A Modular Translation Layer for Static and Deterministic Payload Translation Between Heterogeneous Imperative-Based Agents Using LLMs

Implemented in PEAK environment

Leon Cherchye

Student No.: 1242327

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Artificial Intelligence

Supervisor: Professor Luís Filipe de Oliveira Gomes
Technical Supervision: Researcher Bruno Rafael Gonçalves Ribeiro

Porto, July 9, 2025

DECLARAÇÃO DE INTEGRIDADE

DECLARAÇÃO DE INTEGRIDADE

Declaro ter conduzido este trabalho académico com integridade. Não plagiei ou apliquei qualquer forma de uso indevido de informações ou falsificação de resultados ao longo do processo que levou à sua elaboração.

Declaro que o trabalho apresentado neste documento é original e de minha autoria, não tendo sido utilizado anteriormente para nenhum outro fim.

Declaro ainda que tenho pleno conhecimento do Código de Conduta Ética do P.PORTO.

ISEP, Porto, (dia) de (mês) de (ano)

[illegible]

Abstract

In systems where many different agents need to work together, it is often hard for them to understand each other's messages. This is because each agent may use different formats, words, or meanings in their communication. This is known as the *payload translation* problem. Without a proper solution, agents cannot work together, which reduces the overall performance and usefulness of multi-agent systems.

This research presents a new solution: a modular translation layer that allows agents to understand each other without changing their own internal logic or code. The translation layer works by reading the agent's behavior file to find out what kinds of messages it sends and receives. It then uses a large language model (LLM) to understand the meaning and structure of these messages. Based on that, it builds a formula that translates messages from one agent to another. Once these formulas are created, the system no longer needs the LLM. Messages are translated at runtime using only simple and fast code.

This system solves two main problems. First, it deals with *syntactic differences*, where messages have different structures or data formats. It solves this by keeping the full structure of the original message and only changing the values. Second, it handles *semantic ambiguity*, where different agents use different names for the same thing, or the same name for different things. The system uses LLMs to look at the context and explain each message field, so it can find the right matches between keys of two agents. This makes translation more accurate and avoids mistakes.

Unlike traditional methods that rely on shared formats or standards, this approach uses direct translation between agent pairs. This avoids the loss of detailed information that often happens when converting messages into a simplified standard format. The result is a more flexible and powerful system that can preserve the richness of agent messages.

The system was tested in different scenarios, and the results showed high translation accuracy, even reaching 100% in most of the scenario's. It also proved to be efficient, as the translation process during runtime is fast and does not rely on external services. However, the system is not perfect. It still requires human checking in new or complex cases, and there is no built-in feedback loop to detect mistakes.

In conclusion, this dissertation shows that it is possible to create a simple, scalable, and effective translation system that helps agents communicate without needing to change their original behavior. This is especially useful in systems with many different types of agents, such as smart cities, IoT networks, or autonomous robots. The approach lays a strong foundation for future improvements in agent communication and integration.

Keywords: Payload Translation, Heterogeneous, Multi Agent Systems, interoperability, LLMs, Distributed

Resumo

Em sistemas onde muitos agentes diferentes precisam trabalhar juntos, muitas vezes é difícil para eles entenderem as mensagens uns dos outros. Isso acontece porque cada agente pode usar formatos, palavras ou significados diferentes em sua comunicação. Isso é conhecido como o problema de *tradução de payload*. Sem uma solução adequada, os agentes não conseguem cooperar, o que reduz o desempenho e a utilidade geral dos sistemas multiagentes.

Esta pesquisa apresenta uma nova solução: uma camada de tradução modular que permite que os agentes se compreendam sem a necessidade de alterar sua lógica ou código interno. A camada de tradução funciona lendo o arquivo de comportamento do agente para descobrir que tipos de mensagens ele envia e recebe. Em seguida, utiliza um modelo de linguagem grande (LLM) para entender o significado e a estrutura dessas mensagens. Com base nisso, constrói uma fórmula que traduz mensagens de um agente para outro. Depois que essas fórmulas são criadas, o sistema não precisa mais do LLM. As mensagens passam a ser traduzidas em tempo de execução usando apenas código simples e rápido.

Este sistema resolve dois principais problemas. Primeiro, ele lida com *diferenças sintáticas*, onde as mensagens têm estruturas ou formatos de dados diferentes. Ele resolve isso mantendo toda a estrutura original da mensagem e apenas alterando os valores. Segundo, ele trata da *ambiguidade semântica*, onde diferentes agentes usam nomes diferentes para a mesma coisa, ou o mesmo nome para coisas diferentes. O sistema usa LLMs para analisar o contexto e explicar cada campo da mensagem, permitindo encontrar as correspondências corretas entre as chaves dos dois agentes. Isso torna a tradução mais precisa e evita erros.

Diferente dos métodos tradicionais que dependem de formatos ou padrões compartilhados, esta abordagem usa tradução direta entre pares de agentes. Isso evita a perda de informações detalhadas que normalmente ocorre ao converter mensagens para um formato padronizado e simplificado. O resultado é um sistema mais flexível e poderoso, capaz de preservar a riqueza das mensagens dos agentes.

O sistema foi testado em diferentes cenários e os resultados mostraram alta precisão na tradução, chegando a 100% na maioria dos cenários. Também se mostrou eficiente, pois o processo de tradução em tempo de execução é rápido e não depende de serviços externos. No entanto, o sistema não é perfeito. Ainda requer verificação humana em casos novos ou complexos, e não possui um mecanismo de feedback embutido para detectar erros.

Em conclusão, esta dissertação mostra que é possível criar um sistema de tradução simples, escalável e eficaz que ajuda os agentes a se comunicarem sem a necessidade de mudar seu comportamento original. Isso é especialmente útil em sistemas com muitos tipos diferentes de agentes, como cidades inteligentes, redes de IoT ou robôs autônomos. A abordagem estabelece uma base sólida para melhorias futuras na comunicação e integração entre agentes.

Palavras-chave: Payload Translation, Heterogeneous, Multi Agent Systems, interoperability, LLMs, Distributed

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Contextualization

Payload translation between heterogeneous imperative-based agents refers to the process of converting the content of a message (the "payload") from the format and meaning used by a sending agent to a format and meaning understandable by a receiving agent, when these agents operate using different internal representations or communication conventions. This is crucial because, in multi-agent systems, agents with diverse capabilities and backgrounds often need to exchange information to achieve collective goals, but they might be designed independently with no inherent common ground. [1] [2]

This process presents a significant challenge primarily due to the inherent heterogeneity of devices, systems, and their underlying data models in distributed environments like the Internet of Things (IoT). Without effective payload translation, agents cannot understand or act upon the information exchanged, leading to fragmented ecosystems and hindering seamless collaboration. [1] [2]

One of the sub problems for payload translation is the **syntactic discrepancy** problem. This problem occurs when agents use different encoding rules or data formats for their messages. For instance, one agent might send information formatted as JavaScript Object Notation (JSON), while another expects Extensible Markup Language (XML) or Comma-Separated Values (CSV). There are also structural differences within the same data format, such as transforming one JSON structure into another JSON structure. Even if the underlying meaning is conceptually the same, the receiving agent's decoding rules are incompatible with the sender's encoding rules, making the message unreadable and unusable. A successful translation addresses this by converting the data structure from one format to another. [1] [3]

Beyond mere format, there is also the **semantic ambiguity** problem. Semantic ambiguity refers to a mismatch in the meaning or interpretation of information exchanged between agents. Agents might use different vocabularies, naming conventions, units of measurement, or coding schemes for the same real-world concept. For example, one system might describe "air temperature" with the property name "temp_celsius" and an integer value, while another uses "ambient_temp_degC" with a float value. Even if both are communicating temperature in Celsius, the differing labels and value types make direct interpretation impossible without a common understanding or a translation mechanism that maps these disparate terms to equivalent meanings. This is often addressed through shared ontologies or semantic models. [1] [3]

While standardization aims to create common ground, relying solely on it for interoperability can lead to unintended **information loss or dilution**. If a richer, more detailed original data model is translated into a simpler, more generic standard, specific nuances or granular details present in the source might not have a direct equivalent in the target standard and are therefore discarded or generalized. For instance, an environmental sensor might provide detailed readings for various pollutants (e.g., PM2.5, PM10, CO2), but a generalized "air quality" standard might only support a single, aggregated "Air Quality Index." Translating to this standard could result in the loss of the individual pollutant levels, which may be critical for certain specialized analyses or decision-making. This means that while communication becomes possible, the richness of the data is compromised, potentially limiting the effectiveness of the receiving agent. [4] [3]

Payload translation challenges are prevalent in various real-world contexts, particularly in large-scale distributed systems. In smart home environments, devices from different manufacturers (e.g., smart lights, thermostats, speakers) often use proprietary data models and communication protocols, necessitating translation to enable coordinated actions based on a user's natural language request. Similarly, in industrial automation and the Internet of Things (IoT), disparate sensors, actuators, and control systems within a factory floor or smart city might communicate data in unique formats and semantics, requiring robust translation to achieve cohesive operation and data analysis. Emergency services and traffic management systems also face this, integrating data from diverse sources like health monitors, vehicle sensors, and city infrastructure, where each source may have its own specific data representation. [5] [1] [6]

The problem of payload translation remains highly relevant today, even with advancements in AI and communication protocols. The rapid proliferation of heterogeneous IoT devices and the emergence of new AI agent paradigms mean that the landscape of data formats and semantic definitions is constantly expanding, not shrinking. Many legacy systems persist, which were built without current interoperability standards in mind, requiring continuous efforts to integrate them with newer technologies. Furthermore, while new protocols like Model Context Protocol (MCP) or Agent Communication Protocol (ACP) offer standardized ways for agents to interact and share context, the fundamental need to translate the content of messages into a usable form for different specialized agents remains. The complexity of manual "adapter" development for every possible pair of systems is unsustainable at scale, underscoring the urgent need for more automated, flexible, and robust payload translation solutions that can handle both **syntactic and semantic differences without significant information loss**. [7] [1] [4] [3]

This dissertation presents a new approach to solving the payload translation problem between heterogeneous imperative-based agents. Recent research shows that differences in message formats (syntax) and meaning (semantics) often prevent agents from understanding each other. To address this, a distributed architecture is proposed in which each agent has access to a Large Language Model. This avoids dependence on a central translation service, eliminating the risk of single points of failure. In addition, the distributed setup increases scalability and robustness, as agents can operate and translate messages independently. This makes the system more adaptable to dynamic and decentralized environments, where agents may be added, removed, or updated without requiring changes to a central component.

The approach adds an extra translation layer on top of existing agents without changing their internal logic or structure. To make payload translation work, the system first extracts the API structure of each agent by reading the unmodified behaviour file(s). This shows which messages the agent can send and receive. After identifying these messages, the system links each message from one agent to the matching message(s) of other agents. This linking allows the creation of translation formulas between specific messages of agent pairs. By only using the original behaviour file of the agents, the approach keeps the agents' behaviour unchanged while enabling communication between different systems. This makes the translation layer modular and easy to add, helping the system scale and work well in complex multi-agent setups.

The proposed method ensures *deterministic translation*, where identical input messages always result in the same output. This improves reliability and consistency in agent communication. To increase efficiency, LLMs are used only during an initial setup phase to generate static translation formulas between agent pairs. These formulas allow fast message translation at runtime without repeatedly invoking the LLM. By generating formulas specifically between each agent pair, the need for strict standardization is avoided. As a result, rich and detailed information can be preserved during translation, preventing the loss or generalization of domain-specific data as long as both agents support these data. The combination of one-time LLM use, agent-specific translation logic, and lightweight execution makes this approach suitable for scalable deployment in real-world systems, such as smart cities, IoT networks, and autonomous agent environments.

1.2 Research Questions and Objectives

In this dissertation, an answer is sought to the main research question: **"How can a modular translation layer enable accurate and scalable payload translation between heterogeneous imperative-based agents without requiring modifications to their internal logic?"** To support this main research question, five secondary research questions have been defined:

- **SRQ1** – Can the translation layer be deployed across different agents without requiring manual intervention or changes to their internal logic?
- **SRQ2** – How can the run-time translation be made deterministic and efficient?
- **SRQ3** – How can the problems of syntactic discrepancies and semantic ambiguity be solved?
- **SRQ4** - Does this solution solve the information loss or information dilution problem that exists with standardization methods?
- **SRQ5** – Is human oversight required to validate or correct message mappings, or is the system consistent and accurate enough to operate fully autonomously?
- **SRQ6** – What are the current limitations of the approach, and how could future work improve coverage, accuracy, or adaptability of the translation process?

Based on these research questions, we define a set of **objectives** that this dissertation aims to achieve. These objectives provide a concrete direction for the investigation and form the foundation for the proposed analysis and solution:

- **O1** – Understand and describe the architecture of agent communication within MAS, including relevant protocols and abstraction layers.
- **O2** – Assess the determinism and consistency of LLMs when used in static, imperative environments.

- **O3** - Ensure the translation is deterministic and static, producing consistent and identical outputs for the same input.
- **O4** - Achieve efficient translation, minimizing processing time and ensuring reliability with minimal overhead.
- **O5** - Maintain distributivity, enabling the translation to work across distributed systems without introducing unnecessary complexity.
- **O6** - Ensure no modifications are required to the agents' input behavior files, preserving their original functionality.
- **O7** - Resolve syntactic discrepancies by correctly handling differences in payload structure between agents during translation.
- **O8** - Address semantic ambiguity by effectively resolving differences in meaning between agent communication formats.
- **O9** - Prevent information loss or information dilution, ensuring the integrity of the data during translation.
- **O10** – Ensure that the model is scalable to handle a growing number of agents, maintaining performance and efficiency as the system expands.
- **O11** – Ensure support for one-to-many message mapping.
- **O12** – Benchmark and validate the performance and robustness of the implementation.
- **O13** – Evaluate the autonomy and reliability of the proposed translation framework and determine whether human intervention is necessary.
- **O14** – Identify potential for future enhancements.
- **O15** - Identify current limitations of the model.

Research Question	Related Objectives
SRQ1	O1, O5, O6, O11
SRQ2	O2, O3, O4, O10
SRQ3	O7, O8
SRQ4	O9
SRQ5	O12, O13
SRQ6	O14, O15

Table 1.1: Correlation between Research Questions and Objectives

1.3 Document Organization

This document is organized to thoroughly explore the research questions and objectives defined at the start of the study. The first section reviews existing research to gather basic knowledge and address the first two objectives. It will focus on analyzing relevant papers and studies to build a strong theoretical foundation. At the end, research objectives 1 and 2 will be assessed.

The following chapter briefly outlines the materials used during the design and development of the model. This section provides insight into the tools, methodologies, and technologies that shaped the implementation.

Chapter 4 delves into the detailed explanation of how the model was implemented and the rationale behind key decisions. This chapter addresses objectives 3 through 11, as well as objective 14, providing a in-depth exploration of the technical aspects and design choices.

Next, a chapter is dedicated to the formulation and evaluation of test cases. This section answers objectives 10, 12, 13 and 15, and also provides further insights into objective 14. The evaluation will assess the model's performance, reliability, and autonomy, based on the defined criteria.

The final chapter, Chapter 6, concludes the document by revisiting the research questions. It will assess how well the main research question has been addressed by looking at all the secondary research questions. Additionally, this section will reflect on the significance of the work, summarize key insights from the study, and suggest directions for future research and potential improvements.

Chapter 2

State of the Art

This section investigates the state of the art in the distributed translation of communication within heterogeneous and open multi-agent systems, aiming to address the first two research objectives outlined in Chapter 1. It begins by introducing the basic concepts of agents and multi-agent systems (MAS) to establish a foundational understanding. Subsequently, agent-oriented programming approaches are discussed to frame the software paradigms used in MAS. Communication mechanisms within MAS are then analyzed to understand how agents interact and share information. The discussion continues with the identification and explanation of different layers of heterogeneity that pose challenges to agent communication and translation. The role of large language models (LLMs) in addressing these semantic translation issues is then explored, including an overview of their core principles, techniques for semantic mapping, and an analysis of their consistency and variability in static environments. Following this, a translation system architecture is presented that integrates outer language translation with the Rule Responder system. Finally, the section concludes by examining techniques for extracting payload information from imperatively programmed agents.

2.1 Basics Agents and Multi Agent Systems

An **intelligent agent** is usually described as an automated, flexible, and independent system that can observe its environment using sensors and act on it using actuators. There is no single agreed-upon definition, because agents can take many different forms, such as software programs, robots, or systems connected over a network. However, the definition by Russell and Norvig is commonly accepted. An intelligent agent is known for several important abilities: it can reason and make logical decisions, it can act on its own without needing constant control, and it can communicate and work together with other agents. It also reacts to changes in its environment and takes action to reach its goals. These abilities are widely accepted as the main features of intelligent agents in the research literature [8] [9].

In **Agent-Oriented Programming (AOP)**, agents are the main building blocks of a system, just like objects are in Object-Oriented Programming (OOP). While OOP focuses on objects that store data and use methods, AOP focuses on agents that can make decisions, have goals, and interact with other agents. In AOP, the logic of the program is based on what the agent wants to achieve, how it behaves, and how it communicates with others. This approach is useful for building systems where different parts need to work on their own and cooperate in changing environments. The following section provides a closer look at the different ways agent behavior can be defined in AOP [8].

Classical agents are built using predefined rules and algorithms. They perform well on specific tasks but struggle to generalize to new situations, often requiring retraining when placed in different environments. In contrast, **LLM-based agents** are trained on large amounts of text and can understand natural language. They are better at generalizing across different tasks and can follow complex instructions using reasoning. These agents can also reflect on their past actions and adjust their behavior over time. In LLM-based agent architectures, the language model acts as the main reasoning engine and is supported by additional components such as planning, memory, and tool usage. Combining all these parts into one unified agent is a new and growing area of research [10].

Multi-Agent Systems (MAS) are a part of Artificial Intelligence and a branch of Distributed AI. Unlike Distributed Problem Solving, where coordination is fixed in advance, MAS focus on the behavior and interaction of independent agents that are loosely connected. A MAS includes many agents working together in a shared environment to reach their own or common goals, often by cooperating or competing. Key features of MAS are knowledge sharing and communication between agents [9].

The main advantages of MAS include efficiency through task distribution and parallel work, easy scalability by adding new agents, fault tolerance where other agents can take over if one fails, and flexibility allowing agents to adapt to changing environments. MAS are used in real-world areas like power grid management, traffic control, robotics, simulations, and energy optimization. These benefits make MAS a strong and effective approach for solving problems in a decentralized way [9] [8] [11].

This dissertation focuses on translating messages between heterogeneous classical agents within a Multi-Agent System (MAS). These agents behave deterministically under identical conditions and rely on predefined logic, without learning or adaptation. To support this translation, it is essential to understand how agent behavior is defined in Agent-Oriented Programming (AOP). The next section outlines key AOP paradigms that describe how agents act, decide, and communicate—providing the foundation for analyzing such classical agents.

2.2 Agent-Oriented Programming Approaches

To extract message APIs from agent behavior definitions, it is important to first understand how agent behavior can be defined. In real-world systems, this behavior can take many forms, which is referred to as Agent-Oriented Programming (AOP). AOP includes different approaches for designing and implementing intelligent agents and multi-agent systems. Each approach defines how an agent behaves, how it manages knowledge, and how it interacts with its environment. These differences lead to unique strengths and make each approach suitable for different types of applications. From a scientific perspective, studying these methods shows a trend toward more flexible, clear, and strong software design practices for building complex and distributed systems.

Rule-based approaches describe agent behavior using a set of “if-then” rules. This clearly separates business logic from other parts of the software. These agents react when new information comes in or when their internal state changes. A big advantage is that rules can often be written in a way that’s easy to read and understand, even by non-programmers. This makes updating or reusing parts of the system much easier. Instead of rewriting whole programs, it is often enough to change a few rules. This is very helpful for building systems that need to adapt quickly. Rule-based AOP is used in many areas such as finance, telecom, and e-government, and also in distributed systems like Grid and Cloud computing. Technologies like Rql, Jess, CLIPS, PyKnow and Drools are popular tools in this category [12] [13].

Imperative or procedural approaches define agent behavior with step-by-step instructions using regular programming languages. This method is important for making agents that interact directly with hardware or need very precise control. It allows developers to fine-tune performance and integrate agents with other systems. Well-known examples of such tools include SPADE and JADE, which provide frameworks for building agents using imperative programming concepts. This style can also be mixed with rule-based methods, creating hybrid agents that use both logic and direct commands. For example, PySB uses Python to model biological systems, while CArtAgO lets agents interact with their environments in Java. These tools show how imperative programming supports flexible and reusable agent systems [12] [14] [15] [16].

Finite State Machine (FSM) approaches use a simple structure where agents move between defined states based on inputs. Each state has specific actions, and transitions depend on certain conditions. This setup is fast and easy to understand, making it ideal for situations where quick reactions are more important than complex thinking. FSM-based systems are especially useful in environments that change rapidly. Well-known implementations of FSM concepts include JADEX and Transitions, which support state-based behavior modeling for reactive agents [12].

Belief-Desire-Intention (BDI) models give agents a way to think more like humans. These agents have beliefs (what they know), desires (what they want to achieve), and intentions (the plans they commit to). This model helps agents reason about their goals and actions in a logical way. BDI agents are easy to understand and very useful in social or dynamic environments where behavior needs to change based on new situations. They can stick to their goals but also adjust if needed. Examples of BDI systems include PRS, AgentSpeak(L), Jason, JACK, and Jadex, which are used in fields like transportation, logistics, and smart environments [17] [12].

Declarative or logic-based approaches describe agent behavior using facts and logical rules, focusing on what the agent should know or do, not how it does it. This makes the system more flexible and easier to maintain. Changes can be made by updating the rules without touching the core code. These systems are powerful for reasoning and can check consistency, infer new facts, and make smart decisions. They are especially useful in smart environments and knowledge-based systems. Tools like Rql, Prolog, GOLOG, Datalog, and Description Logics are used here. They support intelligent features like ontologies and reasoning, as seen in AgentSpeak-DL [17] [13] [12].

Workflow-based approaches organize agent behavior into clear steps or tasks, often controlled by rules. This makes it easier to manage complex actions across multiple agents. Workflows can adjust based on changing situations, which helps with flexibility and control. These systems are used in areas like Grid and Cloud computing to manage resources, schedule tasks, and ensure services meet quality goals. SiLK, Camunda an JADE flows are such tools, allowing workflows to be written as rules and then executed by inference engines. This approach helps automate complex, distributed processes [12].

Ontology-based approaches use formal models (ontologies) to define shared knowledge and concepts in a system. These models help agents understand each other and work together, even if they were built separately. Ontologies describe things, their properties, and how they relate to each other. This supports better communication, reasoning, and decision-making. Ontologies are used in smart systems to manage context and support flexible behavior. For example, the CoBrA system uses ontologies to describe people and places in smart environments. Tools like OntoMAS and Onto2JaCaMo show how ontologies can help design and generate agent systems automatically [12] [17] [16].

The following table provides an **overview** of all approaches:

Approach	**Description Type**	**Example Systems**	**Strong in...**
Rule-based	IF-THEN rules	Rql, Drools, CLIPS, PyKnow	Reasoning, decision-making
Imperative	Functional logic	SPADE, JADE	Full control, scripting
FSM (Finite State Machine)	State transitions	JADEX, Transitions	Reactive systems
BDI (Belief-Desire-Intention)	Cognitive model	Jason, JADEX	Intention-based planning
Declarative/Logical	Logical inference	Prolog, GOLOG	Planning, semantics
Workflow-based	Process/step plans	Camunda, JADE flows	Business and service workflows
Ontology-based	Semantic knowledge	CoBrA, OntoMAS, Onto2JaCaMo	Interoperability, web agents

Table 2.1: Comparison of Agent Oriented Programming (AOP) Approaches

Having reviewed the various approaches to defining agent behavior, it becomes clear that multiple, fundamentally different paradigms exist. For the purposes of implementation and further analysis, the focus will be placed on agents defined imperatively, as the chosen implementation is based on the PEAK framework.

2.3 Communication in MAS

Effective communication is crucial in MAS. Redundant or unnecessary communication may increase computational costs and system instability. The focus now shifts to the communication aspect within a Multi-Agent System (MAS). First, various communication approaches used in MAS environments are explored. Subsequently, particular attention is given to Agent Communication Languages (ACLs), as they represent the most prominent and widely adopted method. [9] [11]

Communication approaches
Communication between agents can take different forms, each with its own approach and characteristics. One such approach is based on speech acts where statements are used to change the environment by creating new conditions or commitments. In this model, agents act as speakers whose messages aim to influence the beliefs of other agents. Another common communication method is message passing, where agents exchange structured messages directly. This can occur either point-to-point, where messages are sent between specific known agents, or by broadcasting messages to all neighboring agents. For effective communication, agents must share a common message format and structure [11].

A different communication approach uses blackboard systems, where agents share information by writing to and reading from a common memory space known as a blackboard. A control component oversees the blackboard and notifies agents about relevant updates. Although powerful and useful for coordination, this method introduces a risk because the blackboard can become a single point of failure if it becomes unavailable [11] [9].

Agent Communication Languages (ACL)
To ensure that agents understand each other clearly, especially in systems where agents are different from each other, an Agent Communication Language (ACL) is used. An ACL

provides a standard format for messages and a shared set of terms and meanings called an ontology. This ontology helps keep the meaning of messages the same across different agents and situations by using agent-independent rules [11] [9].

There are two main ways to design an ACL. The procedural approach treats communication as sending procedural instructions using scripting languages like JAVA, TCL, AppleScript, or Telescript. However, this approach has drawbacks: it needs specific information about message recipients, which is often missing or incorrect. This can cause problems for agent performance, and combining shared scripts is difficult. Because of this, the procedural approach is usually not preferred for ACL design. The **declarative approach**, on the other hand, uses clear and precise statements to define facts, rules, and assumptions. These statements should be easy to understand without needing extra knowledge of complex languages and must be concise to reduce communication cost and avoid mistakes [11] [9].

An ACL usually has three main parts. First, the vocabulary defines the set of words and terms that agents use to communicate. Second, the inner language translates the message content into a logical form that all agents can understand. Examples are KIF (Knowledge Interchange Format), KRSL, and LOOM. KIF extends first-order logic and can describe data, rules, negations, and more. Third, the outer language handles the structure of communication so it works independently of syntax and ontologies. KQML (Knowledge Query and Manipulation Language) is a common example, with a layered design where the communication layer uses low-level packet messages, while agents use different internal languages that are interpreted locally [9].

Let us clarify the concept of agent communication and its components with a simple example in which an agent sends a request to retrieve the geolocation of "lax" [9]:

```
(ask  :Content (geolocation lax(?long ?lat))
:language KIF
:ontology STD_GEO
:from location_agent
:to location_server
:label Query- "Query identifier")
```

The responding agent returns the geolocation information as follows [9]:

```
(tell :content "geolocation(lax, [55.33,45.56])"
:language standard_prolog
:ontology STD_GEO)
```

This example can be analyzed by looking at three key components of agent communication. First, the **outer language** used here is KQML (Knowledge Query and Manipulation Language). KQML defines the message format and communication protocol between agents using a layered structure. The communication layer, which is low-level and packet-oriented, contains metadata such as the sender (`:from location_agent`), receiver (`:to location_server`), and a query label (`:label Query- "Query identifier"`). The message layer specifies the type of communicative act, for example, `ask` for the request in the first message and `tell` for the reply in the second message.

Second, the **inner language** represents the logical content of the communication. In the first message, KIF (`:language KIF`) is used, which extends first-order predicate calculus. The content (`geolocation lax(?long ?lat)`) expresses a logical query for the geolocation of "lax". In the reply, the responding agent uses standard Prolog (`:language` `standard_prolog`) to represent the content as "`geolocation(lax, [55.33,45.56])`". This shows that agents can use different internal languages and that interpretation can be handled locally. Inner languages ensure that message content is concise, clear, and context-aware.

Third, the **vocabulary or ontology** defines a shared set of terms used by the agents. Both messages refer to the same ontology, `:ontology STD_GEO`, which includes terms such as `geolocation`, `lax`, `location_agent`, and `location_server`. A well-designed ACL supports extensible vocabularies and multiple ontologies depending on the application domain. This shared ontology guarantees semantic consistency and enables agents to interpret terms independently of each other.

This example clearly illustrates the roles of the ACL components: KQML structures the message format and communication protocol, KIF and Prolog act as inner languages expressing the logical content of messages, and the STD_GEO ontology ensures consistent interpretation of terms across agents [9].

Despite the standardization provided by an Agent Communication Language (ACL), payload translation remains an important challenge because even if agents use the same outer language and share an ontology, differences can still exist in how they represent and structure the actual message content. Payload translation mainly occurs at the level of the **inner language** and the **ontology**, where the inner language encodes the logical content of the message, and differences here mean that agents might use different syntaxes or representations for similar concepts. Meanwhile, the ontology defines the vocabulary and relationships between terms, but agents may have different domain-specific extensions or interpretations. Therefore, to achieve correct and meaningful communication, a translation mechanism is needed that can interpret and convert message content between the internal languages and adapt the payload according to the specific ontologies used by the communicating agents. This translation ensures that the same information is understood consistently by heterogeneous agents, even when they differ in their internal message representation or vocabulary usage [9].

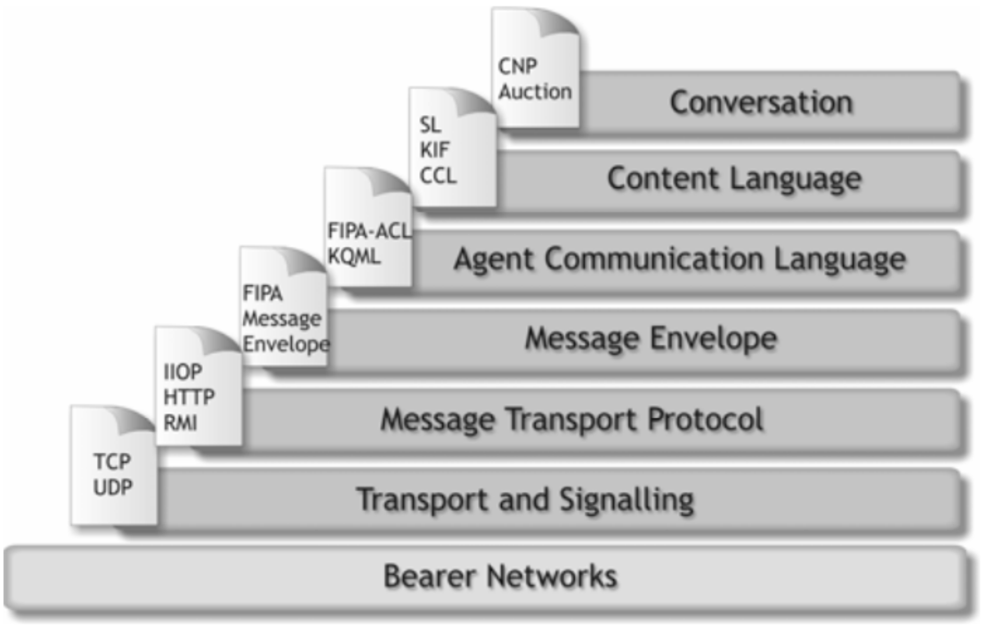


Figure 2.1: A layered model of agent communication [18]

2.4 Different Layers of Heterogeneity as a Challenge for Communication Between Agents

Before starting the design of the implementation, it is important to first understand the broader environment of the problem. Looking beyond the specific scope of this dissertation and considering other types of heterogeneity helps to understand the full complexity in which the solution will operate. This broader view supports better design decisions, helps to identify potential risks, and ensures that the solution remains useful in real-world multi-agent systems. Even though the focus is limited, being aware of challenges outside this scope gives valuable context and makes the system more future-proof and adaptable to changing environments.

Communication among agents in Multi-Agent Systems (MASs) can exhibit heterogeneity across several dimensions, which presents significant challenges to effective interaction and collaboration. This heterogeneity arises from various components within the MAS and impacts the ability of agents to exchange knowledge and coordinate actions. Conversely, specific components can be employed to mitigate heterogeneity [19].

Sources of Communication Heterogeneity
When designing complex multi-agent systems, communication between different agents often faces significant challenges due to various layers of heterogeneity. To understand these obstacles, we can imagine peeling an onion, starting from the broadest, most external layers of abstraction and moving inward to the most granular details of message content and interaction [19].

Although the main focus of this work is on understanding heterogeneity within the **payload**—the actual message content exchanged between agents—it is important to first explore the broader communication landscape in which this problem is situated. Looking beyond the payload, at platform, protocol, and language differences, helps build a more complete understanding of how real-world agent systems function and fail. Even though some of these aspects fall outside the scope of the direct solution, they play a crucial role in shaping the problem space. By gaining this full perspective, more realistic and effective *design decisions* can be made, ensuring that payload translation mechanisms align with the broader communication environment and remain applicable in open and heterogeneous systems[19].

At the outermost layer, we encounter **Platform Heterogeneity**. This encompasses the underlying computer architectural configurations and operating systems on which agents are built, along with the various Message Transport Protocols (MTPs), such as HTTP, Java-RMI, or SOAP, which manage the physical transmission of data. Even when these protocols share similar high-level designs, their technical details for sending and receiving messages can vary, creating significant barriers to direct communication. Furthermore, the choice of development environment (e.g., programming language, specific agent framework) during an agent's creation inherently restricts its available communication methods and adherence to broader standards, forming a foundational challenge for interoperability [19].

Moving inward, the next layer is **Agent Communication Language (ACL) Heterogeneity**. This layer dictates the basic grammar and structure agents use to exchange information in peer-to-peer interactions. It presents challenges because there can be differences in the fundamental message structure and the set of supported performatives (the specific types of communicative acts an agent can execute, like informing, requesting, or querying). Even if agents aim to conform to well-known standards such as KQML or FIPA-ACL, their specific implementations may deviate or they may introduce custom performatives, leading to potential misinterpretations of the intended communicative act. While these performatives frame the message's purpose, they are distinct from the actual data conveyed [19] [7].

At this point, we are delving into the abstraction layers that are directly related to the actual **payload** of the message being sent by an agent, focusing on its structure, meaning, and the conversational context [19].

The **content language** used within an ACL represents a more refined aspect of this layer, as it specifies the actual language in which the message's core information, the payload, is encoded and described. A significant heterogeneity problem arises when agents utilize different content representation languages (for example, FIPA SL, RDF CL, or KIF CL), because their encoding rules become incompatible with the receiver's decoding rules. This directly affects the ability to understand the raw information contained within the message body, necessitating complex efforts, such as comparing grammars and developing translators, to enable agents to accurately decipher the message's content. This layer fundamentally defines the structure of the payload itself [19] [11].

Further inward, we encounter **Ontology Heterogeneity**, which is a critical challenge affecting the shared meaning of the communicated information that constitutes the payload. An ontology fundamentally defines the terms, relations, and rules that make up the vocabulary of a specific topic, essentially representing an agent's conceptual understanding of a particular domain. When agents operate with different ontologies, even within the same application domain, they can exhibit discrepancies at the conceptual level, hierarchical level, and in the semantic relationships between concepts. This makes it difficult to correctly interpret the intrinsic meaning of the data contained within the message payload itself, hindering true semantic understanding [19] [11].

Finally, at what can be considered the innermost layer of communication challenges impacting the payload's utility, we have **Interaction Protocol Heterogeneity.** This layer specifies the rules and the expected sequences of messages that govern a coherent conversation between agents. It dictates how agents coordinate their actions over time to achieve a common objective, influencing the flow and progression of their collaborative efforts. Heterogeneity here can stem from differences in the type of dialogue expected (e.g., negotiation, inquiry, persuasion), the representation language used to formally describe these protocols (such as Petri Nets or Finite State Machines), and the underlying control structures that shape the conversational turns. Even if agents can understand the individual messages and their semantic content (the payload), a mismatch in these protocols can lead to coordination breakdowns, as they might anticipate different responses or follow divergent conversational patterns, ultimately impeding the effective use and contextual interpretation of the payload within an ongoing dialogue [19].

For the implementation, the focus will be on **addressing content language heterogeneity and ontology heterogeneity**. These relate directly to the problems mentioned in the introduction: content language heterogeneity corresponds to the **syntactic discrepancy problem**, while ontology heterogeneity corresponds to the **semantic ambiguity problem**. Interaction protocol heterogeneity falls outside the scope of this dissertation. It will therefore be assumed that both agents share the same interaction protocol, meaning they expect the messages being translated and are able to process them. If responses are sent, it is assumed that these are also expected by the original sender, and that both agents can handle the same sequence of message exchanges.

2.5 Large Language Models (LLMs)

Large Language Models (LLMs) possess semantic knowledge and a strong sense of linguistic structure, which enables the semantic mappings required for protocol and data format translation. However, a key question arises: to what extent can we fully trust an LLM to operate autonomously in this context? In other words, is the model consistently accurate when mapping file structures and domain-specific terminology?

Another important consideration is the determinism of LLMs. Are their outputs consistent given identical inputs, or is there inherent variability in their responses? Understanding and evaluating this determinism is crucial for deploying LLMs in interoperability-critical applications, where consistent and predictable behavior is essential.

Finally, have there already been models or systems proposed that explicitly perform syntactic and semantic schema mappings using LLMs? If so, how successful have these systems been, particularly when dealing with complex mapping scenarios such as 1-to-many or many-to-many relationships? These questions guide the following review of existing approaches, including models such as Magneto and ReMatch.

2.5.1 Overview of Large Language Models (LLMs)

Definition and Capabilities

Large Language Models (LLMs) are a category of neural network-based models characterized by having billions of parameters. Typically, LLMs refer to transformer-based language models trained on vast amounts of (mostly unlabeled) text data using self-supervised learning [20] [21].

LLMs exhibit exceptional performance across a range of Natural Language Processing (NLP) tasks. They demonstrate a strong understanding of natural language and can generate coherent, contextually appropriate responses. Their applications include, but are not limited to, text synthesis, translation, summarization, question answering, and sentiment analysis. Once trained, LLMs can be utilized through well-designed prompting strategies for various tasks [20] [21].

LLMs have significantly impacted AI by being applied in domains such as education, communication, content creation, scientific research, healthcare, and entertainment. They also support classical NLP tasks like word-level prediction, sequence tagging, relation extraction, and sentence generation [20] [21].

Breakthrough Compared to Earlier Models

Language modeling has evolved since the 1940s. Earlier statistical approaches like n-gram models suffered from limitations in capturing long-term dependencies and contextual information. These were succeeded by neural language models [20] [21].

Statistical Language Models (SLMs) relied on the Markov assumption and were widely used in the 1990s. However, they struggled with high-dimensionality problems when estimating higher-order probabilities [20] [21].

The introduction of the Transformer architecture marked a significant advancement. Pre-trained Language Models (PLMs) based on Transformers showed remarkable success across NLP tasks. A major breakthrough occurred when researchers discovered that scaling up model parameters led to enhanced capabilities [20] [21].

When LLMs exceed a certain parameter threshold (e.g., tens or hundreds of billions), they exhibit emergent abilities, such as in-context learning and reasoning. These capabilities, absent in smaller models like BERT, remain only partially understood but mark a transformative leap in AI language processing [20].

Training Process and Key Factors

The training process of Large Language Models (LLMs) typically consists of two main stages. During the pre-training phase, the model is exposed to large and diverse text corpora, allowing it to learn the structure, semantics, and patterns of natural language. This phase builds the foundation for general language understanding and text generation. After pre-training, a post-training phase follows, which includes techniques such as instruction tuning and alignment tuning. These methods help refine the model's behavior for specific tasks and align its outputs more closely with human expectations [20].

Several critical factors influence the success of LLM training. First, the quality and diversity of the pre-training corpus play a key role in the model's ability to generalize across different topics and domains. Common data sources include websites, books, conversations, scientific literature, and code. Second, the model architecture is essential, with the Transformer architecture being the most widely used due to its support for parallel computation and scalability. Third, given the immense scale of modern models, efficient optimization techniques are needed to speed up training while maintaining performance [20].

Another important aspect is the scale of model parameters. Larger models have been observed to perform better and even exhibit emergent capabilities that smaller models do not.

Finally, training such large-scale models requires powerful hardware infrastructure. This includes specialized computing clusters and significant energy resources, making LLM training both technically demanding and costly [20, 21].

Technical Foundations: Language Management and Intuition

LLMs manage and "understand" language through several core technical mechanisms [21].

First,**tokenization** is a fundamental preprocessing step that converts raw text into smaller units called tokens. These tokens may represent characters, subwords, or full words, depending on the chosen algorithm, such as WordPiece, Unigram Language Model (UnigramLM), or Byte Pair Encoding (BPE). Tokenization enables the model to process text numerically, forming the essential bridge between textual data and neural computations.

At the heart of most LLMs is the **Transformer architecture**, which allows for efficient parallel processing of sequences using self-attention mechanisms. This design contrasts with earlier sequential models like Recurrent Neural Networks (RNNs) and enables the modeling of long-range dependencies, which is crucial when working with large datasets and complex linguistic structures.

Attention mechanisms, especially self-attention, play a central role in the architecture by letting the model determine relationships between all tokens in an input sequence. This mechanism assigns contextual importance to each token based on its relevance to others, allowing the model to construct rich, context-aware representations of language. This capability is a key factor in the model's understanding of meaning and nuance.

During the pre-training phase, LLMs are optimized using unsupervised objectives such as **causal language modeling**—predicting the next token given prior context—or masked language modeling (MLM), where certain tokens are hidden, and the model must infer them. These objectives train the model to recognize patterns, grammar, and semantic relationships across vast and diverse corpora, building a deep reservoir of language knowledge.

Finally, LLMs use **word embeddings** to represent words as vectors in a continuous space. These embeddings capture both syntactic and semantic properties, with modern models generating contextualized embeddings—dynamic representations that adjust depending on the word's surrounding context. This allows for fine-grained interpretation of meaning, contributing to what is often described as the model's intuitive grasp of language [22].

Together, these mechanisms enable LLMs to perform sophisticated natural language processing tasks by deeply modeling the structural and semantic complexity of human language [20].

2.5.2 Semantic Mapping

Large Language Models (LLMs) possess advanced semantic understanding and a strong grasp of linguistic structure, enabling them to perform the semantic mappings required for protocol and data format translation. Several studies have demonstrated their capabilities in this domain [23].

Magneto, for instance, leverages LLMs to generate syntactically diverse training data for fine-tuning smaller language models (SLMs). This process exploits the LLMs' understanding of semantic equivalence despite syntactic variation. LLMs are also employed as rerankers to refine candidate matches identified by SLMs, highlighting their ability to assess subtle semantic nuances. Carefully crafted prompts enable LLMs to evaluate and rank attribute matches. Magneto's success in achieving high accuracy across domains suggests that LLMs are effective at identifying semantic correspondences [23].

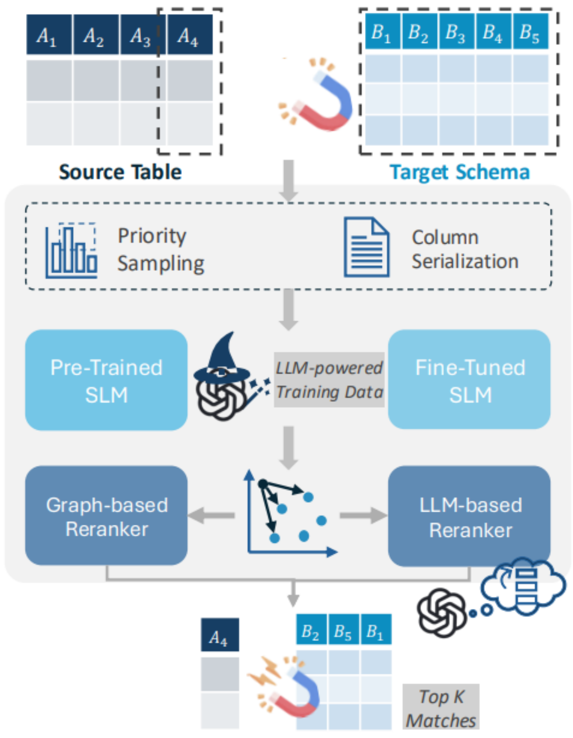


Figure 2.2: Magneto overview. Given source and target tables, the framework identifies matches in two phases: A small language model, automatically fine-tuned using LLM-generated data, retrieves and ranks candidate matches, which are then assessed and reranked by an LLM. This process makes it possible to efficiently identify matches and generalize across domains and datasets. Different schema matching strategies can be derived by adopting different combinations of methods for these phases [23].

ReMatch introduces a retrieval-enhanced approach to schema matching using LLMs without requiring model training or access to the source database. It constructs passage-based representations of source and target attributes and tables, then uses embedding models to assess semantic similarity and retrieve relevant candidate tables. Passage-based representations refer to structuring information as coherent text segments or paragraphs that provide rich contextual descriptions, enabling more effective semantic understanding by language models. An LLM subsequently ranks attribute matches using the contextual information encoded in these document representations. ReMatch's effectiveness on large, real-world schemas underscores the ability of LLMs to perform semantic mapping by leveraging existing knowledge [24].

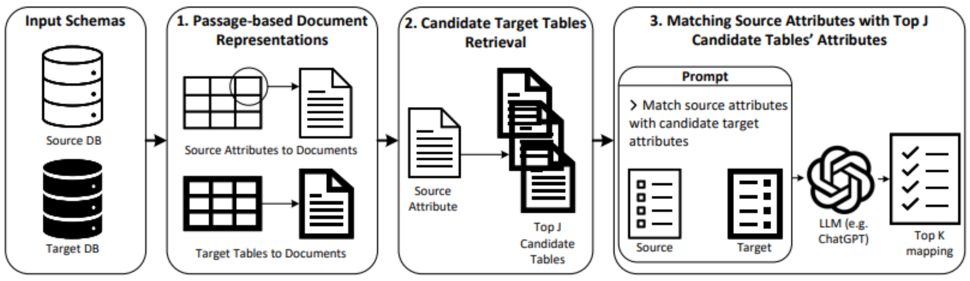

Input Schemas
Source DB
Target DB
1. Passage-based Document Representations
Source Attributes to Documents
Target Tables to Documents
2. Candidate Target Tables Retrieval
Source Attribute
Top J Candidate Tables
3. Matching Source Attributes with Top J Candidate Tables' Attributes
Prompt
> Match source attributes with candidate target attributes
Source
Target
LLM (e.g. ChatGPT)
Top K mapping

Figure 2.3: Overview of the ReMatch method [24].

In addition, another study explores various **prompting strategies** ("task scopes") to deploy LLMs for schema matching based solely on attribute names and descriptions. The results indicate that LLMs can detect semantic correspondences, often outperforming string similarity baselines. However, the performance strongly depends on the quality and quantity of contextual information in the prompts. This study shows that LLMs have the potential to accelerate schema matching tasks using only schema documentation [25].

Despite the promising results of using large language models (LLMs) for schema matching, fully relying on them to operate autonomously remains a complex and open question. Several limitations have been identified in the literature that affect the reliability and consistency of LLM-based approaches [25].

One major concern is the inconsistent accuracy of LLMs across different tasks and versions. While more advanced models such as GPT-4 generally perform better, their accuracy is sensitive to both the size of the task scope and the amount of context provided. Too little context can lead to insufficient understanding, while too much can overwhelm the model or distract from relevant features [25].

Another issue is the variability in how decisively LLMs make decisions. Rather than providing clear binary matches for attribute pairs, models often produce uncertain or vague responses. The decisiveness score proposed by Parciak et al. shows that this confidence level depends on how the task is framed and how complex the prompt is [25].

In addition, while LLMs are generally considered consistent enough for practical use, variations in performance are still reported. This is especially evident in tasks that involve many-to-many (N-to-M) mappings, where standard deviations in F1-scores, precision, and recall indicate some instability in the results [25].

LLMs are also prone to certain reasoning flaws, such as focusing too heavily on surface-level name similarities. This can lead to incorrect mappings when attribute names appear similar but actually refer to different concepts, especially when the model fails to consider supporting documentation or attribute intent [25].

Finally, most current research and evaluations center on simple one-to-one matches. Although some systems, like Magneto and ReMatch, have shown potential for more complex mappings (such as one-to-many or many-to-many), enabling LLMs to handle these automatically without human guidance remains a significant challenge [24].

Conclusion: Semantic Mapping

In conclusion, LLMs exhibit the semantic knowledge and linguistic competence required for data format translation and schema matching. They are capable of understanding context and identifying deeper semantic relations beyond surface-level similarities. Nevertheless, relying entirely on LLMs for this task is currently premature. Accuracy, consistency, and robustness can vary depending on context and task formulation, and LLMs are prone to certain types of systematic errors. As such, human oversight and validation remain crucial. Combining LLMs with complementary approaches—such as reranking and prompt engineering, as seen in Magneto—appears to be a promising path for balancing the strengths and limitations of LLMs in semantic mapping tasks.

2.5.3 Consistency and Variability in LLM Outputs

Recent research has investigated the consistency and variability of large language models (LLMs) under different prompting and decoding conditions.

Controllability of LLMs

The study titled *A Control Theory of LLMs* conceptualizes LLMs as discrete stochastic dynamical systems, where prompts act as control inputs. A central notion is the *reachable set*—the collection of all outputs that can be produced from a given initial state and prompt. Empirical evidence demonstrates that for over 97% of tokens in the Wikitext dataset, the correct next token is reachable using prompts of up to 10 tokens. Remarkably, even low-probability tokens can become the most likely outputs with prompts as short as four tokens, indicating a high degree of controllability [26].

The study also introduces the metric of *k-ϵ controllability*, which evaluates how well a model can be guided towards specific outputs within a prompt length limit. This framework reveals that token reachability is not solely dependent on the initial probability assigned by the model. The Self-Attention Control Theorem provides a condition for output unreachability, suggesting inherent limitations in generating certain outputs. Moreover, a log-linear relationship between prompt length and the fraction of unreachable outputs underscores the nuanced nature of LLM behavior [26].

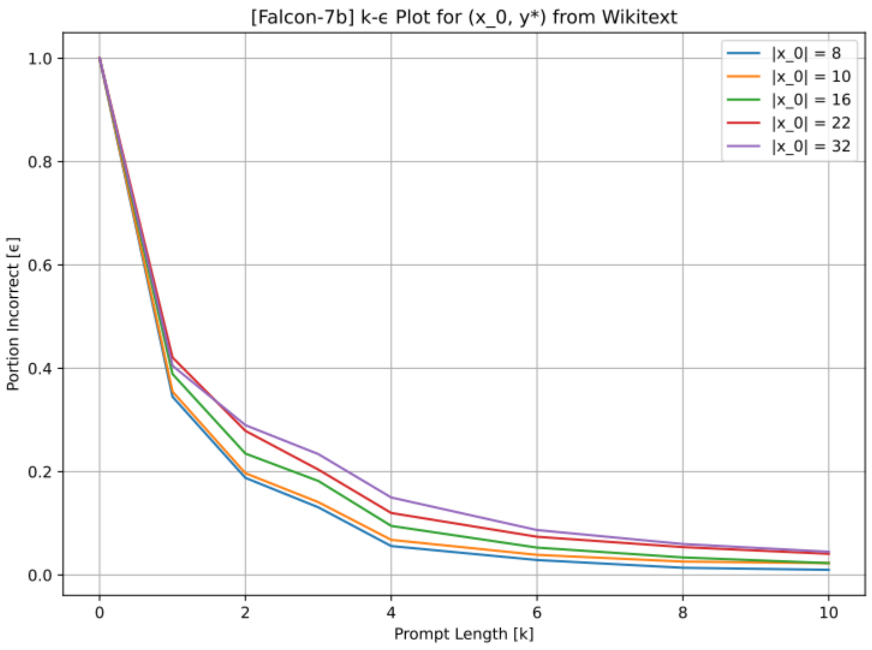


Figure 2.4: k-ϵ values on initial state $\mathbf{x}_0$ and target output token y^* from Wikitext. 97.16% of the instances were solved with a prompt of length $k \leq 10$ [26].

Non-Determinism in LLM Evaluations

In *Evaluation of LLMs Should Not Ignore Non-Determinism*, the authors highlight a critical oversight in current evaluation practices: the assumption of determinism. LLMs can produce varied outputs for the same input, primarily depending on the decoding strategy employed. Greedy decoding yields deterministic outputs by selecting the most probable next token, while sampling introduces variability by drawing from a probability distribution [27].

Across multiple benchmarks, greedy decoding generally outperforms sampling, except in contexts like AlpacaEval. Tasks with constrained output spaces, such as MMLU and Mix-Eval, exhibit consistent performance across decoding methods. In contrast, tasks involving mathematical reasoning (GSM8K) and code generation (HumanEval) are highly sensitive to sampling variability, with performance gaps exceeding 10 points. Techniques like Direct Preference Optimization (DPO) can mitigate some of this variance. However, increasing sampling temperature significantly degrades reasoning and coding capabilities. Interestingly, larger models do not consistently exhibit lower sampling variability compared to smaller counterparts [27].

Performance in Compound AI Systems

The work *Towards the Scaling Properties of Compound AI Systems* explores systems that combine the outputs of multiple LLM invocations to improve answer quality [28]. Two main architectures are examined in this context. The first, called Vote, involves querying the LLM multiple times with the same input and then selecting the most common output using majority voting. This technique is similar to Google's CoT@32 approach. The second, known as Filter-Vote, adds an extra step before voting: the outputs are first evaluated using predefined criteria, and only those that pass the filter are included in the voting stage. This strategy is used in systems like AlphaCode 2 [28].

A central insight from the study is that making more LLM calls does not always lead to better results. Both the Vote and Filter-Vote architectures show non-monotonic performance: initially, accuracy improves as more outputs are aggregated, but after a certain point, adding more responses causes performance to decline. This behavior is linked to the concept of *query difficulty*, which determines how beneficial or harmful repeated querying can be in different scenarios [28].

The authors categorize questions into two types: "easy" and "hard" [28]. For easy queries, making additional model calls generally improves performance because majority voting increases the chance of selecting the correct answer. In contrast, for hard queries—where incorrect answers are more likely at the outset—calling the model more often tends to reinforce these incorrect outputs. As a result, performance on hard queries actually worsens with additional calls [28].

When a dataset contains a mix of both easy and hard queries, this difference in behavior creates an interesting pattern in the overall performance curve. Initially, aggregate performance improves due to the gains on easy queries. However, as the number of model calls increases further, the negative impact on hard queries begins to outweigh these benefits. This leads to a turning point where performance peaks and then starts to decline [28].

Importantly, even when individual model calls are deterministic (e.g., via greedy decoding), the overall system output may still vary. This is due to the fact that small differences in the set of outputs across repeated runs can lead to different majority outcomes, especially when model output margins are small or the queries are complex. Therefore, the consistency of compound systems depends not only on the base model but also on the aggregation dynamics [28].

To address these challenges, Chen et al. developed an analytical scaling model that predicts the performance of Vote and Filter-Vote systems based on the number of LLM invocations and the difficulty distribution of queries. This model can guide the selection of the optimal number of model calls to maximize overall performance [28].

Conclusion: Consistency and Variability in LLM Outputs

LLMs do not produce entirely consistent outputs for identical inputs. Output variability stems from factors such as the decoding method, sampling parameters, prompt structure, and task complexity. Greedy decoding offers deterministic results, while sampling introduces significant variability, particularly harmful in tasks requiring precise reasoning or coding.

The control-theoretic view reveals that LLMs are highly steerable via prompting, yet not all outputs are guaranteed to be reachable. Prompt length and token probability are important but not sole determinants of reachability.

Compound AI systems that aggregate multiple LLM outputs further complicate consistency. Their performance can follow non-monotonic trends depending on the number of calls and the nature of the queries. Even with deterministic base models, aggregation methods such as majority voting can yield different results due to slight variations in the output set.

The study emphasize that understanding these dynamics is essential for effectively deploying compound LLM systems. Their analytical model serves as a practical tool to balance performance and cost by identifying the optimal number of LLM invocations. Overall, careful consideration of non-determinism, controllability, and aggregation effects is crucial for building reliable and efficient LLM-based applications.

2.6 Integration of Outer Language Translation with the Rule Responder System

Rule Responder is an infrastructure designed to enable effective translation and interoperability among heterogeneous, rule-based agents. This interoperability is achieved by using Reaction RuleML as a standardized, platform-independent exchange format for communicating rules, events, actions, and queries. To support this, dedicated translation services perform bidirectional conversion between internal representations of rule engines—such as Prova, OO jDREW, Euler, or Drools—and the Reaction RuleML format. As a result, Rule Responder allows for distributed communication and collaboration across diverse agent platforms [29].

This study focuses on analyzing the translation processes involved in both the outer language and the payload, with particular attention to how a newly proposed payload translation implementation could enhance the overall framework. The investigation begins by examining the decoupling mechanism between the outer language—which includes transport protocols and agent communication languages (ACLs)—and the payload. This structural separation is critical for maintaining flexibility and modularity across agent components [29].

Subsequently, the methods used for translating the outer language are analyzed in more detail, including how Reaction RuleML encapsulates communication directives and integrates seamlessly with the Enterprise Service Bus (ESB) to support various protocols. This is followed by an in-depth exploration of how payload translation is currently handled, identifying key limitations in expressiveness and fidelity when mapping platform-specific rule content into the common format [29].

Finally, the potential integration of the new payload translation approach is discussed. By contributing translated payloads in a compatible format, this implementation could strengthen the existing architecture, offering improved robustness and precision. It may also provide a more dynamic and scalable solution for managing communication between agents within a distributed environment [29].

Decoupling of outerlanguage translation and payload translation

The Rule Responder architecture is designed using a layered structure that separates the *inner language*, which defines the agent's core rule logic and data handling, from the *outer language*, which relates to communication protocols and context. This separation is made possible by using Reaction RuleML as a universal message format, combined with translation services that handle the conversion between platform-specific languages and the shared format [29].

At the heart of this design is Reaction RuleML, which acts as an abstraction layer. It provides a neutral, platform-independent way of representing message content, making it easier to keep internal logic and external communication separate. The format of Reaction RuleML messages makes this separation clear: it distinguishes between the actual content of the message (the payload) and the instructions for how the message should be processed (the directive and metadata). This makes it easier to manage both what the message says and how it should be delivered or acted upon [29].

Translation services play a key role in maintaining this separation. They convert internal message formats used by specific rule engines into the standard Reaction RuleML structure, and vice versa. This allows agents using different technologies to communicate without needing to understand each other's internal logic directly [29].

In addition, the Enterprise Service Bus (ESB) functions as the backbone of communication in the system. It handles the actual transport of messages using a variety of supported protocols, such as HTTP or JMS. Because the ESB is responsible for this transport layer, individual agents don't need to manage low-level communication details. This further reinforces the separation between message content and the transport mechanism [29].

Together, these layers allow Rule Responder to support complex, distributed systems made up of many different types of agents. By clearly separating internal reasoning from external communication, the architecture makes it easier to scale, maintain, and coordinate agents within a virtual organization [29].

Outer Language Translation Capabilities

The term *outer language* refers to the way agents and services communicate with each other in a distributed system. Within the Rule Responder framework, several important features are supported to enable effective translation and interoperability between different systems and technologies [29].

Communication between agents is handled using an Enterprise Service Bus (ESB), such as Mule ESB, which serves as the middleware for message transport. This ESB supports over 40 different communication protocols, including both synchronous and asynchronous types like SMTP, JDBC, TCP, HTTP, and XMPP. Thanks to this wide range of supported protocols, agents in the Rule Responder system can easily exchange messages with external services, regardless of the specific transport technology being used [29].

To ensure that agents understand each other despite differences in internal logic or platform, Rule Responder uses Reaction RuleML as a common, platform-independent message format. This format includes elements of Agent Communication Languages (ACLs) as part of the message's pragmatic context. Each `<Message>` element contains specific attributes that guide how the message should be handled. One such attribute is `directive`, which provides a pragmatic instruction using well-known ACL primitives, such as `acl:query-ref` or `acl:inform-ref`. Additionally, a `Performative` field is used to define the message envelope, helping agents understand the intent and structure of the communication [29].

This structured design ensures that messages can be clearly and consistently translated between Reaction RuleML and any platform-specific formats, even as they pass through different transport layers. It allows for flexible yet controlled interactions between agents, maintaining interoperability across a diverse range of systems [29]. .

Payload Translation in Rule Responder Integration

Within the Rule Responder framework, payload translation is handled by dedicated translation services that convert between the specific syntax of different rule engines and the standardized Reaction RuleML format. This process enables agents built on different platforms to communicate effectively using a shared rule exchange language [29].

At the heart of each Rule Responder agent is a platform-specific rule engine, such as Prova, OO jDREW, Euler, or Drools. These engines are responsible for the core decision-making and reaction logic. However, they differ significantly in how they handle rule types, represent state, and resolve conflicts. Because of this diversity, a common translation mechanism is required to allow consistent interaction across different agent implementations [29].

To enable this interaction, Reaction RuleML is used as the shared communication format. This standardized language supports the exchange of various elements, including rules, actions, events, queries, and data. Regardless of which rule engine an agent uses internally, Reaction RuleML acts as the bridge between systems, allowing them to understand and react to shared content [29].

Translation services play a central role in this architecture. When a Reaction RuleML message arrives at an agent, it is converted into the local rule engine's platform-specific syntax so it can be executed. Similarly, when the engine produces results or sends out rule bases, these are translated back into Reaction RuleML before being transmitted over communication channels such as HTTP or JMS. This bidirectional translation ensures that agents can both receive and send understandable content, regardless of their internal logic format [29].

To perform these translations, the system uses technologies such as XSLT stylesheets and JAXB, which help transform structured XML content between formats. These translation services are embedded in the transport layer of each agent's connections and are integrated into the Enterprise Service Bus (ESB). The ESB acts as the central middleware that handles communication between all distributed agents in the system [29].

Limitations of a Standardized Format like Reaction RuleML

A key limitation of using a standardized format like Reaction RuleML is that it cannot fully represent all the details and unique features—referred to as "additional logical expressivity features"—that exist in different rule engines. This limitation can cause problems when translating messages or rules between systems [29].

One of the main issues is that some specific functionalities of rule engines are difficult to express in a common format. For example, advanced type systems such as Prova's multi-sorted logic, Java's class hierarchies, or OWL/RDFS ontologies may not translate cleanly. Similarly, procedural features—like invoking Java objects dynamically or using built-in functions for SQL, SPARQL, or XQuery—are often too complex to be captured precisely in Reaction RuleML. Rule engines may also use advanced modular structures with scopes or guards, which are not always supported in a generalized standard [29].

Because Reaction RuleML is designed to work as a common language between different platforms, it must act as a baseline for communication. This requirement often forces the system to simplify or generalize specific features. As a result, certain language-specific details may be lost or only partially represented. This can lead to a loss of granularity or a less precise version of the original rule, which may affect how well agents understand or apply the translated message [29].

External Payload Translation and Donation in Rule Responder

It is possible to handle the translation of message content, or payload, outside the main system and then make this translated content available within the Rule Responder framework. This method supports the system's aim of using a shared, platform-independent format called Reaction RuleML for exchanging information [29].

In this approach, an external service is responsible for converting platform-specific message formats into the standardized Reaction RuleML format. This translation includes not only the core message content but also important communication details such as the sender and receiver. By doing this outside the main system, it becomes easier to ensure that all messages follow the same structure, regardless of their origin [29].

Once translated, these messages can be delivered directly into the Rule Responder system. From there, the system forwards the messages to the appropriate agents—software components that are designed to react to these kinds of messages. Because the translation has already been handled externally, the agents do not need to perform any additional conversion and can immediately act on the content they receive [29].

Even though the translation process happens outside the system, the translated content can still be shared with other agents. This can be done by publishing the messages or sending them directly. However, to ensure that all agents interpret the messages correctly, the system relies on shared semantic knowledge, such as ontologies. These shared semantics help maintain a common understanding between agents, even when the messages come from different platforms [29].

Potential Contribution of the Proposed Implementation

As identified in the limitations of payload translation, a standardized format is currently used for payload information. This approach relies on abstracting and converting data to the standardized format and back to the original payload. Despite multiple optimizations, these limitations persist. It is evident that a standardized format cannot capture all the nuances present in specific situations [29].

The objective of the implementation in this dissertation is to provide a dedicated translation between each pair of agents, thereby removing the need for a standardized format. The implementation presented in this dissertation serves as a preliminary demonstration that this methodology is feasible. Consequently, it may be possible to enhance the existing Rule Responder system by integrating the approach proposed here [29].

While this concept remains exploratory, there is potential for significant improvement through further research, optimizations, extensions, and ongoing advancements in large language model (LLM) technology. Such developments could contribute valuable enhancements to the implementation described in this dissertation [29].

2.7 Extracting Payload Information from imperatively programmed Agents

After implementing the system, it is important to evaluate whether existing methods can more effectively extract the message-sending and -receiving APIs in imperatively programmed agents. This step is crucial, as payload extraction is one of the model's four core components and is particularly prone to inconsistencies and failure. Its structural complexity and sensitivity make it a key point of vulnerability. To reduce these risks, this section explores alternative techniques that could support, enhance, or even replace the current method if they offer greater robustness or completeness. Being able to extract communication semantics directly from agent code can greatly improve monitoring, debugging, and interoperability between agents.

Understanding and extracting message structures, along with their associated vocabulary and data types, in imperatively declared multi-agent systems requires a detailed analysis of agent architecture, particularly their communication protocols, role definitions, and tool interfaces. Frameworks like **MetaGPT** demonstrate how explicitly defined **Standard Operating Procedures (SOPs)** and structured communication outputs can guide agents to produce reliable, formatted documents such as Product Requirements Documents (PRDs) and system design specifications. These documents often include well-defined fields like user stories, requirement pools, file lists, and data structures, each with predictable formats and data types, such as lists of strings or standardized path formats. The structured nature of these outputs allows for the identification of keys and expected values by examining the schema associated with each role's responsibility. MetaGPT further enforces structure through a shared message pool where agents publish and subscribe to messages, creating a centralized but filtered communication mechanism. This design ensures that agents only receive task-relevant information, preventing overload while maintaining efficient collaboration [30].

In addition to structured design, the use of predicates in imperative static analysis tools offers another way to uncover internal message logic. Tools like **PredicateFix** or GoInsight use predicates—Boolean expressions embedded in analysis rules—to evaluate properties of code fragments. In imperative environments, these may appear as 'if' conditions or specialized function calls that encapsulate expected code patterns. For instance, a predicate like `assignStmt(V, E, L)` might reflect an assignment operation, involving a variable, an expression, and a location, each representing a distinct element of the message structure. By analyzing such predicates, researchers can determine the kinds of structures the analyzer recognizes, the vocabulary it expects, and the conditions under which specific outputs are triggered. Since these rules are embedded in source code, **static analysis of the tools** themselves can expose the internal schema—key names, data types, and acceptable values—used for reasoning over program artifacts [31].

This structural clarity is also evident in the **Flow-of-Action framework**, designed for root cause analysis in microservices. Here, the tools available to agents are essentially predefined functions with clear input and output specifications. For example, a tool like `match_sop` receives a string indicating fault type or context and returns an integer representing an SOP number. Another tool, `generate_sop_code`, takes an SOP object and produces executable code, demonstrating how each function enforces specific schemas for message exchange. Even the SOPs themselves are structured as objects with a name and a sequence of steps, offering a reusable and interpretable template for agent interaction. Because each function or tool acts as an interface between agents and actions, examining their definitions reveals exactly what keys are used, what data types are expected, and what information is transmitted or transformed [32].

To improve the reliability and manageability of such systems, several strategies have emerged from recent research. The use of SOPs is a widely endorsed practice, as it injects human-like reasoning and standardization into agent workflows while **reducing hallucinations or incoherent outputs from language models**. MetaGPT highlights the advantage of using structured documents and diagrams, rather than free-form natural language, to reduce ambiguities during collaboration. Its publish-subscribe mechanism enhances scalability and precision, allowing agents to filter messages and focus only on relevant tasks. Beyond design patterns, control mechanisms play an essential role in operational robustness. For example, MetaGPT includes an executable feedback loop where the Engineer agent generates, runs, and iteratively debugs code, creating a self-correction mechanism that ensures outputs are not only syntactically but also functionally correct [30].

The Flow-of-Action framework introduces a related control model through the concept of **action sets**. Rather than acting immediately, agents first generate a set of reasoned possible actions, then choose the most appropriate one. This balances the stochastic nature of language models with the need for deterministic reasoning, allowing agents to operate under uncertainty while avoiding irrational behavior. Another fundamental principle in both MetaGPT and Flow-of-Action is specialization through **role-based collaboration**. In MetaGPT, roles like Product Manager, Architect, and Engineer divide tasks into manageable responsibilities, while in Flow-of-Action, roles such as MainAgent, CodeAgent, JudgeAgent, and ObAgent enforce checks and balances across the workflow. This division of labor reduces the cognitive burden on individual agents, enhances overall accuracy, and allows for internal cross-verification of outputs [30] [32].

Extracting message structures effectively depends on **access to specific inputs**. The most direct source is the agent source code, which contains the actual implementation of communication logic, state transitions, and tool interactions. Design documents or SOP specifications provide additional insights into the expected outputs and workflows. Equally important are the APIs and function signatures of tools used by agents, as they define exact data schemas—what input types are accepted and what outputs are produced. Profiles of agent roles, goals, and message formats are also crucial for mapping message flows and their structure. These resources collectively offer a complete picture of the communication landscape and allow for reverse engineering of internal message schemas [30] [32] [31].

Finally, the role of **Large Language Models** (LLMs) in these systems is central. MetaGPT, PredicateFix, and Flow-of-Action all rely on LLMs to generate structured outputs, whether they be documents, corrected code, or operational plans. In PredicateFix, models such as GPT-4o and Claude-3.5-Sonnet are used in conjunction with Retrieval-Augmented Generation (RAG) and curated examples to produce accurate bug fixes in a structured JSON format. In Flow-of-Action, LLMs convert SOPs into executable procedures and generate action plans. Despite their strengths, LLMs are known to hallucinate or misinterpret parameters, which reinforces the need for SOPs, structured interfaces, and robust control mechanisms to constrain behavior and improve reliability. The success of these agent systems lies not only in their individual components but in the rigor with which communication and data flow are defined and enforced [30] [32] [31].

By combining the techniques discussed with the current implementation for extracting message structures, along with the associated vocabulary and data types, the reliability, robustness, and consistency of the model can be significantly enhanced. A key strength of these implementations is their use of static code to guide, support, and control the Large Language Model (LLM). Static code is inherently more reliable and predictable, which is a major advantage for a system that aims to achieve consistency and reliability. A potential direction for further research would be to explore these methods in more detail, investigating whether they can be integrated or fully implemented to improve the overall system described in this dissertation.

2.8 Conclusion

O1 – Understand and describe the architecture of agent communication within MAS, including relevant protocols and abstraction layers. In a Multi-Agent System (MAS), agents communicate to share information and coordinate tasks. This is especially important when agents are built differently, such as in heterogeneous environments. Most MAS use Agent Communication Languages (ACLs) like KQML to standardize communication. An ACL separates messages into three layers. The *outer language* defines the structure of the message and includes metadata like sender and receiver. The *inner language* holds the logical content, such as a request or response, often written in formats like KIF or Prolog. The *ontology* provides shared meanings for terms, so agents interpret messages consistently.

For classical agents with fixed behavior, communication follows strict patterns based on predefined logic. Although they may use the same outer language, differences in message structure or ontology can still cause misunderstandings. This makes payload translation a critical task. Translation must handle differences in logic representation and vocabulary use. Understanding these communication layers and how agents encode content is essential for designing systems that can translate messages between different classical agents correctly and reliably.

O2 – Assess the determinism and consistency of LLMs when used in static, imperative environments. Large Language Models (LLMs) are not fully deterministic. Even with the same input, different outputs may occur depending on decoding methods. Greedy decoding produces consistent outputs by always choosing the most likely next token. In contrast, sampling methods introduce randomness, which can cause significant variability. This is especially problematic in environments that require stable and repeatable behavior, such as static, rule-based agent systems.

Studies show that while LLMs are highly controllable through prompt design, not all outputs are guaranteed to be reachable. This controllability depends on prompt length and other internal model dynamics, not only on token probabilities. Moreover, when LLMs are used in compound systems (e.g., voting or filtering multiple outputs), even deterministic models can lead to inconsistent overall outcomes due to variations in aggregation.

For use in imperative systems, LLMs should be used with strict constraints: fixed prompts, greedy decoding, and limited aggregation. Without such controls, LLM behavior may break the determinism expected in traditional software environments. The Gemini 2.5 Flash model used in the implementation in Chapter 4 applies greedy decoding by default when the decoding parameters are set to `"temperature": 0.0, "top_k": 1`. Fixed prompts will be used to keep the output as deterministic and consistent as possible [33].

Chapter 3

Used Materials

This chapter explains the tools and models used to build the communication translation system. It starts with the PEAK platform and the SPADE framework, which support communication between agents. Key features like message structure and metadata are discussed, as they are important for translation. The chapter also describes how different language models were used during development, and why the Gemini 2.5 Flash model was chosen for the final version.

3.1 The PEAK Platform and the SPADE Framework

For the development of the code, the PEAK framework was primarily used. This choice was motivated by collaboration with the GECAD research group at ISEP, who designed and actively maintain the platform. PEAK is a Python-based framework created to simplify the development, deployment, and management of Multi-Agent Systems (MAS). It supports building complex ecosystems where diverse agent communities can cooperate, share information, and coordinate tasks efficiently [34] [35].

PEAK is built on top of the SPADE framework, which uses the XMPP protocol to enable reliable communication between agents, even across different machines or networks. The framework includes features for monitoring agent behavior, analyzing their interactions, and managing their lifecycle, helping developers create robust and maintainable MAS applications [34] [35].

Using PEAK requires Python 3.9.6 or later and access to an XMPP server to handle agent communication. The framework's modular design makes it flexible and suitable for various MAS scenarios, from research prototypes to production systems. Comprehensive documentation and example code can be found on the official website: `https://www.gecad.isep.ipp.pt/peak` [34] [35]

PEAK is built on top of the **SPADE** (Smart Python Agent Development Environment) framework, which is a Python-based toolkit for developing multi-agent systems (MAS) that communicate via the XMPP protocol. SPADE provides an abstraction for agents, services, and communication mechanisms, enabling developers to implement autonomous and interactive agents. Its integration with XMPP ensures robust and asynchronous communication, making it suitable for dynamic, open, and heterogeneous environments. [34] [35] [36]

In SPADE, each agent is defined as a Python class that extends the `spade.agent.Agent` class. Agents contain one or more behaviours, which encapsulate the logic for handling tasks or processing messages. Behaviours are executed concurrently and can be scheduled according to periodic, cyclic, or one-shot strategies. [36]

Agents are uniquely identified by their JID (Jabber ID), a standard identifier in the XMPP protocol. For example [36]:

```
agent = MyAgent("example@domain")
```

Communication in SPADE is implemented using the XMPP protocol, which supports structured, asynchronous message passing. Messages in SPADE conform to the FIPA-ACL (Foundation for Intelligent Physical Agents - Agent Communication Language) message format, allowing rich semantic structures and metadata annotations. [36]

Messages are instances of the `spade.message.Message` class and contain three key components [36]:

- **Metadata:** Information such as the message's performative (e.g., `inform`, `request`). [36]
- **Body:** The main content of the message, often encoded as plain text or structured JSON. [36]
- **To/From fields:** The JID of the sender and recipient agents. [36]

Below is a simple example of how a SPADE agent constructs and sends a message:

```
from spade.message import Message

msg = Message(to="receiver@domain")     # Receiver JID
msg.set_metadata("performative", "inform")  # FIPA performative
msg.body = "{"Temperatur": 22}"        # Message body

await self.send(msg)
```

In this case, the message uses the `inform` performative to share factual information and the body contains the actual payload. [36]

To handle incoming messages, agents define behaviours with a message template. This template filters messages based on metadata, in this case based on the performative field of the metadata. [36]

```
from spade.behaviour import CyclicBehaviour
from spade.template import Template

class ReceiveBehaviour(CyclicBehaviour):
    async def run(self):
        msg = await self.receive(timeout=10)
        if msg:
            print(f"Received: {msg.body}")

template = Template()
template.set_metadata("performative", "inform")
agent.add_behaviour(ReceiveBehaviour(), template)
```

This allows agents to handle only relevant messages and ignore unrelated ones, which is crucial for semantic translation and payload extraction. [36]

For the context of this dissertation, the SPADE communication model presents several important characteristics. First, its structured messaging—with a clear separation between metadata and message body—simplifies the task of identifying and translating payloads. The ability to filter incoming messages using message templates enables precise targeting of translation efforts to specific types of messages or interactions. Finally, SPADE's support for open and distributed environments makes it suitable for heterogeneous multi-agent systems, where reliable communication translation is essential. [36]

The PEAK platform, underpinned by the SPADE framework, provides a infrastructure for implementing distributed, interactive agents. Its structured communication model, based on FIPA-ACL over XMPP, aligns well with the needs of payload translation and interoperability in heterogeneous multi-agent systems. By using metadata-aware message structures and behavioural templates, SPADE facilitates controlled message exchange—essential for extracting and translating payloads in a consistent and deterministic manner. [34] [35] [36]

3.2 Used Models for Development

During the development of the implementation, multiple large language models were used. This influenced the design and development process significantly. Switching between models caused rework, delays, and changes in some components. Newer models are more capable, simplifying prompt design and post-processing. However, differences in output format and structure required adjustments to handle outputs efficiently and leverage the improved capabilities.

Key differences between models include skill level, consistency, completeness, optimization behavior, and speed. These factors affected prompt design and system structure. Skill level means how well a model understands instructions, follows logic, and completes tasks. Newer models generally perform better on complex tasks. Early models often produced inconsistent output formats, causing issues in components that relied on predictable structures, such as the Key Mapping module where unordered or invalid mappings were common.

Completeness is another important factor. Earlier models often failed to return full or usable results, especially when generating structured outputs like mappings or tables. As a result, manual checks were needed to fill in missing values or correct misinterpretations. In addition, less powerful models had difficulty handling complex prompts or multi-step instructions, leading to incomplete or incorrect results. These limitations became worse when the input size increased or the prompt involved more intricate logic. Stronger models were able to process larger prompts more reliably, without dropping important information.

Optimization behavior also varied from model to model. Some models responded well to fine-tuning or system-level prompts, while others became unstable or overly sensitive to small changes. This made it difficult to maintain consistency across different sessions or versions. Speed was another critical factor, especially during development. Slow models made it hard to iterate quickly, which blocked testing and debugging. One of the local models used, despite having good accuracy, was too slow to be practical for real-time development.

The original idea behind the model selection strategy was to start with weaker models and gradually move to stronger ones. This was intended to make the system robust by designing it to handle imperfect outputs. In practice, however, this caused more problems than it solved. Time was lost debugging issues that did not exist when using better models, and the additional logic required to handle weak outputs became unnecessary once a stronger model was used. The expected benefit of robustness did not outweigh the cost in time and complexity.

After the release of the Gemini 2.5 Flash preview model, the development process transitioned completely to this model. It offered a combination of high skill, speed, reliability, and completeness that was unmatched by earlier models. Gemini handled complex and detailed prompts effectively, including those with nested instructions or structured output requirements. Prompt clarity remained important, but the model was far more intelligent in interpreting subtle differences. Surprisingly, even the inclusion of emojis in the prompt improved performance in some cases, as Gemini interpreted them as helpful visual cues. By default, Gemini 2.5 Flash uses greedy decoding when the parameters `temperature: 0.0` and `top_k: 1` are set, which ensures consistent and repeatable outputs — a key requirement for this implementation, as discussed in Chapter 2.

In summary, although the project started with weaker models that introduced several limitations and delays, the final implementation was completed using a state-of-the-art, high-performance model. Earlier models influenced the structure and logic of some components, but switching to Gemini significantly improved output quality and development speed. The ability of Gemini to produce consistent, complete, and logically structured responses made it a crucial factor in achieving a stable and efficient implementation.

Chapter 4

Design and Implementation

The objective of this implementation and proof of concept is to focus on the translation of the *payload*, which refers to the actual content embedded within a specific agent communication protocol. As illustrated in Figure 2.1 under Section 2.1 *basics Agents and Multi Agent Systems*, the focus lies on translating the agent-specific language and structure used for communication, rather than the protocol format itself. In this figure, the given structure is referred to as the *content language*. However, both the payload and the content language protocol belong to the same level in the communication framework, as the content language defines the structure of the payload.

In the PEAK framework, the payload is defined by two main fields: `body` and `metadata`. These fields contain all structured information in JSON format, organized as key-value pairs, which are exchanged between agents. The goal of this model is to translate this structured content between two agents that use different JSON formats and different vocabularies. This translation is to be achieved using Large Language Models (LLMs) in the most efficient manner possible.

The final model should achieve the following objectives:

- **O3** - Ensure the translation is deterministic and static, producing consistent and identical outputs for the same input.
- **O4** - Achieve efficient translation, minimizing processing time and ensuring reliability with minimal overhead.
- **O5** - Maintain distributivity, enabling the translation to work across distributed systems without introducing unnecessary complexity.
- **O6** - Ensure no modifications are required to the agents' input behavior files, preserving their original functionality.
- **O7** - Resolve syntactic discrepancies by correctly handling differences in payload structure between agents during translation.
- **O8** - Address semantic ambiguity by effectively resolving differences in meaning between agent communication formats.
- **O9** - Prevent information loss or information dilution, ensuring the integrity of the data during translation.
- **O10** – Ensure that the model is scalable to handle a growing number of agents, maintaining performance and efficiency as the system expands.
- **O11** – Ensure support for one-to-many message mapping

- **O14** – Identify potential for future enhancements.

These objectives must be kept in mind during the system design to achieve a robust, efficient, and reliable payload translation process between heterogeneous agents. At the end of this chapter, the conclusion will revisit these objectives to assess whether they have been fulfilled and, if so, explain how each one was achieved.

4.1 Assumptions

Before the model explanation is provided, it is necessary to establish some key assumptions on which the model relies to ensure correct operation:

- **Assumption 1:** It is assumed that the communication *expectations* of both agents are aligned. In agent communication, the sequence of messages plays a critical role, especially when agents follow a request-response structure. This structure can be deeply nested and context-dependent, which significantly complicates the translation process between heterogeneous agents. For the purposes of this dissertation, such complexities are considered out of scope. The implementation proceeds under the assumption that both agents adhere to the same message exchange expectations, i.e., they follow the same sequence and order of interactions during communication.
- **Assumption 2:** Additionally, it is assumed that the messages do not contain optional fields, as supporting these would complicate the process and was beyond the scope and time constraints of this dissertation.
- **Assumption 3:** Lastly, it is assumed that each agent has access to a Large Language Model (LLM), either locally or through an external service, but only invokes it when introducing a new translation or modifying an existing one. During regular communication, predefined translations are used without repeated LLM inference, ensuring efficiency.

4.2 Full Translation Workflow Overview

This section will first provide a high-level overview of the entire workflow. The following sections will then dive deeper into the main components of the workflow. Before implementing the components, several important decisions are made at this high-level stage. Specifically, decisions need to be made on how to implement the system efficiently and in a distributed manner. During the explanation, the terms **target** and **source** will be used frequently. "Target" is used when referring to what the system wants to translate to, such as target message, target key, or target agent. These are the receivers in this context. "Source" refers to everything related to the things that need to be sent.

Using Large Language Models (LLMs) can be slow and require a lot of computing power, so it is not practical to call the LLM for every message that needs to be translated. To solve this, a method was developed where the LLM is called only once. The result of this single call is then used to translate messages in a fixed and predictable way, without needing the LLM again. The idea is that the LLM can generate a formula that captures both the structure and meaning of the messages, so future translations can be done without its help. This method supports the goals of scalability (O10) and a fixed, predictable translation process (O3). Because the translation is done in a static way, even many translations during runtime cause little or no extra delay, making the system much more efficient.

The use of Large Language Models (LLMs) is, of course, the opposite of efficient in terms of resources and time. It is not practical to call the LLM for every message that needs to be translated. Therefore, a method has been developed to call the LLM only once. With the result of this LLM query, the translation can be performed deterministically and statically. The model is designed with the perspective that a formula is needed to translate messages from one format to another, considering structural and semantic differences, without requiring further LLM intervention. In this way, the scalability objective (O10) and the deterministic, static objective (O3) are promoted by making the translation process static and highly efficient. This ensures that, even with multiple runtime translations, there is minimal to no overhead.

The **formula** must be message-specific, meaning a formula will be generated for each message. The formula for each message consists of 3 files: template.json file, key_mapping.csv file, and voc_mapping.csv file. The template.json file contains the keys and structure of the message to which translation is needed. This is the target message. The key_mapping.csv file contains the mapping of the target keys (keys in the target message) to the source keys (keys in the message that needs to be translated). The voc_mapping.csv file contains the mapping from the source vocabulary (the possible values for a given source key) to the **target vocabulary** (the possible values for a given target key). Using these three files, a deterministic, static translation can be performed (objectives 3 and 4). More details on how this works will be provided later.

A second decision made here concerns the **distributivity** of the model. A central point that must always be accessed for every generation of a translation formula is not desired. To ensure distributivity, the model will generate the translation formula locally on the agents themselves. Once the agents have located each other, they should be able to generate a translation formula between them without the involvement of a third party.

This also means that the translation will be performed directly, without the need for a standardized form (template or common ontology) between the agents. Thanks to this implementation, there is no need for an intermediate standard form if both agents can process the most specific form of the information. This makes it easier to interpret the information abstractly and avoid rounding errors. For example, if agent 1 and agent 2 can process values with two decimal places, but an intermediate standard form can only interpret values with one decimal place, the value 0.14 would be translated to 0.1 and received by agent 2 as 0.1. Without this intermediate standard form, the value 0.14 can be sent directly, ensuring no information loss or information dilution. This distributed implementation will achieve objectives 5 and 9.

In this way, a message only needs to be translated once between a pair of agents, because after the translation, the message is in the form it needs to be. There is no intermediate format. The messages are always translated by the agent that is sending the message. This means there is no need for reception logic to be implemented on the receiving agent's side. The receiving agent can simply use its original receiving API, because the message it will receive will always be in the format it can process. In this way, objective 6 is also accomplished on the receiver's side. As a result, all translation complexity is handled by the sender, who wants to send the message and thus must translate it. Below, you can find a flow chart illustrating how a send and reply message will be exchanged using the concept described above.

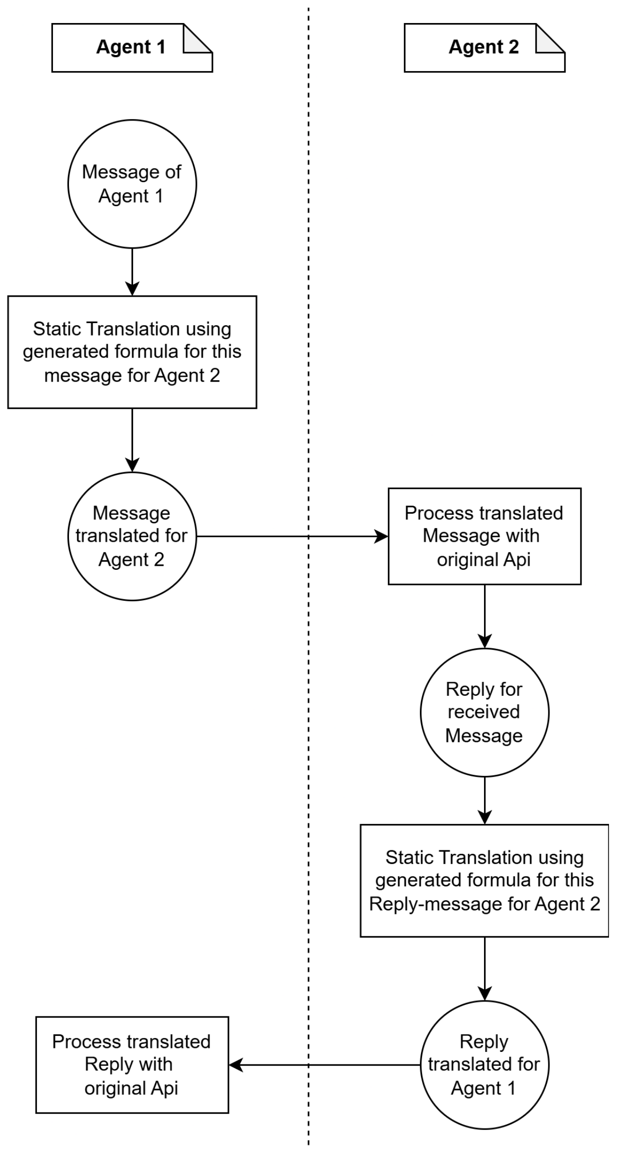

Agent 1
Agent 2
Message of Agent 1
Static Translation using generated formula for this message for Agent 2
Message translated for Agent 2
Process translated Message with original Api
Reply for received Message
Static Translation using generated formula for this Reply-message for Agent 2
Reply translated for Agent 1
Process translated Reply with original Api

Figure 4.1: Simple example illustrating run-time static translation where Agent 1 sends a message to Agent 2, and Agent 2 replies, based on the previously generated translation formulas.

It is important to note a limitation in the current setup: the model currently uses the Gemini API, which means that requests are sent to a central model managed by Google. This could be interpreted as a centralized component, thereby compromising the goal of full distributivity. Ideally, the LLM should run locally on the agent to achieve complete decentralization. If local deployment on each agent is not feasible, a compromise could involve running the model on a privately hosted server. This would allow the organization to retain control over the server while enabling multiple agents to access the model. Such an approach would also improve privacy, assuming the server is maintained within the organization. Nevertheless, the actual message translation process during runtime remains fully distributed, as it runs locally on the agent without requiring further external calls.

Now that the method for performing the actual translation has been established, the logic needs to be designed so that, starting from the unmodified behavior files (as defined in objective 6) both the agents can generate formulas for each message to be sent (source message) to each other. In broad terms, this happens in two steps. First, each agent will individually extract its own sending and receiving APIs. This is the **Message Structure Extraction Phase**. As a result, all the necessary information for creating the formula should be ready before moving on to the next step, the **Formula Generation Phase**. In this phase, the receiving API information from the agents are sent to each other, allowing the agents to generate formulas to translate its sending messages into a format the other agent can process. A detailed explanation of both phases will be provided in the following sections.

By splitting the preparation (Message Structure Extraction Phase) and the actual formula generation (Formula Generation Phase) into two parts, the aim is to support the scalability objective (O10). The Message Structure Extraction Phase should only need to occur once per agent, as long as its receiving and sending APIs are not modified or updated. This makes this phase highly scalable when the number of agents increases. The Formula Generation Phase, however, needs to be triggered for each pair of agents, which could present a scalability challenge. It can be argued that the agent itself determines whether it wants to engage in the translation process with another agent, thus controlling how many agents it wants to undergo the Formula Generation Phase with. The increase in time will be linear, meaning generating formulas for 3 agents will take 3 times longer than for just one agent. This should not cause a significant scalability issue, as long as the agent does not constantly need to create new formulas with multiple agents. Therefore, this scalability criterion depends on the intentions of the agent and is controllable by the agent. Later in the results section, the amount of overhead this process causes will be evaluated, along with the actual scalability bottleneck.

Below is a simple diagram of the steps taken between Agent 1 and Agent 2, who wish to generate formulas for each message they want to send to each other. These are the formulas that were used in the flow chart above (Figure 4.1). Agent 1 has not yet completed its Message Structure Extraction Phase, so this will need to be done first. Agent 2, however, has already completed this phase and only needs to proceed to the next phase, the Formula Generation Phase. At the end, both agents will have formulas for the messages they want to send to each other, and runtime communication can proceed as shown in the flow chart above (Figure 4.1).

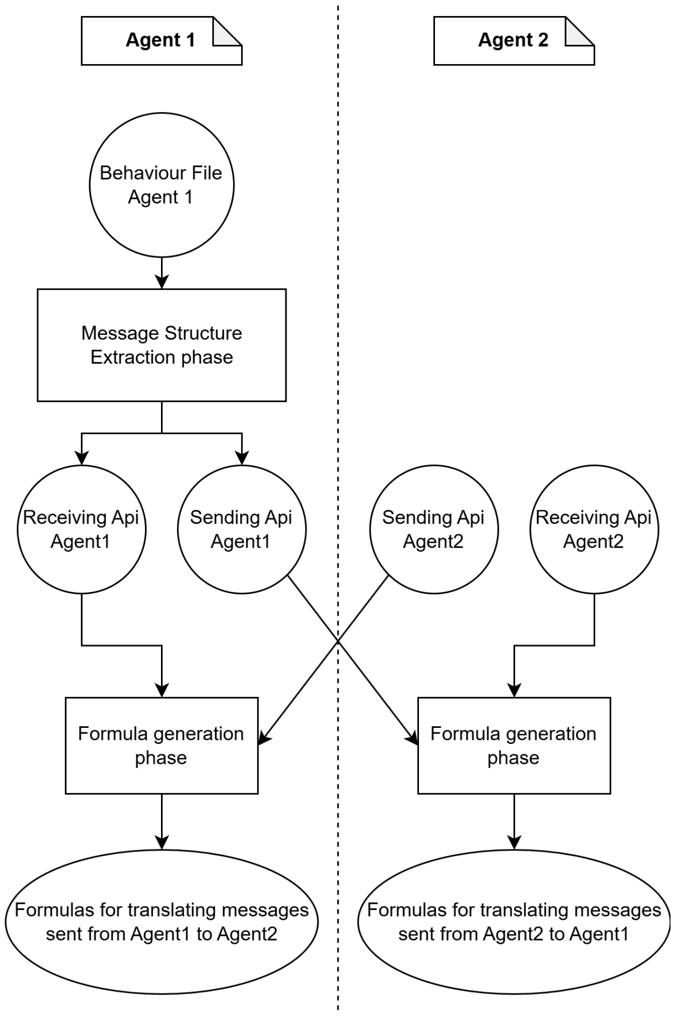


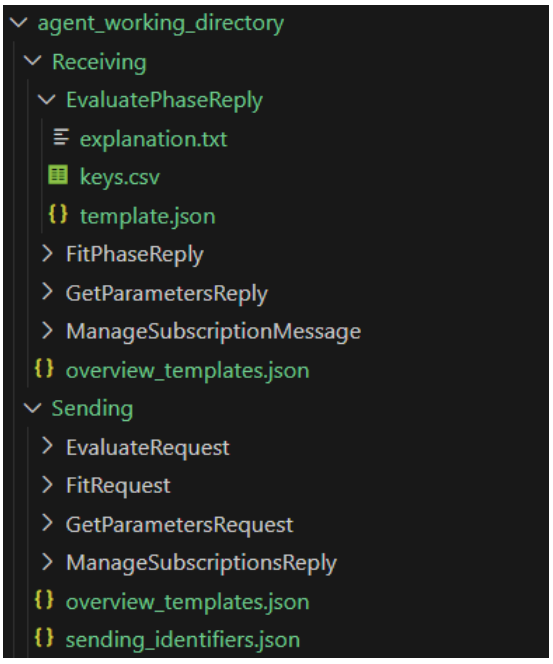
Figure 4.2: Workflow illustrating the steps taken by Agent 1 and Agent 2 to generate translation formulas, including Agent 1's Message Structure Extraction and the Formula Generation Phase between Agent 1 and 2.

For further explanation, the three phases will be described in detail in the following sections. First, the process of extracting the sending and receiving APIs in the **Message Structure Extraction Phase** will be explained in depth. This process occurs individually for each agent. Next, the focus will be on how the **Formula Generation Phase** works between two agents who have already completed the previous phase. Finally, the process of how an agent can translate a message using the generated formula into a specific target message for a target agent will be explained in the **Runtime Runtime Translation Phase**.

4.3 Message Structure Extraction Phase

In this phase, the receiving API and sending API of an agent are extracted and prepared to function in the next phase. The results of extracting both APIs are stored in a directory created for this agent, where all the necessary information for the following phases will be organized. In this phase, the agent is essentially set up to go through the steps required to eventually perform the run-time static translation. So this is the phase where many important design decisions are made regarding how the information should be structured so that it can be easily accessed in the next steps. The structure must remain intact when dealing with multiple target agents (the agents to which the translation is made), multiple messages to be sent (and thus translated), considering the one-to-many mapping.

At the end of this phase, a directory structure will be created that looks like the following: As you can see here, a working directory called "agent_working_directory" will first be



```
agent_working_directory
  Receiving
    EvaluatePhaseReply
      explanation.txt
      keys.csv
      template.json
    FitPhaseReply
    GetParametersReply
    ManageSubscriptionMessage
    overview_templates.json
  Sending
    EvaluateRequest
    FitRequest
    GetParametersRequest
    ManageSubscriptionsReply
    overview_templates.json
    sending_identifiers.json
```

Figure 4.3: Directory structure showing the organization of stored information after the Message Structure Extraction Phase.

created, where all the information needed for this model will be organized. By the end of this phase, there will be two subdirectories under this directory: "Sending" and "Receiving." In each of these subdirectories, every message the agent can send or receive will have its own subfolder. For each message, four files will be generated: "keys.csv", "template.json" and "explanation.txt"". More details about these files will be provided later. These files are message-specific and will be placed in the corresponding message folder for either sending or receiving messages.

Finally, you can see that the sending messages will have two additional collective files: "overview_templates.json" and "sending_identifiers.json". The receiving messages will only have the collective file "overview_templates.json". The purpose of all these files will become clearer in later explanations.

So, it is important to note that all the information is stored in one working directory of the agent. Later, in addition to the "Sending" and "Receiving" subfolders, an extra subfolder called "Translation" will be created. This folder will contain all the translation logic and formulas. But this will be explained in more detail in the next section. The entire functionality of this model will be stored in a single folder. Therefore, if an agent wants to implement the translation functionality, only one additional folder needs to be created, where all the necessary data will be stored. This approach is highly modular and easily transferable, ensuring that the structure remains clean.

The functionality described below will be executed twice: once for outgoing messages and once for incoming messages. These processes will populate the "Sending" and "Receiving" subfolders, respectively. First, the structure of the messages will be extracted, along with the keys, vocabulary, or data type for each message. Subsequently, metadata for these messages and keys will be generated, which will be important in the Formula Generation Phase.

4.3.1 Extraction of Messages, Keys and Vocabulary/Data Types

First, specific structural information needs to be extracted from the agent's behavior file, as well as from any additional files when the agent's logic is distributed across multiple sources. In such cases, the main behavior file is provided as the user query input, while the other files are included as context for the LLM. This is split in such a way that the focus is automatically placed on the main behavior file, with the additional files treated as context from which the LLM may extract information. This setup is ideal for the intended functionality. Furthermore, only the original, unmodified behavior files are used in this process. No changes are made to the agent code itself, allowing the proposed method to function as an abstraction layer on top of the existing agent logic. As a result, the extraction mechanism can be integrated without requiring any modifications to the agent, which directly supports Objective 6 (O6).

In order to fully reconstruct the sending and receiving APIs of an agent, a well-defined set of information must be extracted. This includes **all messages** that the agent is capable of sending or receiving, depending on whether the phase is running in the sending or receiving iteration. Identifying all these messages is essential, as an incomplete view of the messages that can be sent or received will result in an incomplete message mapping. Any messages that are not extracted during this phase will remain invisible to subsequent mapping processes and, as a consequence, will not be included in the translation system.

Once the relevant messages have been identified, the next step is to extract the **internal payload structure** of each message. In the Peak framework, the payload is typically composed of two components: `metadata` and `body`. These sections may contain various fields and can also be nested. It is therefore necessary to determine not just the top-level structure, but also the hierarchy and relationships between keys within the payload. This structural information is crucial during the translation phase and plays a key role in resolving syntactic mismatches between systems, as defined in Objective 7 (O7). Furthermore, this structural data is later stored in the `template.json` file, which constitutes one of the three core components of the translation formula. As such, this information is essential for the correct functioning of the Runtime Translation Phase.

In addition to the message structure, it is also important to extract all the **keys** used inside each payload. These keys will later be matched with keys from other agents, and together they form the basic parts of the translation formula. But having just the keys is not enough. For every key, the correct **data type** must also be found. This is important because during the Runtime Translation Phase, it might be necessary to convert one data type to another. For example, a value that is an `int` in one agent may need to become a `float` or a `bool` in another. Even if two keys mean the same thing, their data types might still be different. This difference must be handled correctly to make sure the translation works as expected.

Finally, for keys with the data type `string`, it is also important to extract all possible string values that can appear. This allows us to build a vocabulary list for each key, which can help match data better between agents during the translation process. If a string field can take on any value—like IDs or locations—labeled as `arbitrary_string`. In this context, concept of **Identifiers** will ne introduced. These are string values where there is only one possible option. In other words, for a certain key, the string value is always the same. This makes it useful for identifying the message, as a message can be recognized based on the fixed value associated with that key.

It is important to note that the following data types are currently supported:

- `bool`
- `int`
- `float`
- `list`
- `dict`
- `string`: string values with a fixed and finite vocabulary
- `arbitrary_string`: string values that can take on any arbitrary value (such as identifiers or locations)

Together, these four types of extracted information—messages, payload structures, keys, and data types (with possible vocabularies)—form the complete foundation of the translation process. All this information is extracted in a single LLM query, and the output is formatted in such a way that it can easily be captured in a Python dictionary. This allows for simple and efficient manipulation of the data in later stages of the pipeline. The prompt used to perform this extraction is shown below.

> You will receive the code of a Python agent with multiple behaviors.
> Analyze **only the word1 messages** that this agent word2 example1.
>
> For each individual word1 message, do the following:
>
> Task Overview
>
> You will analyze and reconstruct the full structure of each message observed in the system. Follow the rules below precisely and in order.
>
> —
>
> 1. Reconstruct the Full Message Structure
>
> Your main task is to extract and reconstruct every message into its structural components. For **every single message instance**, follow these steps **strictly**:
>
> 1. **Split each message into exactly two files**:
> - ‘body.json‘ → Contains **all key-value pairs** from ‘msg.body‘.
> - ‘metadata.json‘ → Contains **all key-value pairs** from ‘msg.metadata‘.
>
> 2. **Only extract from ‘msg.body‘ and ‘msg.metadata‘**.
> - Do **not** include other fields like ‘msg.to‘, ‘msg.sender‘, ‘msg.thread‘, etc.
> - The final structure must be **limited to body and metadata only**.

3. **Fine-grained message separation is required**:
- If two messages **differ in structure** (e.g., different keys, missing fields, or different data formats), treat them as **completely separate messages**, even if they are of the same type (e.g., both are INFORM messages).
- Do **not generalize or merge** messages unless their body and metadata structure are **100

4. **Omit empty sections**:
- If a message contains no body → do **not** create ‘body.json‘.
- If a message contains no metadata → do **not** create ‘metadata.json‘.

—

2. Apply the Value Rules

Apply the following rules when defining the values inside the ‘body.json‘ and ‘metadata.json‘ files:

1. **Strings → always use a list**
- Even if there is only **one possible value**, wrap it in a list: ‘["ready"]‘ instead of ‘"ready"‘

2. **Multiple string values**
- If multiple **distinct** string values are possible, use a list with **all** of them: ‘["pending", "confirmed", "cancelled"]‘

3. **Arbitrary or unpredictable strings**
- If the value can vary or is not predictable, use this exact placeholder: ‘["arbitrary_string"]‘

4. **IMPORTANT: Do NOT invent string values**
- Only include string values that are:
- Present in the provided **behavior file**, or
- Mentioned in **additional info files**
- Do NOT invent or hallucinate values such as ‘"example"‘ or ‘"unknown"‘ unless they literally appear in the input data.

5. **Numeric and boolean values**
- Use the type name as a string: ‘"int"‘, ‘"float"‘, ‘"bool", "list", "dict"‘ - Do not include example values like ‘42‘ or ‘true‘

6. **Never use null**
- Do not use ‘null‘, ‘None‘, ‘"null"‘ or ‘"None"‘

—

3. Output Format

Present the final result using the following format, as a clean Python dictionary: template_string

This prompt contains variables which are defined as:

```
if direction == "Receiving":
    word1 = "incoming"
```

```
        word2 = "receives"
        example1 = ""
    elif direction == "Sending":
        word1 = "outgoing"
        word2 = "sends"
        example1 = " (i.e., messages constructed using .make_reply() or self.send(...))

    template_string = {
        "NameMessage1": {
            "body": {
                ...
            },
            "metadata": {
                ...
            }
        },
        "NameMessage2": {
            "body": {
                ...
            },
            "metadata": {
                ...
            }
        }
    }
```

The data types are written as string values for each key, using labels such as `"int"`, `"float"`, `"bool"`, `"list"`, or `"dict"`. If the value is a string, all possible string values are collected in a list. If the string field can accept any value—such as an ID or location—the list contains only one value: `"arbitrary_string"`. This structure allows efficient extraction of all necessary information into a single Python dictionary, where each message is represented with its structure, keys, and the corresponding values or data types. The result is a clear and linked representation that supports further processing in the translation system.

An important addition to the prompt is the instruction ‘‘`Fine-grained message separation is required`’’, which was introduced to address the issue where the LLM occasionally merged structurally similar messages into one by grouping their identifiers into a shared vocabulary list. This merging occurred because the model perceived such messages as identical or closely related. The instruction explicitly guides the model to treat each message individually, even when structures are similar. Interestingly, an observed side effect of this instruction was that the LLM began splitting messages with optional keys into separate variants, such as `delivery_with_time` and `delivery_without_time`, based on the presence or absence of certain fields. Although this behavior was inconsistent and requires further prompt refinement for reliability, it reveals a potential method for identifying and representing optional keys in the system. While the current implementation assumes the absence of optional keys, these insights could be useful for future system extensions.

After the extraction step, a **post-processing** phase is required to standardize and clean the LLM's output. This is necessary because the model could produce inconsistent or unexpected formatting. One common example is the inconsistent representation of boolean values. To resolve this, a predefined list of possible variants—such as "True", "false", "Boolean", etc.—is used to normalize all such entries to the standard form "bool". This normalization method is applied to every data type. For instance, the terms "int", "integer", and "Integer" are all mapped to "int", while similar groupings are defined for "float", "list", "dict", and "arbritrary_string" values.

In some cases, the LLM may return an invalid structure that cannot be corrected by the post-processing step, such as missing brackets or incorrectly nested values. When such issues occur, the response is entirely discarded and the original query is re-executed. This retry mechanism increases the robustness of the system by ensuring a valid and consistent structure is eventually produced. Although this has become largely unnecessary with the introduction of the more reliable Gemini model, the mechanism is still kept in place as a fallback to handle rare edge cases and guarantee system robustness and stability.

Another frequent inconsistency involves how string vocabularies are formatted. If a key has only one string value, the model sometimes returns it as a single string instead of a list. Conversely, it may incorrectly wrap scalar values like "bool" or "float" in a list. These cases are resolved by analyzing the value or the first element of the list. If the value is not part of any known data type list, it is assumed to be a valid string and converted into a single-item string list. If the list contains a known data type keyword as its first element, the entire list is replaced with the appropriate data type string. This strategy simplifies handling of inconsistent model output and guarantees that the final structure can be parsed and used reliably in further processing.

Post-processing the output is an essential part of working with LLMs, especially when using earlier models such as llama3.2:3b and gemma3:4b. Although it is less critical with more advanced models like Gemini, it remains necessary due to the complexity of the required output structure, which involves handling multiple elements simultaneously. In later steps where LLMs are also used, post-processing becomes much less important or even unnecessary, as those queries involve simpler instructions and more straightforward output formats.

Next, an **optimization** step is applied to handle inconsistencies using a **Stability-Based Filtering** mechanism. In this method, a threshold variable is defined, and the number of occurrences of each exact output variant is counted. When one specific version of the output appears more times than the defined threshold, it is considered correct and forwarded to the next stage. This approach is intentionally very strict: the complete extracted dictionary must be exactly the same across multiple outputs—including formatting and spelling—to be accepted. Such strict filtering is appropriate in the static environment in which the system operates, where even minor differences cannot be tolerated. However, this mechanism strongly relies on the success rate of the model itself. If the model consistently produces the exact same incorrect output, this version will also be accepted without verification. A detailed analysis of the reliability of this method and the role of the threshold parameter is discussed later in Chapter 5.

After completing all previous steps, the result is an output dictionary that represents the extracted message structure. The example shown corresponds to the "sending" iteration of the extraction phase, although the "receiving" iteration produces an output with the exact same structure. In this example, the first message contains both a body and metadata. The body, in this case, is a key that directly serves as an identifier with the value "SUBSCRIBE", while the metadata contains a key "RESOURCE" with another identifier, "MANAGE_SUBSCRIPTIONS". The next two messages do not include a body field but do contain multiple keys in the `metadata` field. Each message has exactly one identifier and includes keys with their corresponding data types as their value. The `"PARAMETERS"` key in the second message holds a vocabulary list with three possible string values, representing the three supported features for that message. These vocabulary lists will be important in later stages. This structured output forms the complete result of the extraction part of this phase and will now be used to prepare the data for the **Formula Generation Phase**. Metadata will be added to the messages and keys, and all information will be organized and stored in structured files as preparation for the **Formula Generation Phase**.

```
{
    "SubscribeFederationMessage": {
        "body": [
            "SUBSCRIBE"
        ],
        "metadata": {
            "RESOURCE": [
                "MANAGE_SUBSCRIPTIONS"
            ]
        }
    },
    "FitPhaseReply": {
        "metadata": {
            "RESOURCE": [
                "FIT"
            ],
            "PARAMETERS": ["feature_1", "feature_2", "feature_3"],
            "N_EXAMPLES": "arbritary_string",
            "METRICS": "arbritary_string"
        }
    },
    "EvaluatePhaseReply": {
        "metadata": {
            "RESOURCE": [
                "EVALUATE"
            ],
            "LOSS": "arbritary_string",
            "N_EXAMPLES": "arbritary_string",
            "METRICS": "arbritary_string",
            "BATCH_SIZE": "int",
            "ADD_GRAPH": "bool"
        }
    }
}
```

4.3.2 Collecting Metadata of Messages and Keys

Before the receiving APIs between agents can be exchanged and the formula can be generated, additional metadata must first be added to the messages and their corresponding keys. This step is essential for the subsequent matching process, where messages and keys need to be aligned with their equivalents across agents. It is important that this information is not only complete and accurate, but also clearly structured and strictly relevant. Modern, more capable LLMs are highly sensitive to fine-grained details, which is advantageous, but also demands caution. Any unnecessary or incorrectly formulated metadata can negatively impact the matching performance. Therefore, each message and key must be described precisely and meaningfully, with carefully selected metadata to support robust and consistent mapping in later stages.

The **explanation for each individual key is generated first**, as these key-level descriptions are also used as input for the subsequent LLM query that generates the message-level explanations. This order is intentional, as including the explanation of each key in the input helps preserve and incorporate the nuanced meaning of the keys into the message descriptions. As a result, the generated message explanations are more accurate and contextually meaningful representations of what each message conveys. This helps when the messages are being mapt in a later stage, in the Formula Generation Phase.

This query takes three types of input: (1) the output from the previous extraction step, (2) the main agent behavior file, which is included in the main user query, and (3) optionally, additional files that provide supplementary descriptions related to the agent behavior file. These contextual files can be included in the same way as in the previous LLM query. Based on this input, the LLM is instructed to identify all keys present in the extracted output and to search through the provided files to determine what each key represents and what role it plays. The objective is to extract a clear explanation for each key based on its use and context in the available files. Finally, a specific output format is defined, which is structured in such a way that it can be easily converted to a `pandas` DataFrame—a tabular data structure in Python that is highly suitable for data manipulation and analysis. The resulting table contains three columns: the name of the message to which the key belongs, the full path to the key within the message structure (this is necessary in cases where two keys may have the same name but have different parent keys), and the generated explanation of the key. This structure ensures that each key can be uniquely identified and clearly described, even in complex or nested message formats. To achieve this, The LLM is provided with the following information (Note: `"word1"` refers to either `"incoming"` or `"outgoing"`, depending on whether the focus is on receiving or sending messages, respectively.):

> Instructions
>
> You will receive:
> 1. A Python dictionary where each key is the name of a word1 message sent by an agent.
> - Each message contains nested content (e.g., 'body', 'metadata', etc.), possibly with deeper levels.
> 2. A separate Python file (agentBehaviourFile) describing the agent's behavior.
>
> Task
>
> Your task is to extract and explain **all keys**, including deeply nested ones, from the messages in the dictionary.
>
> For each key, provide:
> - The name of the message it appears in
> - The full path to the key (e.g., 'body.sensor.temperature')
> - A **clear and detailed explanation** of what the key represents, based strictly on:

- The context of the message itself
- The agent behavior described in the provided file

Important Rules
- Only use the messages and keys present in the dictionary.
- Do not invent or assume additional message names or fields.
- Do not include any information about data types in the explanations.
- Focus purely on the purpose, meaning, and role of the key.

Output Format

Format the output as a table with columns separated by '|'.
Each row must be on a new line.

Columns:
MessageName | KeyPath | Explanation

Example:
TemperatureReport | body.temperature | The measured temperature value reported by the sensor
TemperatureReport | metadata.timestamp | The time at which the message was generated
TemperatureReport | body.location.latitude | The latitude component of the device's reported location

With all necessary information now available, the relevant files can be generated and stored as final results, as illustrated at the beginning of this section. These files are derived from the dictionary extracted in the first step. This dictionary is processed message by message. For each message, a folder is created—either under `Sending` or `Receiving`, depending on the corresponding iteration of the process—named after the message itself. Inside each message folder, two files are generated: `template.json` file and `keys.csv`.

First, a `template.json` file is created by extracting the identifiers from the list from the message and placing them as values in the template, while all non-identifier fields are replaced with `None`. This template serves as a structural placeholder for the *Runtime Translation Phase*, where the actual values will be filled in. You can see how this file could look in the example below, You can see this message has 2 identifiers ("command"-field and "performative"-field) and 1 variable field (the "temperature"-field). Its purpose is to preserve the complete message structure and to resolve **syntactic discrepancies** (O7), a concept that will be further detailed later.

```
{
    "body": {
        "command": "set_temperature",
        "temperature": "None"
    },
    "metadata": {
        "performative": "home_command"
    }
}
```

Second, a `keys.csv` file is produced for each message, containing the table output from the second LLM query, now augmented with datatype information and vocabulary for each key. This information is retrieved from the dictionary generated during the first LLM query and matched to the table entries via the full key path. The `vocabulary` field is only filled in for keys that actually have a defined vocabulary, so only for the "string" data types. For all other keys with a different data type, this field remains empty. For **sending messages**, the `keys.csv` file will **also include identifiers**. This is because, in certain cases, identifiers may appear as variable values in other messages. Including them enables us to establish more links between keys, thus increasing the number of potential matches. This design decision was made based on experimental observations during development.

During the processing of the messages, two additional files are generated: `overview_templates.json` and `sending_identifiers.json`.

An `overview_templates.json` file is created for each message and saved directly under the "Sending" or "Receiving" folder. This file contains information about all extracted message templates for each direction (i.e., sending or receiving). It was originally implemented to help with debugging, but it is now also used as input for the message mapping phase. The file gives a clear overview of all possible messages together with their associated keys.

A `sending_identifiers.json` file is only generated for the messages that need to be sent. It is also stored directly in the "Sending" folder. This file stores all identifiers related to the sending messages. It is a simple dictionary that keeps track of the number of identifiers and their corresponding key-value pairs for each sending message. Later, this file is used during the run-time translation phase to recognize outgoing messages based on their identifiers. This allows the correct translation formula to be applied.

At this stage, the final step of the Message Structure Extraction Phase is performed: **adding a short and clear explanation to each extracted message**. These explanations will help later when matching the sender's messages to the corresponding messages of the receiver. In this step, the language model is asked to create a description for each message from the sender's point of view. The goal is to explain the sender's intention—what the agent is trying to do by sending the message. By focusing on the message's purpose, it becomes easier to connect messages that have the same communicative intent. Additionally, it is also clearly stated what the message is not, based on other possible messages the agent can send or receive. This clarification helps avoid confusion when mapping messages between different agents. The prompts used in this step differ significantly between the sending and receiving iterations. Below are the two separate prompts used:

The prompt for receiving messages:

> You will receive two inputs:
>
> 1. A list of messages that a sending agent can send, including their content (body and optionally metadata).
>
> 2. The API specification of the receiving agent.
>
> Your task is to explain, **from the perspective of the sending agent**, why it would send each message to this particular receiving agent.
>
> In other words: **What is the sender's intention or goal** when sending each specific message to this receiver? What outcome or interaction is the sender aiming to achieve, based on the receiving agent's capabilities?

Additionally, for each message, **explicitly state what it is not**, by clarifying how it differs from other possible messages (e.g. "this is not a report message, which would be used to communicate results", or "this is not a refusal, which indicates inability or unwillingness").

Output format:

```
{
    "MessageName": "<A clear and concise explanation of the sender's intention
    for sending this message to the receiver. Also explicitly state what this
    message is **not**, by distinguishing it from other messages with similar
    or overlapping purposes.>"
}
```

Base your explanation solely on the content of the messages and the sending agent's API.

The prompt for sending messages:

You will receive two inputs:

1. A list of messages that a sending agent can send, including their content (body and optionally metadata). 2. The API specification of the sending agent (the one producing these messages).

Your task is to explain, **from the perspective of the sending agent**, why it would produce each message in general.

In other words: **What is the sender's intention or goal** when generating each specific message? What outcome or interaction is the sender aiming to initiate, based on its own behavior and capabilities?

Additionally, for each message, **explicitly state what it is not**, by clarifying how it differs from other possible messages (e.g. "this is not a decline message, which would reject a request", or "this is not a status report").

Output format:

```
{
    "MessageName": "<A clear and concise explanation of the sender's intention
            for sending this message to this receiver>",

    "MessageName": "<A clear and concise explanation of the sender's intention
            for sending this message to this receiver>",

    "MessageName": "<A clear and concise explanation of the sender's intention
            for sending this message to this receiver>"
}
```

Base your explanation solely on the content of the messages and the sending agent's API.

In addition, a list of explanations for all keys contained within each message is provided to the large language model. This allows the model to generate message-level explanations based not only on the agent's behavior and message structure, but also on the semantic role and intent of individual keys within each message. This information is passed as contextual input to the LLM in the following format:

```
These are the variable or parameter fields found in the given messages.
Use this information when writing explanations for the corresponding messages:
{explanation_keys_per_message}
```

4.3.3 Pseudo code

This section presents the pseudocode for the entire phase, offering a clear overview of all the steps taken along with their corresponding inputs. First, the message structure is extracted from the behavior file(s). Then, the extracted output is postprocessed and evaluated using stability-based filtering. If the result meets the stability requirement, it is considered stable and referred to as `filtered_messages` in the pseudocode. Next, explanations are generated for each key in the messages. After that, the `keys.csv` and `template.json` files are created for every message. Once this is done, the `overview_templates.json` and (for the sending direction only) the `sending_identifiers.json` files are generated. Finally, a description of each message is created and stored in the corresponding message folder as `explanation.json`.

```
for each direction in ["Sending", "Receiving"]:

    repeat:
        response = call API to extract messages in JSON format using:
            - model_name
            - input Python file
            - current direction
            - extra info files

        messages = clean and process the extracted messages
        count = count how often the extracted structure appears

    until count >= required stability threshold (Stability Based
    Filtering)

    filtered_messages = stable messages

    explanation_keys = call API to generate explanation keys for all
    messages using:
        - model_name
        - agent's behaviour file
        - current direction
        - extra info files
        - filtered messages

    create a directory for this direction (e.g. "Sending" or "Receiving
    ")

    for each message in filtered_messages:
        create a message-specific directory
        store explanation keys in a CSV file (keys.csv)
        store message template (template.json)

    store all templates together in overview file (overview_templates.
    json)

    if direction is "Sending":
        store identifiers used in messages in a file (
    sending_identifiers.json)

    explanation_messages = call API to generate explanations per message
     using:
        - model_name
        - agent's behaviour file
        - filtered messages
        - current direction
        - explanation keys

    for each message in filtered_messages:
        store the explanation for this message (explanation.txt)
```

Listing 4.1: Pseudocode for extracting and storing message templates

4.3.4 Result

At this stage, the following components have been obtained for each message:

- `template.json`: This file will be used by the sender to fill in the `None` values with actual data. It effectively represents the structural blueprint of the message to be translated. This file ensures Objective 7 "Resolve syntactic discrepancies by correctly handling differences in payload structure between agents during translation." (O7)
- `keys.csv`: This file will later be used to map target keys to source keys using the generated explanation. It also enables detection of necessary data type conversions. In the case of string-to-string translations (excluding arbitrary strings), the vocabulary lists will be matched to facilitate semantic mapping. The explanation in this file will be used to solve objective 8 "Address semantic ambiguity by effectively resolving differences in meaning between agent communication formats." (O8)
- `Explanation.txt`: A concise explanation of the intention behind each message. This will be used to link the outgoing messages of Agent 1 to the incoming messages supported by Agent 2, based on shared communicative goals.

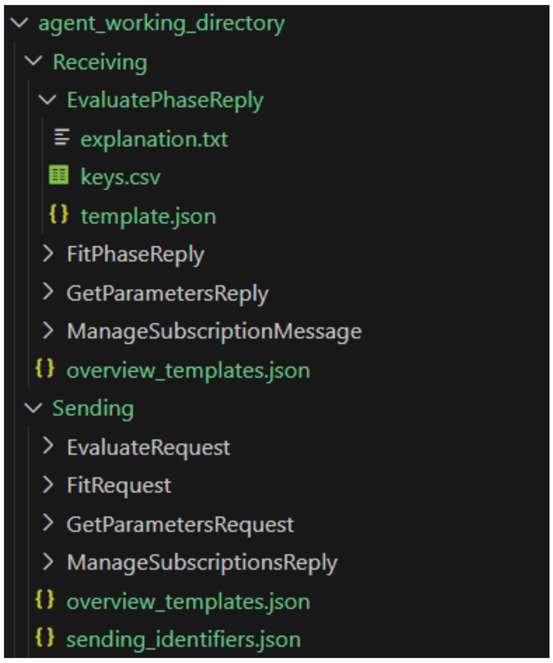
Additionally, two global components were generated:

- `overview_templates.json`: Aggregates all templates for both the sending and receiving APIs into a single file. This overview is used during the message mapping phase to provide the LLM with full structural context.
- `sending_identifiers.json`: Contains identifiers for all sending messages and is essential during the Runtime Translation Phase to determine which message should be transmitted based on the current context.

The information is stored as follows, as shown in the beginning of the explanation of this phase:

4.3.5 Design Decisions and Insights

This subsection discusses several design decisions corresponding to the Message Extraction Phase. Some of these are earlier attempts that were eventually discarded, with reasons for their removal explained. Others highlight important choices that were implemented, along with the motivations behind them. There are also a few ideas listed here that may be implemented in the future. These points are grouped together in this subsection to keep the explanation clear and to concentrate the key insights in one place.



```
agent_working_directory
  Receiving
    EvaluatePhaseReply
      explanation.txt
      keys.csv
      template.json
    FitPhaseReply
    GetParametersReply
    ManageSubscriptionMessage
    overview_templates.json
  Sending
    EvaluateRequest
    FitRequest
    GetParametersRequest
    ManageSubscriptionsReply
    overview_templates.json
    sending_identifiers.json
```

Figure 4.4: Directory structure showing the organization of stored information after the Message Structure Extraction Phase.

Prompt Specificity and Model Sensitivity

When designing the prompt for structure extraction, it proved significantly more effective to provide a detailed and explicit instruction set. Initial experiments with less capable language models showed that such detailed prompts occasionally led to worse performance. This was likely due to the model's limited ability to parse and integrate complex instructions, often resulting in confusion and inconsistent outputs.

However, with more advanced models such as Gemini, it is feasible to construct comprehensive prompts that are both precise and lengthy, while still achieving consistent and accurate results on large inputs. The model demonstrates a strong capacity to follow intricate guidelines when the prompt is well-structured and unambiguous.

It is also crucial to maintain strict precision in wording. For example, the use of `arbritrary_string` as value was misused in cases where only a single valid string was expected—effectively an identifier. Debugging revealed that this issue stemmed from one phrasing in the prompt such as "list the multiple possible string values for each key as value in the output." The word *multiple* introduced unintended ambiguity, which negatively impacted the model's output in approximately 30% of the cases. This highlights how a single imprecise word can substantially degrade the success rate, emphasizing the importance of prompt clarity and linguistic accuracy.

Iterative Prompt Refinement and the Risk of Overfitting

Empirical observations suggest that recurring errors in model outputs can often be resolved through targeted prompt refinements. By carefully analyzing what the model perceives and how it reasons—closely resembling human-like logic—it becomes possible to identify problematic interpretations and prevent them through improved phrasing. This typically involves inspecting intermediate outputs, understanding the model's perspective, and iteratively adjusting the prompt to guide the model more effectively.

However, it is important to note a key limitation of this approach. In the context of this study, the refinement process was not validated across a wide range of test scenarios. As such, there is a risk that the applied fixes may overfit to the specific test data or examples used during development. While efforts were made to ensure generalizability by reasoning at a higher level and avoiding overly specific adjustments, this cannot be guaranteed due to limited test coverage and time constraints.

Filtering Method

Due to the high complexity involved in extracting the structure from behaviour files, a more forgiving filtering method was initially pursued: the count-based filtering method. The extraction process requires accounting for many details, and even small errors can cause the entire output to be marked as incorrect, which is computationally expensive. Moreover, if minor mistakes occur repeatedly in multiple places, reaching consensus on a fully correct answer can take an indefinite amount of time. For these reasons, initial experiments focused on the count-based filtering approach.

This method relies on two configurable parameters: the number of runs used to generate the output from the language model, and a threshold to filter messages. The mechanism runs the code a fixed number of times, counts how often each identical message is produced, and retains messages that meet or exceed the threshold. In this way, the method attempts to isolate stable messages by frequency of occurrence.

At first glance, the count-based method appeared promising, especially with earlier models where output inconsistency was a significant problem. However, testing revealed several limitations. Notably, messages could be absent from the final set if all their variants individually failed to meet the threshold, increasing the risk of missing relevant messages. Additionally, there was a delicate balance to maintain between the two parameters: setting the threshold too low (e.g., below 50%) allowed multiple variants of the same message to appear, causing redundancy, while setting it too high risked excluding inconsistent messages that were occasionally absent or varied.

Later, with the advent of the Gemini model—capable of reliably producing well, structured and correct outputs, the Stability Based Filtering method became a more effective alternative. This stricter approach requires exact matches of the entire structure, reducing false negatives caused by minor variations and ensuring a higher overall quality of the retained results.

There is an more in depth analyse of the characteristics of the stability-based filtering method in Chapter 5.

Excluding Data Types in Key Descriptions

It is important to explicitly **prohibit the inclusion of datatype information** in the explanation of each key. Initially, the LLM automatically incorporated data type descriptions into the key explanations, as these were present in the output generated during the extraction step. However, during the testing phase of the mapping functionality, this led to cases where keys with clearly identical roles and meanings were not mapped to each other, solely due to differing data types. Although the prompt did not instruct the LLM to consider data types as a factor, the model implicitly used them as a filtering criterion. To resolve this issue, the decision was made to remove data type references at this stage, ensuring they are not part of the generated key explanations. This is a deliberate choice: datatype knowledge is not required for the semantic mapping of messages and keys, but is only relevant later during runtime translation, where type conversions may be necessary. By removing this information here, the mapping process remains focused exclusively on the meaning and role of each key, without being biased by implementation details such as data types.

4.4 Formula Generation Phase

In this phase, actual information exchange between agents is initiated. The core idea is that the receiver shares the results of the Message Structure Extraction Phase (i.e., the three key components) with the sender. Based on this information, the sender generates a formula for each message it is capable of sending. As a result, when the sender wishes to transmit a message to the receiver, it will first translate the message into a format that the receiver can understand. This approach eliminates the need for any processing logic on the receiver's side, as the sender prepares messages in a compatible structure.

The information sent by the target agent will be stored under the `ReceiverApi` folder. First, the receiver agent sends its entire `Receiving` folder to the sender. Once received, the sender stores this data in the directory path `ReceiverApi/Agent_id`. In this phase, the sender will create the appropriate translation formula based on the data found in this folder. The content within `ReceiverApi/Agent_id` therefore serves as the input reference for building the translation formula for each message to this target agent.

The formula generation process consists of three main matching steps:

1. **Message Matching:** For each message the sender intends to transmit, it identifies a corresponding message supported by the receiver that serves the same communicative purpose or intent.
2. **Key Matching:** Once the corresponding receiver message has been identified, the sender maps the relevant keys between the source and target messages.
3. **Value Matching:** Finally, the sender ensures compatibility between the matched values. This involves performing any necessary data type conversions or aligning vocabulary terms when both keys hold non-arbitrary strings.

All three steps rely on the pattern recognition capabilities of large language models (LLMs).

4.4.1 Message Matching

In this step, all sending message explanations are provided to the LLM in a single input block. For each message, its associated keys and corresponding explanations are included as bullet points to provide full structural and semantic context. The same is done for all possible receiving messages. The LLM is then prompted to identify links between the sending messages and one or more compatible receiving messages, based on shared communicative intent and structural compatibility. The prompt is structured as follows:

> Instructions
>
> You will receive one or more **sending messages**, each containing:
> - A **name**
> - A **detailed explanation** with two parts:
> 1. **What the message *does* mean** – its intent, function, and purpose.
> 2. **What the message *does not* mean** – explicitly excluded purposes or

uses.
- A **list of keys** (e.g., ‘metadata.status‘, ‘body.code‘), each with a brief explanation of its meaning and role.

You will also receive a set of **receiving messages**, each structured the same.

—

Task

For each sending message, identify which receiving messages it matches, based on:
- The **communicative intent**: what the message aims to achieve.
- The **semantic content**: what information it conveys.
- The **structure and meaning of the keys**: whether the sending message contains all necessary data expected by the receiving message.
—

Matching Criteria

1. **Intent First**
- Match only if the sending and receiving messages have the same communicative goal (e.g., to request, report, confirm).

2. **Respect Exclusions**
- If the sending message explanation specifies what it *does not* mean, do **not** match it to receiving messages that imply such meanings.

3. **Key Coverage Required**
- All keys required by a receiving message must be present in the sending message, either by name or clear semantic equivalence.
- A match is only valid if the sending message **fully satisfies** all key requirements of the receiving message.

4. **One-to-Many Allowed**
- A single sending message may match multiple receiving messages.
- However, each receiving message must be satisfied **independently** — shared keys are fine, but each one’s full requirements must be met.

5. **No Superficial Matching**
- Do not rely on keyword overlap or similar phrasing.
- Match only when **intent, content, and key structure** are clearly aligned.

—

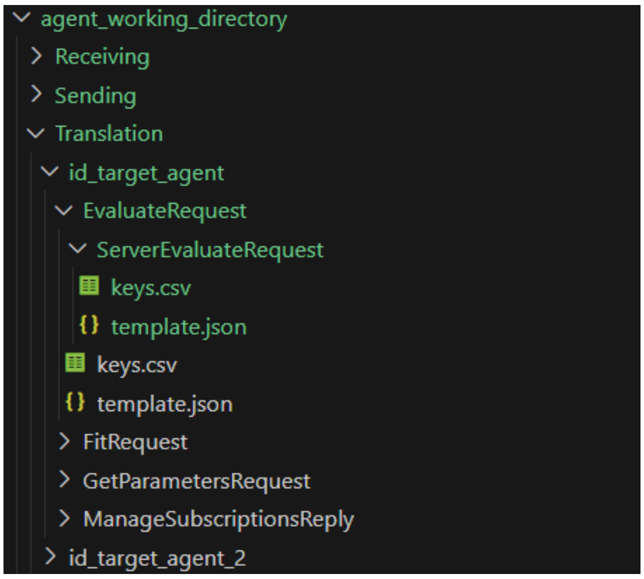
Output Format

Use the following format per match:
SendingMessageName | ReceivingMessage1, ReceivingMessage2, ...

If no receiving messages match:
SendingMessageName | None

Optimization
The number of source keys (including identifiers) and the number of target keys (excluding identifiers) is counted. A check is then performed to ensure that there are enough source keys available to match all required target keys. If the number of target keys exceeds the number of source keys, the mapping is automatically marked as invalid. This is because each target key is considered mandatory and must have a corresponding source key, in line with the defined assumptions.

Result
Finally, the directory structure containing the linked information is organized to facilitate its use in subsequent steps:



```
agent_working_directory
  Receiving
  Sending
  Translation
    id_target_agent
      EvaluateRequest
        ServerEvaluateRequest
          keys.csv
          template.json
        keys.csv
        template.json
      FitRequest
      GetParametersRequest
      ManageSubscriptionsReply
    id_target_agent_2
```

Figure 4.5: Structure of the directory after message matching.

All results from the Formula Generation Phase are stored in the `Translation` directory. Within this directory, a separate folder is created for each target agent involved in a translation-based communication process. Inside each target agent folder, all relevant information required for the translation process is organized.

For each message that can be sent by the source agent, a dedicated folder is created. Within this folder, subfolders are generated for each message into which the original message can be translated. This supports one-to-many mappings.

Each source and target message folder contains two files: `keys.csv` and `template.json`. These files are generated during the Message Structure Extraction Phase. Naturally, the files under the source message folder (e.g., `ServerEvaluateRequest`) originate from the source agent, while the files under the target message folder (e.g., `ServerEvaluateRequest`) are from the target agent and were obtained through its receiving API, which was shared before this phase began.

4.4.2 Key Matching

For each message, the corresponding source and target keys are identified using the `keys.csv` files prepared earlier. This mapping is established through the following prompt, which is presented to the language model to guide the key matching process:

> You will be given two input lists:
>
> 1. A list of source keys, each with:
> - a key name
> - a description explaining what the key represents or is used for
>
> 2. A list of target keys, each with:
> - a key name
> - a description explaining what the key is intended to represent
>
> Your task: Match each target key to the most semantically equivalent key from the source list.
>
> Requirements:
> - Only use keys from the provided source and target list.
> - You must not invent or generate any new keys.
> - Every target key must be matched to exactly one source key.
> - The match should be based on the semantic meaning of the key and its description.
>
> Output format: Return the matches in this format:
>
> target_key | matched_source_key target_key | matched_source_key ...
>
> Example:
>
> Input:
>
> Source keys:
> - "start_square": The initial square from which the piece moves
> - "end_square": The square where the piece is moved to
> - "promotion_type": The piece type chosen when a pawn is promoted
> - "game_timer": The remaining time on the clock for the current player
>
> Target keys: - "from": Indicates where a move begins
> - "to": Indicates where a piece lands after a move
> - "promotion": The piece chosen after a pawn reaches the last rank
>
> Output: from | start_square
> to | end_square
> promotion | promotion_type

Optimizations

When the outputs of the LLM are read in, several optimizations are applied by checking each key-pair for common issues. These include verifying whether the `target_key` and `source_key` are both present in the input, detecting duplicates of either key, and identifying any missing `target_keys`.

Any pairs that do not satisfy these criteria are sent back to the LLM for correction. Importantly, only the pairs that fail these checks are resent and the valid mappings are retained. This iterative approach increases the likelihood of obtaining a correct mapping.

Once all checks are satisfied, the validated results are collected into a list along with a count. This mechanism tracks how many times identical DataFrames have been generated. A variable, `stable_required`, is set to a predefined threshold, defining the number of times the same DataFrame must be produced before it is deemed correct and saved to the file. This approach significantly enhances the success rate by requiring consistent results for validation.

Results
The results are stored in the target message directory
(under `id_target_agent/SourceMessageName/TargetMessageName`) in the file `key_mappings.csv`. This file contains several columns: `target_key`, which refers to the key name in the target message; `matched_source_key`, the corresponding key name from the source message; `data_type_target`, indicating the data type of the target key's value; `string_voc_target`, listing possible string values for the target key if applicable; `data_type_source`, showing the data type of the matched source key; and `string_voc_source`, which provides the vocabulary of possible string values for the source key, if relevant.

4.4.3 Vocabulary Matching

The final phase in generating the data translation formula involves mapping the values between each source and target key of all linked messages. For each source-target key pair, it is assessed whether translation of the associated values is necessary. This decision is guided by the data types of the values, which are identified during the Message Structure Extraction Phase and stored in the `keys.csv` file for each message.

Currently, the system supports the following data types: `string` for categorical values such as identifiers or types, `arbitrary_string` for free-form text like IDs or paths, `integer` for whole numbers, `float` for decimal numbers, and `boolean` for true/false values. The immediate focus lies on handling values of the `string` datatype to enable effective value translation.

String to String Matching
To achieve string-to-string matching, source vocabulary terms are aligned with target vocabulary terms. This process uses a Large Language Model (LLM) query utilizing the following prompt:

> You are an advanced linguistic model specializing in identifying equivalent concepts between different vocabularies within specific contexts.
>
> **Instructions:**
>
> The user will provide you with the following information:
> 1. **List of source vocabulary:** A list of terms from the source.
> 2. **List of target vocabulary:** A list of terms from the target.
> 3. **Explanation of the context in which this source vocabulary is used:** A description of the context in which the source terms are used.
> 4. **Explanation of the context in which this target vocabulary is used:** A description of the context in which the target terms are used.

> Your task is to match the terms from the 'target vocabulary' with their equivalent terms (synonyms or terms referring to the same concept) from the 'source vocabulary', *exclusively within the given contexts*.
>
> **Output Format:**
>
> The output should be a plain text string where each line represents a match. Each line will contain two values separated by a pipe symbol ('|').
> The first value will be the term from the target vocabulary, and the second value will be the matched term from the source vocabulary.
>
> **Important:** Only include lines in the output where a match is found. If a target term has no equivalent in the source vocabulary within the specified contexts, it should *not* appear in the output.
>
> **Example Input (hypothetical):**
>
> List of source vocabulary: ["aanvraag", "verzoek", "klant", "afnemer"]
> List of target vocabulary: ["order", "cliënt", "aanvraagdocument"]
> Explanation of the context in which this source vocabulary wordt gebruikt: "This concerns an administrative context where we talk about formal contacts with people who want to receive products or services."
> Explanation of the context in which this target vocabulary wordt gebruikt: "This concerns a commercial context where we talk about orders and the entities placing these orders."
>
> **Example Output (conforming to the format):**
> order | aanvraag
> cliënt | klant

The user query will incorporate the list of target vocabulary, the list of source vocabulary, an explanation of the target key (representing the context of the target vocabulary), and an explanation of the source key (representing the context of the source vocabulary). All this contextual information is stored in the 'key_mapping.csv' file. The output generated by the LLM is subsequently stored in a DataFrame, enabling further optimization steps, as detailed below. The final result is then persisted in 'voc_mapping.csv'.

Arbitrary String to String Matching

For `arbitrary_string` to `string` matching, all potential target strings are initially recorded with an empty corresponding source vocabulary match. This approach allows us to validate during the Runtime Translation Phase whether the provided arbitrary source vocabulary is indeed a valid option for the target string. If invalid, the value will be replaced by a `None` value in the Runtime Translation Phase.

Optimizations for LLM Request

To optimize the LLM request process, post-processing and additional validation are performed on the LLM's response to ensure consistency, completeness, and adherence to the expected format. This includes detecting duplicates within the `string_voc_target` and `matched_source_vocab` entries, and identifying invalid values not present in the respective vocabulary dataframes (`df_target_vocab` and `df_source_vocab`). Any vocabulary mappings that fail these checks are re-submitted to the LLM API in a `while` loop, excluding already validated mappings. This iterative refinement process closely resembles the key mapping procedure.

Once all checks are satisfied, the validated results are collected into a list along with a count. This mechanism tracks how many times identical DataFrames have been generated. A variable, `stable_required`, is set to a predefined threshold, defining the number of times the same DataFrame must be produced before it is deemed correct and saved to the file. This approach significantly enhances the success rate by requiring consistent results for validation.

4.4.4 Result

For a given agent pair, the following components have now been defined for each of their respective outgoing (source) messages intended for the other agent:

1. **Message Mapping:** A mapping from each outgoing source message to one or more incoming (target) messages of the target agent. This mapping is stored within the source agent's directory structure.
2. **Key Mapping:** The mapping of keys for each linked message pair, specifically from the source key to the target key. This data is saved in the generated directory structure within the `key_mapping.csv` file.
3. **Value Mapping (Vocabulary Mapping):** The mapping of values for each linked key, specifically when the value is of type `string` or, in other words, when the value represents a fixed set of strings (vocabulary). This mapping is also stored in the generated directory structure, under the file `voc_mapping.csv`.

Thus, the established directory structure, alongside the `key_mapping.csv` and `voc_mapping.csv` files, collectively constitute the formula for the translation process that will occur in the subsequent phase. This phase needs to be executed only once for any given pair of agents and marks the final stage where Large Language Models (LLMs) are utilized. The underlying principle is to perform a single, computationally intensive operation to prepare all the necessary information for running a correct and static translation. This ensures a deterministic translation, which consistently yields the same output for a given input, thereby providing unambiguous translations without the need for AI inference during runtime.

4.4.5 Design Decisions

The Challenge of Message Matching

A central challenge in this phase was determining how to perform meaningful message matching. Since the goal of translation is to map semantically or structurally different messages to equivalent meanings, it was necessary to identify a common layer of information that all messages should share in order to be considered equivalent.

While the general notion of "contained information" could serve this purpose, it proved too abstract and prone to ambiguity. As a result, the decision was made to anchor message equivalence in the *intent* of the message. In other words: Why is this message sent? What is the sender trying to achieve by sending it? What communicative function does the message fulfill? This notion of *message intent* became the foundational basis for the mapping process.

To consistently extract intent, all message descriptions were written from the sender's perspective. This **perspective alignment** was essential to ensure that all messages could be interpreted on the same conceptual level, making intent comparison more coherent and systematic. During prompt generation, special care had to be taken to distinguish between sending and receiving processes, as explanations from a mixed perspective often caused mismatches or confusion. Writing all descriptions consistently from the sender's point of view significantly improved the quality of the resulting message mappings.

Initially, the message mapping was performed by sending each individual sending message to the LLM alongside the full set of potential receiving messages. Due to the one-to-many nature of the mapping task (each sending message maps to one or more receiving messages), this setup enabled the system to filter out receiving messages that were not relevant to the current sending message. This approach was chosen specifically to avoid the inverse many-to-one mapping, which was not supported in this implementation.

To determine whether a mapping was valid, a structural condition was introduced: the sending message must contain *at least as many keys* as the receiving message. This constraint ensured that every required element of the receiving message could be matched to something in the sending message. It served as a simple but effective way to enforce semantic and structural compatibility between message pairs.

This design was originally adopted to avoid overwhelming the model with too much information in a single request. However, later experiments showed that Gemini could reliably handle full batches of sending and receiving messages in one prompt, enabling a more holistic mapping. This approach brought several benefits: the model could now reason comparatively between sending messages, and message intent explanations could reference distinctions across the full set of messages.

To further improve precision—especially in one-to-many cases—**negative descriptions** were introduced. These are explicit statements about what a message *is not*, based on the other communicative options available to the agent. For example, in test case 1, the sending side included a `FAILURE` message which had no clear counterpart on the receiver's side. In the absence of an explicit match, the model frequently mapped it (20% of the time) to the receiver's `REPORT` message, which was semantically close but not fully correct.

To counteract this ambiguity, negative cues were introduced. The explanation for the `FAILURE` message explicitly stated that it was *not* an `INFORM` message and should only be used for signaling failures. However, the receiving side's explanation for `REPORT` did not specify that it is not a `FAILURE` message, due to a lack of such information in the API specification. Consequently, the asymmetry remained partially unresolved.

Potential for Many-to-One Mapping

Although the current implementation is limited to one-to-many mappings, observations during debugging suggest that a many-to-one mapping approach may also be feasible. Specifically, when the model was prompted to explain why certain sending messages were not mapped, it occasionally included justifications for why it did *not* map to receiving messages that were already matched in previous iterations.

This behavior implies that the LLM is capable of evaluating candidate mappings holistically, considering global constraints across all message pairs. Such reasoning capacity opens the door for implementing a many-to-one mapping mechanism in future versions of the framework, potentially increasing the system's flexibility and alignment with real-world message structures.

Key- and Semantic Mapping: Control-Based Optimization Mechanism

During the key mapping and semantic mapping phases, a static control-based optimization mechanism was implemented. This system was originally designed to compensate for the limitations of less capable language models, which frequently omitted or mishandled parts of the task. The solution involved embedding simple validation checks into an iterative refinement loop. Whenever the model output failed to meet the structural requirements—such as missing keys or mismatched types—the prompt would be automatically corrected and resubmitted for improvement.

Later, after switching to a more advanced model such as Gemini, the need for such a mechanism diminished due to the model's increased consistency and accuracy. However, the control mechanism remains active and is occasionally triggered, indicating that even high-performing models can still benefit from automated error detection and correction during complex mappings.

4.5 Runtime Translation Phase

This is the final phase, executed during each message transmission. Unlike the previous phases, which are one-time preparations conducted before communication, this phase performs the actual translation of the message during runtime. Similar to the Formula Generation Phase, the translation also occurs on the sender's side. Consequently, the sender translates the message such that the receiver can process it using its original behaviours without requiring any additional logic.

This phase can be divided into two main steps: the message identification step and the template filling step. As the names suggest, the first task is to identify which message the source agent is trying to send. This is necessary because the formulas in the system are stored under names generated by the LLM during the Message Structure Extraction Phase. In practice, messages do not have fixed names, and the generated names are not reliable. Once the correct message is identified for a specific source and target agent pair, the corresponding formula is retrieved. Each formula is unique to a specific target agent and a specific source message. After retrieving the formula, the second step begins: the actual translation. In this step, the formula is applied to transform the source message into a format that the target agent can understand, the target message. The translation is static, deterministic, and efficient because it uses a predefined formula. In the case of a one-to-many mapping—where one source message maps to multiple target messages—there is one formula for each individual target message.

Each formula consists of three components prepared in earlier phases: the `template.json` file, the `key_mapping.csv` file, and the `voc_mapping.csv` file. The `template.json` file serves as the blueprint of the target message and contains the full structure, with all keys present and all values set to `None`, unless they are identifiers. The `key_mapping.csv` file contains the specific key mappings for the given source message and target message combination. Similarly, the `voc_mapping.csv` file defines the vocabulary mappings for string-type values for that same message pair.

Identifying the Message

The first step in the translation process is to determine which specific message the source agent is trying to send. To make this efficient, the `sending_identifiers.json` file was created during the Message Structure Extraction Phase. This file contains all identifiers of possible sending messages in a structured format, which avoids the need to open and inspect every message folder and template during runtime. Since this identification step is performed for every message that needs to be translated, reducing file access operations is crucial for minimizing latency, especially when handling large volumes of messages. Reading files is relatively slow in Python, so avoiding unnecessary access has a direct impact on performance. After loading the identifiers, they are sorted from most to least specific—based on the number of unique fields—to ensure that the system first checks the most precise message definitions. This approach ensures that the message with the most specific and accurate match is identified first.

Once the correct message has been identified, the system loads the corresponding folder generated during the *Formula Generation Phase*. This is the folder found under the path "Translation/target_agent_id/sending_message_name_used_by_model". Under this path there are folders for every target message corresponding to this specific sending message for this specific target agent. In case of a one to one mapping there should be only one folder containing the formula. If it is a one to many mapping, there are as many folders as target messages each containing a formula. As mentioned above the formula contains out of the `template.json` file, the `key_mapping.csv` file, and the `voc_mapping.csv` file.

Filling in the Template

This section explains the actual translation process, specifically how the `template.json` file is filled in. First, a flowchart is presented to provide a visual overview of the procedure. Below the flowchart, a detailed explanation is given that describes each step of the process and how the system translates a message by inserting the correct values into the template.

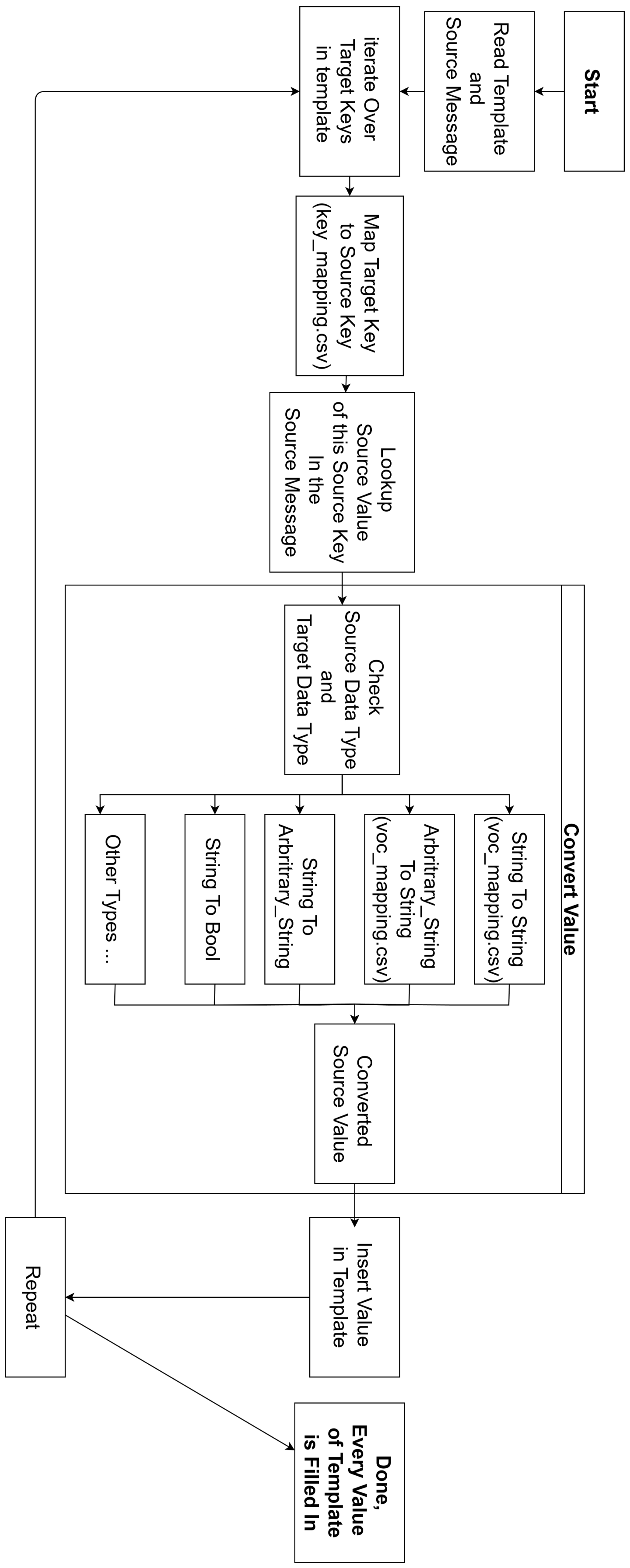

Start
Read Template and Source Message
iterate Over Target Keys in template
Map Target Key to Source Key (key_mapping.csv)
Lookup Source Value of this Source Key In the Source Message
Convert Value
Check Source Data Type and Target Data Type
String To String (voc_mapping.csv)
Arbritrary_String To String (voc_mapping.csv)
String To Arbritrary_String
String To Bool
Other Types ...
Converted Source Value
Insert Value in Template
Repeat
Done, Every Value of Template is Filled In

Figure 4.6: Flowchart of the translation process that converts a source message into a filled in structured target template.

Once the correct formula has been selected for the translation, the system proceeds to fill in the `template.json` file. This process begins by reading both the original message and the `template.json` file as Python dictionaries. The model then **iterates over each key in the template dictionary**, which corresponds to the target keys used by the receiving agent. For each target key, the model consults the `key_mapping.csv` file to retrieve the associated source key from the original message. This retrieved source key is then used to look up in the original source message, the message that needs to be translated, to get the value associated with this source key. This value is the source value and needs to be converted to a target value. In addition, the data types for both the source and target keys are obtained. These represent the expected types of the values linked to the respective keys. Then the value conversion function is called with the target key, target data type, source data type, and the source value as input parameters. This function is responsible for converting the source value into the appropriate target value. Once the conversion is complete, the target value is inserted into the corresponding position in the template. This process is repeated for every target key in the template. When all keys have been processed, the message is fully translated and ready for use by the target agent.

The value **conversion function** first checks whether a data type conversion is required. This conversion can occur between any supported source and target data types. When a string value must be translated into another string value, a vocabulary lookup is performed. These vocabulary translations are defined per target key in the `voc_mapping.csv` file. The system searches for the relevant target key and retrieves the corresponding vocabulary mappings. If the provided source value, which is a word in this case, is found under that key, the corresponding target word is returned. This translated value is then inserted into the translation template, as described in the previous paragraph, thereby completing the translation of that specific word. If the source word is not found in the `voc_mapping.csv` file, the function returns `None`, indicating that no valid translation exists for that word for the intended target agent.

In cases where a value must be converted from `arbitrary_string` to `string`, the value moves from a non-strict to a strict environment. This means that the source value can be any string, but the target value must be one of a predefined set of allowed strings. These allowed target values are also defined in the `voc_mapping.csv` file. The model checks whether the source word appears under the current target key being visited in this file. If a match is found, the source word is considered valid for that key and is returned to be filled into the template. If the source word is not found under the given target key, the system returns `None`, indicating that the value is not supported for that specific key in the target agent. When the opposite conversion is necessary, so from `string` to `arbitrary_string`, the value moves from a strict to a non-strict environment. This makes any checks unnecessary and the source value can just be returned as it is.

When a string needs to be converted to a Boolean value (true or false), the system looks at predefined lists. Values such as `"True"`, `"true"`, and `"1"` are seen as `True`. Values like `"False"`, `"false"`, `"0"`, and `"-1"` are treated as `False`. These lists may need to be expanded in the future to include other similar values. So following these rules a string will be converted to a boolean value and be returned. When the string doesn't match any of the values defined in these lists, None is returened.

For other types of conversions, such as turning a value into an integer, float, or Boolean, the system uses standard Python functions. These include `str(value)` for strings, `int(value)` for integers, `float(value)` for decimal numbers, and `bool(value)` for Boolean values. This basic method works in most situations but may not handle all special cases or unusual input formats. In those cases, extra logic might be needed later on.

Usage of the Runtime Translation Function

To utilize the runtime translation function, only one function needs to be imported from the `apiTranslation.py` file: the `translate()` function. This function requires three parameters: the message to be sent, the `receiver_id`, and the `sender_id`. These IDs must correspond to the ones used during the preparation and Formula Generation Phase. Although it is technically possible to extract the sender and receiver IDs directly from the `from` and `to` fields in the message, the current implementation keeps them as explicit parameters for simplicity and due to time constraints. The `translate()` function is designed to replace the `self.send()` function used in the PEAK framework. The replacement can be implemented as follows:

```
translated_messages = translate(msg, "receiver_id", "sender_id")
if translated_messages:
    for translated_message in translated_messages:
        await self.send(translated_message)
        logger.info("Request sent to HomeAgent1.")
else:
    logger.info("No translation for Message: " + str(msg))
```

A `for`-loop is used here because the `translate()` function supports one-to-many mappings, meaning that a single source message may result in multiple target messages.

In an ideal scenario, the `self.send()` method would be overridden so that no changes are needed in the behaviour files of individual agents. The updated version of `self.send()` would internally check whether translation is required based on the `receiver_id`. If translation is needed, it would automatically call the `translate()` function and send all translated messages. This approach maintains the original purpose of the `self.send()` function but extends it with translation capabilities in a transparent and modular way. As a result, agent behavior files remain untouched, preserving clean and maintainable code. This would then finally fulfill the objective **"Ensure no modifications are required to the agents' input behavior files, preserving their original functionality."** (O6).

4.6 Possible Extensions for Future Work

This section discusses possible future extensions. It is important for understanding the current limitations of the system and for suggesting potential solutionsor extensions that could address these limitations and enhance the model. This section also directly relates to one of the objectives of this dissertation, specifically Objective 14: "Identify potential for future enhancements" (O14).

Exploring the Effectiveness of Fine-Grained vs. Single Query Extraction in LLMs

A single, comprehensive prompt is used to extract all relevant information from the behavior file in one query. However, the question can be raised whether it would be more effective to split the prompt into multiple fine-grained prompts. Intuitively, it might seem that this could improve performance because the tasks would be smaller, reducing the chances of making mistakes since the LLM would need to consider fewer elements. In that case, it might be better to break the extraction down into three steps: first, extracting only the messages, then sending a query with the extracted messages to add the keys, and finally, passing both pieces of information to the last query to place the values with the keys.

Counterarguments to this approach would include the added complexity. Linking the messages with the keys, and the keys with the values, could introduce additional points where errors could occur, potentially having a counterproductive effect. Additionally, the Gemini model seems comfortable with larger prompts, so there was no immediate need to split the prompt. Moreover, one might speculate that if the model forgets to extract something, which is where most errors occur, the likelihood of it forgetting the same information in one of the smaller steps might not be any smaller. This remains speculation and would need to be supported by experimental results. Finally, splitting the queries would require three times as many calls for a single output. Increasing the threshold value of the filter could multiply the number of queries by a factor of three, which would have a significant impact, especially with models that have longer processing times.

For the implementation, a single query with a large, detailed prompt was chosen due to uncertainties about whether splitting the prompt would result in better performance and the added complexity it might introduce. Additionally, this approach could hinder efficiency and would cause the model's daily query limit to be reached three times faster. For these reasons, the single query approach was implemented, but it remains an interesting idea to investigate whether the results would be better or worse than using a large prompt with a single query. If the results were better or worse, how much better or worse would they be?

Output Validation of The Extracted Structure via Logs

An additional idea explored after development is the use of **logfiles** as a secondary validation step and as extra input for the LLM. Although it is not yet clear whether these logfiles would need preprocessing or how the payload structure could be extracted from them, they offer a promising source of information to improve output validation. One of the main challenges at this stage is the lack of a feedback or verification signal to determine if the extracted output is actually correct, since the system currently relies solely on the agent's behavior file. Adding a validation mechanism based on logs could provide useful feedback to confirm output accuracy. Moreover, if correct and incorrect outputs can be distinguished at runtime, invalid outputs could be reused as negative examples to help the LLM avoid similar mistakes. If using existing logfiles proves too complex or unreliable, an alternative approach would be to implement a custom message sniffer that intercepts all sent and received messages and logs them in a structured format adjusted for validation. In short, introducing an output validation source would be highly valuable, and system logs—or a custom logging tool—could be strong candidates for enabling this functionality.

Optional keys

An appealing improvement to the current implementation would be the support for optional keys. To enable this, it is necessary to first identify and extract metadata about optional fields during the *Message Structure Extraction Phase*. This metadata should be stored efficiently, preferably as an additional attribute within the extracted structure. Subsequently, during the key mapping phase, this metadata can indicate that certain fields are optional and therefore do not require a strict counterpart in the target schema.

Another possible enhancement involves supporting one-to-many and many-to-one key mappings. For example, a single key such as `{date: xx}` could be mapped to multiple keys like `{day: xx, month: xx, year: xx}`, and vice versa. The main challenge in implementing these features lies in the structure extraction step, as it remains uncertain how reliably optional fields and complex mappings can be detected across varying message formats. Thorough evaluation and testing would therefore be essential before these improvements can be adopted reliably.

Function-Aware Vocabulary Mapping

A very important improvement would be to also generate explanations for vocabulary values and use those explanations during semantic mapping. Initially, it was assumed that only explaining the key would be enough to understand the context of a vocabulary value. However, after implementation, it became clear that this is not sufficient.

There are cases where the same vocabulary value (i.e., a string) can have a different meaning or trigger different actions in different systems. For example, imagine a field that gives the temperature status of a room, and the value is "room_too_hot". Two systems might both receive this value, but System 1 opens the windows, while System 2 turns on the air conditioning. These are two different actions linked to the same value, which is not acceptable in a translation system.

Even if the keys are correctly matched, the meaning and effect of the vocabulary values also matter. Each vocabulary value—especially if it's a command—should have its own explanation. This was not an issue during testing because no actions were linked to the vocabulary values, so the LLM could understand them based on general knowledge. But when identical values have different effects, they should not be matched.

The implementation of this feature would be similar to the current key-matching approach. The explanation for each vocabulary value should be generated during the Message Structure Extraction Phase. This ensures that the explanation is created only once per agent, which keeps the process efficient. During generation, the behavior files—and preferably the key explanations as well—should be included to provide the full context. These explanations should then be stored in a structured way and passed along to the vocabulary matching phase. The prompt for that phase should be adjusted to make sure the LLM focuses on the function of each value, ensuring that only values with matching functionality are linked.

Handling of Lists and Dictionaries in Translation

At this stage, support is only implemented for handling lists and dictionaries during the *Message Structure Extraction Phase*. This is acceptable, as these datatypes are not relevant in the *Formula Generation Phase*. However, during the *Runtime Translation Phase*, there is currently no dedicated support for these datatypes beyond simple copy-pasting of the list or dictionary values. It is important to note that this copy-pasting behavior has not yet been thoroughly tested, so its correctness and reliability remain uncertain. This opens up potential areas for further research. First, it should be determined whether additional logic is necessary to properly convert lists and dictionaries during translation, or whether the current copy-pasting approach is sufficient in practice. If additional logic proves to be necessary, the type of logic required and how it can be effectively integrated into the translation process must be investigated. Finally, the current copy-pasting approach should be systematically evaluated and tested to ensure its correctness across a variety of use cases.

Unit Conversion

A potential future feature involves implementing unit conversion for numeric values (integers and floats), which was not included due to time constraints. Adding this feature would require modifying the vocabulary matching phase with a new LLM prompt to detect necessary unit conversions, efficiently storing the resulting unit mapping information, and integrating static unit conversion logic into the Runtime Translation Phase.

Key considerations for implementing unit conversion include the extent to which conversions can be implemented flexibly, whether all possible conversions need to be hardcoded manually, and whether a generalizable pattern exists to automate part of the conversion process and reduce development effort. It is also worth considering whether a custom, extendable unit conversion library would be beneficial during deployment. Additionally, the availability of existing Python libraries capable of performing unit conversions efficiently should be evaluated, along with the feasibility of integrating such libraries with LLM-generated outputs.

Error Propagation

For future deployment, structured error propagation is essential. When a vocabulary entry cannot be found (i.e., a `None` value is returned), it is important to verify whether the vocabulary mapping is complete and accurate. If the target agent fails to process a message, a feedback mechanism should be implemented. In the case of incorrectly received messages, a recovery system that dynamically attempts to resolve the issue may be necessary. All errors should be logged in a structured format to support further fine-tuning of the system, either manually or automatically. Given the dynamic nature of LLMs, errors are inevitable; the true challenge lies in capturing, propagating, and correcting these errors efficiently, or triggering regeneration of valid outputs when necessary.

Dynamic Selection in One-to-Many Message Mapping

Currently, the implementation of one-to-many message mapping is deterministic, meaning that when a message is mapped to multiple translated messages, all of them are always sent. While this approach is simple and consistent, it lacks flexibility. In real-world scenarios, it is often the case that a message should indeed be mapped to multiple variants, but only one or a specific subset of these messages should be sent, depending on certain values contained within the original message's fields. As such, a valuable extension of the current system would involve supporting dynamic selection among the translated messages. This would enable context-aware decision-making within the implementation, allowing it to evaluate the content of incoming messages and determine which of the mapped outputs are most relevant in a given situation. By doing so, the system could ensure more accurate and efficient communication, avoiding unnecessary or incorrect message propagation in one-to-many mapping situations.

Extension to XML-Compatible Content Languages

An extension towards compatibility with XML-formatted content languages can be considered, depending on how effectively the LLM can extract structural information from the agent behaviour files. It may be necessary to adjust the prompt to improve structure extraction in such cases. If the extraction process performs better when the resulting structure is represented in XML, adopting this format could be advantageous. Currently, data processing and optimizations rely on Python dictionaries. Therefore, if the structure is extracted in XML, only a conversion step would be needed to transform the XML structure into a Python dictionary. Once converted, all existing processing steps can continue without additional modifications. The same principle applies to the Runtime Translation Phase: during translation, the JSON template is interpreted as a Python dictionary. The system essentially fills in this dictionary, and at the final output stage, converting the dictionary into XML instead of JSON would be sufficient to support XML-based communication.

Many-to-One Mappings

As explained in Section 3.6.5 *Design Decisions*, the current system likely supports many-to-one mappings. Observations indicate that the LLM often attempts to map multiple sending messages to the same receiving message, even without explicit instructions. To improve the reliability of such behavior, it is important to clearly define the boundaries of acceptable mappings within the prompt. Additionally, checks should be implemented to detect whether the model consistently produces the same incorrect mappings. If such patterns are found, the system can be enhanced by refining the prompts through targeted prompt engineering in future iterations. In contrast, inconsistent or random mistakes should be automatically filtered out by the stability-based filtering mechanism.

Portability Across Programming Languages

A promising extension of this implementation is its potential portability to other programming languages. Since the LLM prompts used throughout the system are language-independent and do not contain Python-specific instructions, no changes would be required at the prompt level. The only necessary adjustments would concern the static post-processing logic, which handles language-specific data types.

These changes would primarily affect the *Runtime Translation Phase*. For instance, in Python, Boolean values are capitalized (`True`, `False`), whereas in Java they are lowercase (`true`, `false`). Such differences could lead to incorrect translations if not properly managed during static conversion. A promising strategy for supporting multiple programming languages is to introduce an intermediate conversion layer. In this approach, all value translations follow a standardized path: *source language* $\rightarrow$ *Python* $\rightarrow$ *data type conversion* $\rightarrow$ *Python* $\rightarrow$ *target language*. This setup allows developers to focus only on implementing conversion logic between Python and each additional language, without modifying the core translation process.

Importantly, no modifications would be needed for the *Message Structure Extraction Phase* or the *Formula Generation Phase*. However, designing language-specific prompts for structure extraction could further improve accuracy and reliability when adapting the system to other programming environments.

4.7 Conclusion

In this section, multiple objectives needed to be achieved. As a conclusion to this chapter, the following evaluation determines whether each objective has been fulfilled and explains how it was realized.

O3 - Ensure the translation is deterministic and static, producing consistent and identical outputs for the same input.

This is achieved by constructing a static formula composed of three files: the `template.json` file, the `key_mapping.csv` file, and the `voc_mapping.csv` file. These files serve as parameters for the `translate()` function, which processes them in a static and deterministic way. As a result, any identical input—translating the same original source message to the same target agent—will always invoke the same formula. Consequently, this ensures that the translation remains consistent for each identical message. This approach is discussed in more detail in the "Formula Generation Phase" section for the construction of the formula, and in the "Runtime Translation Phase" section for how the formula is applied to translate a message.

O4 Achieve efficient translation, minimizing processing time and ensuring reliability with minimal overhead.
This is also ensured by performing the runtime translation entirely through static code. By implementing the `sending_identifiers.json` file, the identification of the message to be translated is optimized. The actual translation—filling in the template—is carried out without any involvement of an LLM or computationally expensive operations. As a result, once the formula has been generated, the translation of the message itself is executed in a highly efficient manner.

O5- Maintain distributivity, enabling the translation to work across distributed systems without introducing unnecessary complexity.
This was implemented and explained in the "Full Translation Workflow Overview" section. The model ensures distributivity by allowing each agent to generate the translation formula locally, without relying on a central authority or third party. Messages are translated directly by the sending agent into a format that the receiving agent can understand, removing the need for a common standard format. This approach prevents information loss and keeps the architecture simple and efficient. However, one limitation is that the current implementation depends on the Gemini API, which introduces a centralized element. To achieve full decentralization, the model should ideally run locally on each agent or be hosted on a private server under organizational control.

O6 - Ensure no modifications are required to the agents' input behavior files, preserving their original functionality.
This was achieved in two parts. First, the extraction of message structures is based solely on the agent's behaviour file(s) as input. These files are not preprocessed or anything, we just use the original files without any modifications. This ensures that the entire translation process can begin without requiring any modifications to the agent's files or additional actions from the agent. Second, the translation function that is invoked at runtime can be embedded within the agent's original `send()` function by overriding it. These two measures ensure that absolutely no modifications are needed in the agent's files, successfully fulfilling this objective.

O7 - Resolve syntactic discrepancies by correctly handling differences in payload structure between agents during translation.
This objective is achieved by first extracting the message structures using the LLM. After extraction, the complete structural information of each message is stored in a `template.json` file. This file serves as a blueprint for the messages. During translation, this file is read and only the values are filled in, while the structure remains unchanged. This guarantees that the full structure of the original message is preserved throughout the process, thereby fulfilling the objective.

O8 - Address semantic ambiguity by effectively resolving differences in meaning between agent communication formats.
This issue is addressed by prompting the LLM to generate an explanation for each key based on the context in which the key is used. The context refers to the agent's behavior file(s), which the LLM has access to during explanation generation. Each explanation is detailed and clarifies the meaning, role, and purpose of the key. These explanations are then used in the mapping process by supplying them alongside the keys. A second LLM is prompted to carefully analyze the explanations and to match keys only when their meaning, role, and purpose align. This method ensures that the mapping process goes beyond the semantic

similarity of the key names by incorporating contextual information. As a result, ambiguity is reduced, and equivalent keys with different names can still be correctly matched.

O9 - Prevent information loss or information dilution, ensuring the integrity of the data during translation.
This is also addressed and resolved in the *Full Translation Workflow Overview* section. By implementing a direct translation architecture between each pair of agents, there is no need for a standardized intermediate format. This approach eliminates the risk of information loss due to rounding or omission, which could occur if the intermediate format supports fewer features than the communicating agents. In the implemented system, information is exchanged directly between agents, ensuring that the full capacity of information—supported by both the sender and the receiver—is preserved during translation.

O10 – Ensure that the model is scalable to handle a growing number of agents, maintaining performance and efficiency as the system expands.
This has been taken into account by designing an architecture in which the *Message Structure Extraction Phase* only needs to be executed once per agent, as long as that agent's behaviour remains unchanged. In contrast, the *Formula Generation Phase* must be executed for every unique pair of agents that intend to communicate. Although this is less scalable, it remains manageable since each agent controls its own formula generation process. Moreover, this phase scales linearly: if an agent wants to communicate with five different agents, the formula generation step must be performed five times. However, since each of these steps only needs to be done once per agent pair, and the runtime translation is designed to be highly efficient, the system overall can still be considered scalable. The only bottleneck may occur when an agent needs to generate formulas for communication with a large number of agents simultaneously.

O11 – Ensure support for one-to-many message mapping.
This requirement was fulfilled in the *Message Mapping* subsection of the *Formula Generation Phase*. It was achieved by making three key adjustments. The first adjustment was to explicitly encourage one-to-many mappings in the prompt given to the language model. This guidance made the model more likely to attempt such mappings. The second adjustment involved clearly describing in the message explanations what a message is *not*, based on the definitions of other possible messages from the same agent. This helped prevent overmapping, where a single source message would incorrectly be linked to too many different target messages. The third adjustment was to provide the language model with the sets of keys used in both the source and target messages. The model was instructed that a source message must contain at least as many keys as the target message. This final step effectively eliminated overmapping by ensuring that only source messages with enough relevant information could be matched to a target message. In summary, one-to-many message mapping was successfully implemented by encouraging it in the prompt, while avoiding incorrect matches through precise message descriptions and a minimum key requirement.

O14 – Identify potential for future enhancements.
This is discussed in detail in the *Possible Extensions for Future Work* section. Below is a summary of all the proposed extensions mentioned and explained there. For a full explanation of each item, refer to the respective section:

- Exploring the Effectiveness of Fine-Grained vs. Single Query Extraction in LLMs
- Output Validation of The Extracted Structure via Logs

- Optional keys
- Function-Aware Vocabulary Mapping
- Handling of Lists and Dictionaries in Translation
- Unit Conversion
- Error Propagation
- Dynamic Selection in One-to-Many Message Mapping
- Extension to XML-Compatible Content Languages
- Many-to-One Mappings
- Portability Across Programming Languages

Chapter 5

Model Evaluation

This chapter presents an evaluation of the model. The first section analyzes the stability-based filtering mechanism used in all four generation steps: message structure extraction, message mapping, key mapping, and semantic mapping. Simulations are used to examine the influence of the threshold variable and to assess the reliability of the filtering mechanism at various threshold levels.

Following this, three test cases are introduced, each presented in a separate section. Each case features two agents attempting to communicate using the same API content, but with differences in the payload (message structure and vocabulary), requiring translation between the agents. The first two test cases introduce targeted challenges to benchmark the system's robustness, while the third combines these challenges into a more extended and difficult scenario to evaluate the model's overall robustness and consistency. In each test case, all four LLM-based generation steps—message structure extraction, message mapping, key mapping, and semantic mapping—are evaluated individually. These are the only steps capable of producing incorrect output, and an fault in any one of them leads to an incorrect translation formula. Each part of the test case is executed 30 times to account for variability and to obtain reliable evaluation metrics. The agents involved are fully functional, and when the translation formula is correctly generated, successful communication is achieved over the PEAK framework, despite structural and vocabulary differences in their APIs.

Each test has its own threshold parameter for stability-based filtering. Each test is initially performed with this parameter set to 1 to evaluate the model's success rate under default conditions. If the model fails to achieve a perfect score, the parameter is iteratively increased until the test is passed. Errors are carefully analyzed and will be reported alongside the results of each test. This is important to understand how the model fails, whether the failures are consistent, and what the underlying issues might be. Note that identifiers in the receiving messages are not considered mappable keys, as explained in Chapter 4, "Design and Implementation". Therefore, identifiers of target messages are excluded from the target keys in the key mapping process.

The Gemini model consistently produces well-structured and reliable outputs, outperforming earlier models used in this work; many previous models struggled to deliver consistent results, which made implementation and evaluation difficult. Additionally, it is hard to find other models that are both freely available and sufficiently capable, while also responding similarly to fine-tuning due to differences in model architecture and behavior. This makes it challenging to find a suitable alternative to Gemini for meaningful comparison. Consequently, no other models were included in the evaluation phase of this work. This limitation is not conceptual, but rather practical, and opens a clear path for future research: once more high-quality and freely accessible models become available, it would be valuable to test whether the same results and robustness can be achieved using alternative models. For now, Gemini's superior performance was crucial in ensuring the reliability and accuracy demonstrated throughout this thesis.

At the end of the chapter there will be a conclusion which addresses two key objectives defined at the beginning of the dissertation:

- **O10** – Ensure that the model is scalable to handle a growing number of agents, maintaining performance and efficiency as the system expands.
- **O12** – Evaluate the autonomy and reliability of a proposed or existing communication framework and determine whether human intervention is necessary.
- **O13** – Benchmark and validate the performance and robustness of the implementation.
- **O15** - Identify current limitations of the model.

5.1 Statistical Analysis of the Stability-Based Filtering Method

This section presents a statistical reasoning framework to evaluate the effectiveness and limitations of a stability-based filtering method designed to improve the reliability of outputs from a language model. The method relies on a *threshold* parameter, denoted by n, which dictates that the process halts once any variant of the output has appeared *at least n* times, regardless of the order in which these occurrences appear.

It is important to emphasize that, in our context, the correct answer is not known in advance. The goal is to *identify* the correct answer only based on the outputs generated by the language model. This setup inherently assumes that the correct answer appears with a sufficiently high and stable frequency across repeated generations.

Consequently, the reliability of the filtering mechanism is closely tied to the *robustness* of the language model's output distribution. If the model is consistent—i.e., it tends to produce the correct structure more frequently than incorrect alternatives—the filtering method is expected to converge successfully. However, if the model is unstable or prone to producing multiple plausible but incorrect variants, the filtering method may fail to isolate the correct structure.

To better understand this behavior, a simulation-based framework is implemented to empirically explore under which statistical conditions the method remains robust, and under which conditions it is likely to fail. This includes investigating the relationship between variant probabilities and the selected threshold, as well as quantifying the risk of premature convergence on an incorrect output. The resulting observations form the basis of the statistical analysis.

Simulation 1: Varying Threshold and Error Granularity

In each simulation run, a sequence of outputs is generated by repeatedly sampling from a self-defined probability distribution. This distribution reflects how often the correct output and each type of incorrect output are expected to appear. Specifically, the number of incorrect variants is chosen, a probability is assigned to each incorrect variant $(q_1, q_2, \ldots)$, and consequently, the remaining probability is allocated to the correct output $(p = 1 - \sum q_i)$.

At each step of the simulation, one output is randomly selected based on this distribution, and the number of times each variant appears is tracked. The process terminates as soon as any single variant—whether correct or incorrect—reaches a predefined number of occurrences, referred to as the threshold *n*. The first variant to hit this threshold is treated as the "selected" outcome for that simulation run. By repeating this process 10 000 times, it becomes possible to empirically estimate the likelihood that the correct variant is selected under varying conditions. These conditions include the threshold value *n*, the number of incorrect variants, and the assigned probabilities of all variants. This simulation framework provides a flexible and controlled way to evaluate how robust the filtering method is, allowing for analysis of when it reliably selects the correct output and when it is more prone to failure.

First, the **impact of the number of errors on the overall success probability** is investigated while keeping the success rate fixed at 60%. The simulation is conducted over a range of error counts corresponding to all integer divisors of 40, i.e., from 1 to 40. To maintain simplicity and ensure a fair comparison, the total probability of error is uniformly distributed across the different error types. This approach allows the effect of increasing the number of error categories on the success probability to be isolated, without altering the cumulative error likelihood. The results provide insights into how the granularity of error classification influences system performance at a fixed success rate. The names of the curves in the plot are presented as $x\% \times y$, where x represents the probability of an individual error and y denotes the number of distinct error types.

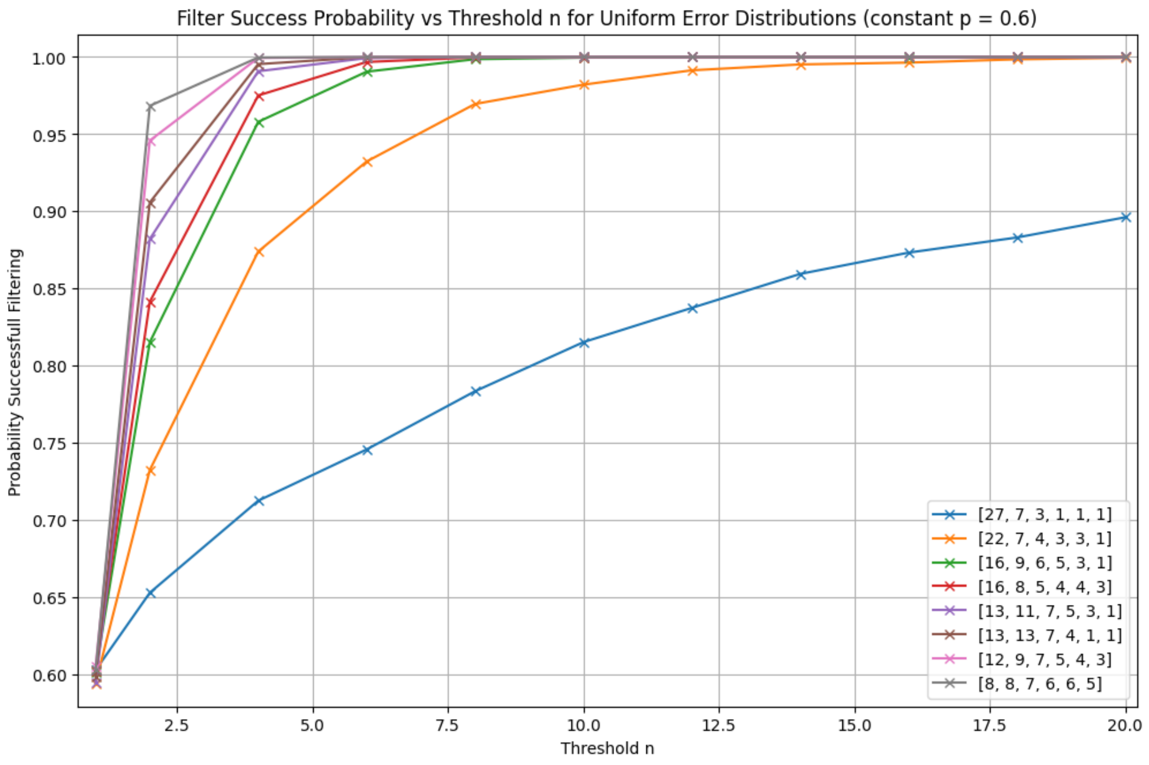


Figure 5.1: Illustration of the simulation results showing the effect of the number of errors on the overall success probability with a fixed success rate of 60%. The names of the curves denote the error probability per error type multiplied by the number of distinct errors.

It is evident that the granularity of errors has a significant impact. Not only the success probability matters for the model, but also the number of error types. Even with a single error type, the filtering method achieves a success probability of almost 90% at an *n*-value of 20, which is not reliable as a system but indicates that the filtering method is effective. It is possible to achieve a 100% success rate, as demonstrated by the given examples. The curve labeled 1% $\times$ 40 attains 100% success at $n = 2$, while 8% $\times$ 5 reaches 100% success at $n = 10$, and 10% $\times$ 4 does so at $n = 12$.

It can be concluded that removing consistent errors is important even if the overall success rate does not increase. When a larger variety of errors occurs at the same success rate, the filtering mechanism's likelihood of success improves.

Next, the number of errors and the success rate are fixed to examine the **effect of error probability distribution**. The success rate is set at 60%, with the number of errors fixed at 6. The distribution of the remaining 40% error probability is then varied across these 6 error types to analyze its impact. The numbers in the legend represent the distribution of error percentages across the issues.

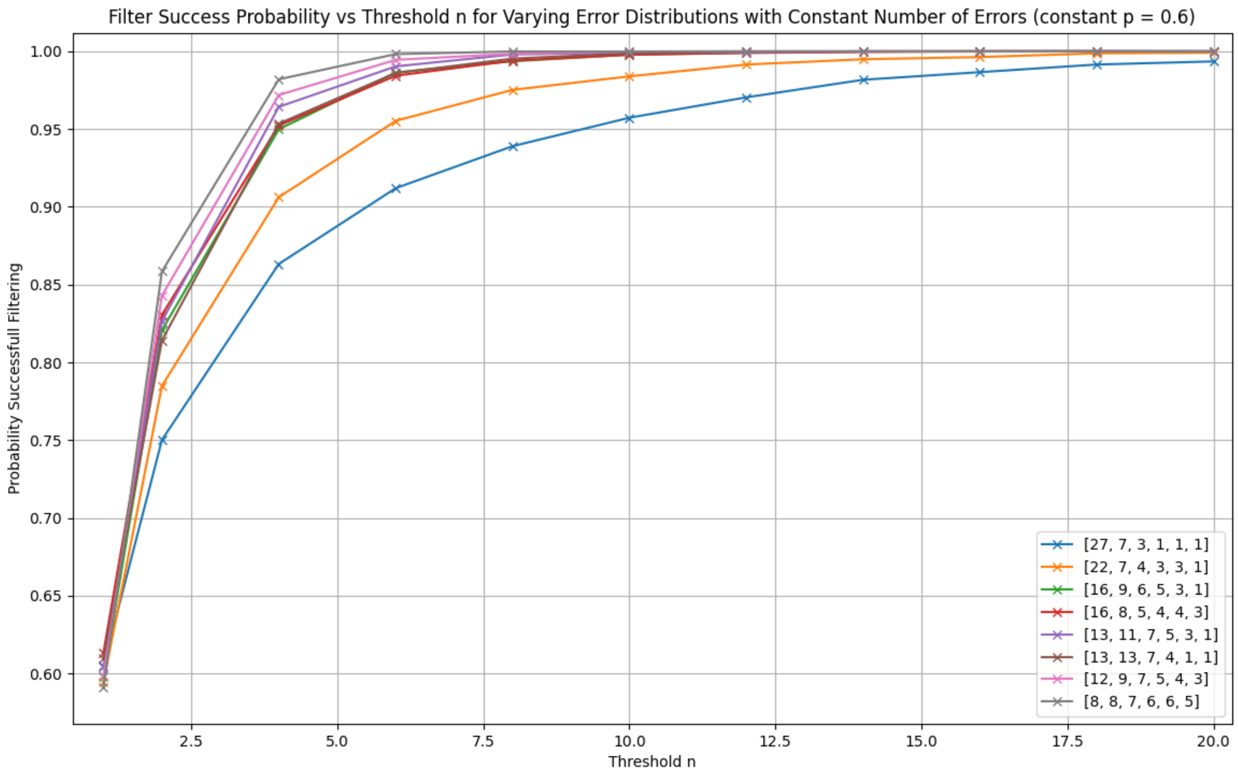


Figure 5.2: Success probability plotted against threshold *n* for different error distributions with a constant success rate of 60%. Each curve represents a distinct distribution of the total 40% error probability over six error types, as indicated by the labels.

It can be concluded that, given a constant success rate and a fixed number of errors, the distribution of errors affects the success probability of the filtering mechanism. Specifically, the more uniform the distribution of errors, the better the results.

Simulation 2: Varying Model Accuracy with Fixed Threshold

In addition to the previously described simulation setup, a complementary experiment is introduced to analyze how the required success rate of the model evolves with the number of error types. In this alternative approach, the threshold value n is fixed at 5, 10, and 15, while the model's success rate p is systematically varied from 0.0 to 1.0 in increments of 0.02. This setup allows for assessing the minimum level of reliability the model must achieve in order for the filtering mechanism to remain effective.

To isolate the influence of the number of incorrect variants, separate simulations are conducted for different error counts. For each case, it is assumed that the total probability mass assigned to incorrect outputs is $1 - p$, which is uniformly distributed across the specified number of error types. In other words, if there are k error types, each incorrect variant receives a probability of $(1 - p)/k$. This uniform error distribution represents a best-case scenario in which no single incorrect output dominates, and all errors are equally likely.

For each configuration, the simulation is executed 10 000 times to compute the probability that the correct variant reaches the threshold n before any incorrect variant does. By plotting the success probability of the filtering method against the model's success rate p, a set of curves is obtained—one for each number of error types—that illustrates how sensitive the filtering mechanism is to the model's accuracy.

This analysis provides insight into the minimum success rate required for reliable filtering, depending on how many equally likely error types are present. In practical terms, it helps identify under which conditions the filtering method is likely to fail due to excessive uncertainty in the model output, and when it can be expected to consistently converge to the correct structure.

These graphs can sometimes give a misleading impression when evaluating whether the filtering mechanism is effective, as they mostly reflect best-case scenarios for a given number of errors and a fixed success rate. It is not practical to generate graphs for all possible error distributions, since the space of such distributions is infinitely large. However, the worst-case scenario is still included: it occurs when the entire error probability is concentrated in a single incorrect output, represented by the curve with only one error type.

To properly interpret the graphs, it is advisable to first estimate the total error probability. If multiple errors occur with roughly equal frequency, the curve corresponding to that number of error types gives a reasonable approximation of the worst case. If the errors are unevenly distributed, it is safest to assume the true worst case, which corresponds to a single dominant error. Nonetheless, the most accurate method for assessing the filtering mechanism's effectiveness remains to run the original simulation with the exact parameters relevant to the specific case.

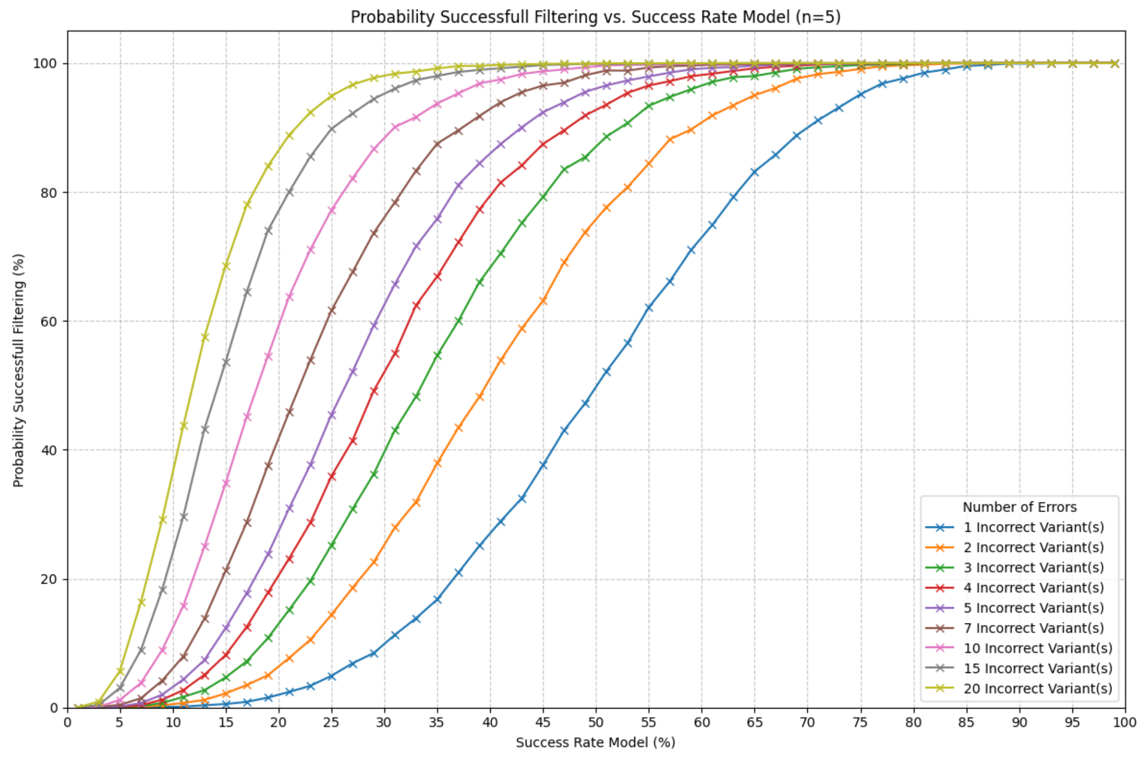


Figure 5.3: Success probability as a function of baseline success rate p, with a fixed threshold $n = 5$. Each curve reflects a different number of uniformly distributed error types over the total error probability $1 - p$. The plot shows how increasing error diversity raises the minimum required success rate for reliable filtering.

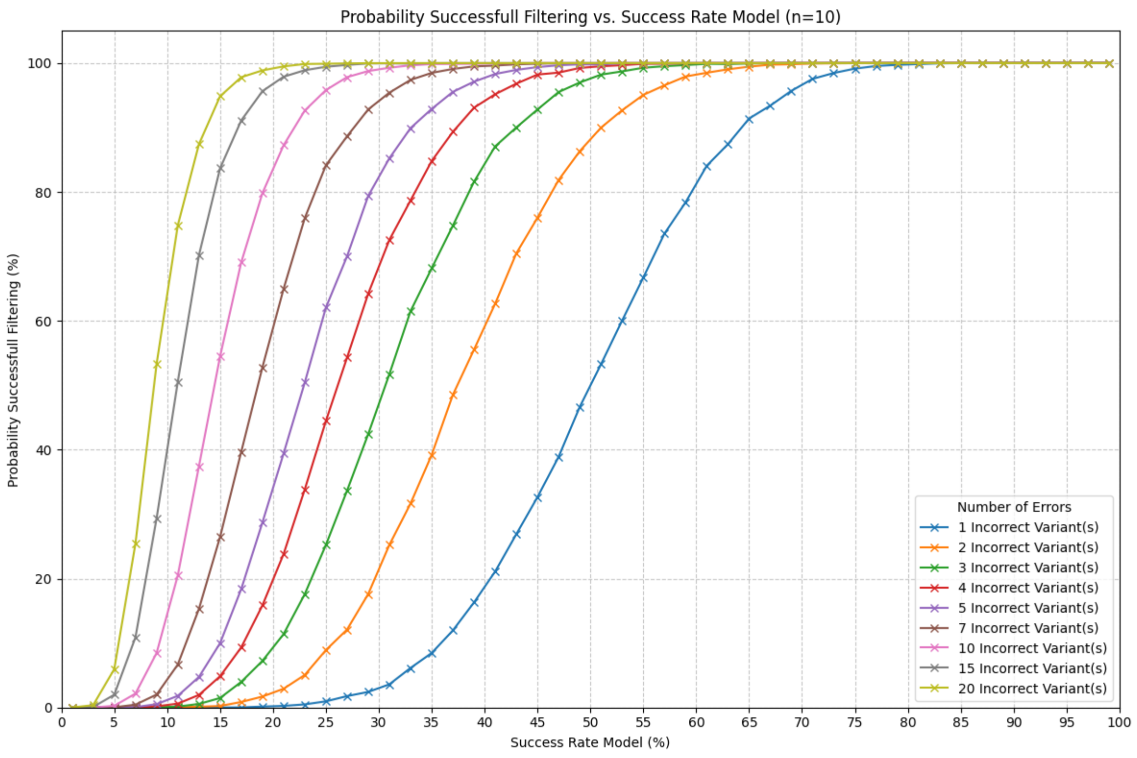


Figure 5.4: Success probability as a function of baseline success rate p, with a fixed threshold $n = 10$. Each curve reflects a different number of uniformly distributed error types over the total error probability $1 - p$. The plot shows how increasing error diversity raises the minimum required success rate for reliable filtering.

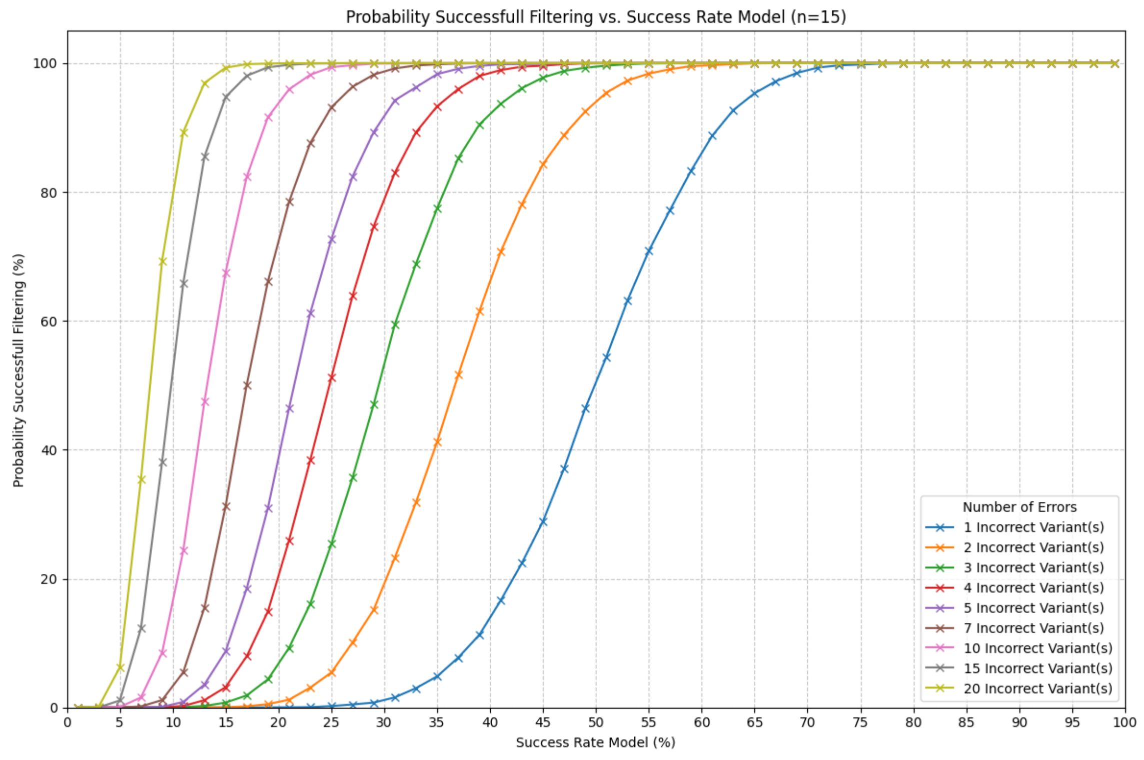


Figure 5.5: Success probability as a function of baseline success rate p, with a fixed threshold $n = 15$. Each curve reflects a different number of uniformly distributed error types over the total error probability $1 - p$. The plot shows how increasing error diversity raises the minimum required success rate for reliable filtering.

5.2 Test Case 1: DeliveryAgent and PlannerAgent

In this test case, the `PlannerAgent` sends a `request` to the `DeliveryAgent` to inquire about the delivery status. The `DeliveryAgent` responds with one of four possible statuses: `"INFORM"`, `"REFUSE"`, `"AGREE"`, or `"FAILURE"`.

This test primarily evaluates the following aspects of the system:

- **Perfect string-to-string conversion:** The source and target string vocabularies are identical in size and equivalent in meaning, allowing for a perfect match. This is tested in the `status` field of the `INFORM` reply.
- **Imperfect string-to-string conversion:** The source and target vocabularies both contain two entries that do not have direct counterparts. This tests whether the LLM only translates vocabulary that exists in both sets. This is evaluated using the `reason` field in the `REFUSE` and `FAILURE` replies.
- **Arbitrary string filtering:** The system's ability to detect and filter out arbitrary strings that do not exist in the target vocabulary is tested. If such a value occurs, it should be replaced with a `None` value. This behavior is examined using the `delivery_point` field in the `INFORM` and `AGREE` messages.

- **Redundant source messages:** The `DeliveryAgent` may send a `FAILURE` message that is not supported by the receiving API of the `PlannerAgent`. The system should recognize this and ensure that such a message is excluded from the message mapping.
- **Structural differences:** The `AGREE` message received by the `PlannerAgent` lacks one key compared to the original. Additionally, in the request sent by the `PlannerAgent`, a key is located in the `body` rather than the `metadata`.
- **Extracting from multiple files:** Both agents utilize a parameter file containing multiple parameters. This test evaluates the system's ability to correctly extract and interpret the structure of messages spread across multiple files.
- **Key name discrepancies:** All key names differ between source and target, testing the robustness of the key mapping mechanism.

So the *DeliveryAgent* is designed to send four types of outgoing messages and receive one type of incoming message. In contrast, the *PlannerAgent* sends one type of outgoing message and is capable of handling three types of incoming messages.

5.2.1 Results DeliveryAgent

Message Structure Extraction

We first evaluate the system using the default parameter values:

- **Success rate:** 100%
- **Average execution time:** 14.38 seconds

The DeliveryAgent has 4 types of outgoing messages and 1 type of incoming message. Additionally, three parameters from a separate file need to be included.

Message Mapping

The message mapping got a perfect score:

- Success rate: 100%
- Average execution time: 14.16 seconds

This test included one redundant message on the sending side, namely the `FAILURE` message. It is impressive that the system was able to achieve a perfect score without applying any optimization strategies (i.e., enforcing a minimum occurrence threshold). This suggests a strong inherent reliability of the model under the default configuration.

This test was executed 40 times to ensure that the results were not due to chance.

Key Mapping

The key mapping achieved a perfect score:

- Success rate: 100%
- Average execution time: 13.32 seconds

Three messages were mapped, containing the following key mappings:

- 4-to-2 mapping
- 5-to-4 mapping
- 4-to-2 mapping

There is no need to test higher values for `stable_required_key_mapping`, since the mapping was successful on the first attempt.

Vocabulary Mapping

The test also achieved a perfect score:

- Success rate: 100%
- Average execution time: 8.55 seconds

This test evaluates both a perfect 7-to-7 and an imperfect 8-to-8 string-to-string vocabulary mapping. The imperfect mapping includes two extra source words and two extra target words, resulting in 10 source and 10 target vocabulary entries, but the intended mapping remains 8-to-8.

The fact that the vocabulary mapping functions correctly without requiring additional optimization runs is a promising indication. It suggests that LLMs are highly capable of performing accurate semantic mappings between two distinct vocabulary sets, even under minimal conditions. This highlights the model's ability to generalize and align conceptually equivalent terms effectively.

5.2.2 Results PlannerAgent

Message Structure Extraction

We first evaluate the system using the default parameter values:

- **Success rate:** 93.33%
- **Average execution time:** 11.22 seconds

For the `PlannerAgent`, one type of sending message and three types of receiving messages must be extracted. In addition, four parameters from an external parameter file must be integrated into the extraction process.

Identified Issue: A total of two issues were identified, each occurring only once across 30 runs, resulting in a probability of 3.33% per error.

Problem 1 (3.33%, receiving): In this case, the LLM confused the content of the incoming `ACCEPT` message with that of the outgoing `REQUEST` message. The result was an amusing yet incorrect mix of message elements.

Problem 2 (3.33%, receiving): Here, the model correctly included the external parameters in the outgoing message but failed to do so for the incoming messages, despite those also requiring the parameters.

Evaluation with Increased Parameters

- `stable_required_extract_structure = 2`
- **Success rate:** 100%
- **Average execution time:** 22.66 seconds

Message Mapping

- **Success rate:** 100%
- **Average execution time:** 9.61 seconds

This fast and accurate performance is expected, as it was only a straightforward one-to-one mapping.

Key Mapping

- **Success rate:** 100%
- **Average execution time:** 10.96 seconds

The execution was also very fast, as there was only one message with a four-to-three key mapping.

Vocabulary Mapping

There are no string vocabulary to be mapped for the sending messages of the `PlannerAgent`.

5.2.3 Overview Results

Results DeliveryAgent:

Part of System	Success Rate	Average Time (s)
Message Structure Extraction	100%	14.38
Message Mapping	100%	14.16
Key Mapping	100%	13.32
Vocabulary Mapping	100%	8.55

Table 5.1: Performance of DeliveryAgent

Results PlannerAgent:

Part of System	Success Rate	Average Time (s)
Message Structure Extraction	93.33%	11.22
Message Structure Extraction (stability-based: 2)	100%	22.66
Message Mapping	100%	9.61
Key Mapping	100%	10.96
Vocabulary Mapping	N/A	N/A

Table 5.2: Performance of PlannerAgent

5.3 Test Case 2: ControllerAgent and HomeAgent

In this test case, two agents are involved: the `ControllerAgent` and the `HomeAgent`. The `ControllerAgent` supports two commands: one to adjust the temperature and another to switch a light on or off. The `HomeAgent` responds to both commands with a reply that includes the updated values. Additionally, every 15 seconds, the `HomeAgent` sends a status update to the `ControllerAgent`, including random values for `humidity`, `temperature`, and `light_status`.

This test focuses on the following challenges and issues:

- **Boolean-to-integer conversion:** When the `ControllerAgent` requests a light status change, the `status` field requires a conversion from boolean to integer format.

- **Integer-to-boolean conversion:** The `HomeAgent`'s reply to the light status adjustment involves converting the `status` field from integer back to boolean.
- **Integer-to-float conversion:** When adjusting the temperature, the `ControllerAgent` sends an integer, which needs to be converted to a float in the `HomeAgent`.
- **Float-to-integer conversion:** The `HomeAgent`'s reply for a temperature adjustment requires converting a float temperature value back to integer for the `ControllerAgent`.
- **One-to-many mapping:** The `HomeAgent`'s status update contains all three values (`humidity`, `temperature`, and `light_status`) in a single message. However, the `ControllerAgent` processes these using three distinct behaviours, one for each parameter. Therefore, the single status message must be mapped into three separate messages, each aligned with a specific behaviour.
- **Structural differences:** In the reply messages for both temperature and light adjustments, the `HomeAgent`'s message includes an extra field not required by the `ControllerAgent`. Moreover, in the one-to-many mapping, several fields present in the source message are absent in the corresponding target messages.
- **Key name discrepancies:** Several keys differ in naming between the two agents, requiring accurate key mapping.

So the *ControllerAgent* is capable of sending two types of messages and receiving five. The *HomeAgent*, on the other hand, sends three types of messages and receives two.

5.3.1 Results ControllerAgent

Message Structure Extraction

We first evaluate the system using the default parameter values:

- **Success rate:** 90%
- **Average execution time:** 14.31 seconds

In this case, two types of sending messages and five types of receiving messages are extracted.

Identified Issues: In total, three extraction errors were observed, consisting of two instances of the first problem and one of the second:

Problem 1 (6.66%, receiving): In two messages, a non-existent field `type` was hallucinated, although it is not defined in the behaviour file.

Problem 2 (3.33%, receiving): A type mismatch occurred. A field expected to be of type `int` was extracted as a vocabulary list of strings (`["0", "1"]`). While this is a plausible semantic interpretation—given that the behaviour file checks for equality with 0 or 1, it remains incorrect in terms of type definition.

Both issues occurred with low frequency. Therefore, increasing the stability threshold in the filtering mechanism would likely eliminate them reliably. For example, setting the threshold to 20 would require the exact same hallucinated or incorrect value to be generated 20 times before the correct structure was generated 20 times, which is highly unlikely. In the next section, we repeat the test with 30 runs and a threshold of 2 to assess whether a 100% success rate can be achieved under al little stricter filtering.

Evaluation with Increased Parameters

- `stable_required_extract_structure = 2`

- **Success rate:** 100%
- **Average execution time:** 32.27 seconds

Message Mapping

- **Success rate:** 100%
- **Average execution time:** 9.23 seconds

This was a 2-to-2 mapping of messages.

Key Mapping

- Success rate: 100%
- Average execution time: 10.54 seconds

Two messages were involved:

- One message with a 4-to-2 key mapping, including 2 redundant keys.
- One message with a 3-to-1 key mapping, also including 2 redundant keys.

Vocabulary Mapping

There are no string vocabulary to be mapped for the sending messages of the `ControllerAgent`.

5.3.2 Results HomeAgent

Message Structure Extraction

- **Success rate:** 96.67%
- **Average execution time:** 12.06 seconds

This agent is capable of sending three types of messages and receiving two types of messages.

Identified Issues: Only one error was observed in this evaluation:

Error (3.33%, sending): In a single run, the model hallucinated an extra key in two of the three outgoing messages. Since this occurred within the same run, it is counted as a single error out of 30.

Evaluation with Increased Parameters

- `stable_required_extract_structure = 2`
- **Success rate:** 100%
- **Average execution time:** 23.44 seconds

Message Mapping

- **Success rate:** 100%
- **Average execution time:** 12.74 seconds

This was a *one-to-many* mapping scenario, where a single status update message needed to be distributed across three distinct target messages. The mapping was performed perfectly without requiring any additional optimization steps. This outcome suggests a strong and effective implementation of the one-to-many mapping mechanism.

This test was executed 40 times to ensure that the results were not due to chance.

Key Mapping

- **Success rate:** 100%
- **Average execution time:** 11.24 seconds

This process involved mapping source keys to target keys for three outgoing messages:

- **Message 1:** 3 source keys mapped to 2 target keys.
- **Message 2:** 3 source keys mapped to 2 target keys.
- **Message 3:** 5 source keys mapped to 1 target key, repeated three times, as this single message was mapped to three distinct messages—each containing only one non-identifier key.

Vocabulary Mapping

There are no string vocabulary to be mapped for the sending messages of the `HomeAgent`.

5.3.3 Overview

Results ControllerAgent:

Part of System	Success Rate	Average Time (s)
Message Structure Extraction	90%	14.31
Message Structure Extraction (stability-based: 2)	100%	32.27
Message Mapping	100%	9.23
Key Mapping	100%	10.54
Vocabulary Mapping	N/A	N/A

Table 5.3: Performance of ControllerAgent

Results HomeAgent:

Part of System	Success Rate	Average Time (s)
Message Structure Extraction	96.67%	12.06
Message Structure Extraction (stability-based: 2)	100%	23.44
Message Mapping	100%	12.74
Key Mapping	100%	11.24
Vocabulary Mapping	N/A	N/A

Table 5.4: Performance of HomeAgent

5.4 Test Case 3: PlannerControllerAgent and DeliveryHomeAgent

In this test case, the PlannerAgent from Test Case 1 and the ControllerAgent from Test Case 2 are combined into a single agent: the *PlannerControllerAgent*. Similarly, the *DeliveryHomeAgent* is formed by merging the DeliveryAgent from Test Case 1 and the HomeAgent from Test Case 2. As a result, this test case integrates all the challenges identified in the previous two cases, which must now be handled simultaneously.

Below is a list of all challenges included. For detailed explanations, refer to the previous test cases:

- **Perfect string-to-string conversion**
- **Imperfect string-to-string conversion**
- **Arbitrary string filtering**
- **Redundant source messages**
- **Structural differences**
- **Extracting from multiple files**
- **Boolean-to-integer conversion**
- **Integer-to-boolean conversion**
- **Integer-to-float conversion**
- **Float-to-integer conversion**
- **One-to-many mapping**
- **Key name discrepancies**

The *PlannerControllerAgent* is capable of sending three types of messages and receiving eight. The *DeliveryHomeAgent* can send seven types of messages and receive three.

5.4.1 Results PlannerControllerAgent

Message Structure Extraction

- **Success rate:** 86.67%
- **Average execution time:** 20.00 seconds

For this agent, the system was required to extract three distinct types of *sending* messages and eight types of *receiving* messages. Additionally, four parameters from a separate parameter file needed to be included, which was also accomplished perfectly even across more messages then the PlannerAgent individually.

Identified Issues: A total of four distinct issues were identified, each occurring only once, corresponding to a probability of 3.33% per error. Such rare errors are easily eliminated by the stability-based filtering mechanism. The real challenge lies in systematic or consistent errors, which can potentially deceive the filtering process by passing the stability checks.

Problem 1 (3.33%, receiving): The LLM incorrectly added `arbitrary_string` as an additional vocabulary value for every key, even when a specific set of valid values was already defined.

Problem 2 (3.33%, receiving): A minor spelling mistake was introduced in the vocabulary list: `Charleroi` was misspelled as `Charlerol`.

Problem 3 (3.33%, receiving): A key mismatch occurred. The key `performed_action` was incorrectly extracted as `action`.

Problem 4 (3.33%, receiving): A type mismatch was observed. A field expected to be of type `int` was extracted as a vocabulary list of string values, namely `["0", "1"]`. While this interpretation is semantically reasonable, since the behaviour file checks whether the value equals 0 or 1, it is still incorrect in terms of type specification.

Evaluation with Increased Parameters

- `stable_required_extract_structure = 2`
- **Success rate:** 100%
- **Average execution time:** 38.51 seconds

Message Mapping

- **Success rate:** 100 %
- **Average execution time:** 11.34 seconds

This is a combination of the PlannerAgent and ControllerAgent, so the message mapping is a combination of the message mappings of the 2 agents combined. This is thus a one-to-one mapping and a 2-to-2 mapping of messages. Which makes this a 3-to-3 mapping of messages.

Key Mapping

- **Success rate:** 100%
- **Average execution time:** 8.96 seconds

This involved mapping source keys to target keys for three outgoing messages:

- **Message 1:** 3 source keys mapped to 2 target keys.
- **Message 2:** 3 source keys mapped to 2 target keys.
- **Message 3:** 5 source keys mapped to 1 target key, repeated three times, as this single message was mapped to three distinct messages—each containing only one non-identifier key.

Three messages were involved:

- One message with a 4-to-2 key mapping, including 2 redundant keys.
- One message with a 3-to-1 key mapping, also including 2 redundant keys.
- One message with a 4-to-3 key mapping, including 1 redundant key.

Vocabulary Mapping

There are no string vocabulary to be mapped for the sending messages of the `PlannerControllerAgent`.

5.4.2 Results DeliveryHomeAgent

Message Structure Extraction

- **Success rate:** 90%
- **Average execution time:** 20.61 seconds

In this case, the agent was expected to extract a total of seven distinct types of *sending* messages and three types of *receiving* messages. Additionally, three parameters from a separate configuration file needed to be extracted, which was accomplished flawlessly.

Identified Issues: A total of three distinct errors were encountered, each occurring with a probability of 3.33%. This diversity benefits the optimization, as it filters out inconsistent outputs effectively.

Error 1 (3.33%, receiving): The model incorrectly split one expected message into two separate parts, distributing keys that should be grouped.

Error 2 (3.33%, receiving): A valid reply message was split into two messages, one containing a specific key and the other without.

Error 3 (3.33%, sending): The model produced structurally inconsistent and semantically confused outgoing messages, reflecting its probabilistic nature.

Evaluation with Increased Parameters

- `stable_required_extract_structure = 2`
- **Success rate:** 100%
- **Average execution time:** 45.31 seconds

Message Mapping

- **Success rate:** 96.67 %
- **Average execution time:** 22.87 seconds

This agent is capable of sending seven messages, while the receiving agent (PlannerController) can receive eight distinct messages. Among the sending messages, the `FAILURE` message is redundant and should not be mapped. Additionally, the status update message must be mapped to three separate receiving messages of the PlannerController Agent. This results in a 7-to-8 message mapping scenario that includes both a redundant sending message and a *one-to-many* mapping case. As such, this test serves as a strong benchmark for evaluating the system's message mapping capabilities.

Identified Issues: In this case, the model failed to map the `INFORM` message to the corresponding `REPORT` message of the `PlannerControllerAgent`. This error was unexpected and unrelated to the inherent challenges of the test scenario. It likely reflects a moment of inconsistency by the language model, a well-known weakness of large language models. The error occurred only once out of 30 runs, corresponding to a probability of 3.33%. As such, it should be easily filtered out by the stability-based optimization strategy.

Evaluation with Increased Parameters

- `stable_required_message_mapping = 2`
- **Success rate:** 100%
- **Average execution time:** 46.07 seconds

Key Mapping

- **Success rate:** 100%
- **Average execution time:** 21.13 seconds

This process was also executed with a perfect success rate of **100%**. It involved mapping source keys to target keys for six outgoing messages:

- **Message 1:** 4 source keys mapped to 2 target keys.

- **Message 2:** 5 source keys mapped to 2 target keys.
- **Message 3:** 4 source keys mapped to 2 target keys.
- **Message 4:** 3 source keys mapped to 2 target keys.
- **Message 5:** 3 source keys mapped to 2 target keys.
- **Message 6:** 5 source keys mapped to 1 target key, repeated three times, as this single message was mapped to three distinct messages—each containing only one non-identifier key.

Vocabulary Mapping

- **Success rate:** 100%
- **Average execution time:** 8.84 seconds

This test evaluates both a perfect 7-to-7 and an imperfect 8-to-8 string-to-string vocabulary mapping. The imperfect mapping includes two extra source words and two extra target words, resulting in 10 source and 10 target vocabulary entries, but the intended mapping remains 8-to-8.

5.4.3 Overview

Results PlannerControllerAgent:

Part of System	Success Rate	Average Time (s)
Message Structure Extraction	86.67%	20.00
Message Structure Extraction (stability-based: 2)	100%	38.51
Message Mapping	100%	11.34
Key Mapping	100%	8.96
Vocabulary Mapping	N/A	N/A

Table 5.5: Performance of PlannerControllerAgent

Results DeliveryHomeAgent:

Part of System	Success Rate	Average Time (s)
Message Structure Extraction	90%	20.61
Message Structure Extraction (stability-based: 2)	100%	45.31
Message Mapping	96.67%	22.87
Message Mapping (stability-based: 2)	100%	46.07
Key Mapping	100%	21.13
Vocabulary Mapping	100%	8.84

Table 5.6: Performance of DeliveryHomeAgent

5.5 Conclusion

In this conclusion, the results of the evaluation will be discussed, and conclusions will be drawn based on those results. At the beginning of this chapter, several objectives were defined to guide the evaluation of the model. These objectives are now revisited to systematically assess whether they have been met, thereby providing a structured conclusion regarding the model's performance in the evaluation.

O10 – Ensure that the model is scalable to handle a growing number of agents, maintaining performance and efficiency as the system expands.
As discussed in Chapter 4, the Formula Generation Phase, which consists of message mapping, key mapping, and vocabulary mapping, can potentially be a bottleneck for the scalability of this model. The execution time increases linearly with the number of agents that are included in the translated communication. An important note is that this scaling depends on the number of agents that are initially contacted, so normally each target agent should only count once in this scaling metric.

As shown in the results, the Formula Generation Phase works efficiently. The longest time measured for this phase is still under 1 minute and 30 seconds. This was measured with a filter threshold of 1 for the key mapping and vocabulary mapping, and a threshold of 2 for the message mapping. Still, this is a good result. To estimate how long this phase would take with a parameter value of 4, a conceptual estimation can be made by multiplying the time by the factor of the parameter value for a best-case scenario. If the performance is slightly worse and 2 errors are made in the 4 runs, which means each step is executed 6 times in total, the total time would be 5 minutes and 18 seconds. This is still a very good and efficient result. Therefore, the Formula Generation Phase does not form a strict bottleneck.

O12 – Benchmark and validate the performance and robustness of the implementation.
In three distinct test scenarios, various challenges were implemented to evaluate the model's ability to handle them. The results are very positive, indicating that the model can consistently produce correct outputs even in complex situations involving multiple challenges and larger agent APIs (test case 3).

A success rate of 100% was achieved with a filtering threshold as low as 2, demonstrating strong model consistency. This confidence can be further increased by raising the threshold without significantly impacting resource usage. Overall, the model operates efficiently and quickly, with execution times consistently under one minute for each step at threshold 2. Execution time depends largely on the size of the input data, but the low baseline runtime without filtering suggests that higher thresholds remain feasible.

Most difficulties were encountered during structure extraction and occasionally in message mapping, but overall error rates were low and errors were well distributed, favorable conditions for the filtering method. Key mapping proved very robust and consistent, with no observed errors, as expected given its straightforward nature. Vocabulary mapping showed similar reliability.

In summary, the model demonstrates strong robustness against various challenges and their combinations, with a lowest success rate of 86.67% without filtering optimization in the structure extraction phase. No consistent errors were detected, and errors appeared uniformly distributed, representing an ideal scenario for the filtering approach. Overall, the implementation can be regarded as highly successful.

O13 – Evaluate the autonomy and reliability the proposed translation framework and determine whether human intervention is necessary. The accuracy of the implementation appears promising, as the first two test cases address distinct challenges that complicate the process, while the third combines these challenges into one big test case. Despite being limited to only three test cases, a variety of potential problem scenarios have been effectively evaluated. Notably, the final test case achieves a strong success rate even without applying the filtering method. Introducing the filtering method with a higher threshold, such as 10, should further enhance the system's robustness.

Based on these results, one could argue that the implementation is robust enough for fully autonomous deployment. Achieving a 100% success rate across all tests with a threshold of 2 indicates high consistency. To ensure greater robustness and reliability, increasing the threshold to 5 or even 10 is advisable.

However, it is crucial to temper this optimism. Evaluating only three test cases is insufficient to draw definitive conclusions, particularly in complex environments like multi-agent systems where agents may differ significantly. The primary risk of limited testing is overfitting, meaning the solution may perform well only on the tested scenarios. Although care was taken during development to mitigate overfitting, a broader testing regime is needed for conclusive validation.

Furthermore, given that this model is still in an early stage, human oversight remains essential at every step. There are undoubtedly minor issues to resolve, and additional testing and optimization are required before the system can be considered fully mature.

O15 - Identify current limitations of the model.
A fundamental and straightforward limitation when translating messages between two heterogeneous agents is that both agents must support the same basic functionality. This means that even if one agent can send a message and the translation system can perfectly convert it into the format of the other agent, the translation will still fail if the receiving agent does not recognize or support the purpose of the message.

In practical terms, this refers to the underlying API endpoints or commands available to each agent. For example, if one agent sends a message to "start a task," but the receiving agent does not have any functionality to start tasks, then translation is not possible, no matter how well the message is structured or how accurately it is converted. Although the message format, vocabulary, or structure may vary between agents, the actual meaning—the intent behind the message—must be something that both agents can understand and act upon. Therefore, successful translation depends not only on language or structure, but also on functional compatibility between agents.

Another limitation is the lack of mechanisms to verify whether the output produced by the translation formula is actually correct. Since the system generates translations for interactions that do not inherently exist, and the agents themselves are not designed to support translation-based behavior, there is currently no runtime validation mechanism to confirm the correctness of the translated output. At this stage, the only safeguards available are reliance on the accuracy of the large language model (LLM) and the filtering mechanism designed to select the most likely correct result.

For message structure extraction, partial validation may become possible through log-based feedback, as discussed in the proposed extensions in Chapter 4. Such feedback would provide valuable insight into whether the model is performing as expected. For the other phases, it is conceivable that with further engineering effort, some partial validation or rule-based error detection could be implemented, helping to catch certain types of mistakes. However, at present, only a hand full of such mechanisms are in place. This means the system almost fully depends on the proper functioning of the LLM, and any errors in translation may go undetected until a downstream failure occurs.

For this reason, the development of an advanced and well-designed error handling system, including support for negative feedback, is critical for any future deployment in production environments. Such a system would play a key role in detecting failures early and improving the reliability of the translation framework.

Chapter 6

Conclusion

This section evaluates whether the main research question — "**How can a modular translation layer enable accurate and scalable payload translation between heterogeneous imperative-based agents without requiring modifications to their internal logic?**" — has been effectively answered. This will be done by first addressing each of the secondary research questions, which collectively support the answer to the main research question. Subsequently, a concise summary will be provided that directly answers the main research question, based on the insights obtained from the secondary questions. Finally, a reflective conclusion is presented regarding the implemented model, emphasizing the achieved results, the relevance of these results, and the potential for future developments.

6.1 Reviewing the Secondary Research Questions

Answer to SRQ1 – Can the translation layer be deployed across different agents without requiring manual intervention or changes to their internal logic?

Yes, the translation layer can be deployed across different agents without requiring manual intervention or changes to their internal logic. This is made possible through several key design choices that ensure modularity, independence, and compatibility with existing systems.

The system supports distributivity by allowing each agent to generate and use its own translation logic independently. There is no need for a central authority or shared format; messages are translated locally by the sending agent into a format that the receiving agent can understand. This greatly reduces system complexity and ensures that communication remains efficient and reliable, even in distributed environments. The only caveat is the current reliance on the Gemini API, which introduces a centralized dependency on the google servers. A fully decentralized version would involve deploying the language model locally or on a private server.

Importantly, no modifications are required to the agents' original behavior files. Message structures are extracted directly from these files without any preprocessing or alterations. Furthermore, translation logic can be integrated by overriding the agent's existing `send()` function, which preserves the agent's original behavior and avoids intrusive changes. This ensures that agents remain functional as originally designed while still gaining translation capabilities.

Altogether, these features confirm that the translation layer is modular and adaptable enough to operate across different agents without requiring manual changes to their internal structure, while still maintaining accuracy, efficiency, and scalability.

Answer to SRQ2 – How can the run-time translation be made deterministic and efficient?
A deterministic and efficient runtime translation can be achieved by designing the translation process around a static and repeatable structure, the formula. This is done using three fixed files: `template.json`, `key_mapping.csv`, and `voc_mapping.csv`. These files are used as input for the `translate()` function, which does not rely on any randomness or external computation. As long as the same message is translated under the same conditions, the output will always be identical. This guarantees full consistency, which is critical for reliable communication between agents. More details about this process are provided in the sections that describe formula generation and runtime translation.

Efficiency is ensured by avoiding complex or expensive operations during runtime. Once the formula has been generated, the translation is handled through basic static code. The system uses a precomputed `sending_identifiers.json` file to quickly determine which formula should be used. No language model or external service is involved during this runtime phase. As a result, messages can be translated almost instantly, with very low computational overhead.

Scalability is addressed by dividing the system into clear phases. The extraction of an agent's message structure only needs to be done once, unless the agent changes its behavior. The more intensive Formula Generation Phase only needs to be performed once for each pair of agents that will communicate. Although this generation step grows linearly with the number of agents, it remains manageable because each translation formula is reused indefinitely once created. The runtime translation itself remains fast and efficient, no matter how many agents are in the system. This approach allows the system to scale effectively, even in large and dynamic environments.

Answer to SRQ3 – How can the problems of syntactic discrepancies and semantic ambiguity be solved?
Syntactic discrepancies can be resolved by preserving the full structure of the messages during translation. This is achieved by first using a language model to extract the complete structure of each message. The extracted structures are then stored in a `template.json` file, which acts as a fixed blueprint. During translation, this template is used to reconstruct the message, with only the values being filled in while the original structure remains unchanged. This ensures that any differences in message formatting between agents are properly handled, and the integrity of the structure is maintained throughout the process.

Semantic ambiguity is addressed by focusing on the meaning and usage context of each key, not just its name. A language model generates detailed explanations for every key, based on how the key is used in the agent's behavior file. These explanations describe the key's meaning, role, and purpose. When mapping keys between two agents, another language model compares these explanations and only links keys when their meanings align. This reduces the risk of incorrect mappings caused by similar names with different meanings or different names with the same

Answer to SRQ4 - Does this solution solve the information loss or information dilution problem that exists with standardization methods?
Yes, this solution avoids the problem of information loss or dilution commonly found in standardization-based methods. In many traditional systems, messages must first be converted into a shared standard format before being sent to another agent. This intermediate format often supports only a limited set of features, which can result in information being

simplified, rounded off, or completely omitted. As a result, some important details may be lost during translation.

The solution presented here uses a direct translation approach, where each pair of agents communicates through a custom translation formula without converting messages into a shared standard. This ensures that the full content of the message is preserved, as long as the receiving agent supports the necessary features. By translating directly from one agent to another, the system maintains the integrity and richness of the original information, fully avoiding the reduction in detail that happens with standardized formats.

Answer to SRQ5 – Is human oversight required to validate or correct message mappings, or is the system consistent and accurate enough to operate fully autonomously? Human oversight is currently still required to ensure correctness, although the system shows strong potential for autonomous operation. Extensive testing was carried out across three different scenarios, each designed to introduce specific challenges. The system demonstrated impressive performance, achieving up to 100% accuracy when a filtering threshold of 2 was applied. Even in the most complex scenario—where multiple challenge types and large agent APIs were combined—the model maintained a high level of reliability. These results suggest that the system is capable of functioning consistently and efficiently in controlled settings.

Despite this promising performance, the current stage of development requires caution. Only three test cases have been evaluated so far, which is not enough to fully guarantee robustness in all real-world situations. Multi-agent environments can be highly diverse, and additional testing is necessary to confirm that the model generalizes well. There is also the risk of overfitting, where the solution might perform well only in familiar or previously tested cases.

Therefore, human oversight remains important for now, especially in cases involving new or complex agents. While the filtering mechanism and overall framework show strong potential for autonomous use, more extensive testing and further refinement are needed before the system can operate completely without supervision.

Answer to SRQ6 – What are the current limitations of the approach, and how could future work improve coverage, accuracy, or adaptability of the translation process? The current translation approach faces several important limitations that affect its accuracy, coverage, and adaptability. First, there is a fundamental requirement that both communicating agents must support the same core functionality. Even if a message is perfectly translated at the structural and linguistic level, the translation will still fail if the receiving agent does not recognize or support the intended purpose of the message. This issue lies outside the scope of message formatting or vocabulary and instead concerns the functional compatibility between agents. Without shared capabilities, meaningful translation cannot occur.

Another major limitation is the absence of a proper validation mechanism for checking whether the translation output is correct. Since most translations are created by an LLM and used without runtime confirmation, mistakes in translation may go unnoticed until a downstream error appears. This makes the system highly dependent on the model's internal reasoning and the filtering mechanism, which are not foolproof. Without structured feedback, errors cannot be systematically corrected or even detected. This limitation directly affects the system's reliability and restricts its deployment in production environments.

Additionally, the current system assumes that agents communicate in a predictable request-response format and that all message fields are mandatory. These simplifications reduce complexity but also limit flexibility. The absence of support for optional fields, error handling, or dynamic message sequencing further constrains the range of situations the system can manage.

Future work could address these limitations in several ways. Introducing log-based validation could provide feedback about the correctness of translations. Supporting optional fields would improve adaptability to real-world data structures. A better understanding of vocabulary semantics, particularly for functionally meaningful values, could prevent incorrect message behavior. Handling lists and dictionaries more intelligently, enabling unit conversions, and supporting dynamic selection in one-to-many mappings would also expand the system's scope and precision. Furthermore, improving error detection and propagation would help recover from translation failures and guide system corrections.

Finally, extending the model to support XML-based content formats, many-to-one mappings, and cross-language compatibility would make the approach more robust and portable. All these enhancements could improve not just accuracy, but also the practical usability of the system in diverse environments.

6.2 Answer to Main Research Question

The research question guiding this dissertation was: *"How can a modular translation layer enable accurate and scalable payload translation between heterogeneous imperative-based agents without requiring modifications to their internal logic?"*

The proposed solution demonstrates that a modular translation layer, powered by a large language model (LLM), can indeed serve as an effective intermediary between agents with different payloads, which involves different message structures, vocabularies, and formats. The key innovation lies in the system's modular design, which separates the translation process into three distinct phases: message structure extraction, formula generation and runtime translation. Each phase operates subsequently but independently, making the overall system highly adaptable, scalable and maintainable.

By relying on LLM-driven prompts and schema inference based on behavior files—rather than hardcoded transformation rules—the translation layer can dynamically interpret the structure and intent of agent messages. Although the system requires access to the agents' behavior descriptions as input, it does not depend on their internal implementation logic or source code. This allows the system to generate translation formulas without modifying the agents themselves. As a result, the encapsulation and autonomy of each agent are preserved, which is essential for ensuring scalability and reusability in heterogeneous environments.

Furthermore, once translation formulas are generated and stored, no further LLM calls are needed during live communication. This design choice ensures that runtime performance remains efficient and predictable. The use of a filtering mechanism to select the most reliable LLM outputs further contributes to the accuracy of the generated translations.

In summary, the dissertation shows that modularity, combined with strategic use of LLMs for initial inference, can provide a scalable and accurate translation mechanism that functions independently of agent internals. Although the system still has limitations—particularly in validation, error handling, and semantic disambiguation—it lays a strong foundation for a translation framework that enables seamless communication between otherwise incompatible imperative-based agents.

6.3 Final Reflection

Overall, this implementation successfully meets its primary objective: building a system that introduces an abstraction layer on top of existing agents, enabling distributed and static payload translation without requiring any modifications to the agents' internal logic.

The design ensures that the Message Structure Extraction phase is performed only once per agent, making it both efficient and scalable. This allows the application of a stricter threshold in the filtering mechanism to guarantee robustness, as it only happens once per agent. The message mapping phase, executed once for each relevant agent pair, requires more cumulative effort as the number of agents increases. However, since it too is a one-time setup, it represents a worthwhile investment that enables fast, low-cost, static, and personalized translation at runtime.

One key limitation of this work lies in the scope of testing. While the implementation performed excellently on the defined test cases, the limited number of scenarios makes it difficult to draw definitive conclusions about its general applicability. This is especially relevant in multi-agent environments where high variability is common. Constructing and evaluating realistic test cases is time-consuming, and due to time constraints, more extensive testing was not feasible. Ideally, additional test scenarios and stress tests on the four core components of the system would be included as future work.

Another limitation of this work in practical applications is that the model only addresses the problem of payload translation and assumes that agents are able to discover each other and initiate communication. However, in real-world scenarios, heterogeneous agents often use different communication protocols and Agent Communication Languages (ACLs). Systems like the Rule Responder System (explained in Chapter 2) can adopt translated payload and manage heterogeneity at the lower layers of communication. This system also includes its own payload translation mechanism, but it relies on standardization, which limits full information transfer and dynamic adaptability. In this context, the proposed architecture for a translation model can offer a valuable contribution to the broader challenge of enabling communication between imperative-based agents in heterogeneous environments.

Despite these limitations, the dissertation fulfills its objective of proposing and prototyping a model for the distributed generation of static translation formulas using Large Language Models (LLMs). While not production-ready, the results clearly demonstrate the feasibility of the proposed architecture.

With LLMs continuously improving in accuracy, consistency, and capacity, this concept is likely to become increasingly viable. Even during the short implementation window (March to June), the field progressed significantly. The release of the Gemini model on May 20, which was used in this project, provided a substantial boost in quality and implementation speed.

In summary, this dissertation presents an efficient and functional model for distributed payload translation in open, heterogeneous multi-agent systems using LLMs. While still in a prototypical stage, the concept offers a strong foundation for future development and optimization.

Bibliography

[1] Nedaa H. Ahmed et al. “Internet of Things Multi-protocol Interoperability with Syntactic Translation Capability”. In: *International Journal of Advanced Computer Science and Applications* 12.9 (2021). doi: `10.14569/IJACSA.2021.0120969`. url: `http://dx.doi.org/10.14569/IJACSA.2021.0120969`.

[2] Abul Ehtesham et al. *A survey of agent interoperability protocols: Model Context Protocol (MCP), Agent Communication Protocol (ACP), Agent-to-Agent Protocol (A2A), and Agent Network Protocol (ANP)*. 2025. arXiv: `2505.02279 [cs.AI]`. url: `https://arxiv.org/abs/2505.02279`.

[3] Mario de la Parte et al. “SISS: Semantic Interoperability Support System for the Internet of Things”. In: *IEEE Internet of Things Journal* PP (Jan. 2025), pp. 1–1. doi: `10.1109/JIOT.2025.3577776`.

[4] Jacob Nilsson et al. “AI Concepts for System of Systems Dynamic Interoperability”. In: *Sensors* 24.9 (2024). issn: 1424-8220. doi: `10.3390/s24092921`. url: `https://www.mdpi.com/1424-8220/24/9/2921`.

[5] Negin Jahanbakhsh et al. “Leveraging Retrieval-Augmented Generation for Automated Smart Home Orchestration”. In: *Future Internet* 17.5 (2025). issn: 1999-5903. doi: `10.3390/fi17050198`. url: `https://www.mdpi.com/1999-5903/17/5/198`.

[6] J. Moreira, L. Ferreira Pires, and M. Van Sinderen. *SEMIoTICS: Semantic Model-Driven Development for IoT Interoperability of Emergency Services*. Universitat Politecnica de Valencia, Jan. 2025.

[7] Naveen Krishnan. *Advancing Multi-Agent Systems Through Model Context Protocol: Architecture, Implementation, and Applications*. 2025. arXiv: `2504.21030 [cs.MA]`. url: `https://arxiv.org/abs/2504.21030`.

[8] Rafael C. Cardoso and Angelo Ferrando. “A Review of Agent-Based Programming for Multi-Agent Systems”. In: *Computers* 10.2 (2021). issn: 2073-431X. doi: `10.3390/computers10020016`. url: `https://www.mdpi.com/2073-431X/10/2/16`.

[9] P. G. Balaji and D. Srinivasan. “An Introduction to Multi-Agent Systems”. In: *Innovations in Multi-Agent Systems and Applications - 1*. Ed. by Dipti Srinivasan and Lakhmi C. Jain. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–27. isbn: 978-3-642-14435-6. doi: `10.1007/978-3-642-14435-6_1`. url: `https://doi.org/10.1007/978-3-642-14435-6_1`.

[10] Pengyu Zhao, Zijian Jin, and Ning Cheng. *An In-depth Survey of Large Language Model-based Artificial Intelligence Agents*. 2023. arXiv: `2309.14365 [cs.CL]`. url: `https://arxiv.org/abs/2309.14365`.

[11] Ali Dorri, Salil S. Kanhere, and Raja Jurdak. “Multi-Agent Systems: A Survey”. In: *IEEE Access* 6 (2018), pp. 28573–28593. doi: `10.1109/ACCESS.2018.2831228`.

[12] Costin Bădică, Lars Braubach, and Adrian Paschke. “Rule-Based Distributed and Agent Systems”. In: *Rule-Based Reasoning, Programming, and Applications*. Ed. by Nick Bassiliades, Guido Governatori, and Adrian Paschke. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 3–28. isbn: 978-3-642-22546-8.

[13] T. Tammet et al. "A rule-based approach to Web-based application development". In: *2006 7th International Baltic Conference on Databases and Information Systems*. 2006, pp. 202–208. doi: 10.1109/DBIS.2006.1678497.

[14] Alessandro Beneventi et al. "Integrating rule and agent-based programming to realize complex systems". In: *WSEAS Trans. on Information Science and Applications* 1.1 (2004), pp. 422–427.

[15] Carlos F Lopez et al. "Programming biological models in Python using PySB". In: *Molecular Systems Biology* 9.1 (2013), p. 646. doi: https://doi.org/10.1038/msb.2013.1. eprint: https://www.embopress.org/doi/pdf/10.1038/msb.2013.1. url: https://www.embopress.org/doi/abs/10.1038/msb.2013.1.

[16] Artur Freitas, Rafael H. Bordini, and Renata Vieira. "Model-Driven Engineering of Multi-Agent Systems Based on Ontologies". In: *Applied ontology* 12.2 (2017), pp. 157–188. doi: 10.3233/ao-170182.

[17] Nayat Sánchez Pi. "Intelligent techniques for context-aware systems". PhD thesis. Universidad Carlos III de Madrid, 2011.

[18] Heikki Helin et al. "Efficient Networking for Pervasive eHealth Applications". In: (Jan. 2006).

[19] Maricela Bravo et al. "Multi-agent Communication Heterogeneity". In: *2015 International Conference on Computational Science and Computational Intelligence (CSCI)*. 2015, pp. 583–588. doi: 10.1109/CSCI.2015.167.

[20] Wayne Xin Zhao et al. "A Survey of Large Language Models". In: *ArXiv* abs/2303.18223 (2023). url: https://api.semanticscholar.org/CorpusID:257900969.

[21] Mohaimenul Azam Khan Raiaan et al. "A Review on Large Language Models: Architectures, Applications, Taxonomies, Open Issues and Challenges". In: *IEEE Access* 12 (2024), pp. 26839–26874. url: https://api.semanticscholar.org/CorpusID:267675587.

[22] Pranjal Kumar. "Large language models (LLMs): survey, technical frameworks, and future challenges". In: *Artif. Intell. Rev.* 57 (2024), p. 260. url: https://api.semanticscholar.org/CorpusID:271961846.

[23] Yurong Liu et al. *Magneto: Combining Small and Large Language Models for Schema Matching*. 2024. arXiv: 2412.08194 [cs.DB]. url: https://arxiv.org/abs/2412.08194.

[24] Eitam Sheetrit et al. *ReMatch: Retrieval Enhanced Schema Matching with LLMs*. 2024. arXiv: 2403.01567 [cs.DB]. url: https://arxiv.org/abs/2403.01567.

[25] Marcel Parciak et al. *Schema Matching with Large Language Models: an Experimental Study*. 2024. arXiv: 2407.11852 [cs.DB]. url: https://arxiv.org/abs/2407.11852.

[26] Aman Bhargava et al. *What's the Magic Word? A Control Theory of LLM Prompting*. 2024. arXiv: 2310.04444 [cs.CL]. url: https://arxiv.org/abs/2310.04444.

[27] Yifan Song et al. *The Good, The Bad, and The Greedy: Evaluation of LLMs Should Not Ignore Non-Determinism*. 2024. arXiv: 2407.10457 [cs.CL]. url: https://arxiv.org/abs/2407.10457.

[28] Lingjiao Chen et al. *Are More LLM Calls All You Need? Towards Scaling Laws of Compound Inference Systems*. 2024. arXiv: 2403.02419 [cs.LG]. url: https://arxiv.org/abs/2403.02419.

[29] Adrian Paschke and Harold Boley. "Rule Responder: Rule-Based Agents for the Semantic-Pragmatic Web." In: *International Journal on Artificial Intelligence Tools* 20 (Dec. 2011), pp. 1043–1081. doi: 10.1142/S0218213011000528.

[30] Sirui Hong et al. *MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework*. 2024. arXiv: 2308.00352 [cs.AI]. url: https://arxiv.org/abs/2308.00352.

[31] Yuan-An Xiao et al. *PredicateFix: Repairing Static Analysis Alerts with Bridging Predicates*. 2025. arXiv: 2503.12205 [cs.SE]. url: https://arxiv.org/abs/2503.12205.

[32] Changhua Pei et al. *Flow-of-Action: SOP Enhanced LLM-Based Multi-Agent System for Root Cause Analysis*. 2025. arXiv: 2502.08224 [cs.SE]. url: https://arxiv.org/abs/2502.08224.

[33] Google AI. *Prompting Strategies | Gemini API*. https://ai.google.dev/gemini-api/docs/prompting-strategies. Accessed: 2025-06-30. 2024.

[34] Bruno Ribeiro et al. “Python-based Ecosystem for Agent Communities Simulation”. In: *SOCO 2022* 1 (2022), pp. 1–10. doi: 10.1109/SOCO2022.000123.

[35] GECAD Group. *PEAK: Python-based framework for heterogeneous agent communities*. https://gecad-group.github.io/peak-mas/. GitHub repository: https://github.com/gecad-group/peak-mas. 2025. url: https://gecad-group.github.io/peak-mas/.

[36] José R. Flórez and contributors. *SPADE: Smart Python multi-Agent Development Environment*. https://spade-mas.readthedocs.io/. Accessed: 2025-06-30. 2024.