Università degli Studi di Firenze
School of Engineering - Department of Information Engineering

Bachelor's Thesis in Computer Engineering

Designing Self-Aware Multi-Agent AI Systems: A Two-Fold Framework Based on AIBOM and Reflective Architecture

Candidate
Niccolò Caselli

Supervisor
Prof. Enrico Vicario

Co-supervisor
Marco Becattini

Academic Year 2024/2025

Contents

Abstract

The increasing complexity of modern AI systems—especially those based on large language models (LLMs) and multi-agent architectures—demands new methodologies to ensure system-level reliability, traceability, and adaptability. Existing tools offer limited visibility into software and knowledge dependencies, leaving a gap in accountable and maintainable cognitive workflows. This thesis addresses that gap by proposing a two-fold framework combining the Reflection architectural pattern with an extended notion of the Software Bill of Materials (SBOM), adapted for AI systems as the Artificial Intelligence Bill of Materials (AIBOM). This novel integration—largely unexplored in current literature—enables runtime adaptability and structured traceability. The architecture features a knowledge layer managing workflow meta-models and an operational layer for task execution. Reflection supports semantic interoperability across heterogeneous components whose interactions are not predefined. A use case in AI for Network Engineering (AI4NE) and Network Engineering for AI (NE4AI) demonstrates how cognitive workflows dynamically route requests across cloud resources based on evolving constraints (e.g., latency, energy efficiency, and computational cost). This work opens several research directions and lays the groundwork for further investigation into structured multi-agent architectures and their alignment with forthcoming AI governance regulations.

Chapter 1

Introduction and Background

1.1 Introduction

The advent of Large Language Models (LLMs) marks a significant advancement in cutting-edge artificial intelligence systems, demonstrating remarkable performance across a wide range of tasks. Nevertheless, deploying a single LLM-based agent for complex, real-world applications exposes several well-known limitations. These include the lack of long-term memory, difficulties in tailoring model behavior to specific sub-tasks, and a restricted capacity to ground outputs in verifiable data sources. Consequently, a growing shift is observed in both research and industry towards multi-agent systems, wherein agents are assembled, each optimized for distinct tasks and characterized by specific configurations (e.g., varying knowledge bases, hyperparameter tuning, training data, resource management, etc.). This evolution, exemplified by recent frameworks such as SALLMA [3], allows for the design of more robust, scalable, and adaptable systems that better accommodate the diverse requirements of contemporary use cases.

The paradigm of multi-agent architectures gives rise to what we term a

cognitive workflow[1]: a structured and coordinated process in which intelligent agents interact to manage context-aware decisions, integrate external tools, and adapt to evolving objectives and data. Unlike traditional workflows, cognitive workflows are designed to support runtime reasoning and reconfiguration, making them particularly powerful, while also introducing a considerable degree of complexity.

As the number of agents, tools, data sources, and configurations grows, so too does the difficulty of ensuring reliability, portability, and maintainability[2]. Notably, tracing the dependencies, configurations, and interactions of each agent becomes essential for establishing trust in the behavior of the overall system. To tackle these challenges, we propose applying principles from software supply chain management—particularly, the concept of the Software Bill of Materials (SBOM)—to multi-agent AI systems. This contributes to the emerging trend of the AI Bill of Materials (AIBOM), presented in Section 1.2. Furthermore, the Reflection architectural pattern is employed to promote self-awareness and dynamic reconfiguration, establishing a structured method for trust and maintainability in AI-driven workflows. When applied to cognitive workflows, reflection enhances interoperability across agents and facilitates runtime coordination aligned with evolving system goals.

The remainder of this chapter is organized as follows: Section 1.2 reviews the historical background of SBOM and the emerging concept of AIBOM, along with relevant regulatory considerations. Section 1.3 examines the cur-

[1]Although the term cognitive workflow has been used across various domains, its application in this context is limited. We argue that it best captures the organization of semantic workflows that require context adaptation and intelligent decision making, a view supported by the recent literature.

[2]These qualities—reliability, portability, and maintainability—are classified as primary software quality characteristics by the `ISO/IEC 25010` standard.

rent state of SBOM, its adoption in AI, and the open challenges. Chapter 2 presents the proposed architectural framework and its implementation. Ultimately, Chapter 3 demonstrates a relevant use case in the fields of *AI for Network Engineering* (AI4NE) and *Network Engineering for AI* (NE4AI), while Chapter 4 discusses results, limitations, and future research directions.

1.2 Historical Context

Building on SBOM foundations, this section examines the evolution of SBOMs into responsible AI tools amid an expanding regulatory landscape.

1.2.1 Software Bill of Materials

The concept of a Software Bill of Materials (SBOM) has emerged in response to the growing need for traceable and secure software, along with its underlying dependencies. The U.S. National Telecommunications and Information Administration (NTIA) defines the SBOM as a "nested inventory for software, a list of ingredients that make up software components" [18]. That is to say, the SBOM acts as the software counterpart of the Bill of Materials (BOM), a well-established concept in the industrial and manufacturing sectors. An SBOM provides a structured inventory of all software libraries, modules, and dependencies that make up a given system. Therefore, this detailed record enables the parties to address critical security concerns, such as identifying which modules rely on a specific library version (e.g., "Which services use library X, version 1.2?") to mitigate vulnerabilities. Although the concept of an SBOM is not new, its adoption has been hampered by challenges related to integration into existing workflows, as well as the lack of standardization [7]. However, SBOM's popularity has grown significantly following the issuance of an Executive Order [4] by the White House under President Joe Biden in May 2021. This directive, part of a broader initiative to enhance national cybersecurity, mandates that all federal agencies produce SBOMs for their software components. Taking into account the increasing regulatory requirements, it is reasonable to assume that SBOM techniques adoption will rise as more corporations become aware of the associated benefits (See Section 1.3 for statistics on SBOM adoption and usage).

1.2.2 Extending SBOM to AI: AIBOM

In recent years, the number of software applications and open source projects that rely on machine learning (ML) and artificial intelligence (AI) components has exponentially increased. As a result, SBOM techniques have begun to be experimented with within the context of AI systems. The concept of the AI Bill of Materials (AIBOM) is still emerging and has not yet been extensively documented or covered in the literature. Additionally, discussions on the topic have been limited by a restricted group of researchers.

However, this scenario is likely to evolve, as recent developments have significantly impacted the sector: on 13 March 2024, the European Parliament voted in favor of the *AI Act*, which has entered into force on 1 August 2024. This act is the first comprehensive regulation of Artificial Intelligence within the European Union. It features a four-category classification of AI systems: unacceptable risk (e.g., social scoring systems), which are prohibited, high-risk AI systems, limited-risk AI systems, and minimal-risk AI systems, which remain unregulated. Within the context of this research, a key aspect of the act is that providers of General Purpose AI (GPAI) models, including foundation models like OpenAI's GPT, are mandated to provide technical documentation and instructions for use, comply with the Copyright Directive, and publish a summary of the training data sources. Several points of the regulation stress the importance of dataset quality and relevance, ensuring transparency, and providing information to deployers. Additionally, recital 71 emphasizes that the technical documentation should be updated regularly throughout the lifetime of the system and that "high-risk AI systems should technically allow for the automatic recording of events, by means of logs, over the duration of the lifetime of the system" [8].

Beyond software security, this focus on dataset quality and transparency highlights the ethical challenges associated with ML systems, particularly in ensuring fairness and reliability, as well as avoiding biased outcomes. For example, a widely cited case study [24] involved Google in 2015 when software engineer Jacky Alciné drew attention to a major flaw in Google Photos' image recognition algorithm: it had been classifying black people as "gorillas". The error stemmed from multiple factors, most notably the incompleteness of the training data, which lacked sufficient diversity and failed to represent the broader population. Two years later, the press pointed out that the issue had yet to be addressed.

Lastly, while the *AI Act* is the first-ever legal framework on AI, it is worth noting that on 8 December 2020, the United States issued Executive Order 13960 [9], titled *Promoting the Use of Trustworthy Artificial Intelligence in the Federal Government*, through the Department of the Treasury. Although less prescriptive than the *AI Act*, this document emphasizes that federal agencies must design, develop, acquire, and use AI in a manner that fosters public trust. Furthermore, Section 3 designates *Principles for the Use of AI in Government*, highlighting the importance of transparency, accountability, reliability, safety, and resilience as fundamental requirements.

In conclusion, it is evident that the rapidly growing international regulatory landscape highlights the rapid evolution of AI governance and underscores the pressing need for innovative approaches to ensure legal compliance and the dependability of systems.

1.2.3 Literature Review

Although a review of the literature reveals that the term AIBOM is not entirely novel[3], it gained prominence through the work of Australian researchers, who have introduced it as an emerging concept and have been recurrent contributors to this area of study. Notably, Xia et al. [25] highlight the distinction between SBOMs for conventional software and those tailored for AI systems. Meanwhile, in *Trust in Software Supply Chains: Blockchain-Enabled SBOM and the AIBOM Future* [26], Xia et al. propose a blockchain-based architecture that leverages Verifiable Credentials (VCs) and smart contracts to foster trust and overcome the issues associated with SBOM sharing and accountability. These are fundamental steps towards AIBOM as the authors state "the advent of AI and its application in critical infrastructure necessitates a nuanced understanding of AI software components, including their origins and interdependencies". Moreover, the idea of exploiting verifiable credentials to improve the traceability of ML systems is not unique to this work, and is also found in other studies, such as Barclay et al. [1]. This further underscores the importance of establishing trust in data originators.

One of the most comprehensive insights on the topic is provided by Stalnaker et al [20], who conducted interviews with 138 practitioners from five stakeholder groups, including professionals with AI/ML backgrounds, and

[3] To the best of our knowledge, the term AIBOM was first used by Brian Ka Chan in 2017.

highlighted key challenges and emerging opportunities in SBOM usage. According to the authors, AIBOMs have the potential to improve the reproducibility of machine learning models and facilitate dataset verification across academic studies. By providing transparency on model training—detailing aspects such as architecture, hyperparameters, and the use of pre-trained components—they help ensure accountability. Additionally, the concept of DataBOM[4], first introduced by Barclay et al. [2], is explored in its relationship with AIBOM as it can assist developers in detecting whether a model was trained on biased or unethically sourced datasets, as well as in preventing Data Poisoning attacks[5]. Nevertheless, when the interviewed stakeholders were asked whether the two documents should be merged into a single one, less than 10% agreed on their integration.

Lastly, Lu et al. [16] have studied system-level design patterns to foster the engineering of what they define as *responsible-AI-by-design*. Among the solutions presented, notable measures include the Bill of Materials, Ethical Knowledge, Ethical Sandbox, and Ethical Twin. The latter can act as an operational infrastructure component that facilitates the monitoring of an AI system's runtime behavior and decision-making through an abstract simulation model that uses real-world data. Additionally, the necessity of an Ethical Sandbox is highlighted as a strategy to separate AI components from non-AI components by running them in an isolated, self-contained environment.

[4] Data Bill of Materials

[5] An adversarial attack where malicious data is injected into the training set to manipulate or degrade the model's behavior.

1.3 State of the Art

This section examines the current state of SBOM documentation in AI systems, with a focus on industry trends, key challenges, and existing gaps related to its integration into multi-agent workflows

1.3.1 Current Trends in Industry Adoption

In January 2022, the Linux Foundation released the *SBOM Readiness Report* [15], surveying 412 organizations worldwide. The report highlights that 98% of the respondents are concerned about software security, making it the first organizational priority. Additionally, 82% of organizations are familiar with the concept of SBOM, 90% have started their SBOM journey, while 76% are actively addressing SBOM needs. Although the Linux Foundation projects an increase in SBOM usage, significant challenges persist.

A more recent study by Xia et al. [25] underlines that the adoption of SBOM is not as optimistic, identifying three key aspects of SBOM's state of practice. Based on data from 17 interviews and 65 survey responses, the study reveals that at least 83.1% of surveyed organizations do not receive SBOMs along with third-party open-source or proprietary software, thereby complicating SBOM integration with external components and limiting its uptake among software vendors.

Moreover, the misalignment between anticipations and reality regarding SBOM readiness is further highlighted by Kloeg et al.'s work [14], which states that 69% of the interviewed stakeholders observed minimal demand from their clients or did not express demand to their suppliers for SBOM.

An up-to-date, comprehensive analysis of the challenges is provided by Dalia, Di Sorbo, and Canfora [7] who identified ten challenges hindering

SBOM adoption, including the lack of a unique standard. Several tools—including CycloneDX, SPDX, and Syft—are presented in a comparative examination according to different criteria, concluding that the enabling technologies have not yet achieved maturity and full automation.

1.3.2 State of SBOM Integration in AI

Concerning the integration of SBOM tools into AI and ML contexts, Stalnaker et al. [20] assert that standards and support for such tools are nearly absent in these specific sectors. Moreover, their research reveals that 85% of the interviewees with a Machine Learning background were unfamiliar with any tool support for generating AIBOMs, while 90% were unaware of tooling for DataBOMs. Nevertheless, the study provides valuable insights. Notably, it reports that CycloneDX (starting from version 1.5) has added a Machine Learning Bill of Materials to its specification [6]. Furthermore, it draws attention to the Dataset Cards [11] of Hugging Face[6] which, according to two interviewed participants, could serve as a form of DataBOM. Dataset Cards are documents that store metadata on datasets, including data source, format, and possible bias. Furthermore, it should be noted that Hugging Face supports Model Cards [12], which simplify the documentation of model architecture, training and validation datasets, and evaluation metrics. The theoretical foundation for Model Cards was already established by Mitchell et al. [17] in 2019, who introduced this framework as a fundamental step towards responsible democratization of machine learning, aiming to standardize ethical practices and ensure transparent model reporting. Their work proposed a well-defined schema for Model Cards, outlining key fields and

[6]Hugging Face is a leading AI platform that serves as a centralized repository and package manager for AI models and datasets. See: https://huggingface.co.

emphasizing their complementarity with dataset documentation paradigms, such as *Datasheets for Datasets* [10].

1.3.3 Unresolved Gaps in Existing Work

Although the adoption of SBOM techniques in traditional software engineering is expected to increase, substantial gaps persist in the literature and industry practice regarding their application to AI systems. A significant limitation lies in the absence of a structured methodology for managing *cognitive workflows* in multi-agent systems. Existing research primarily discusses SBOMs in a general ML context, leveraging model cards [17] or DataBOMs [2] to address ethical considerations and provide documentation of dataset sources, or blockchain-based methods [1, 26] to foster accountability. However, these approaches do not address the unique challenges posed by multi-agent architectures, where multiple AI components interact dynamically in distributed environments. Additionally, while the concept of AI Bill of Materials (AIBOM) is emerging, its development has, to date, remained theoretical, with no concrete strategies for its integration into AI agent orchestration or demonstrations of how SBOM data can be effectively harnessed in practice within the system, such as for querying an agent's dependencies. To the best of our knowledge, only the already cited work by Lu et al. [16] has focused on system-level design patterns within the lifecycle of a provisioned AI system.

Furthermore, the dynamic nature of these workflows—where agents may self-adapt, reconfigure, or exchange knowledge—presents additional complexity that is not accounted for in traditional frameworks. Active use of SBOM information for governance and maintenance within the system remains unexplored despite the increasing regulatory requirements for regular system monitoring and documentation updates [8, 9], as highlighted in Section 1.2.

In summary, two critical gaps persist in the literature. First, there is a **lack of system-level design patterns** for multi-agent workflows that can accommodate the complex interdependencies and dynamic interactions between autonomous agents. Second, there is an absence of **comprehensive AI traceability tools** that ensure provenance, accountability, and transparency across the entire system lifecycle, from initial deployment to ongoing adaptation and knowledge exchange.

Chapter 2

A Two-Fold Framework for AIBOM in Cognitive Workflows

This chapter presents our proposed methodology for incorporating SBOM-driven approaches (extended to the AIBOM scope) within a multi-agent cognitive workflow. The chapter is structured as follows: Section 2.1 outlines the conceptual design and meta-level orchestration strategies that form the foundation of our approach; Section 2.2 introduces the SALLMA architecture, a multi-agent system whose principles are employed in our proposal; finally, Section 2.3 details the design and implementation of the proposed system, providing concrete examples of how the theoretical framework translates into a working solution.

Supporting materials, including comprehensive UML diagrams and our testing strategy, are provided in Appendix A.

2.1 Our Proposal

This study proposes a two-fold framework designed to address the critical gaps previously identified in the literature (see Section 1.3.3). It specifically targets the persistent challenges of reliability, structural adaptability, and comprehensive traceability within AI-driven multi-agent systems. The framework comprises two distinct aspects that together underpin the proposed approach.

The first aspect involves a dedicated design pattern adapted from the Reflection Architectural Pattern (see Section 2.1.1), tailored for agent-based systems. This pattern, defined as a means of "changing the structure and behavior of software systems dynamically" [5]—is used to enable self-awareness and run-time adaptability.

The second aspect addresses traceability by extending standard Software Bill of Materials (SBOM) practices to include AI-specific artifacts, thereby formalizing the Artificial Intelligence Bill of Materials (AIBOM). The AIBOM provides a structured record of models, datasets, hyperparameters, prompts, tools, and agent configurations, supporting transparency and accountability across the AI system lifecycle.

2.1.1 The Reflection Architectural Pattern

We argue that leveraging the Reflection Pattern offers clear benefits for the AIBOM. To support this claim, we first provide an overview of the pattern and its key mechanisms.

The Reflection architectural pattern enables software systems to dynamically adapt their structure and behavior at runtime. As described by Buschmann et al. [5], it achieves this by dividing the system into two levels: a **meta-level**, which holds *metaobjects* encapsulating information about system properties (such as type structures or communication mechanisms), and a **base level**, which implements the core application logic. The key idea is that changes to metaobjects, via a well-defined **metaobject protocol** (MOP) that ensures consistent modifications, directly influence the behavior of the base level without altering its source code. Consequently, the implementation of the base level uses the meta-objects to remain independent of all aspects prone to alteration.

An overview of the general structure of a reflective architecture is illustrated in Figure 2.1, which highlights the layered interaction. This separation supports system evolution, flexibility, and the ability to respond to changing requirements.

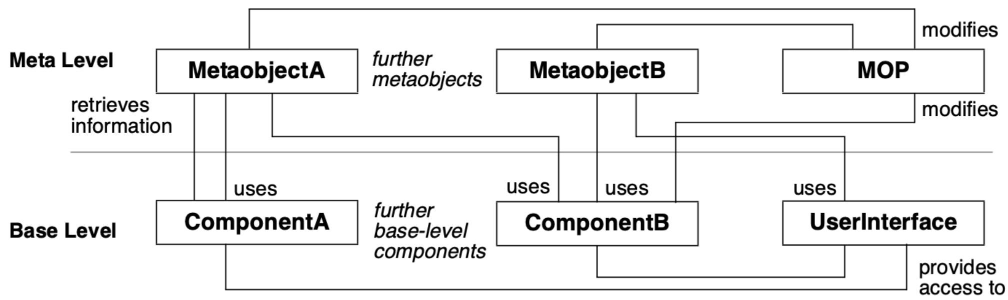

Figure 2.1: Architectural overview of a reflective system, illustrating the meta and base levels and their interactions. From *Pattern-Oriented Software Architecture*, Buschmann et al. [5].

In the context of this study, the presented pattern is fundamental as it allows for the dynamic management of a *repository of meta-models*, without the need to define in advance the components with which the agents or the system will interact. As a result, agent interactions are dynamically modeled, leading to a structured cognitive workflow. In summary, a reflective approach yields the following key benefits:

- **Dynamic reconfiguration**: The system can adapt its behavior and structure at runtime based on changes to its meta-level, without requiring redeployment.
- **Separation of concerns**: By decoupling the application logic (Base Level) from the Knowledge Layer (Meta Level)—which is subject to more frequent changes—maintainability and extensibility are significantly improved.
- **Semantic interoperability**: Components with no prior knowledge of each other can interact seamlessly. This enables the integration of heterogeneous agents into cohesive workflows, allowing the composition of complex and adaptive interactions.
- **Explainability and traceability**: By observing the system's components and behaviors, the meta-level establishes a foundation for a self-aware and auditable architecture. This allows the system to track its composition, explain the rationale for the decision, and trace the results.

2.1.2 Integration of the Two-Fold Framework

The novel and primary contribution of this work lies in integrating the Reflection Pattern, which provides structure and adaptability, with the AIBOM formalism, which ensures traceability. By coupling the comprehensive AIBOM inventory with the Reflection pattern, each cognitive workflow gains visibility into:

- **Its own structural elements** — including agent composition, model versions, training data references, and external libraries.

– **Its meta-level** — a supervisory layer that monitors changes, performs reconfiguration, and enforces security or compliance rules.

– **Runtime adaptations** — triggered by environment changes, updates to agent dependencies, or new user requirements, all logged and documented within the AIBOM to maintain continuous traceability.

This integration of reflection and AIBOM serves as the foundation for a self-aware and self-adapting multi-agent system. The architecture we propose includes:

– **Core AI Agents**: Responsible for distinct tasks (e.g., user intent detection, route planning, resource allocation).

– **Meta-Level Controller**: Dynamically adjusts agent configurations by referencing the AIBOM, ensuring only approved/compatible components are used.

– **AIBOM Repository**: Storing SBOM-like records for each AI component and its lifecycle events.

By adopting this layered approach, we aim to demonstrate how cognitive workflows can be more traceable, transparent, and compliant with emerging AI regulations, especially where multi-agent collaboration requires detailed knowledge of each component's provenance and behavioral characteristics.

2.2 LLM-based System of Agents

AIBOM can be a valuable tool for managing complex systems composed of LLM agents by introducing the level of transparency and traceability typically associated with Software Bills of Materials. As foundation models rapidly transform the methodologies used to develop and deploy software-intensive systems, emerging approaches must embrace this shift by adopting novel architectural designs that address distributed, real-world scenarios rather than relying on conventional centralized solutions. In this context, the present study builds upon the work of Becattini, Verdecchia, and Vicario at the Software Technologies Laboratory, University of Florence [3], extending their proposed **SALLMA** architecture.

2.2.1 SALLMA Architecture Overview

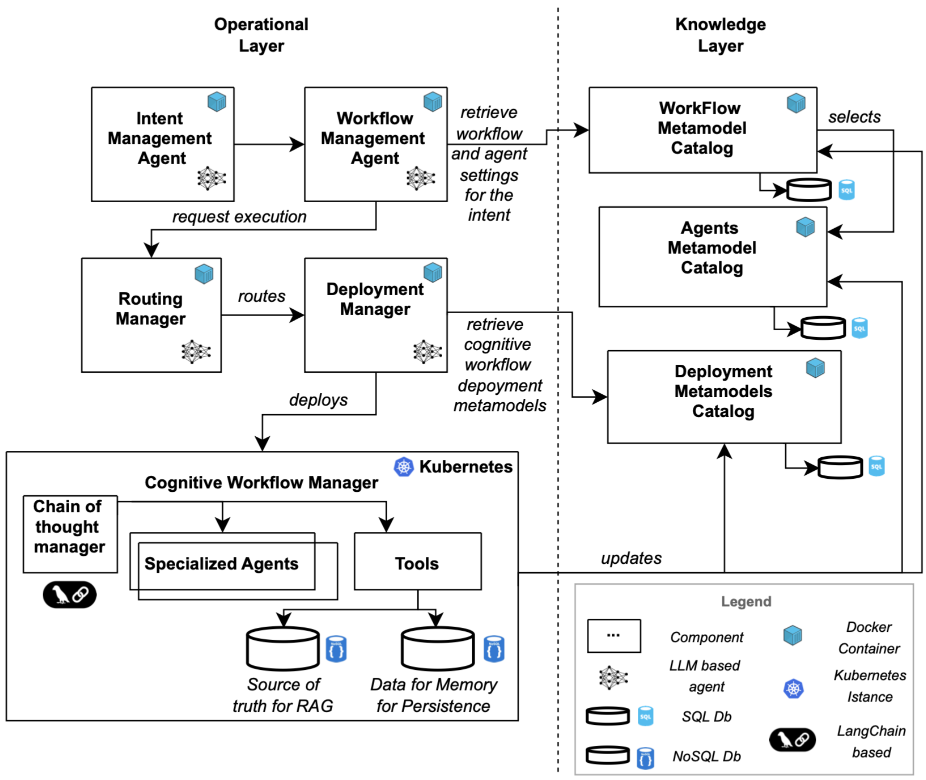

Figure 2.2: SALLMA Architecture Overview

An introduction to the concept of SALLMA (Software Architecture for LLM-Based Multi-Agent Systems) will first be provided. SALLMA is a distributed, multi-agent architecture featuring an Operational Layer for real-time task execution and a Knowledge Layer for workflow and agent configuration meta-model management. Engineered from first principles, it is inherently modular and adaptive, designed to orchestrate multiple large language model (LLM) agents across both cloud and edge environments. While primarily a conceptual architecture, a proof of concept has been implemented to evaluate its functional suitability.

By introducing a dual-layer structure, SALLMA addresses the limitations inherent in single-agent LLM systems, enabling dynamic adaptability of the system to the diversity of tasks in real-world applications. Notable limitations commonly observed in single-agent approaches include:

– Suboptimal hyperparameter configurations for specific requests.

– Lack of persistent memory.

– Limited access to ground-truth data.

– Challenges in reliability and sustainability management.

Compared to a single monolithic AI agent, an ensemble of specialized agents enables dynamic task decomposition and organic specialization: each agent can focus on its strengths, and tasks can be distributed dynamically across the system. Furthermore, this approach introduces fault tolerance: if one agent fails, others can continue execution, resulting in greater robustness and reliability.

A comprehensive overview of the SALLMA architecture is illustrated in Figure 2.2, highlighting the distinction between its two core layers. A brief explanation of each layer follows.

SALLMA Operational Layer

The Operational Layer acts as the dynamic core that is responsible for handling real-time interactions and orchestrating task-specific agents. Essentially, it is where the LLM agents "live" and interact, and is thus responsible for the decision-making process of the workflow. The primary elements of the Operational Layer are:

- **Intent Management Agent**: Parses the request and sets up the system to retrieve workflows using predefined intents stored in the Knowledge Layer.
- **Workflow Management Agent**: Breaks tasks into subtasks, assigning them to specialized agents.
- **Routing Manager**: Determines whether an existing workflow instance can handle the request or a new instance is required.
- **Deployment Manager**: Deploys workflows by leveraging the configurations and resources outlined in the Knowledge Layer.
- **Cognitive Workflow Manager**: Orchestrates tasks within a containerized environment, leveraging specialized agents, a chain-of-thought process, persistent memory for context-aware execution, and foundational data for reliable retrieval-augmented generation (RAG).

SALLMA Knowledge Layer

The Knowledge Layer is dedicated to storing and managing information that agents use or produce. It serves as the foundation for SALLMA's metamodel management, maintaining an inventory of reusable workflows and agent configurations. Agents in the Operational Layer rely on the Knowledge

Layer both to retrieve relevant data and to update it with new findings or changes in state. Among its components, the most notable include:

- **Workflow Metamodel Catalog**: Stores predefined workflow configurations optimized for different tasks.
- **LLM Configuration Catalog**: Contains predefined configurations for each LLM agent (e.g., hyperparameters, resource allocations, and so on).
- **Deployment Metamodels Catalog**: Holds a set of deployment models that detail the proper cognitive workflow configurations for each operational scenario (e.g., cloud or edge environments).

2.2.2 Extending SALLMA

The proposed framework integrates SBOM management into the SALLMA multi-agent system architecture. By extending SALLMA, we introduce a systematic way to document and track the composition of each agent within the Knowledge Layer and to use that data in the Operations Layer for better decision-making and security inspections. In contrast to traditional practices that rely on the static generation of SBOMs and their passive use for tracking dependencies, this novel approach marks a paradigm shift toward dynamically leveraging SBOM knowledge within the system across its entire lifecycle. The key idea is that as the multi-agent system evolves, it will generate and maintain an SBOM that describes all its software components (agents, models, libraries, and dependencies); this self-describing inventory of components information will be actively used by the system for governance and maintenance, enhancing trust in what the AI is executing.

A limitation of the approach outlined in SALLMA is the lack of a clear formalization regarding information ownership, specifically whether it should reside within the Knowledge Layer or be delegated to external systems and accessed via retrieval augmented generation (RAG). In this context, an AI-BOM approach could be beneficial, as it allows a more proper and elegant differentiation between the knowledge layer—where meta-models of the cognitive workflows are stored—and the data level, where data belongs.

2.3 System's Design and Implementation

The following section details the architecture and core components of the proposed system in its design and final implementation. Building on the conceptual foundations introduced earlier, it outlines the structural patterns, persistence strategies, execution mechanisms, and dynamic adaptation features that enable the orchestration of cognitive workflows. Every design choice is motivated by the need for extensibility, modularity, and runtime flexibility in managing AI agents and their meta-structures. The demonstration implementation is carried out in Java, leveraging the emerging Spring AI framework from the Spring ecosystem to support AI engineering tasks (see Appendix B for a detailed explanation of this choice). As a proof of concept, although SALLMA was conceived as a distributed architecture, the concrete implementation proposed to support the study is based on a single application where workflow nodes are instantiated in memory rather than deployed as separate services. This simplification maintains the core concepts while allowing for comprehensive validation of the system's logic and interaction patterns.

2.3.1 Pattern Outline

As previously introduced, the proposed architecture adheres to the SALLMA philosophy, adopting a dual-layered structure comprising the Knowledge Layer and the Operational Layer. Leveraging the Reflection Pattern alongside aggregation and subclassing, the architecture achieves a high degree of flexibility. The Operational Layer hosts the execution-time entities: instance nodes and workflows; each instance is linked to a corresponding meta-model, which belongs to the Knowledge Layer and governs its actual runtime behavior. Across both layers, as illustrated in Figure 2.3, specialized nodes encapsulate different AI models and external tools (e.g., LLMs, Embedding Models, RESTful Services, Vector Databases, etc.).

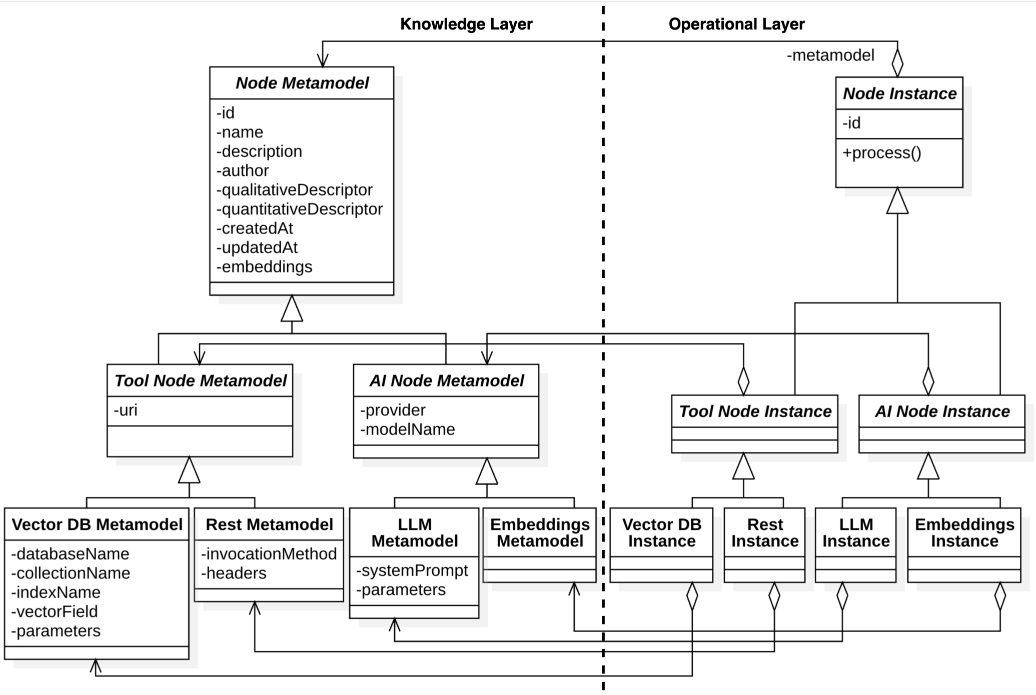

Figure 2.3: Dual-layer architecture diagram showing Knowledge Layer meta-models (left) and their corresponding Operational Layer instances (right), illustrating the inheritance relationships between components.

The nodes can be combined into workflows, which are essentially *directed acyclic graphs* (DAGs). Each workflow is represented as a graph consisting of nodes and edges, where nodes correspond to specific meta-models with their associated versions, and edges define the connections between nodes within the workflow structure. Ideally, multiple workflows can be composed together, allowing simple workflows to be reused and integrated into more complex processing pipelines. For example, the Retrieval-Augmented Generation[1] (RAG) pattern can be modeled as a sub-workflow consisting of the following nodes:

1. **Embedding node**: Calls an embedding model to generate a vector representation of the text input.

2. **Vector database node**: Performs a vector similarity search against a pre-indexed data repository.

3. **LLM node**: Receives the original query along with the retrieved documents and generates a final response.

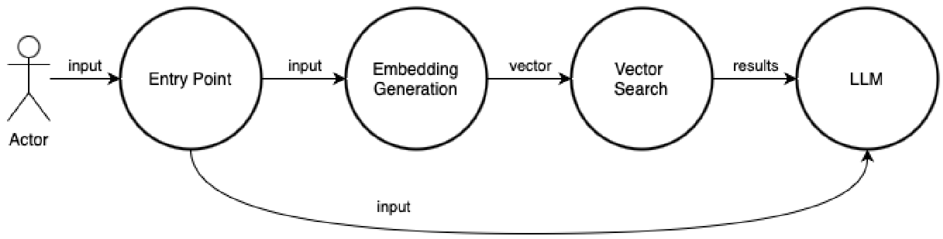

Figure 2.4: Diagram of the RAG pattern workflow corresponding to the embedding, retrieval, and generation steps.

[1]RAG is an AI engineering technique used to retrieve information from an external knowledge base in order to ground LLMs on accurate and up-to-date data.

Data Persistence

A MongoDB database was selected for the meta-models repository due to its NoSQL, document-based approach. This choice is ideal for schema flexibility, especially when dealing with the evolving and heterogeneous data common in multi-agent configurations (e.g., varying parameters, node types, and data formats). Its nested structure also provides efficient storage for SBOMs. Finally, the out-of-the-box Vector Search capability provided by Atlas[2] was key in enabling semantic search for RAG, without requiring additional dedicated databases. The system uses three collections to manage:

- The meta-models of individual nodes
- The meta-modes of workflows
- The catalog of intents handled by the system

Although inheritance in entity modeling can add complexity to data persistence, the combination of `Jackson`'s polymorphic deserialization and `Spring Data MongoDB` effectively abstracts away this complexity. When a subclass instance is persisted, Spring automatically includes a `_class` field in the document, which contains the fully qualified class name. This allows the framework to correctly deserialize the document and resolve the appropriate subclass during *Object-Document Mapping* (ODM).

2.3.2 Structured Data Flow

In workflow-based systems, controlling the flow of data between components is essential to ensure consistency and interoperability throughout the

[2] `https://www.mongodb.com/atlas` - a fully managed Database-as-a-Service (DBaaS) provided by MongoDB.

execution. In the context of this work, we utilize the notion of ports as an abstraction to declare the input and output interfaces of each node within the workflow. This construct promotes modular design, enables validation during data propagation, and assists in the early detection of incompatibilities. To enforce the structural correctness of meta-models, dedicated services are employed to verify the compatibility of ports across connected nodes, ensure type consistency, and validate the satisfaction of all required inputs.

Input and Output Ports

Ports act as standardized entry and exit points for data within a node. A port is identified by a unique key and is defined by a schema that describes the expected data type and structure (e.g., primitive types, objects, arrays), supporting rich and flexible data modeling. Additionally, the port abstraction supports various specialization types which encode domain-specific semantics. For instance, in nodes representing RESTful services, ports can correspond to request body fields, query parameters, path variables, etc. In contrast, LLM-based nodes might define ports in terms of system prompt variables or user messages. An overview of the class structure used to define ports and their schema is provided in Figure 2.5.

LLM Structured Outputs

When working with Large Language Models (LLMs), generating structured outputs is critical to ensure predictable consumption of the model's responses. Spring AI addresses this need through its **Structured Output API** [21], which includes a set of **Structured Output Converters**. These converters transform raw models' output into instances of predefined Java

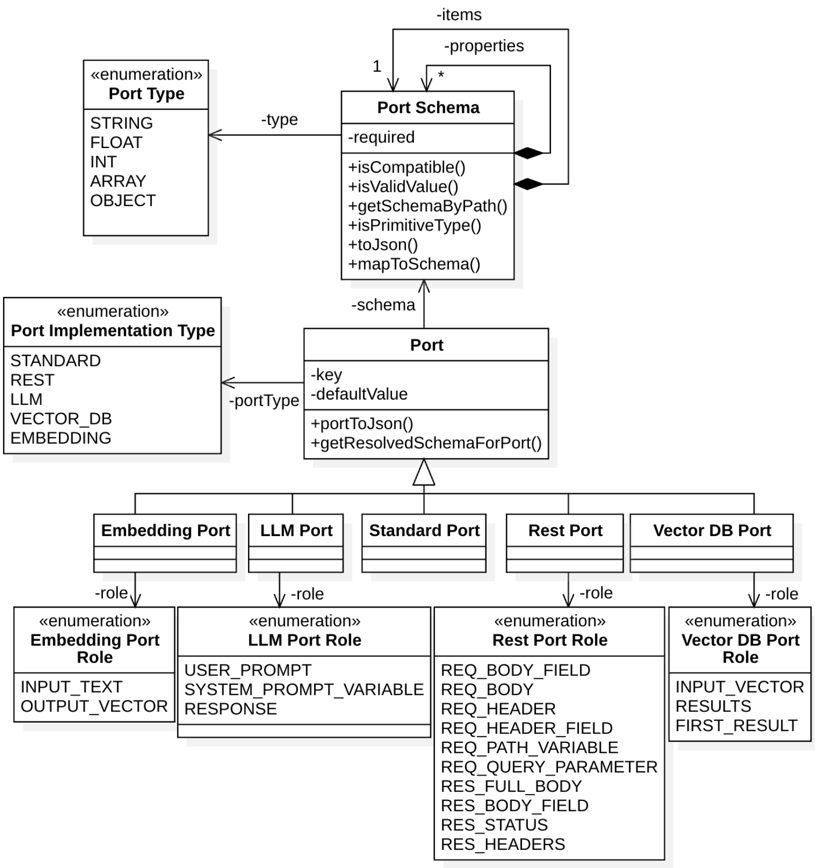

Figure 2.5: UML Class Diagram illustrating the *Port* and *Port Schema* entities, which define the structure and behavior of inputs and outputs for workflows' nodes. For brevity, the figure omits that each port implementation is associated with a corresponding builder class.

classes, leveraging prompt engineering[3] to guide the model's formatting behavior. Essentially, as shown in Figure 2.6, instructions are embedded in the prompts to shape the output according to the expected structure, following which the converter parses the response into a Java object. Nevertheless, as meta-model's ports are dynamically typed and their schema is more expressive than what Spring AI supports natively, a custom converter layer

[3]The process of designing inputs to effectively guide generative AI models toward desired outputs.

was developed. This layer, built on top of the one of Spring AI, also leverages prompt augmentation[4] to map the LLM responses to the corresponding output port schema declared in the meta-model.

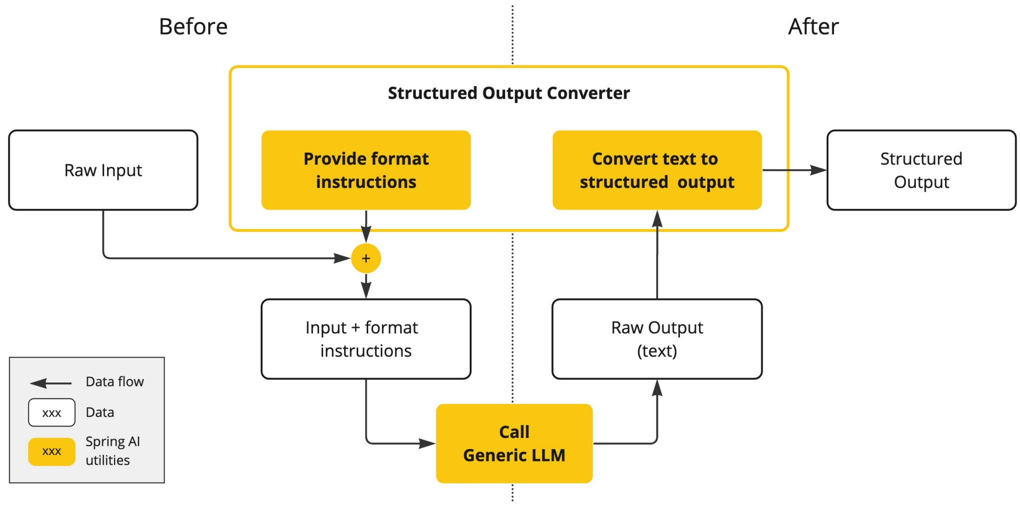

Figure 2.6: Overview of Spring AI's Structured Output Converters. From `https://spring.io/blog/2024/05/09/spring-ai-structured-output`

Edge Bindings

A central concept to facilitate the collaboration of nodes within workflows is the idea of bindings. These specify how the output of one node connects to the input of the following node. This is particularly important in heterogeneous systems where different nodes might use different naming conventions or data formats for interface ports. Bindings operate in two modes:

- **Explicit Bindings**: These are direct mappings between input and output port names. As aliases, they allow ports with different names to be connected explicitly.

[4] A prompt engineering technique that enriches a base input prompt by adding context, examples, or instructions to improve the relevance, accuracy, or specificity of the AI's response.

– **Implicit Matching**: When no bindings are provided, the system attempts to connect ports with the same key name, assuming their schema is compatible.

This mechanism is essential for enabling interoperability between components provided by different vendors or when combining general-purpose nodes with more specialized ones. Importantly, while bindings can be manually defined at design time, they can also be determined dynamically at runtime through the use of an LLM-based agent. In our implementation, this agent is referred to as the `Port Adapter Service`, which enables the adaptiveness of the system. The service takes as input the schemas of two port sets—source and target—and generates a bindings map by matching fields based on names, types, structure, and required fields. It supports nested objects through dot notation[5] and applies renaming where needed to ensure schema alignment.

2.3.3 Workflow Execution

For the purposes of this project, the workflow engine was built entirely from the ground up, despite considering alternatives like *n8n*[6]. While the Operational Layer is the core of the system—where the actual execution of workflows takes place—it heavily relies on the Knowledge Layer. As shown in Figure 2.7, the Knowledge Layer is accessed through the *MOP* (Meta Object Protocol), as already introduced in Section 2.1.1. The MOP encapsulates meta-model services that directly interface with the three repositories: Meta-model Catalogs of Workflow, Nodes, and Intents. Therefore, the MOP acts as

[5]Dot notation is a syntax used to access nested fields in structured data (e.g., `user.address.city`), commonly found in formats like JSON or object schemas.

[6]n8n is a workflow automation tool. See `https://n8n.io/`

the controller of meta-models and provides the only possible way for the Operational Layer to fetch and update them. Additionally, it handles event dispatching and, through dedicated services such as the `Workflow Metamodel Validator` and `Node Metamodel Validator`, ensures the structural integrity of these meta-objects. The complete execution process proceeds as follows:

1. **Intent Detection**: `The Intent Detection Service` identifies the user's intent from natural language input. It retrieves relevant intents from the Knowledge Layer for RAG and extracts unstructured variables. If no matching intent exists, it creates a new one with associated metadata.

2. **Workflow Selection**: The `Routing Manager` selects a workflow corresponding to the detected intent by querying available workflows. If multiple matches are found, selection can be guided by intent-specific scores—derived from user feedback, comparative evaluation, or an *AI judge*[7]—while a temperature-based sampler ensures diversity in the final choice.

3. **Workflow Retrieval**: The `Workflow Instance Manager` retrieves an existing workflow from the registry or instantiates a new one if none exists by fetching up-to-date information from the Knowledge Layer.

4. **Workflow Execution**: The `Input Mapper Service` (LLM-powered) binds the extracted input data to the entry ports of the workflow. The `Workflow Executor` then runs the process, supported by the `Port Adapter Service` (LLM-powered), which resolves any port mismatches experienced during execution.

[7] An AI model that is used to evaluate the output of other AI systems [13]

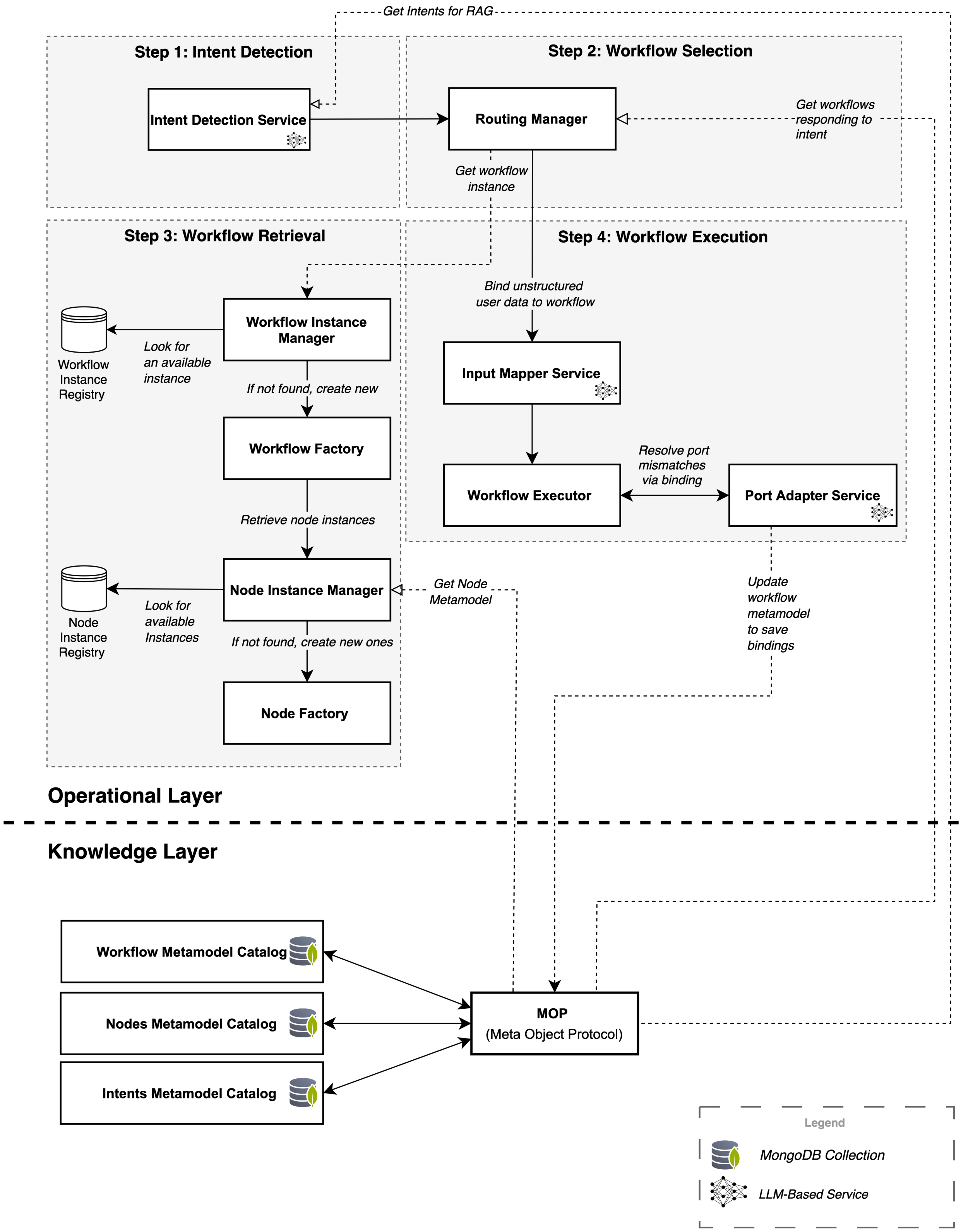

Figure 2.7: Workflow Execution four-step process, showing the interaction between the Operational Layer components and the Knowledge Layer accessed through the MOP and its three meta-model catalogs.

Ports Adaption

As previously mentioned, the `Port Adapter Service` is essential for ensuring system adaptability and the interoperability of components, even when these components are developed independently or by third-party vendors. Since nodes are not necessarily aware of each other, aligning components with mismatched input and output interfaces is critical to achieving component reusability. As discussed in Section 2.3.2, edge bindings between components can be resolved at runtime. This task is handled by the `Port Adapter Service`, which is triggered by the `Workflow Executor` whenever a node about to be executed lacks the necessary input data in the current execution context. If a functional adaptation is found, the Operational Layer interacts with the Knowledge Layer to update the workflow meta-model. As a result, following executions of the same workflow will skip the adaptation step, reducing execution time. Figure 2.8 provides a summary of this dynamic adaptation process. Ultimately, this mechanism significantly improves the **resilience of the system**: when a workflow depends on a node whose interface changes (e.g., due to a breaking change), the system can attempt to automatically resolve any incompatibilities introduced by that change.

Workflow Synthesis

The true potential of this architecture and the SBOM becomes apparent when during the *Workflow Retrieval* phase (Step 3) no pre-assembled workflow capable of handling the user's intent is found in the Knowledge Base. In such cases, the system can leverage *SBOM-like information* in the Knowledge Layer, which becomes central to the process. This enables the dynamic construction of new workflows by intelligently combining existing nodes.

Each node is associated with both qualitative and quantitative descrip-

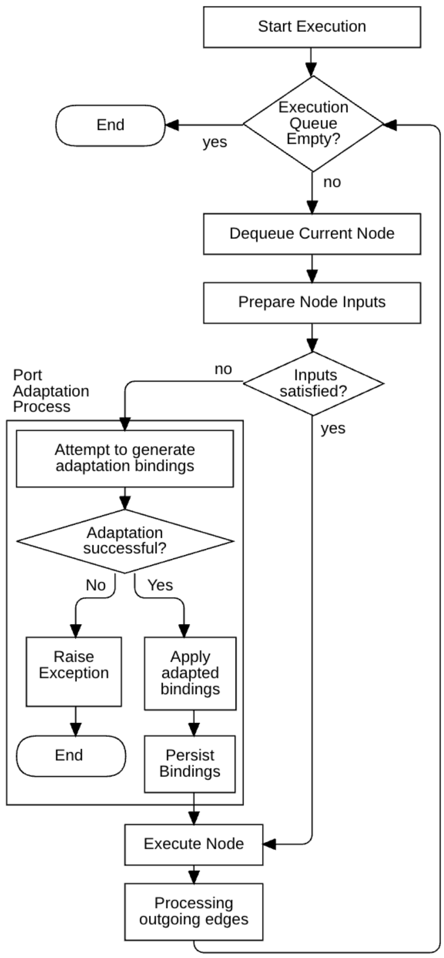

Figure 2.8: Flowchart of the workflow nodes execution process, detailing the dynamic port adaptation process.

tors—structured and unstructured—which describe its functionality and specifications. These descriptors are embedded and indexed to support **Hybrid Search**[8], allowing an LLM agent to retrieve and compose components based on their declared capabilities.

This mechanism enables the system to generate workflows that meet **previously unanticipated requirements**. For instance, if the intent intro-

[8] An information retrieval method that combines semantic search with full-text search, merging results using algorithms such as Reciprocal Rank Fusion to improve relevance.

duces a new constraint, such as a specific latency threshold, the system can construct a workflow by including only nodes whose descriptors indicate compatibility with that latency requirement. Similarly, this applies to verifying the legal compliance or certification status of nodes and tools. Moreover, this mechanism fosters interoperability between vendors, overcoming the major challenges related to the **lack of standardization** in SBOMs and AIBOMs. Even without a shared or predefined standard or grammar for tracking node dependencies and specifications, considering that different providers may disclose only partial data or use varying formats, an AI-driven approach can transcend these limitations.

Once a suitable workflow is synthesized, it can be saved in the Knowledge Base for future retrieval and reuse. New workflows can be evaluated using established methods such as AI-based judgment, human evaluation, or **consensus-based voting**. In the latter, multiple agents specialized in workflow construction independently generate candidate workflows, and a workflow is accepted only if it is produced by a majority of agents.

While this high-level approach is promising and underscores the importance of the meta-model repository, further research is required to formalize and systematically structure the process of automatic workflow synthesis.

2.3.4 Runtime Adaptations

As previously discussed, a central advantage of the Reflection pattern is its support for runtime adaptability. In the proposed architecture, agents both retrieve and modify knowledge during execution to accommodate modifications in the environment, dependency structures, or evolving user requirements. Operational agents interact with the Knowledge Layer to query or update system knowledge in response to runtime conditions. These updates may originate **externally**—such as from software releases, dependency upgrades, or the introduction of new meta-models—or be **triggered internally** by components (e.g., the `Port Adapter Service`; see Section 2.3.2) or by system processes (e.g., the generation of new intents and workflows; see *Workflow Execution*, Section 2.3.3). To ensure synchronization and consistency between the operational level and the Knowledge Layer, meta-model changes are propagated as events. These events, dispatched via the Meta-Object Protocol (MOP), are received by subscribed operational components, which then adapt their configurations accordingly (see Figure 2.9).

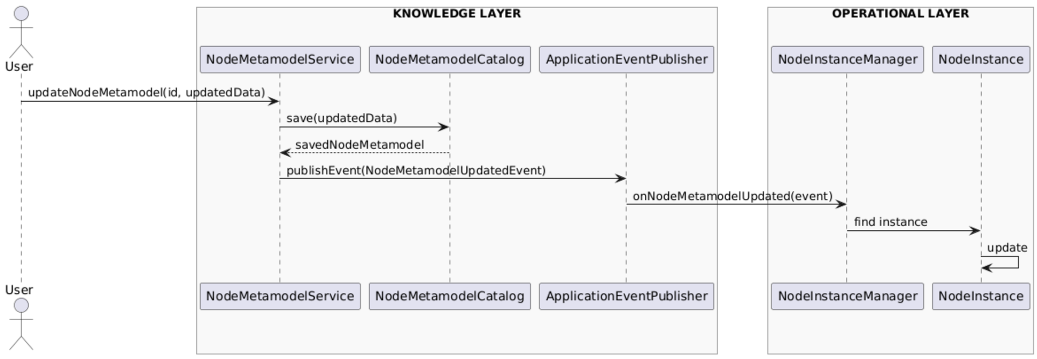

Figure 2.9: Propagation of meta-model updates from the Knowledge Layer to operational components via the MOP.

Meta-models Versioning

In the proposed implementation, the versioning of meta-model catalogs follows the Semantic Versioning[9] standard.

– Minor and patch version changes are applied in place within the existing document.

[9] A versioning scheme that uses the format `MAJOR.MINOR.PATCH` to indicate the type and impact of changes.

– Major version changes (i.e., breaking updates) result in the creation of a new meta-model entry within the catalog.

Therefore, each meta-model is identified by a unique ID and a family ID, which groups all versions belonging to the same logical component. Within a workflow, nodes reference a specific version of a node.

Strategies for Instance Updates

Two primary operational update strategies are employed depending on the nature of the meta-model change:

– **Hot-Swapping**: For non-breaking updates, if the instance is not currently running, the system may swap the meta-model in place, seamlessly refreshing the instance.

– **Re-instantiation**: In cases where the instance is currently active or the update constitutes a breaking change (e.g., altered node dependencies in a workflow), the affected meta-model is marked as deprecated. Any future execution of such instances requires a complete re-creation, even if it was previously persisted in the registry.

Chapter 3

Relevant Use Cases: the AI4NE and NE4AI scenario

To illustrate the methodology in action, we developed a use case that leverages AI both for networking (AI4NE) and networking for AI (NE4AI). Concretely, the system must route user tasks across different nodes (or cloud resources) according to user "intent" and hardware availability, while also relying on specialized AI modules to optimize networking decisions. Below is an overview of how the cognitive workflow proceeds:

1. **User Intent Detection**

2. **Potential Routes Definition**

3. **AI Request Analysis and Hardware Selection**

4. **Routing Finalization**

3.0.1 User Intent Detection

Upon receiving a request from a client (e.g., "Render a real-time analytics dashboard" or "Execute large-scale model training"), a User Intent Agent (an LLM-based agent) interprets the high-level goal and translates it into technical requirements. This step involves:

- Parsing the user's textual or spoken instruction.
- Mapping that intent to known categories (inference task, training job, data filtering, etc.).
- Consulting the AIBOM to understand which AI models or libraries are permissible for this kind of request (e.g., "Model X is suitable for object detection, but not recommended for large-scale text processing.").

All discovered components relevant to the request—such as the required Python libraries, pretrained model references, or potential data pipelines—are recorded (or updated) in the AIBOM to ensure the system remains transparent about which components it is about to use.

3.0.2 Potential Routes Definition

Next, a Routing Planner Agent searches for feasible network paths between the client application and the available compute nodes. Using standard network protocols and topological information, it generates a list of candidate routes. Distinct network constraints (latency, throughput, energy consumption, cost, etc.) are factored in, often requiring multiple rounds of negotiation with other agents or external controllers.

While performing this step, the agent consults the meta-level reflection data:

– Query which nodes currently have the AI libraries or model versions needed.

– Verify compliance constraints from the AIBOM (for example, nodes that must not run a certain unpatched version of a library due to security advisories).

– Eliminate routes if the underlying hardware or software stack is known to be unreliable or unlicensed for the requested model (e.g., GPU licensing restrictions).

3.0.3 AI Request Analysis and Hardware Selection

Once potential routes are defined, the system needs to confirm that the selected nodes can actually fulfill the user's AI request:

– **Hardware Analysis**: A specialized Resource Evaluator Agent checks CPU, GPU, or TPU availability. It correlates the user's task (as identified in the Intent Detection step) with the required hardware acceleration or memory footprint.

– **AIBOM Query**: The agent queries the AIBOM to ensure the model in question is compatible with the node's software environment (e.g., particular library versions, known security patches).

– **Recommendation Generation**: Based on these checks, the agent recommends the best route-node pairs, factoring in performance versus cost.

Because some tasks may rely on large LLMs or sensitive data, the system leverages the AIBOM to confirm compliance with data residency requirements or usage constraints. If mismatches are found (e.g., a node is running an older version of the model that has known biases or vulnerabilities), the

AIBOM triggers a meta-level adaptation to either update the model or redirect the request elsewhere.

3.0.4 Routing Finalization

Finally, a Decision Orchestrator Agent merges outputs from the previous steps:

– It confirms that each node along the selected route meets both networking requirements (latency thresholds, bandwidth availability) and AI-specific constraints (model version, patch level, resource availability).

– It finalizes the routing decision, effectively instructing the traffic or job scheduler to provision the user's task on the chosen path.

– A final log entry is written to the AIBOM, capturing the route chosen, the models and libraries used, and any dynamic modifications performed.

Throughout this process, the system's meta-level continuously updates the AIBOM with new or modified dependencies, ensuring that future queries and audits will accurately reflect the environment in which each decision was made.

3.0.5 Results of Experimentation

To assess the feasibility and advantages of this approach, we deployed a simplified prototype in a controlled lab environment:

– **Experimental Setup**: We simulated a small-scale software-defined network (SDN) with four nodes, each running different hardware (e.g., CPU-only vs. GPU-enabled). A suite of test tasks—ranging from lightweight inference jobs to more resource-intensive training tasks—was submitted to the system with varying user intents.

– **Metrics**: We tracked (i) route feasibility and correctness, (ii) average planning time, (iii) concurrency overhead when multiple tasks arrived simultaneously, and (iv) the completeness of the AIBOM logs.

Observations:

– The reflection layer effectively excluded nodes with incompatible or unpatched libraries, preventing known security threats and licensing violations.

– Overall routing decisions, combined with the hardware analysis, converged in near real-time for simpler tasks, though more complex tasks introduced additional overhead.

– Developers and auditors could readily trace each decision back to its underlying data (e.g., library versions, node constraints), facilitating clearer accountability.

Despite these positive findings, we encountered certain challenges, including the complexity of maintaining up-to-date meta-information and addressing partial failures when one agent's local state diverges from the AIBOM record. These aspects are further discussed in Chapter 3. Nevertheless, our results demonstrate how integrating multi-agent orchestration with a reflective AIBOM-based approach can offer improved transparency, adaptability, and compliance in an AI-driven network environment, exemplifying the potential of AI4NE and NE4AI synergy.

Figure 3.1: Flowchart of the workflow nodes execution process, detailing the dynamic port adaptation process.

Chapter 4

Discussion and Future Work

This final chapter presents a discussion of the proposed architecture's outcomes and explores its implications, limitations, and future evolution. Section 4.1 examines the key challenges encountered during development and the limits of the present implementation; Section 4.2 outlines future research directions; finally, Section 4.3 presents our concluding observations on the contribution of this work.

4.1 Challenges

The implementation of the proposed framework brought to light several challenges, which can be broadly categorized as architectural limitations of the framework and difficulties encountered during development.

4.1.1 Framework Complexities and Limitations

Performance Overhead

The integration of meta-model catalogs introduced a measurable performance overhead compared to traditional approaches. The necessity to query and update meta-information at runtime, along with validating component compatibility, led to increased latency in workflow execution, as demonstrated by the controlled experiments illustrated in Chapter 3. This overhead is notably pronounced during the starting phases of workflow execution, where intent detection, variable extraction and mapping, and compatibility validation must occur before the actual processing begins. While this performance bottleneck might be a concern in some latency-critical applications, it could be acceptable given the improved transparency, adaptability, and—most importantly—reliability, as demonstrated by the experiments.

Complexity of Meta-Information Management

One of the most notable challenges encountered was maintaining accurate and up-to-date metadata across the system. As the number of nodes and their interdependencies grew, the complexity of tracking configuration updates became substantial. Furthermore, the heterogeneous nature of AI components exacerbates this challenge. Different model providers use varying metadata and configuration formats, making it challenging to define an abstract schema that is suitable for all LLM providers.

State Consistency

The dual-layer architecture presents new challenges related to state consistency. For example, when workflows depend on nodes that are subjected to breaking changes, the system cannot automatically attempt to adapt. Similarly, the situation in which a workflow is running while the nodes are updated concurrently requires better management and is a subject of future inquiry. Such divergences can lead to decisions based on outdated or incomplete information, which can compromise reliability.

Domain Precision

A key complexity of the framework lies in configuring the system and meta-information to operate with fine granularity across diverse domains. Aiming for an out-of-the-box, highly adaptable system risks losing precision in specific applications. This issue was evident during the setup of the AI4NE/NE4AI experiments (Chapter 3). For example, the system's intent detector failed to recognize certain specialized operations as refinements of broader intent categories, suggesting the need for additional domain-specific knowledge. Similarly, variable extraction poses domain-dependent challenges. In some applications, variables must be extracted with absolute fidelity; in others, conversions or qualitative-to-quantitative mappings are required, while some cases demand intelligent aggregation (without fabricating data). Accordingly, core AI components must be dynamically configured per domain, striking a balance between generalizability and domain-specific precision.

4.1.2 Challenges in System Development

Tooling Immaturity

While Spring AI offered a promising foundation for integrating LLM capabilities into Java-based systems, its relative novelty meant that documentation, tooling, and community support were still in the process of evolving. Consequently, several components required custom implementation, and some bugs were encountered—for example, issues with prompt templates and structured outputs failing due to JSON within the prompt, as documented in issue `#2836`[1] and issue `#2347`[2].

Lack of Standardization

As discussed in Chapter 1, while the concept of SBOM is relatively mature in traditional software domains, its extension to AI remains unstandardized. This results in ambiguity when defining the scope and schema of the AIBOM, especially concerning components such as LLM configurations, datasets, prompt templates, and inter-agent interactions. Establishing an effective representation therefore required iterative refinement and custom schema design, given the absence of widely adopted best practices.

Evaluation of LLM-based Components

Validating the correctness and stability of LLM-powered services proved challenging due to their nondeterministic and open-ended nature. Extensive testing was required to guarantee high reliability, especially for the structured data flow. This involved testing complex, deeply nested adaptations of intricate schemas, as well as ensuring strict compliance with the structured output models. Moreover, guaranteeing consistent behavior across different prompts, providers, and model versions necessitated comprehensive integration testing, which incurred notable costs associated with API usage. For

[1] See `https://github.com/spring-projects/spring-ai/issues/2836` (Accessed on June 9, 2025)

[2] See `https://github.com/spring-projects/spring-ai/issues/2347` (Accessed on June 9, 2025)

more details, see the testing strategy outlined in Appendix A.

4.2 Next steps

This section outlines future directions for refining the framework's implementation and identifies key research opportunities.

4.2.1 Prospects for Implementation Refinement

Advanced Workflow Synthesis

Although the current implementation is designed to support autonomous workflow synthesis, substantial research is still required. Nevertheless, this also opens up significant opportunities for advancement, particularly in the following areas:

- **Workflow Optimization**: Developing a more sophisticated core logic that employs multi-objective optimization techniques could enable the system to balance competing objectives—such as performance, cost, energy efficiency, and regulatory compliance—simultaneously.

- **Formal Verification Methods**: Although the proposed implementation already features verification services to check the quality of meta-level artifacts, integrating formal verification techniques would further strengthen correctness guarantees. This is especially critical in safety-sensitive domains.

- **AI as a Judge**: Introducing an AI-based mechanism to automatically assess both the meta-model structure and execution outputs of workflows is necessary. This would enable the consistent evaluation of both automatically generated and manually crafted workflows, assisting in the detection of regressions and ensuring system reliability. Given the open-ended nature of workflow execution and the absence of an exact ground truth, an LLM agent is particularly well-suited for this task.

Security and Privacy Enhancements

Future developments should focus on addressing security and privacy, building on the strategies already discussed in the cited works [1,26]. To improve accountability, integrating trust mechanisms such as verifiable credentials or blockchain-based audit trails could prove effective. In particular, the secure sharing of SBOM data could rely on techniques such as zero-knowledge proofs [23], which allow verification of compliance without revealing specific implementation details.

Enhanced Scalability

To facilitate enterprise-scale implementations, subsequent efforts should concentrate on:

- **Distributed Architecture**: Evolute from the centralized prototype to a fully distributed system, as outlined in the SALLMA design, to ensure scalability and fault tolerance, while maintaining consistency across metamodels.

- **Hierarchical AIBOM Management**: Implement a multi-level AIBOM structure that supports efficient querying and updates at multiple levels of granularity—combining summary-level views for strategic decisions with detailed data for operational analysis.

- **Caching Strategy**: Replace in-memory caching with scalable solutions like Redis and introduce prefetching to anticipate metadata needs, reducing latency and improving system responsiveness.

4.2.2 Directions for Future Research

To lay the foundation for future developments, this study opens up several promising research directions that aim to validate the proposed approach.

The first direction involves **quantitative evaluations** of the system via controlled modifications of the AIBOM. By altering the information in the meta-catalogs, it is possible to assess the system's resilience, sensitivity, and adaptive capabilities. These experiments would yield measurable insights into the robustness of cognitive workflows when faced with changes in component-level descriptors.

A second key direction focuses on **knowledge subtraction scenarios**. This line of inquiry aims to analyze how the removal or degradation of structured or unstructured knowledge affects the system's behavior. Future investigations could explore the effects of excluding specific nodes, metadata fields, or knowledge base entries from the meta-model, assessing whether the system can compensate for the missing information.

4.3 Conclusion

This thesis introduced a two-fold framework that addresses critical gaps in the literature and industry in the domain of multi-agent AI systems. By combining the Reflection architectural pattern with the concept of the AI Bill of Materials, it has been demonstrated how cognitive workflows can achieve enhanced reliability, portability, and maintainability. The convergence of regulatory requirements, emerging business demands, and existing research gaps offers a timely opportunity for implementing solutions such as the one proposed in this study. As artificial intelligence becomes more integral to critical infrastructure and decision-making processes, robust governance mechanisms become indispensable.

In conclusion, this research, situated at the intersection of software engineering and AI engineering, offers both theoretical foundations and practical insights for addressing the growing complexity of modern artificial intelligence systems. Looking ahead, the effectiveness of future multi-agent architectures will depend not only on their technical capabilities but also on their traceability, adaptability, and accountability. The presented framework marks a step toward this vision, fostering the development of more trustworthy and resilient AI solutions. As the field continues to mature, such approaches will be essential for securing public trust and ensuring the reliable integration of AI into high-stakes domains.

Appendix A

Implementation Details and Supplementary Materials

This appendix provides additional technical insight to support the implementation described in Chapter 2.

UML Diagrams

This section presents UML diagrams that illustrate the structural design and interrelationships of the system's core components. Figure A.1 depicts the **Knowledge Layer**, which encapsulates the domain model and the repositories used for Object-Document mapping (ODM). Serving as the backbone, the Meta-Object Protocol (MOP) is realized through five distinct services. These services manage the meta-models and facilitate advanced capabilities such as Hybrid Search.

Conversely, Figure A.2 illustrates the **Operational Layer**. This package includes the implementation of specialized workflow nodes that interact with external systems, such as databases, REST APIs, and AI service providers.

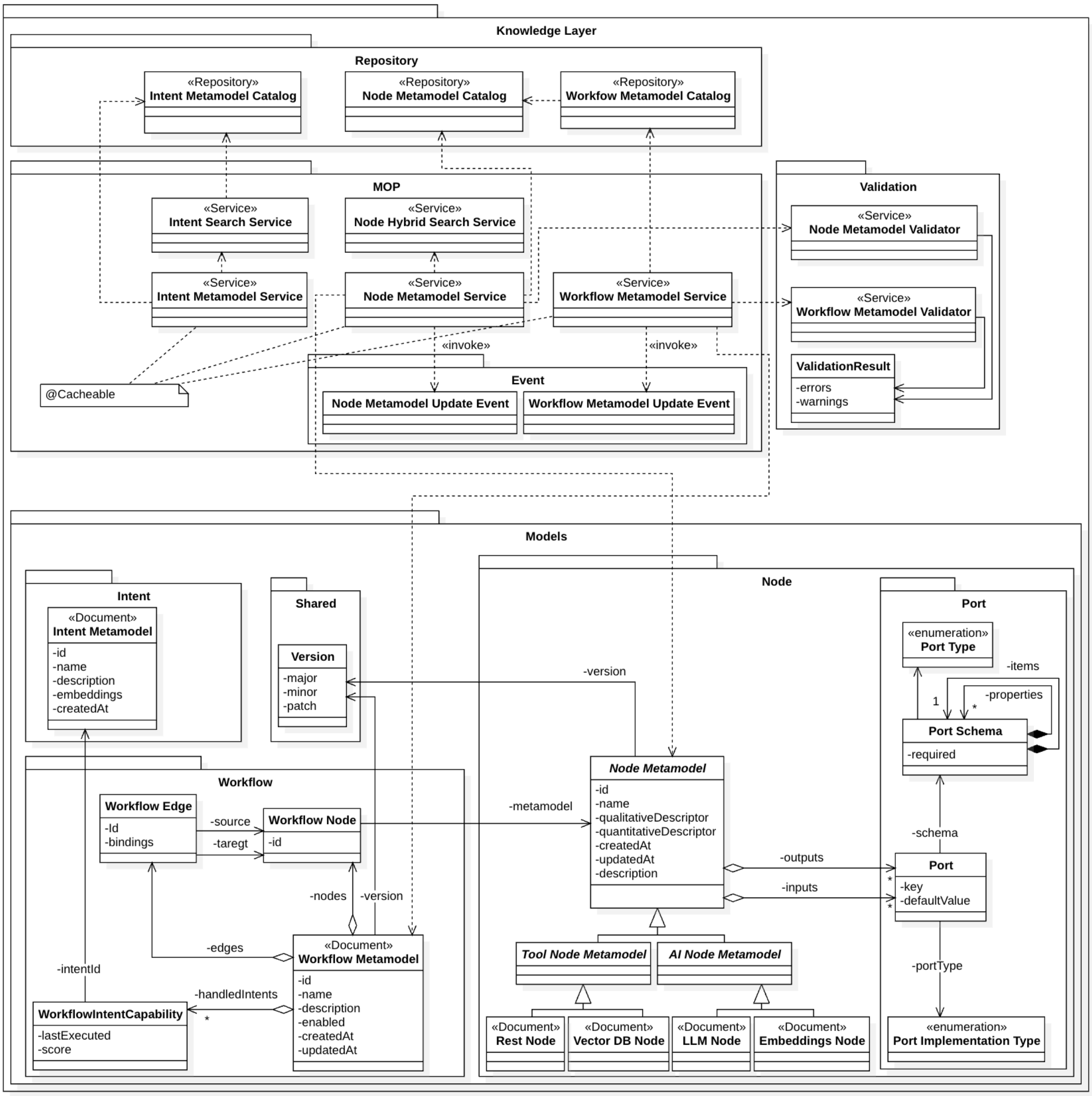

Figure A.1: UML Diagram of the `knowledge` package, illustrating its internal structure and relationships with other components. For conciseness, standard getters/setters and non-essential elements have been excluded.

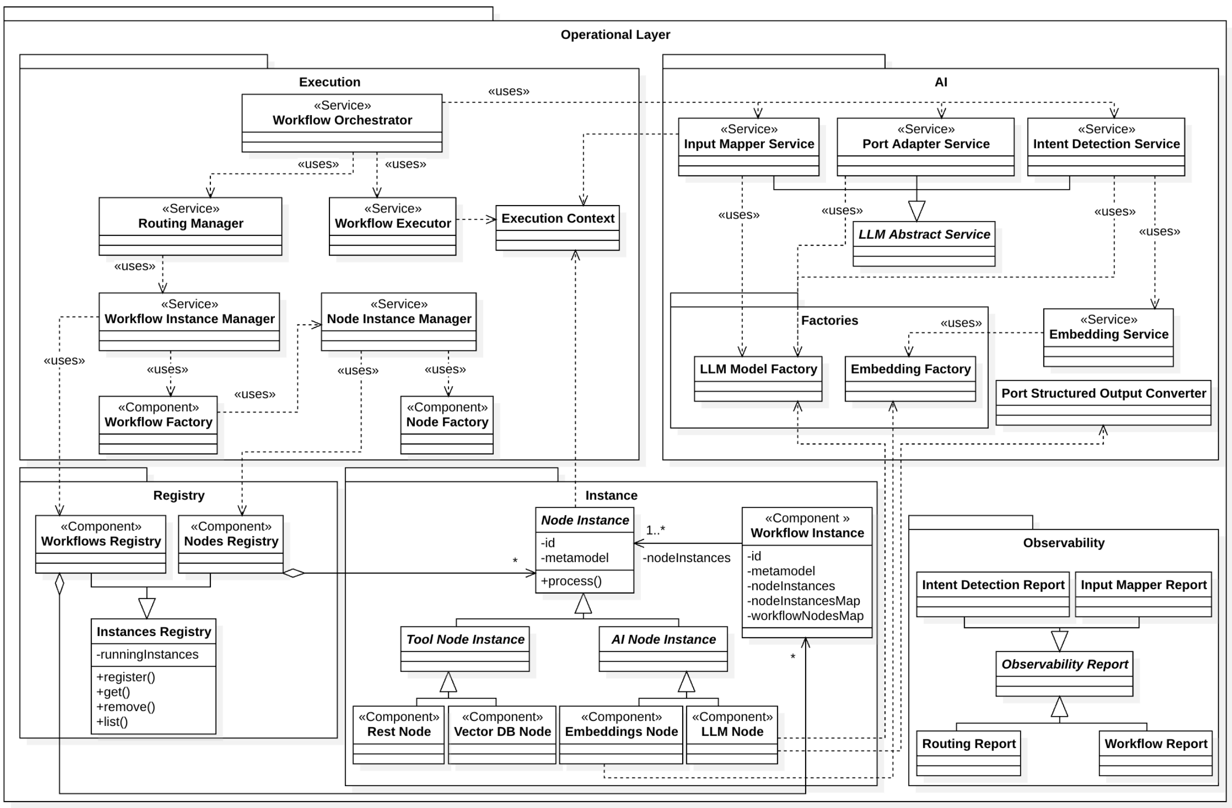

Figure A.2: UML Diagram of the `Operational` package, presenting the core classes and their interactions within the operational layer. It details functionalities including workflow orchestration, AI services, instance management, and observability. Standard getters/setters and non-essential elements are omitted for clarity.

Central to this layer is the `engine` package, which orchestrates the lifecycle of workflows—handling their retrieval, routing, and execution. Workflow and node instantiation are handled by specialized factory components, while two in-memory registries maintain records of active workflow and node instances, respectively. A significant component within the operational layer is the `AI` subpackage, which manages interactions with external providers of large language models and embedding services. It also exposes several essential LLM-powered services:

– `Input Mapper Service`: Translates user-provided request variables into the corresponding workflow inputs.

– `Port Adapter Service`: Resolves data-flow mismatches between workflow nodes.

– `Intent Detection Service`: Identifies and classifies the intent of the user.

To enable dynamic and flexible model instantiation, a factory design pattern has been implemented. This allows LLM and embedding models to be initialized at runtime based on the specifications provided by the knowledge base. This approach encourages high configurability and supports seamless switching between different model providers, thus facilitating experimental evaluation and optimization.

AI Integration

A central aspect of the system is the integration of large language models (LLMs) to enable cognitive workflows. For this purpose, Spring AI was selected as the primary framework (see Appendix B for further details). To evaluate both performance and system adaptability, multiple models and providers were tested across diverse services, demonstrating the flexibility offered by the model factories (see Figure A.2). Ultimately, **Claude 3.5 Sonnet** by Anthropic was selected for the `Input Mapper Service` and `Port Adaptation Service` due to its strong capabilities in schema handling, consistent structured outputs, and lower cost compared to more recent models like Claude 3.7 and Claude 4. For the `Intent Detector Service`, **GPT-4o** by OpenAI was integrated, providing state-of-the-art performance in intent classification. Finally, for generating embeddings, OpenAI's **text-embedding-3-small** was employed to enable efficient vector search supporting intent classification.

Testing Summary

The testing strategy covered unit, integration, and end-to-end tests, validating system correctness and robustness across 104 test cases:

- **Unit testing** involved 48 tests designed to verify the functionality of isolated components, such as metamodel validators and domain model methods.

- **Integration testing** included 54 tests that evaluated the interactions between components and services. LLM-based services were tested on real-world tasks spanning a wide range of scenarios, from simple to complex. Furthermore, the tests covered nodes that interact with external systems, such as database nodes and RESTful services, the latter simulated using WireMock[1].

[1] WireMock is an open-source tool for mocking HTTP-based APIs. See `https://wiremock.org`

– **End-to-end testing** involved two comprehensive tests simulating full workflow execution and adaptation in a production-like environment. These tests covered the entire process—from intent recognition to response generation—within the RAG scenario depicted in Figure 2.4.

A valuable insight from this testing regime is the role of integration tests that incorporate actual (non-mocked) LLM API calls. Although such tests incur considerable costs due to high API usage, they are essential for verifying system stability and functionality, particularly given the inherent non-deterministic nature of LLMs. By designing specific, nontrivial test cases that challenge the model's capabilities, this approach enables robust validation across different foundation models. It ensures resilience against potential regressions or failures resulting from model updates, optimizations, or changes in the underlying model provider.

It is important to note that, while database connections were mocked during the unit and integration testing phases to isolate system components, the end-to-end tests employed a dedicated MongoDB cluster. This environment was configured to closely replicate production conditions, providing a realistic scenario to validate the system's performance.

Test Coverage Report

Table A.1 presents the code coverage metrics of the implemented system.

Table A.1: Code coverage statistics by package.

Package	Class Coverage	Method Coverage	Line Coverage
`API`	50% (3/6)	6% (3/45)	5% (8/139)
`config`	75% (3/4)	60% (3/5)	80% (8/10)
`knowledge`	88% (55/62)	73% (244/333)	63% (902/1415)
`operational`	77% (51/66)	71% (236/332)	75% (1444/1906)
Total (all packages)	**80% (112/139)**	**67% (486/716)**	**68% (2362/3471)**

API Endpoints Overview

The system exposes a RESTful API for manipulating the knowledge catalogs and performing requests. The following table summarizes the available endpoints, HTTP methods, and corresponding controllers.

Table A.2: Overview of exposed API endpoints and their respective controllers.

Endpoint	Method	Controller
`/api/workflows`	GET, POST	WorkflowController
`/api/workflows/execute`	POST	WorkflowController
`/api/workflows/{id}`	GET, PUT	WorkflowController
`/api/nodes`	GET, POST	NodeController
`/api/nodes/embeddings`	POST	NodeController
`/api/nodes/embeddings/{id}`	PUT	NodeController
`/api/nodes/gateway`	POST	NodeController
`/api/nodes/gateway/{id}`	PUT	NodeController
`/api/nodes/llm`	POST	NodeController
`/api/nodes/llm/{id}`	PUT	NodeController
`/api/nodes/rest-tool`	POST	NodeController
`/api/nodes/rest-tool/{id}`	PUT	NodeController
`/api/nodes/vector-db`	POST	NodeController
`/api/nodes/vector-db/{id}`	PUT	NodeController
`/api/intents`	GET, POST	IntentController
`/api/intents/{id}`	GET, PUT, DELETE	IntentController

Source Code Repository

The complete source code developed for this project is publicly available on GitHub at the following URL:

`https://github.com/NiccoloCase/cognitive-workflow`

This repository contains all relevant implementation files, documentation, and instructions necessary to reproduce the results and experiments described in this thesis.

Appendix B

Comparative Analysis of LangChain and Spring AI

This appendix presents a comparative analysis conducted to identify the most suitable framework for integrating large language models (LLMs) within the scope of this thesis project, ultimately motivating the decision to adopt a Java-based approach. The study focused on evaluating LangChain with Python and Spring AI with Java. The objective was to assess development experience, framework capabilities, and underlying design philosophies.

LangChain Overview

LangChain is an open source framework introduced in October 2022, conceived to simplify the integration of LLMs into applications, allowing developers to build complex AI-powered systems. Over time, the LangChain ecosystem has grown to include additional tools such as LangGraph and LangSmith. Actively maintained by a passionate and growing community, LangChain has been widely adopted across various sectors, ranging from startups to large enterprises. The framework is organized around modular components, such as chains, tools, memory, and agents, and offers seamless integration with a variety of foundational models, vector stores, APIs, and data loaders. LangChain also provides support for advanced agent architectures including `ReAct`, `MRKL`, `Plan-and-Execute`, and `BabyAGI`, enabling out-of-the-box capabilities for decision making and task decomposition.

Spring AI Overview

Spring AI is an initiative within the Spring framework that facilitates the integration of generative AI capabilities into enterprise Java applications. Released in a milestone version in May 2024, its first stable release is expected by mid-2025[1]. Some of the most notable features include:

- **Spring Ecosystem Integration:** Seamlessly integrates with the existing Spring environment, enabling developers to leverage AI capabilities within the Spring ecosystem, following established Spring patterns.
- **Enterprise Scalability:** Designed to support enterprise-level applications, enabling the development of AI solutions that scale effectively in diverse environments. It provides robust scalability through its multithreading capabilities and efficient use of the JVM, making Java preferable for handling complex concurrent workloads.
- **Seamless Deployment:** The Spring ecosystem provides a comprehensive framework that facilitates the efficient deployment of AI models in hetero-

[1] According to a recent update from the Spring team (April 2025), a release candidate (RC1) is scheduled for the following month, with a general availability (GA) release planned shortly after, in time for the Spring IO conference in Barcelona. See: `https://spring.io/blog/2025/04/10/spring-ai-1-0-0-m7-released` (accessed April 21, 2025).

geneous environments. Components such as Spring Boot and Spring Cloud support the scalable orchestration of microservices, making Spring particularly well-suited for production-ready, distributed AI systems.

Feature Comparison

Spring AI and LangChain use distinct conceptual approaches. Among the key differences highlighted in Table B.1, the most notable is their support for agents, for which the two frameworks adopt fundamentally different strategies. LangChain follows an agent-first approach and natively supports several agent architectures, integrating a wide range of tools such as web search (allowing retrieval using the entire internet), databases, external APIs, and external applications. Conversely, Spring AI follows a more workflow-oriented approach, emphasizing *Design for Reliability*. Drawing on insights from a research publication by Anthropic [19], it promotes simple, composable patterns rather than complex frameworks. At the time of writing, Spring AI does not directly integrate tools for creating agents. However, the Spring Blog [22] documents *Agentic Patterns* (e.g., Chain Workflow, Parallelization Workflow, Routing Workflow, and Orchestrator-Workers) that can be implemented using the framework. Consequently, there is currently no autonomous decision loop like LangChain's AgentExecutor or a native mechanism for automatic routing or re-planning (such as Plan-and-Execute). But this is likely to change as the project matures.

Table B.1: Comparison of Key Features Between LangChain and Spring AI for LLM-Based Application Development, updated as of April 2025

Feature	**LangChain**	**Spring AI**
Language Support	Python—JavaScript and Java secondary	Java/Kotlin
Design Approach	Modular chains, tools, and memory components	Spring-style abstractions (e.g., templates, dependency injection)
Model Providers	50+ supported providers	15+ supported providers
Agents	Native support for agents and tools	Currently unsupported
Vector Search	Supported	Supported
Web Search	Supported (built-in APIs)	No native support; external integration required
Web Scraping	Supported (e.g., ScrapeGraph, WebVoyager, ScrapingAnt)	No native support; can be integrated externally
State Management	Conversation memory, LangGraph for workflows	Chat memory support
Observability	LangSmith integration for debugging and traces	Observability through Spring ecosystem tools
Deployment	Supports Docker/Kubernetes; flexible runtime environments	Optimized for Spring Boot and cloud-native deployments
REST API Support	Requires web frameworks like Flask, FastAPI, etc.	Built-in Spring Web support
Primary Focus	Rapid prototyping, research, startup-friendly	Enterprise-grade backend integration

Development Experiment

In order to assess the development experience with LangChain and Spring AI, two parallel Retrieval-Augmented Generation (RAG) pipelines were implemented, one in Python and the other in Java. The pipelines were developed using the following components:

- **LangChain:** Utilized `FAISS` for similarity search and OpenAI `GPT-4`, served through `FastAPI`.
- **Spring AI:** Employed `SimpleVectorStore` in-memory vector store, `GPT-4`, and exposed via REST APIs through `Spring Web`.

The implementation of this comparative analysis is available in the corresponding GitHub repository[2]. Please note that this project serves as a proof-of-concept to evaluate the development process and is not intended for production use.

Development Insights

During the experimentation, several factors were taken into account, with the key differences summarized in Table B.2.

A significant observation concerns the ease of setup, documentation, and community support for the two frameworks, which impact the overall development experience. Although LangChain provides extensive documentation, it can sometimes be fragmented and difficult to navigate—especially due to frequent updates that lead to outdated and conflicting information. Furthermore, the lack of clear overviews and guided introductions can leave begin-

[2] `https://github.com/NiccoloCase/ielts-evaluator-langchain-springai`

ners feeling overwhelmed. However, these issues are mitigated by an active community that consistently produces high-quality articles and resources. Additionally, LangChain is designed to be effortless to set up, especially with its straightforward installation process via `pip`. In contrast, Spring AI benefits from being part of the well-established Spring ecosystem. While the documentation is still evolving and on-line resources remain limited, its integration with Spring Initializr and its more focused scope make it easier for newcomers to get started. Furthermore, there is a reasonable expectation that Spring AI will adhere to the same principles of stability, backward compatibility, and enterprise-grade documentation of the Spring project.

Lastly, it is worth noting the relative verbosity of Java compared to Python, which can impact development speed, especially in tasks such as string manipulation and prompt construction. In this regard, Spring AI faces a disadvantage: operations that are concise in Python often require additional boilerplate in Java. This is especially evident in tasks like JSON parsing and construction. Spring AI addresses these challenges by offering higher-level abstractions, including helper classes like `PromptTemplate` and predefined prompt structures, which reduce manual string handling and streamline prompt creation. Nonetheless, Java's strong typing, robust IDE support, and mature build tools—such as Maven and Gradle—contribute to improved maintainability and reliability.

Conclusive Framework Choice and Motivation

During the early design phase of this thesis, the choice between Java and Python was made. Python's extensive support for state-of-the-art LLM agents initially suggested it as the natural choice. However, Java's align-

Table B.2: Development Experience: LangChain vs Spring AI

Aspect	**LangChain (Python)**	**Spring AI (Java)**
Ease of Setup	High	Moderate
Modularity	High	High
Integration with Backend	Low	High
Documentation	Comprehensive	Growing
Community Support	Large and active	Emerging, backed by Spring ecosystem

ment with established software engineering principles—especially in terms of tooling, maintainability, and scalability—ultimately led to its preference. Its robust static typing system is particularly suited for implementing the reflective architecture proposed in this study: although type annotations were introduced in Python 3, Java's enforced type safety provides a more formal structural foundation for clearly demonstrating the Reflection Pattern. Furthermore, given that the project addresses themes such as Responsible AI and Software Transparency, Java's long-standing role in enterprise systems was a decisive factor. The inherent emphasis of the Spring ecosystem on scalability, security, and maintainability directly supports these objectives. Additionally, since the project was conceived to develop a structured meta-architecture for AI workflows from the ground up, advanced LangChain features—such as complex toolchains and built-in agentic patterns—were neither necessary nor aligned with the study's goals. Lastly, the novelty of Spring AI within the academic literature offered a valuable opportunity for original contribution.

References

[1] Iain Barclay, Alun Preece, Ian Taylor, Swapna Krishnakumar Radha, and Jarek Nabryzski. Providing assurance and scrutability on shared data and machine learning models with verifiable credentials. *Concurrency and Computation: Practice and Experience*, 34(7):e6997, 2022.

[2] Iain Barclay, Alun Preece, Ian Taylor, and Dinesh Verma. Towards traceability in data ecosystems using a bill of materials model. *arXiv preprint arXiv:1904.04253*, 2019.

[3] Marco Becattini, Roberto Verdecchia, and Enrico Vicario. Sallma: A software architecture for llm-based multi-agent systems. In *SA Trends 2025*. Software Technologies Laboratory, University of Florence, 2025.

[4] Joe Biden. Executive order on improving the nation's cybersecurity, 2021. `https://www.govinfo.gov/content/pkg/FR-2021-05-17/pdf/2021-10460.pdf`. Accessed: 2025-03-26.

[5] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, Chichester, UK, 1996.

[6] CycloneDX. Machine Learning Bill of Materials (ML-BOM), 2024. `https://cyclonedx.org/capabilities/mlbom/`. Accessed: 2025-03-13.

[7] Gregorio Dalia, Corrado A. Visaggio, Andrea Di Sorbo, and Gerardo Canfora. Sbom ouverture: What we need and what we have. In *Proceedings of the 19th International Conference on Availability, Reliability and Security (ARES 2024)*, pages 1–9, Vienna, Austria, July 2024. ACM.

[8] European Union. Regulation (EU) 2024/1689 of the European Parliament and of the Council of 13 June 2024 laying down harmonised rules on artificial intelligence (Artificial Intelligence Act). `https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=OJ:L_202401689`, jun 2024. Official Journal of the European Union, L 1689, 13 June 2024. Accessed: 2025-03-13.

[9] Federal Register. Promoting the use of trustworthy artificial intelligence in the federal government. `https://www.federalregister.gov/documents/2020/12/08/2020-27065/promoting-the-use-of-trustworthy-artificial-intelligence-in-the-federal-government`, December 8 2020. Accessed: 2025-03-25.

[10] Timnit Gebru, Jamie Morgenstern, Briana Vecchione, Jennifer Wortman Vaughan, Hanna Wallach, Hal Daumé III, and Kate Crawford. Datasheets for datasets. *Communications of the ACM*, 64(12):86–92, 2021.

[11] Hugging Face. Dataset Cards, n.d. `https://huggingface.co/docs/hub/datasets-cards`. Accessed: 2025-03-13.

[12] Hugging Face. Model Cards, n.d. `https://huggingface.co/docs/hub/model-cards`. Accessed: 2025-03-13.

[13] Chip Huyen. *AI Engineering*. O'Reilly Media, Inc., 2024.

[14] Berend Kloeg, Aaron Yi Ding, Sjoerd Pellegrom, and Yury Zhauniarovich. Charting the path to sbom adoption: A business stakeholder-centric approach. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security (ACM AsiaCCS)*, pages 1770–1783. ACM, 2024.

[15] Linux Foundation. The state of software bill of materials (sbom) and cybersecurity readiness, Jan 2022. https://www.linuxfoundation.org/tools/the-state-of-software-bill-of-materials-sbom-and-cybersecurity-readiness/. Accessed: 2025-03-13.

[16] Qinghua Lu, Liming Zhu, Xiwei Xu, and Jon Whittle. Responsible-ai-by-design: A pattern collection for designing responsible artificial intelligence systems. *IEEE Software*, 40(3), 2023.

[17] Margaret Mitchell, Simone Wu, Andrew Zaldivar, Parker Barnes, Lucy Vasserman, Ben Hutchinson, Elena Spitzer, Inioluwa Deborah Raji, and Timnit Gebru. Model cards for model reporting. In *Proceedings of the Conference on Fairness, Accountability, and Transparency*, pages 220–229. ACM, 2019.

[18] National Telecommunications and Information Administration. Software bill of materials. https://www.ntia.gov/page/software-bill-materials, 2021. Accessed: March 12, 2025.

[19] Erik Schluntz and Barry Zhang. Building effective agents. https://www.anthropic.com/engineering/building-effective-agents, 2024. Published by Anthropic. Accessed: 2025-04-22.

[20] Trevor Stalnaker, Massimiliano Di Penta, Nathan Wintersgill, Oscar Chaparro, Daniel M. German, and Denys Poshyvanyk. Boms away!

inside the minds of stakeholders: A comprehensive study of bills of materials for software systems. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*, pages 1–13, 2024.

[21] Christian Tzolov. Spring ai - structured output. `https://spring.io/blog/2024/05/09/spring-ai-structured-output`, 2024. Published by Spring Blog. Accessed: 2025-05-20.

[22] Christian Tzolov. Building effective agents with spring ai (part 1). `https://spring.io/blog/2025/01/21/spring-ai-agentic-patterns`, 2025. Published by Spring Blog. Accessed: 2025-04-22.

[23] W3C. Verifiable credentials data model v1.0 - zero-knowledge proofs. `https://www.w3.org/TR/vc-data-model/#zero-knowledge-proofs`, n.d. Accessed: June 9, 2025.

[24] Titus Winters, Tom Manshreck, and Hyrum Wright. *Software Engineering at Google: Lessons Learned from Programming Over Time*. O'Reilly Media, 2020.

[25] Boming Xia, Tingting Bi, Zhenchang Xing, Qinghua Lu, and Liming Zhu. An empirical study on software bill of materials: Where we stand and the road ahead. In *2024 IEEE/ACM 45th International Conference on Software Engineering (ICSE '24)*, 2024.

[26] Bonning Xia, Dawen Zhang, Yue Liu, Qinghua Lu, Zhenchang Xing, and Liming Zhu. Trust in software supply chains: Blockchain-enabled sbom and the aibom future. In *2024 IEEE/ACM 4th International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS) and 2024 IEEE/ACM Second International Workshop on Software Vulnerability (EnCyCriS/SVM '24)*, pages 1–8, 2024.