: In this initial stage there is the definition of project objectives, scope, resources, constraints, and risks. This phase establishes the overall vision and feasibility of the initiative.

2. **Analysis**: This phase focuses on gathering, examining, and documenting both functional and non-functional requirements. It establishes a clear understanding of user needs and system specifications, providing the foundation upon which all subsequent development activities are built.

3. **Design**: In this phase, the documented requirements are translated into a coherent technical blueprint. It defines the software architecture, data structures, user interfaces, and system workflows, ensuring that the solution is well-structured and ready for implementation.

4. **Implementation**: During this phase, the system is constructed through coding. Developers translate design specifications into executable software using programming languages, frameworks, and tools.

5. **Testing & Integration** : The developed system is rigorously tested through unit tests, integration tests, system tests, and acceptance tests, to ensure functional correctness, performance efficiency, and reliability.

6. **Maintenance**: After deployment, the software requires ongoing updates, bug fixes, and enhancements to adapt to changing requirements, technologies, and operating environments.

Figure 2.3: The Software Development Life Cycle (SDLC) phases.

From a theoretical perspective, the the Software Development Life Cycle (SDLC) embodies core principles of process management, risk mitigation, and quality assurance. It provides a structured framework for coordinating complex technical activities, ensuring that development is systematic, traceable, and aligned with organizational goals.
Over time, several models have been proposed to structure the SDLC.
The **Waterfall Model** adopts a strictly sequential flow of phases, moving linearly from requirements analysis to deployment.
The **V-Model** builds on this approach by explicitly linking each development stage with its corresponding testing activity, emphasizing verification and validation.
In contrast, more recent **Iterative and Agile Approaches** emphasize adaptability, incremental delivery, continuous feedback, and close collaboration with stakeholders.

In contemporary practice, the SDLC serves not only as a methodological guideline but also as a foundation for integrating emerging paradigms. Examples include *DevOps*, which bridges development and operations through automation and shared responsibility, and *Continuous Integration/Continuous Deployment (CI/CD)*, which accelerate software delivery while maintaining quality standards.
More recently, Large Language Model (LLM)-based agents have been introduced as active participants in the SDLC, assisting in tasks such as requirements analysis, automated code generation, testing, and system monitoring [6].

Chapter 3

System Design

This chapter presents the architectural foundations of the system, which enables the orchestration of cloud infrastructure through natural language interaction. The primary objective is to provide an intelligent interface between human users and cloud platforms, capable of translating high-level textual requests into structured, validated, and executable actions.

At its core, the system is built around a team of specialized agents powered by Large Language Models (LLMs). The agent-based design promotes modularity, task specialization, and parallel execution, making the system both flexible and scalable.

By combining advanced reasoning capabilities with external tool integration and contextual memory, the system is able to automate complex workflows while preserving human oversight and control.

The following sections describe the design principles, agent architecture, memory mechanisms, planning strategies, and execution flow that underpin the system.

3.1 Design Goals and Principles

The primary objective of the system is to provide a modular, extensive, and interpretable framework that enables users to automate AWS cloud infrastructure tasks without writing code or relying on the Management Console. This is made possible through a multi-agent architecture that integrates Large Language Models (LLMs) with a library of specialized tools for executing cloud operations.
The design of the system is guided by several core principles.

Modularity ensures that core functions such as planning, parameter extraction, execution, validation, and logging are encapsulated within dedicated, self-contained components. This separation ensures that each component can be independently developed, tested, replaced, or extended without affecting the others.

Closely related is the principle of **Separation of Concerns**, which enforces a strict division between responsabilities. Planning determines what needs to be done; parameter extraction and validation identify which data is required for each action; and execution carries out the tasks using the appropriate tools.
This clear delineation promotes readability, simplifies debugging, and supports clean architectural layering.

Another fundamental principle is **Parallelism**. Although some cloud operations are inherently sequential due to interdependencies, many steps, particularly planning and parameter extraction tasks, can be performed concurrently.
The system is designed to exploit this parallelism to reduce latency and improve responsiveness, especially when handling multi-action prompts.

Equally important is the principle of **Traceability**. Every decision made by the system, along with the extracted parameters, the tools invoked, the execution results, and the resource metadata is systematically recorder.
These logs are exported in both human-readable (Markdown) and machine-readable (JSON) formats.

Finally, the system enforces **Structured Reasoning** through the use of Pydantic models. Each agent is required to produce output that conforms to a predefined schema, ensuring that all actions are interpretable, predictable, and safe to execute. This not only reduces the risk of misinterpretation but also enhances the reliability and correctness of the overall pipeline.

3.2 Architecture Overview

The system is organized into a set of clearly separated modules, each responsible for a specific part of the process: from understanding the user's natural language request, to extracting the necessary parameters, and finally executing the appropriate AWS actions. Each module is stored in its own directory, making the codebase clean, modular, and easy to maintain or extend.
Below is a description of each main component in the architecture.

1. `core/` – Main Agents

This folder containts the three core agents that form the reasoning and coordination backbone of the system:

- **`planner.py`**: Receives the user prompt and decompose it into a sequence of structured AWS actions. It also defines execution order based on dependencies between actions.
- **`extractor.py`**: Extracts the parameters required for each action using an LLM and returns a validated object conforming to predefined Pydantic schemas.
- **`executor.py`**: Executes the planned actions in the appropriate order. It invoke the corresponding AWS tools, monitors task status, and handles execution errors.

2. `aws_actions/` – AWS Tool Layer

This folder contains the low-level implementations of AWS cloud operations using the Boto3 Python SDK. Each script corresponds to a specific AWS service or capability.
Here is a list of the tools currently available:

- **`bucket.py`**: Create or reuses an S3 bucket.
- **`lambda_function.py`**: Creates a Lambda function.
- **`handler.py`**: Generates and uploads Lambda handler code.
- **`trigger.py`**: Adds a trigger (EventBridge rule) to schedule Lambda function invocations.
- **`iam.py`**: Create or reuses an IAM role.
- **`policy.py`**: Attaches an inline IAM policy to a role.
- **`permission.py`**: Adds permissions to allow Lambda invocation.
- **`api_gateway.py`**: Exposes a Lambda function through an API Gateway endpoint.
- **`dynamo.py`**: Manages DynamoDB tables and records with operations such as create, insert, update, retrieve, and delete.
- **`ec2.py`**: Manages EC2 instances (launch, start, stop, terminate).
- **`cost.py`**: Retrieves AWS cost information using AWS Cost Explorer.

3. `schemas/` – Parameter Validation

This folder defines all input validation logic, implemented using Pydantic schemas. Each schema is associated with a specific AWS tool and specifies the expected input format, including both required and optional parameters.
These schemas ensure that all mandatory parameters are correctly extracted and typed before execution begins.
At the same time, malformed or incomplete requests are detected early, ensuring that tools never receive invalid input.
In addition to validation, the schemas provide strong typing and serve as a form of documentation for each tool's interface, making the system both more reliable and easier to maintain.

4. `models/` – Action Definitions

This folder defines shared Pydantic data structure **`action_schema.py`**, which standardizes the representation of AWS actions across the system. The schema specifies the core fields that describe an action:

- **`action`**: Identifies the type of AWS operation to be performed.
- **`text`**: Stores the portion of the original prompt associated with the action.
- **`description`**: Provides a human-readable explanation of the task, making logs and documentation more interpretable..

5. `my_logging/` – Logging and Reporting

This package implements the logging infrastructure used for both runtime monitoring and post-execution analysis. It includes two main modules

- **`logger.py`**: Offers general-purpose logging utilities.
- **`task_logger.py`**: Records the history of executed tasks along with the extracted parameters, execution results and overall system status.

Logs are automatically exported in both **JSON** and **Markdown** formats, including task status, execution timestamps, LLM token usage, and estimated costs (USD) based on OpenAI pricing.

6. `tests/` – Prompt-Based Testing

This folder contains a full suite of **24 prompt-based tests** designed to simulate the real user requests. The goal of these tests is to validate the entire pipeline, from planning to parameter extraction and execution. All test cases are defined in the **`tests.py`** file, and each run automatically logs its output while producing a summary report with the corresponding pass/fail status. Through this mechanism, the test suite provides assurance of correctness and reliability while also serving as a safeguard against regressions during further development.

Running Example

To illustrate how the modules interact, the following example presents through a complete workflow in which a natural language request is translated into specific AWS operations.

Deploying a Scheduled Lambda Function

User request : *"Create a Lambda function called `LambdaExample` that runs every day at 9 AM, using the provided IAM role, and store its logs in the specified S3 bucket."*

Assumptions :

- The user has valid AWS credentials with sufficient permissions to manage the required resources.
- Default values for region and account are either preconfigured or explicitly provided.
- Existing resources are reused whenever available, following a *create-or-reuse* policy.
- All resource names supplied by the user comply with AWS naming conventions.

Planned actions (`core/planner.py`) :

1. Create or reuse the specified S3 bucket for logs.
2. Create or reuse the specified IAM role with basic execution policy and S3 write permissions.
3. Create the Lambda function associated with the IAM role.

4. Generate and upload the handler code package.
5. Create an EventBridge rule to trigger the Lambda daily at 09:00 AM, and configure the necessary invocation permissions..

Parameter extraction and validation (`core/extractor.py` + `schemas/`) : During this stage, the system derives all necessary parameters from the natural language request and validates them against the corresponding Pydantic schemas.

Parameter	**Value**
Lambda name	LambdaExample
Description	Executes daily and stores execution logs into the specified S3 bucket
Handler	Deployment package (lambdaexample.zip) containing the handler script `lambdaexample.py`
S3 Bucket name	Provided by the user
IAM Role name	Provided by the user
Policies	`AWSLambdaBasicExecutionRole` + Custom inline S3 write policy
Schedule expression	`cron(0 9 * * ? *)` (daily at 09:00)

Execution mapping (`core/executor.py` + `aws_actions/`) :

- `bucket.py` → Create or reuse S3 bucket for logs.
- `iam.py`, `policy.py` → Create or reuse role; attach inline policy for S3.
- `lambda-function.py` → Create Lambda bound to the IAM role.
- `handler.py` → Generate and Upload handler code.
- `trigger.py` → Create EventBridge rule and permission to invoke Lambda.

Outputs, Logging and Traceability (`my_logging/`) : All operations are logged to ensure transparency and reproducibility.
The system produces both human-readable (Markdown) and machine-readable (JSON) reports which include: planned actions, extracted parameters, invoked tools, execution results, and timings.
Additionally, the logger tracks LLM token usage and estimates AWS API call costs.

Failure Handling :

- Parameters are validated against Pydantic schemas. If required fields are missing or invalid, execution is halted and the system requests explicit user input to correct or provide the missing values.
- The create_or _reuse policy prevents duplication by reusing existing resources when available.
- Temporary AWS service failures are managed through bounded retries with exponential backoff, which reduces overload and increases resilience to intermittent issues.

3.3 Communication and Collaboration

The system relies on the tightly coordinated interaction of three specialized agents: the **planner**, the **extractor**, and the **executor**. Each agent is responsible for a distinct stage of the orchestration pipeline:

- **Planner**: Interprets the user's natural language request and decomposes it into a sequence of structured AWS actions ordered according to their logical dependencies.
- **Extractor**: Retrieves and validates the parameters required for each action (e.g., bucket names, Lambda function names, schedules), ensuring completeness and type safety through Pydantic schemas.
- **Executor**: Executes each action in the correct order, invoking the appropriate AWS tools and tracking execution status.

Rather than relying on asynchronous messaging system or external orchestrators, the agents communicate directly through function calls, exchanging structured data in the form of validated Pydantic objects.
Each agent completes its task and then explicitly transfers control to the next by

returning structured output.
This approach, referred to as *sequential modular delegation*, allows each agent to remain isolated and focused on its specific responsibility, while ensuring smooth coordination and traceability across the pipeline.

Sequential Interaction Flow

The collaboration between agents is organized as a strictly ordered pipeline.

The process begins with the **planner**, which interprets the user's natural language request through an LLM and decomposes it into a structured sequence of AWS actions. The planner's output is constrained by a predefined schema (`MultiActionSchema`), ensuring syntactic correctness and compatibility with downstream components.

Once the sequence of actions has been produced, it is passed to the **extractor**. At this stage, a second LLM-based invocation derives the parameters required for each action. Each AWS operation is represented as a validated Pydantic object, such as `BucketSchema` or `LambdaFunctionSchema`, ensuring that all inputs are complete, consistent and type-safe.

In the final step, the **executor** receives the complete sequence of action–parameter pairs and executes them in the correct order. It invokes the corresponding AWS operation, while simultaneously monitoring the execution status to prevent redundant actions and to ensure the robustness of the workflow.

Orchestration

Although the system doesn't explicitly include a "supervisor" agent, the planner implicitly assumes this coordinating role by establishing the correct task order according to action dependencies. In this way, orchestration emerges naturally from the planning process itself, which keeps the overall architecture lightweight while still preserving a coherent global execution strategy.

This design choice brings several advantages. Since each agent operates independently and communicates through well-defined input/output schemas, the system maintains a high degree of modularity. At the same time, the deterministic flow of information across agents ensures transparency and makes the reasoning process fully traceable. Finally, the architecture remains inherently extensible, as new agents or tools can be incorporated without disrupting the existing workflow, enabling the system to evolve alongside future requirements and technological

advances.

Execution and Tool Invocation

The final stage of the pipeline is the execution layer, where the planned actions are carried out through a suite of modular tools. Each tool is implemented as an independent Python function built on top of the Boto3 SDK and is invoked directly by the executor. To promote clarity and maintainability, tools are organized into dedicated modules named after the AWS resource they manage (e.g., `bucket.py`, `lambda_function.py`, `trigger.py`).

A distinctive aspect of the execution layer is its reliance on strict validation: tools accept only structured and pre-validated parameter objects, ensuring both syntactic correctness and semantic consistency before execution.

Furthermore, the tools are designed with module isolation in mind, meaning that each operates independently of the others. This separation not only simplifies debugging but also increases the predictability and robustness of the overall system. So potential errors are localized and prevented from propagating across components. At the same time, this design facilitates incremental development and testing, since individual tools can be updated, replaced, or extended without introducing unintended side effects in the rest of the pipeline.

Finally, execution errors are managed through a framework of uniform error handling. Failures are centrally captured and logged in a standardized format, guaranteeing transparency and traceability. In interactive mode, the user may also be prompted to provide missing or corrected information, allowing the system to recover gracefully without compromising reliability.

This design results in a pipeline that is modular, auditable, and robust—bridging the gap between high-level user intent and concrete infrastructure deployment.

3.4 Memory and Contextual Reasoning

To manage complex, multi-step user requests and ensure consistent execution across dependent operations, the system employes a lightweight yet powerful memory mechanism designed to support contextual reasoning, parameter disambiguation, and dependency resolution, which are essential for coherent execution within a single interaction cycle.
Although memory is not persisted across sessions, its temporary nature is sufficient

to guarantee continuity during short-lived interactive tasks while also safeguarding user privacy and maintaining real-time responsiveness.

Conversational Memory

The memory is implemented as a structured **conversational history**, which evolves dinamically during the session.
Rather than storing only the user's raw prompts, it captures the entire execution trace, that includes the original instructions, the sequence of structured actions generated by the planner (together with their type, description, and associated text), and the validated parameters extracted by the extractor.
This shared state is passed between all agents (planner, extractor, executor) and updated at every step. Internally, it is encoded as a sequence of structured messages, including `HumanMessage`, `AIMessage`, and `SystemMessage`, which are provided as context to the LLM during each round of parameter extraction.
This enables the model to reason with full awareness of the task sequence and prior references.

The memory is ephemeral, retained only for the duration of the current session. It is discarded when the session ends or it is manually reset by the user. This design choice is motivated by both privacy considerations and a focus on real-time responsiveness, as the system is primarily intended for short-lived interactive tasks. Despite its non-persistent nature, the system maintains a `task_log.jsonl` file that records all actions, extracted parameters, and timestamps. This log supports debugging, auditing, and post-hoc analysis.

One of the key benefits of the memory mechanism is its ability to enable contextual disambiguation. This process resolves ambiguities in user input by drawing on conversational context, semantic cues, recency, and logical flow.
For instance, when a user states:

> "Deploy the Lambda using the same bucket as before."

the phrase *"the same bucket"* is ambiguous in isolation. A traditional system would require the user to explicitly restate the bucket name. By contrast, a memory-enabled LLM can retrieve prior context, through the full conversation history, and correctly infer that this is the intended bucket.

Beyond resolving ambiguities, the memory mechanism can also infer missing details from the recent context. For example, it may automatically use the most recently created role ARN if one was just generated. In addition, it can adapt to user corrections or refiniments, such as switching to a newly specified bucket.

This approach allows the interaction to remain fluid and natural, reducing the need for repetitive clarification and making the system more adaptive to real conversation dynamics.

Handling Long Conversations

To support complex interactions, the system includes the entire conversational history when performing parameter extraction.
Unlike approaches that impose strict artificial truncation or message limit, this design allows the model to reconstruct the entire request context whenever needed. This capability is particularly valuable in scenarios where later actions depend on earlier steps in the dialogue.

Although the current implementation does not yet employ truncation or summarization, the pipeline remains robust thanks to its modular design.
The planner breaks down the requests into smaller, well-designed blocks and the extractor processes each block independently. This ensures that prompts remains within manageable bounds while still preserving the continuity of the overall conversation.

Limitations

Although the current short-term memory mechanism is effective within a single session, it has inherent limitations.
Most notably, it does not persist information across sessions, which prevents the system from tracking deployed infrastructure over the long term. In addition, the mechanism relies exclusively on the language model's ability to reason over the conversation history, without the support of symbolic memory or a structured knowledge base.

Another important limitation concerns scalability in extremely long sessions.
Since the entire conversational history must ultimately be represented within the model's context window, very large dialogues may exceed available capacity. Moreover, even when the window is not fully saturated, longer sessions inherently make it harder for the model to retrieve and prioritize the most relevant details from the context. Addressing these challenges would require strategies such as hierarchical memory or dynamic summarization in order to preserve continuity without overwhelming the model.

Looking ahead, future iterations of the system could overcome these limitations by

introducing a persistent memory layer capable of storing parameter values, resource metadata and session context. This would enable multi-session continuity and team-level collaboration, allowing users to reference infrastructure deployed days or weeks earlier.

Despite these constraints, the current architecture already demonstrates the strength of combining a transient but structured conversational memory with a capable language model. Even without persistent storage, this approach is sufficient to enable rich, context-aware reasoning within complex, multi-step workflows in a single session.

3.5 Human Feedback and Control

Although the system is designed to operate autonomously once a natural language request is received, human input and oversight remain essential for ensuring reliability, security and transparency. To this end, the architecture integrates lightweight humain-in-the-loop mechanisms that provide flexibility, support error recovery, and reinforce user control without undermining automation.

Authentication and Credential Management

Before any cloud action is executed, the system requires the user to authenticate by providing their AWS credentials.
These include the `AWS_ACCESS_KEY-ID` and `AWS_SECRET_ACCESS_KEY`, which are collected at the beginning of the session and securely stored in a local `.env` file.
This approach ensures that credentials remain under the sole ownership of the user and that no action is taken without explicit authorization. By embedding authentication as a prerequisite step, the system maintains both operational integrity and user trust, preventing unauthorized access while enabling seamless orchestration once credentials are verified.

Session Initialization

Once authentication has been successfully completed (as described in the previous subsection), the system initializes the environment and provides the user with a welcome prompt.
At this stage, the agent explicitly list the available command and awaits the first request.

```
Welcome to the AWS Agent!
How can I help you today?
```

```
Available commands:
  • 'exit'          → quit the agent
  • 'reset'         → reset the conversational context
  • 'tokens'        → show GPT tokens used so far and estimated cost
  • 'reset tokens'  → reset token and cost counters

Write your request:
```

This initialization step provides a clear entry point for the user, ensuring transparency and lowering the barrier to interaction by making the system state and controls immediately visible.

Interactive Parameter Resolution

In some cases, the extractor may be unable to retrieve or validate the necessary parameters from the user prompt either because information is missing, ambiguous, or incorrectly formatted.
When this occurs, the system enters a fallback mode explicitly prompting the user to supply or correct the required values through an interactive interface. This guarantees that execution begins only when all parameters are complete, consistent, and confirmed.

Rather than interrupting the workflow with continuous requests for confirmation, the system follows a principle of semi-autonomous execution.
Once the necessary preconditions are met and all parameters are validated, the orchestration pipeline proceeds without further intervention. Human interaction is thus limited to moments where it is strictly necessary, such as resolving ambiguities or completing initial setup (e.g., authentication).
This design strikes a balance between autonomy and user control, ensuring both ease of use and robustness in edge cases.

Chapter 4

System Implementation

4.1 Development Stack and Libraries

The system is developed in **Python**, using a modular and folder-based architecture. Each module encapsulates a specific functionality, enabling the pipeline to be extended or modified with minimal impact on the rest of the system.
Development is conducted using **Visual Studio Code (VSCode)**, which offers integrated support for Python and integrated tools for version control and environment management.

4.1.1 LangChain

Figure 4.1: LangChain Logo.

LangChain is a powerful open-source framework designed to simplify the development of applications based on LLMs by providing high-level abstractions and integrations.

On the development side, LangChain provides a composable interface for building `agents`, `chains`, `tools`, and `workflows` leveraging both the LangChain and LangGraph libraries. Once a prototype is ready, it can be deployed through the LangGraph Platform, which transforms experimental prototypes into production-ready APIs. Finally, the companion tool LangSmith enables developers to monitor and refine their applications by inspecting execution traces, evaluating performance, and analyzing agent behavior in detail.

LangChain is composed of several interoperable packages, each contributing to a specific layer of functionality:

- **`langchain_core`**: Defines the fundamental abstractions that underpin the entire ecosystem, including models, prompts, message objects, and execution logic.
- **`langchain_openai`, `langchain_anthropic`, other provider-specific packages**: Provide streamlined integrations with popular LLM providers. These modules handle the communication between LangChain and external APIs, abstracting away low-level details and providing developers with uniform interface for working with different models.
- **`langgraph`**: Introduces a graph-based paradigm for agent orchestration, allowing workflows to be represented as nodes and edges. This makes it possible to design flexible, non-linear reasoning processes where agents can branch, loop, or converge depending on the task.
- **`langchain_community`**: Acts as hub for third-party extensions, tools and components. It extend's the framework capabilities beyond its core modules, providing access to a broad and evolving set of resources tailored for diverse use cases.

In this project, the language model is accessed primarily through the `ChatOpenAI` class provided by the `langchain_openai` module.
This wrapper provides a high-level interface to OpenAI's chat-based models, such as GPT-4, and extends their functionality with features that are crucial for building robust applications. Among these are support for streaming responses, the ability to generate structured outputs, and seamless integration with LangChain's memory management and tool-calling components.

Listing 4.1: LLM initialization using ChatOpenAI

```
from langchain_openai import ChatOpenAI
import os

llm = ChatOpenAI(
    model="gpt-4o",
    temperature=0,
    max_tokens=512,
    api_key=os.getenv("OPENAI_API_KEY"),
    timeout=30,
    retry_settings={"max_retries": 3}
)
```

The constructor supports various parameters:

- **`model`**: Defines the name of the model to be used, such as `"gpt-4o"`, `"gpt-4"` or `"gpt-3.5-turbo"`. This parameter determines the underlying capabilities, cost, and performance characteristics of the system.

- **`temperature`**: Regulates the degree of randomness in the model's responses. Lower values (e.g., 0.0-0.3) make the output more deterministic and reproducible, while higher values (e.g., 0.7-1.0) encourage greater creativity and variability.

- **`max_tokens`**: Sets a limit for the number of tokens that can appear in the generated output. This parameter is useful for controlling verbosity, ensuring concise responses, and preventing excessive resource usage.

- **`api_key`**: Provides the authentication key required to access OpenAI's API.

- **`timeout`**: Defines the maximum waiting time (in seconds) for a response before the request is aborted. This prevents the client from hanging indefinitely in case of network issues or slow responses.

- **`retry_settings`**: Configures the retry policy when errors occur, such as network failures or rate-limit responses. Parameters may include the maximum number of retries and backoff strategies.

Chat models such as GPT-4 process inputs as a sequence of structured messages, where each message is associated with a predefined role.
This design is aligned with OpenAI's API and supports multi-turn, stateful interactions by clearly distinguishing between different types of contributions in the conversation. The main roles are:

- **system**: Defines high-level instructions and behavior guidelines for the model, typically used to establish context, enforce constraints, or specify the assistant's persona and objectives.
- **user**: Represents the human input, usually expressed in natural language. These messages drive the conversation by providing questions, commands, or clarifications.
- **assistant**: Contains the model's responses. Beyond delivering outputs to the user, these messages also serve to preserve conversational state, allowing the model to maintain continuity across multiple turns.

Listing 4.2: Basic message exchange with ChatOpenAI

```
from langchain_core.messages import SystemMessage, HumanMessage

messages = [
    SystemMessage(content="You are an AWS automation assistant."),
    HumanMessage(content="Create a Lambda function that writes to an S3
    bucket.")
]

response = llm.invoke(messages)
```

This message-based architecture enables precise control over conversational context and enables the model to reason effectively across multiple turns. Such capabilities are particularly valuable in scenarios that require long-horizon planning or the retention of agent memory.

One of the most powerful features of LangChain is its support for **tool-augmented reasoning**. In this paradigm, a language model is not limited to text generation alone but can also interact with external tools, APIs or services. Rather than relying solely on its internal knowledge, the model can dynamically decide which tool to invoke based on the user query. The tool's output is then integrated back into the reasoning process, enabling the model to act as an active problem solver rather than a passive text generator.

LangChain agents often implement reasoning strategies such as **ReAct** (Reasoning + Acting), which combine step-by-step reasoning with tool invocation in an iterative loop. In this approach, the model alternates between thoughts (internal reasoning) and actions (tool calls), using the results of each action as new observations for the next step. The cycle continues until a final answer is produced. Concretely, the process can be described as follows:

1. Interprets the user query and reason about the next step (*Thought*).
2. Chooses an appropriate tool if needed (*Action*).
3. Constructs a valid input and execute the tool.
4. Receive and analyze the tool's output (*Observation*).
5. Repeat the reasoning-acting cycle as necessary until the problem is solved.
6. Deliver the final response to the user (*Final Answer*).

LangChain provides the `create_react_agent()` helper function, which simplifies the construction of agents based on the ReAct paradigm.
This function binds together a language model, a set of tools, and an optional memory backend into a fully operational agent.

Listing 4.3: Creating a ReAct agent with LangChain

```
from langgraph.prebuilt import create_react_agent
from langchain_tavily import TavilySearch
from langgraph.checkpoint.memory import MemorySaver

tools = [TavilySearch()]
memory = MemorySaver()

agent_executor = create_react_agent(llm, tools, checkpointer=memory)
```

Once instantiated, the agent can handle multi-turn dialogues, stream intermediate reasoning steps in real time, and resume prior sessions using thread identifiers. This makes it possible to preserve context between different user interactions.

Listing 4.4: Streaming a tool-augmented agent interaction

```
input_message = {
    "role": "user",
    "content": "Search for the weather in Rome."
}
config = {
    "configurable": {"thread_id": "user-session-123"}
}

for step in agent_executor.stream(
    {"messages": [input_message]},
    config,
    stream_mode="values"
):
    step["messages"][-1].print()
```

4.1.2 Pydantic

Figure 4.2: Pydantic Logo.

To ensure type safety, input validation, and structured output parsing, the system employs **Pydantic**, a Python library for data modeling and schema validation.. Schemas are defined as subclasses of `BaseModel`, where each class attribute corresponds to an expected field. At runtime, Pydantic automatically validates any data instance against the declared schema, raising errors when the input does not conform..

In addition to type annotations, the `Field` function allows developers to enrich model definitions with metadata, such as descriptions, default values, and constraints (e.g., minimum or maximum length).

LangChain natively seamlessly with Pydantic through the `with_structured_output()` method. This mechanism enables the LLM to return output directly in a structured format., eliminating the need for manual string parsing or fragile regex extraction. As a result, outputs can be consumed immediately as typed Python objects.

In practice, the developer begins by defining a schema as a Pydantic class that specifies the fields required for a given task.
In practice, the developer first defines a schema as a Pydantic class specifying the fields required for a task (e.g., an `Action` object). The language model is then instantiated (e.g., via `ChatOpenAI`) and wrapped with `llm.with_structured_output(schema=Action)`. When the model produces a response, the system automatically parses it into the corresponding Pydantic object, ensuring that the output is validated, interpretable, and directly usable for downstream execution.

Listing 4.5: Example of a structured schema for AWS actions

```
from pydantic import BaseModel, Field
from typing import Dict, Any
from langchain_openai import ChatOpenAI

class Action(BaseModel): resource: str = Field(..., description="The
    type of AWS resource to interact with (e.g., 's3', 'ec2').")
    operation: str = Field(..., description="The specific operation to
    perform on the resource (e.g., 'create', 'delete').") parameters:
    Dict[str, Any] = Field(..., description="Optional parameters required
     for the operation, provided as key-value pairs.")

llm = ChatOpenAI(model="gpt-4", temperature=0)

structured_llm = llm.with_structured_output(schema=Action)
```

Using Pydantic in this way provides several key advantages in an LLM-based agent system.
First, it improves robustness and safety: malformed or incomplete outputs raise validation errors instead of silently propagating through the pipeline.
Second, it enhances clarity and maintainability, since the expected structure of agent actions is explicitly defined in code, making the system easier to read, validate, and extend.
Third, the approach supports composability, as different action schemas can be declared for different resource types (e.g., `LambdaAction`, `DynamoDBAction`), depending on the planning context.
Finally, once validation succeeds, the resulting object can be safely executed, which is particularly important when interfacing with external systems such as AWS, where reliability and correctness are critical.
By combining schema definition, runtime validation, and structured output parsing, Pydantic ensures that every step of the orchestration pipeline remains interpretable, predictable, and trustworthy. This tight integration not only reduces implementation complexity but also strengthens the overall reliability of the agent framework.

4.1.3 Boto3

Figure 4.3: Boto3 Logo.

All interactions with AWS are executed through **Boto3**, the official AWS SDK for Python. In the proposed multi-agent system, Boto3 serves as the primary interface for cloud operations, enabling agents to programmatically interact with a wide range of AWS resources. It is used not only for direct service operations but also as the execution backend for the tools integrated within the LangChain agent framework.

To promote modularity and maintainability, the system organizes AWS service interactions into a dedicated `aws_actions` package. Each service is encapsulated in its own Python module, exposing functions that wrap Boto3 client calls with additional logic for error handling and interactive fallback. These service-specific modules are not accessed directly by the user; instead, their invocation is coordinated by a central executor agent, which selects and executes the appropriate operations based on the user's request.

At the core of every interaction lies the concept of a **client**. A Boto3 client is a low-level object that provides programmatic access to the operations of a specific AWS service, closely mirroring its API. Clients are created by instantiating a session with the necessary AWS credentials and region configuration, thereby establishing a secure and authenticated connection to the service.

Listing 4.6: Initializing AWS clients using Boto3 and environment variables

```
import boto3
import os

session = boto3.Session(
    aws_access_key_id=os.getenv("AWS_ACCESS_KEY_ID"),
    aws_secret_access_key=os.getenv("AWS_SECRET_ACCESS_KEY"),
    region_name=os.getenv("AWS_REGION"),
)
s3_client = session.client("s3")
lambda_client = session.client("lambda")
...

s3_client.upload_file("lambda.zip", "my_bucket", "lambda.zip")
```

Each Boto3 client provides a dedicated interface to a specific AWS service. It exposes the full set of API operations for that service while abstracting away low-level concerns such as authentication, request signing, retries, and HTTP communication.
Error handling is also standardized through structured exceptions (e.g., `ClientError`), which allows failures to be managed in a consistent predictable way.

Within this system, clients for services such as S3, Lambda, IAM, EventBridge, DynamoDB, EC2, and Cost Explorer are instantiated once at runtime and stored in a dictionary. This design ensures that service connections are efficiently reused, while still allowing tool functions to dynamically access the appropriate client depending on the type of task being executed.

4.2 Tools Integration

Code Generation

The system can automatically generate executable Python code for AWS Lambda functions starting from high-level natural language prompts.
The code generation logic is implemented in `handler.py`, where the user's textual description of the desired logic is passed to a large language model such as GPT-4o. The model responds with valid Python code, which is automatically extracted and saved into a `.py` file (e.g., `lambda.py`). The file is then automatically compressed into a `.zip` archive and stored locally, making it immediately ready for deployment. At a later stage, this archive is consumed by the executor, which uses the Boto3 SDK to deploy the Lambda function in AWS.
Although the generated code is not tested or executed directly within the system

itself, the toolchain ensures that the output is correctly structured, syntactically valid, and aligned with AWS deployment requirements.

Operational Tracking

The system exposes its runtime behavior through structured exceptions and uniform logging. When interacting with AWS services, errors are returned as typed exceptions such as `ClientError`, making it easy to trace a logged action back to the exact API call and condition that triggered the failure. Since each log entry includes both the action name and its parameters, developers can readily reconstruct the request, reproduce the issue, and apply corrective measures or rollbacks with minimal effort.

Cost Tracking (AWS)

To monitor infrastructure spending, the system integrates with AWS Cost Explorer, retrieving detailed usage data over a specified time window. Produces a clear breakdown of costs per service and per day, while omitting zero-cost entries to keep the report focused. This functionality provides immediate operational insight, helping validate the financial impact of automated deployments and highlighting unusual or unexpected spending patterns.

Cost Tracking (LLM Usage)

In addition to AWS costs, the system also monitors the usage of language model itself, keeping track of both token consumption and the associated expenses for each request. The text exchanged with the model is tokenized using tiktoken according to the selected architecture, and the number of input and output tokens is multiplied by per-thousand pricing defined in a local price table.
To make monitoring practical, the system includes utility functions that print concise summaries of usage and estimated costs on demand. Additionally, the system provides a mechanism to reset the token counter manually, allowing users to start a new cost tracking cycle at any time.

Automated Documentation

Every test or user interaction performed through the system is documented automatically in both JSON and Markdown formats.
For each test session, the system records the test name, the original user prompt, the full sequence of executed actions, the extracted parameters with their values and the execution results.It also logs information about language model usage,

including token consumption and estimated costs. These metrics are captured both at the granularity of individual actions and at the level of the entire session, giving users full visibility over expenses and resource consumption.

4.3 Error Handling and Recovery Mechanisms

Error handling in the system is designed to ensure robustness, fault isolation, and guided recovery in the face of common AWS failures and input-related issues. Rather than failing silently or halting entirely, the system incorporates mechanisms to detect, log and respond to errors in a structured and interactive way.
Each AWS action implements its own error-handling logic. These mechanisms are defined at the tool level and rely on structured exception management using `botocore.exceptions.ClientError`.

For instance, the S3 bucket creation tool is able to distinguish between different error conditions. If a bucket is already owned by the user (`BucketAlreadyOwnedByYou`), the action is marked as complete and execution continues without interruption. Conversely, if the bucket name is already taken globally (`BucketAlreadyExists`), the system resets only that parameter and prompts the user to provide a new name. In the event of unexpected errors, the failure is reported clearly, logged in detail, and the workflow terminates gracefully rather than crashing.

A similar strategy is applied in the Lambda creation tool, where specific error patterns, such as invalid or unusable IAM role ARNs, incorrect handler naming, malformed or empty deployment packages, or conflicts with existing function names, are intercepted (`InvalidParameterValueException`).
Each of these conditions triggers a tailored recovery path: the system resets only the affected parameter (e.g., `role_arn` or the `zip_path`) and requests clarification from the user instead of restarting the entire process. This localized recovery minimizes disruption during multi-step executions and preserves continuity across the pipeline.

Whenever possible, the system attempts to recover from errors by leveraging the conversational memory context. If a parameter cannot be inferred, the user is prompted interactively to provide a valid value. This fallback behavior provides resilience against partial failures and ensures that task execution continues with minimal interruption, while feedback is surfaced immediately through real-time console output.

To further improve robustness, retry mechanisms are applied at the action level. Since each task is validated in isolation through its corresponding Pydantic schema

and execution state is preserved between modules, failed actions can be retried independently without re-running earlier steps.
Temporary issues such as throttling or rate limiting can also trigger time-delayed retries (for example, in response to a `TooManyRequestsException`), allowing the system to recover automatically from transient errors.

Before any tool is invoked, input parameters are validated against a strongly typed Pydantic schema. This upfront validation prevents malformed requests and allows the system to catch missing or incorrect values early reducing the risk of cascading failures.
In addition, every user session generates a complete execution trace that captures both successful operations and failed attempts.
These logs are recorded in a structured format that supports both human readability and automated analysis.

To support both human readability and automated processing, logs are exported in two complementary formats: a Markdown report that summarizes the session in a concise and accessible way, and a JSON file that records the same information in a structured format suitable for auditing, debugging, or further analysis.

Chapter 5

Evaluation and Testing

This chapter presents the evaluation framework used to test the system across a wide range of cloud-related tasks. The testing phase aimed to validate the system's ability to correctly interpret natural language instructions, extract parameters, select appropriate actions, and execute them via AWS services.
The evaluation focused on both functional correctness and robustness against incomplete or ambiguous input.

5.1 Benchmark Design

A total of 24 test cases were designed and implemented to rigorously evaluate the system's ability to interpret and execute natural language prompts.
The benchmark suite was constructed with the goal of maximizing coverage across typical AWS use cases, as well as testing edge cases that simulate incomplete, ambiguous, or multi-action user requests.
Each test corresponds to a distinct scenario designed to mirror real-world cloud infrastructure requirements.
The test suite covers a variety of AWS services, grouped into the following categories:

- **Lambda Functions**: Tests in this category validate the system's ability to create Lambda functions, generate Python handler code, zip and deploy it. Some tests intentionally omit parameters (e.g., function name, IAM role) to evaluate the system's capability to infer missing details from under-specified prompts.
- **S3 Buckets**: Tests in this category validate the system's ability to create and configure S3 buckets for storage operations. Several tests omit parameters (e.g., bucket name) to assess whether the system can correctly recover or generate them when not explicitly provided.

- **API Gateway Integration**: Tests in this category validate the system's ability to expose Lambda functions through HTTP endpoints. They verify the correct configuration of routes, methods (GET, POST), and integration settings. Each test concludes with an endpoint invocation to ensure that status codes, headers, and payloads match the expected results.
- **Triggers (EventBridge)**: Tests in this category evaluate the system's ability to schedule Lambda executions using AWS EventBridge rules. They focus on interpreting temporal instructions (e.g., "every day at 8 AM") and correctly associate them with the appropriate function.
- **IAM Role and Policy Management**: Tests in this category assess whether the system can create IAM roles, attach inline policies, and assign these roles to services. They also examinate role reuse, policy scope, and naming consistency, particularly when prompts are underspecified.
- **DynamoDB Operations**: Tests in this category cover a complete CRUD (Create, Read, Update, Delete) workflow on a sample table. Each operation is executed independently, verifying parameter extraction, data type correctness, and error handling in database interactions.
- **EC2 Operations**: Tests in this category validate system's ability to manage EC2 instances, including launching, starting, stopping, terminating, and checking status. Prompts provide varying levels of detail, requiring the system to infer sensible defaults or flag missing parameters.
- **Multi-action workflows**: Tests in this category assess the system's ability to handle prompts that require multiple operations within the same pipeline (e.g., creating two Lambda functions with shared IAM roles, wiring permissions, and uploading files to S3 buckets). They evaluate how well the system can decompose, plan, and orchestrate sequential actions.
- **Contextual memory tests**: Tests in this category evaluate the system's short-term memory. For example, a prompt may create a bucket, while the subsequent prompt refers to "the previously created bucket." The system must correctly retrieve and apply this contextual information.
- **Minimal prompt handling**: Tests in this category evaluate the system's behavior under vague or partial instructions. They assess its ability to handle under-specification, apply fallbacks, trigger clarification logic, or gracefully fail when necessary.

Each test is executed through the `run_test()` method of the evaluation module. This method logs the input prompt, resulting actions, timestamp, and token usage.

5.2 Evaluation Methodology

The evaluation methodology was designed to systematically assess the system's ability to interpret natural language input, reason over cloud-related tasks, and execute them effectively. Each test followed a standardized pipeline designed to reflect the operational flow of the multi-agent system.
The procedure can be outlined into the following key stages:

1. **Prompt Execution**: Natural language instructions are submitted to the system via the test harness. Prompts range from simple single-action commands to complex, multi-step workflows.

2. **Structured Parameter Extraction**: Leveraging predefined Pydantic schemas tailored to each AWS service, the system parses the prompt and extracts all relevant parameters. This step ensures both syntactic and semantic validation before execution.

3. **Action Execution and Validation**: Once parsing is successful, the system invokes the appropriate AWS service through the Boto3 SDK. Each action is monitored to confirm correct completions, with exceptions captured for debugging and analysis.

4. **Result Aggregation**: For every test, detailed metadata is recorded, including the original prompt, extracted parameters, execution outcome (success/failure), LLM token usage, and timestamp. This information supports both quantitative benchmarking and qualitative inspection.

5. **Documentation and Reporting**: All test results are stored in both JSON and Markdown formats. This dual-format approach enables automatic report generation while also ensuring human-readable documentation for reproducibility and traceability.

For robustness testing, several experiments deliberately omitted essential parameters to evaluate the system's fallback strategies and error handling capabilities. In addition, some prompts combined multiple actions to test sequencing, orchestration, and context management.

5.3 Results and Analysis

The evaluation demonstrated the system's ability to accurately interpret and execute a diverse set of cloud-related directly from natural language instructions. Overall, the results highlights not only the breadth of supported AWS functionality but also the robustness of the underlying reasoning and execution framework.

Table 5.1: Overall distribution of success, partial success, and failure outcomes across 24 tests

Result Type	Number of Tests	Total Percentage (%)
Fully Successful	21	87.5%
Partial Success	1	4.2%
Complete Failure	2	8.3%

Table 5.2: Performance results by AWS service category

Category	Tests	Success Rate (%)	Failures
Lambda Functions	4	100%	0
S3 Buckets	2	100%	0
API Gateway	3	67%	1
EventBridge	2	100%	0
IAM Management	3	100%	0
DynamoDB Operations	4	100%	0
EC2 Operations	3	100%	0
Multi-action flows	2	50%	1
Contextual memory	1	100%	0

Note: The last column reports only complete failures.
Cases of partial success are excluded from this summary and are discussed in the main text.

Successful Cases

Out of 24 tests, 21 completed fully successfully (87.5%).
Lambda tests consistently managed code generation, handler packaging, and deployment without errors. The system also demonstrated reliability in creating IAM roles, attaching inline policies, and assigning them to Lambda functions.
S3 buckets operations were equally robust, handling both new bucket creation and interactions with existing buckets, including storange and retrieval tasks.
DynamoDB operations covered the entire CRUD spectrum with correct parsing of primary keys, accurate type handling, and precise key-based updates and deletions.
EC2 workflows (launch, stop, terminate, status) executed as expected whenever valid identifiers were provided.
In more complex, multi-step scenarios, the system correctly decomposed prompts into ordered actions.
Contextual memory tests further confirmed that conversational state is reused

effectively.
Even under minimal prompts, execution succeeded after the system interactively collected missing parameters. This fallback mechanisms prevented unsafe assumptions while ensuring correctness by explicitly validating all required inputs.

Failure Cases

Two complete failures (8.3%) and one partial success (4.2%) were observed.
The first complete failure occurred in the multi-action flow scenario. While the Lambda functions were created successfully, the API Gateway integration failed with a `ResourceNotFoundException`.
The second complete failure occurred also in the multi-action flow scenario. Here, the ambiguous wording led to invalid IAM policy names and a misinterpreted S3 write, preventing end-to-end execution.
The partial success was recorded in a chained DynamoDB workflow. The table was created successfully, but item insertion was skipped after being flagged as a duplicate.

Efficiency and Robustness

Table 5.3: Avg efficiency and robustness metrics calculated over 24 tests

Metric	Value
Tokens per prompt	1,545
Execution time per test	22.1 s
Orchestration overhead	48.4%
Cost per test	$0.00037

Note: Orchestration overhead refers to the share of time spent by the agent in planning, prompt interpretation, disambiguation, and conversational context management, rather than in the direct execution of AWS API calls. Cost values vary depending on the LLM used; in this evaluation, all tests were executed with GPT-4o.

The table reports aggregate efficiency and robustness metrics from the 24-test benchmark.
On average, each test consumed 1,545 tokens per prompt, accounting for both inputs and model outputs. Token usage increases naturally with multi-step workflows, clarification turns, and verbose tool instructions.
The mean end-to-end runtime was 22.1 s, measured wall-clock from orchestration start to the final assertion. This duration includes all stages of execution, so not

only AWS API calls, but also planning, parameter validation, logging, and LLM round-trip latency.

A notable result is the 48.4% orchestration overhead. Roughly half of the total runtime was spent on reasoning and coordination rather than direct APIs execution. This overhead reflects activities such as plan construction, contextual retrieval, disambiguation of underspecified inputs, and retry/backoff handling. These results point to clear opportunities for latency reduction, including prompt streamlining, more structured I/O to limit token exchanges, and batching or parallelizing independent AWS operations where feasible.

Finally, the average LLM cost per test was $0.00037 (GPT-4o), which is negligible compared with typical cloud charges for provisioning and data transfer. Reported costs cover only LLM usage and naturally vary with model choice, network conditions, and AWS service load. The orchestration overhead figure refers specifically to *time*, not cost.

LLM Pricing Considerations

An additional dimension of the evaluation concerns the choice of the underlying LLM model. While the experiments were primarily conducted using GPT-4o, alternative models offer different trade-offs between accuracy, latency, and cost. Table 5.4 reports the official pricing (per million tokens) at the time of writing for several OpenAI models, distinguishing between input and output tokens.

Table 5.4: Comparison of LLM pricing per 1M tokens (USD)

Model	Input Price ($)	Output Price ($)
GPT-4o	0.0050	0.0200
GPT-4o-mini	0.0006	0.0024
GPT-4.1	0.0030	0.0120
GPT-4.1-mini	0.0008	0.0032
GPT-4.1-nano	0.0002	0.0008
o4-mini	0.0040	0.0160

On average, each benchmark test consumed approximately 1,545 tokens (input + output), corresponding to negligible costs even for the most expensive models. For example, executing one full benchmark run of 24 tests with GPT-4o amounts to less than $0.01 in LLM usage, economically insignificant when compared with typical AWS resources charges. Nevertheless, the table also indicate that lighter

models (e.g., GPT-4.1-nano or GPT-4o-mini) can reduce costs by up to one order of magnitude. While this difference is minor at small scale, it becomes relevant when executing thousands of automated prompts per day in enterprise deployments.
Consequently, model selection should balance cost efficiency against reasoning accuracy. Smaller models are well suited for routine orchestration tasks with predictable patterns, whereas more advanced models remain preferable for complex, ambiguous, or high-stakes scenarios where accuracy justifies the higher token expediture.

5.4 Discussion and Insights

The benchmark evaluation highlights the strengths and limitations of the proposed multi-agent system for cloud orchestration. Overall, the system demonstrated strong functional accuracy, adaptability, and interpretability in transforming natural language instructions into concrete, executable cloud operations.
Several important insights emerged from the testing phase:

- **Modularity and Extensibility**: The modular architecture, which separates prompt interpretation, parameter extraction, and execution, enables seamless integration of new AWS services and reasoning strategies without affecting existing functionality. This design supports both vertical scalability (additional services) and horizontal scalability (more reasoning strategies), ensuring maintainability over time.

- **Importance of structured prompting**: The use of structured intermediate schemas (e.g., Pydantic classes) proved critical for validating parameters prior to execution. This approach reduces the risk of errors, hallucinations, or misinterpretations. Structured prompting also enhances traceability and debugging, as every decision point in the workflow is explicitly defined.

- **Effective Short-Term Memory**: Tests confirmed that the system is capable of maintaining short-term memory across consecutive prompts. This allows the agent to reference previous operations without requiring redundant information from the user. Such context retention improves user experience and emulates natural interaction patterns, where users expect the system to "remember" prior steps within a session.

- **Moderate LLM Usage**: Even with complex of many multi-step prompts, token usage remained moderate throughout the benchmark. This is attributable to the system's ability to reuse context, avoid verbose completions, and rely on tool-based execution rather than generating long textual outputs. Efficient

token consumption ensures the system remains cost-effective when scaled in production settings.

- **Value of Edge Cases**: Stress tests with missing parameters, ambiguous prompts, or conflicting actions, were instrumental in uncovering design limitations and improving system robustness. In particular, minimal prompts exposed areas where fallback mechanisms were insufficient, suggesting the need for improved clarification routines or interactive disambiguation. These tests emphasize that robust NLP-driven systems must be evaluated not only on typical scenarios, but also on ill-formed or incomplete instructions that closely resemble real-world usage.

Several directions for improvement emerge from these findings.
A first direction would be to introduce **transactional undo and rollback mechanisms**, allowing failed actions to be automatically compensated preserving infrastructure consistency.
In parallel **adaptive retry strategies** could differentiate between transient errors (e.g., throttling or network delays) and permanent ones (e.g., invalid parameters), applying exponential backoff or idempotent re-execution when appropriate.

Another promising avenue is **continuous benchmarking**. So running the full extendable test suite regularly in a sandbox environment while at the same time providing long-term visibility into operational costs and efficiency.

Looking further ahead, the introduction of **persistent memory** would open the way to multi-session continuity and collaborative workflows. Such a memory could also be queried in a RAG way, allowing the system to reuse past user prompts and their outcomes, re-executing actions automatically if previous attempts had failed.

Finally, **advanced visualization tools** could make the system more transparent and user-friendly. Rich interactive dashboards, coupled with graphical representations of workflows and error traces, would strengthen auditability, compliance, and user trust.

Chapter 6

Conclusion and Future Work

This thesis project has presented the design and implementation of a multi-agent system capable of orchestrating AWS cloud infrastructure directly from natural language input.
The core objective was to allow users to launch and manage AWS services through textual prompts alone, without directly interacting with the AWS Management Console.
This objective was successfully achieved with the development of a fully functional prototype that integrates Large Language Models (LLMs) with a set of modular tools built on the Boto3 SDK.

The evaluation phase, based on 24 carefully designed test cases, confirmed the system's effectiveness in interpreting, validating, and executing a broad range of cloud-related tasks. Strong performance was observed not only in typical use cases but also in complex multi-action workflows.
Furthermore, the system demonstrated resilience when handling under-specified prompts, leveraging defaults, contextual memory, and interactive fallback mechanisms to ensure continuity and correctness.
Overall, the results show that intelligent agents powered by LLMs can serve as a practical and reliable interface for cloud automation, bridging the gap between high-level natural language instructions and low-level infrastructure deployment.

6.1 Limitations

While the results are encouraging, several limitations emerged during the project that should be acknowledged.

- **Prompt Sensitivity**: Parameter extraction remains strongly dependent on the clarity and specificity of user prompts. Vague or ambiguous instructions,

may lead to misinterpretations or invalid parameters generation.

- **Error Handling**: Although basic fallback mechanisms and retry strategies were implemented, the system lacks more advanced recovery capabilities such as automated rollback in the event of partial or failed deployments.
- **Scope Restriction**: The prototype currently supports only AWS services. Despite the modularity of the architecture, extending support to other cloud providers (e.g., Azure or Google Cloud) would require additional development and validation effort.
- **Evaluation Scale**: The evaluation relied on 24 test scenarios designed for functional coverage. Broader testing with larger prompt sets and diverse user groups would be necessary to fully assess scalability, robustness, and real-world applicability.

6.2 Future Work

Building on this foundation, several promising directions can be explored to extend both the practical applicability and the research contributions of this work:

- **Multi-cloud support**: Extending the tool layer beyond AWS to include providers such as Microsoft Azure and Google Cloud Platform would broaden applicability and enables comparative studies of orchestration strategies across different environments.
- **Graphical user interface with multimodal interaction**: A user-friendly front-end could make the system accessible to a wider audience by complementing natural language with visual infrastructure diagrams and interactive dashboards. Enabling multimodal interaction through voice commands, sketches, or hybrid methods, would further enhance accessibility for both technical and non-technical users.
- **Rollback, recovery and validation**: Introducing undo mechanisms and automated rollback for failed operations would improve reliability and prevent inconsistent infrastructure states.
- **Long-term memory and context management**: While short-term memory was effective, persistent memory across sessions could support more complex multi-step projects and continuity in collaborative workflows.
- **Integration with DevOps pipelines**: Connecting the system to CI/CD tools would embed natural language orchestration into established software

engineering workflows, bridging conversational interfaces with automated infrastructure delivery.

- **Cost optimization and monitoring**: Integrating agents capable of analyzing and suggesting cost-efficient alternatives would help align infrastructure automation with financial constraints.
- **Security and compliance**: Automatic enforcement of IAM best practices, role management policies, and regulatory compliance checks would strengthen system trustworthiness.
- **Research opportunities**: Academically, the system could serve to benchmark the robustness of LLM-based agents in structured orchestration tasks, explore human–agent collaboration in DevOps workflows, and explore explainability in multi-agent reasoning pipelines.

In summary, this thesis demonstrated the feasibility of using LLM-driven agents as intelligent orchestrators for cloud infrastructure.
By addressing current limitations and pursuing the proposed extensions, future work could evolve this prototype into a production-ready platform while contributing to ongoing research in AI-assisted software engineering.

Appendix A

Test

Test 1 — Lambda Function + Bucket S3

Involved services: Lambda, IAM, S3

Prompt

Create a Lambda function named `LambdaTest1` using IAM role `arn:aws:iam::849107555072:role/AdminLambda` and write `"Test 1 OK"` to `test1.txt` in the existing bucket `bucket-test1-uno`.

Result

SUCCESS.
Handler generated, Lambda created and executed, file correctly written in the S3 bucket.

Test 2 — Lambda Function + API Gateway

Involved services: Lambda, IAM, API Gateway

Prompt

Create a Lambda function named `LambdaTest2` using IAM role `arn:aws:iam::849107555072:role/AdminLambda`, return `"Test 2 OK"`, and expose it via API Gateway at path `/test2` with method `GET`.

Result

SUCCESS.
Lambda created and deployed. API Gateway endpoint exposed successfully.

Test 3 — Lambda Function + Bucket S3 + Scheduled Trigger

Involved services: Lambda, IAM, EventBridge, S3

Prompt

Create a Lambda function named `LambdaTest3` using IAM role `arn:aws:iam::849107555072:role/AdminLambda`; trigger it every day at 08:00 and write `"Test 3 OK"` to `test3.txt` in existing bucket `bucket-test3-tre`.

Result

SUCCESS.
Lambda created and scheduled via EventBridge. Daily trigger runs as expected.

Test 4 — Lambda Function + Bucket S3 + Trigger + API Gateway

Involved services: Lambda, IAM, EventBridge, API Gateway, S3

Prompt

Create a Lambda function named `LambdaTest4` using IAM role `arn:aws:iam::849107555072:role/AdminLambda`. Every day at 08:00 write `"Test 4 OK"` to `test4.txt` in existing bucket `bucket-test4-quattro`. Expose the function via API Gateway at `/test4` with method `GET`.

Result

SUCCESS.
Bucket accessed, handler generated, Lambda created, trigger configured, and API Gateway exposed.

Test 5 — DynamoDB — Create Table

Involved services: DynamoDB

Prompt

Create a DynamoDB table named `Utenti` with primary key `user_id` of type `String`.

Result

SUCCESS.
Table `Utenti` created with PAY_PER_REQUEST billing mode.

Test 6 — DynamoDB — Put Item

Involved services: DynamoDB

Prompt

Insert a new item into table `Utenti` with `user_id = "123"` and `name = "Mario"`.

Result

SUCCESS.
Item inserted correctly.

Test 7 — DynamoDB — Get Item

Involved services: DynamoDB

Prompt

Retrieve the item from table `Utenti` where `user_id = "123"`.

Result

SUCCESS.
Item retrieved successfully.

Test 8 — DynamoDB — Update Item

Involved services: DynamoDB

Prompt

Update the item in table `Utenti` with `user_id = "123"` by adding `stato = "attivo"`.

Result

SUCCESS.
Item updated successfully.

Test 9 — DynamoDB — Delete Item

Involved services: DynamoDB

Prompt

Delete the item from table `Utenti` where `user_id = "123"`.

Result

Result: SUCCESS. Item deleted successfully.

Test 10 — S3 — Create Bucket

Involved services: S3

Prompt

Create a new S3 bucket named `bucket-test10-dieci`.

Result

SUCCESS.
Bucket created successfully.

Test 11 — EC2 — Launch Instance

Involved services: EC2, IAM, Security Group

Prompt

Launch an EC2 instance with `AMI ID = ami-0022eedc61881ccdc`, `type = t2.micro`, `ssh-key = chiavi-sergio`, `security-group = allow_ssh`.

Result

Result: SUCCESS. EC2 instance launched correctly.

Test 12 — EC2 — Start Instance

Involved services: EC2

Prompt

Start the EC2 instance with `ID = i-092f6bda6f9f310fa`.

Result

FAILURE.
InvalidInstanceID.NotFound — instance not found.

Test 13 — EC2 — Stop Instance

Involved services: EC2

Prompt

Stop the EC2 instance with `ID = i-092f6bda6f9f310fa`.

Result

SUCCESS.
EC2 instance stopped successfully.

Test 14 — EC2 — Terminate Instance

Involved services: EC2

Prompt

Terminate the EC2 instance with `ID = i-092f6bda6f9f310fa`.

Result

SUCCESS.
EC2 instance terminated successfully.

Test 15 — EC2 — Check Status

Involved services: EC2

Prompt

Check the status of the EC2 instance with `ID = i-092f6bda6f9f310fa`.

Result

SUCCESS.
Status retrieved successfully.

Test 16 — IAM Role + S3 Policy + Lambda

Involved services: IAM, Lambda, S3

Prompt

Create an IAM role named `RoleTest16` for a Lambda function. Add a write policy for an S3 bucket to the same role and assign it to function `LambdaTest16`.

Result

SUCCESS.
IAM role created, S3 policy attached, role assigned to Lambda.

Test 17 — IAM Policy for Lambda Role

Involved services: IAM

Prompt

Add an inline policy granting full access to an S3 bucket to IAM role `RoleTest16`.

Result

SUCCESS.
Inline policy attached successfully.

Test 18 — Multi-Actions — Two Lambda Function writing to bucket S3

Involved services: Lambda, IAM, S3

Prompt

Create two Lambda functions:
– `LambdaTest18-Uno` writes `"Message 1"` to `multi1.txt` in bucket `bucket-test18-diciotto`
– `LambdaTest18-Due` writes `"Message 2"` to `multi2.txt` in the same bucket.
Both use IAM role `arn:aws:iam::849107555072:role/AdminLambda`.

Result

SUCCESS.
Both Lambdas created and able to write to the S3 bucket.

Test 19 — Multi-Actions — Two Lambda Function exposed via API Gateway

Involved services: Lambda, IAM, API Gateway

Prompt

Create two Lambda functions:
– `LambdaTest19-Uno`, exposed via API Gateway at `/log1` with method `GET`, returns `"Log1"`
– `LambdaTest19-Due`, exposed via API Gateway at `/log2` with method `GET`, returns `"Log2"`.
Both use IAM role `arn:aws:iam::849107555072:role/AdminLambda`.

Result

FAILURE.
Both Lambdas created, but API Gateway exposure failed (ResourceNotFoundException: function not found).

Test 20 — Contextual Memory — Create Bucket S3

Involved services: S3

Prompt

Create a bucket named `bucket-test20-venti`.

Result

SUCCESS.
Bucket created successfully.

Test 21 — Contextual Memory — Lambda uses previous Bucket S3

Involved services: Lambda, IAM, S3

Prompt

Create a function named `LambdaTest20` using IAM role `arn:aws:iam::849107555072:role/AdminLambda` that writes `"Test 20 OK"` to `test20.txt` in the bucket created in Test 20.

Result

SUCCESS.
Lambda created and writing to S3 bucket.

Test 22 — Chained Actions — DynamoDB create and insert

Involved services: DynamoDB

Prompt

Create a DynamoDB table `LogTable` with `partition_key = ID` of type `String`, then insert an item with `ID = "log1"` and `name = "Francesco"`.

Result

PARTIAL SUCCESS.
Table created but item insertion failed ("Action already executed").

Test 23 — Minimal Prompt — Create Bucket S3

Involved services: S3

Prompt

Create a bucket named `bucket-test23-ventitre`.

Result

SUCCESS.
Bucket created successfully.

Test 24 — Minimal Prompt — Lambda Function with Daily Trigger

Involved services: Lambda, EventBridge

Prompt

Create a Lambda function and connect it to a trigger that runs every day at 10:00.

Result

SUCCESS.
Lambda created and scheduled daily trigger at 10:00 configured.

Bibliography

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. «Attention Is All You Need». In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2017, pp. 5998–6008. URL: https://arxiv.org/abs/1706.03762 (cit. on p. 3).

[2] Zhenfeng Fan, Jinhao Duan, Yawen Zeng, Weiming Lu, and Xiangnan He. «Large Language Models as Generalist Agents». In: *arXiv preprint arXiv:2309.14365* (2023). URL: https://arxiv.org/abs/2309.14365 (cit. on p. 6).

[3] Lei Wang et al. «A Survey on Large Language Model based Autonomous Agents». In: *Frontiers of Computer Science* (2024). DOI: 10.1007/s11704-024-40231-1. URL: https://doi.org/10.1007/s11704-024-40231-1 (cit. on p. 7).

[4] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. «A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications». In: *arXiv preprint arXiv:2402.07927* (2024). URL: https://arxiv.org/abs/2402.07927 (cit. on p. 12).

[5] Yifan Yu et al. «Communication Strategies in Multi-Agent Systems with Large Language Models». In: *arXiv preprint arXiv:2502.14321* (2025). URL: https://arxiv.org/abs/2502.14321 (cit. on pp. 14, 15).

[6] Xiang Xie, Boyang Li, Chuanqi Zhang, Xin Xia, Zhenchang Xing, David Lo, and Ahmed E. Hassan. «Software Engineering meets Large Language Model Agents: A Survey». In: *arXiv preprint arXiv:2408.02479* (2024). URL: https://arxiv.org/abs/2408.02479 (cit. on p. 20).