Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Towards Advancing Modular AI Planning Systems: Insights from the PlanX Toolbox Redesign

Daniel Six

Course of Study:	Informatik
Examiner:	Dr. Ilche Georgievski
Supervisor:	Dr. Ilche Georgievski
Commenced:	September 1, 2024
Completed:	April 1, 2025

Abstract

As Artificial Intelligence (AI) planning systems grow in complexity and are deployed in increasingly diverse application contexts, existing frameworks often struggle with modularity, reusability, and tool interoperability. These limitations hinder the integration of heterogeneous planning tools and reduce the adaptability of planning systems to evolving requirements. This thesis addresses these issues by redesigning the PlanX Toolbox, an open-source framework for composing AI planning functionalities based on Service-Oriented Architecture (SOA) and Component-Based Software Engineering (CBSE) principles.

Through a detailed architectural analysis of the original PlanX system, several constraints were identified, including rigid service interactions, limited extensibility, and tightly coupled parsing, converting, and planning components. In response, a redesigned architecture was developed featuring a modular *Generation Unit* that unifies conversion and plan generation while maintaining parsing as an independent component. This restructuring improves the separation of concerns and supports more flexible composition of planning workflows. The redesign also introduces standardized message structures and dynamic planner selection, simplifying the integration of new components.

The evaluation includes the integration of the Fast Downward planner, which revealed both the potential and the limitations of reusing standardized input formats like Planning Domain Description Language (PDDL) across diverse planning backends. While conversion and generation could be modularized effectively, parser reusability remained constrained by tool-specific assumptions and preprocessing steps. These findings highlight key insights into the practical challenges of building modular AI planning systems: true composability requires not only standardized interfaces but also architectural awareness of planner internals.

By proposing and validating a refined system architecture, this thesis provides actionable design patterns for enhancing maintainability, extensibility, and integration in AI planning systems. The findings offer both theoretical and practical contributions to advancing modular AI planning frameworks.

Kurzfassung

Mit zunehmender Komplexität und Vielfalt der Einsatzbereiche stoßen bestehende Artificial Intelligence (AI) Planning-Systeme häufig an ihre Grenzen hinsichtlich Modularität, Wiederverwendbarkeit und Interoperabilität. Diese Einschränkungen erschweren die Integration heterogener Planning-Tools und verringern die Anpassungsfähigkeit von Planungssystemen an sich ändernde Anforderungen. In dieser Arbeit wird das PlanX Toolbox-Framework überarbeitet, um diese Herausforderungen zu adressieren. Dabei kommen Prinzipien der Service-Oriented Architecture (SOA) und Component-Based Software Engineering (CBSE) zum Einsatz.

Eine detaillierte Analyse der ursprünglichen Architektur der PlanX Toolbox identifizierte zentrale Schwächen wie starre Service-Interaktionen, eingeschränkte Erweiterbarkeit und eine enge Kopplung von Parsing-, Converting- und Planning-Komponenten. Als Lösung wurde eine neue Architektur entwickelt, die eine modulare *Generation Unit* einführt, welche die Konvertierung und Planerzeugung zusammenführt, während das Parsing bewusst als eigenständige Komponente beibehalten wird. Diese Trennung verbessert die Flexibilität der Systemkomposition und erleichtert den Austausch einzelner Komponenten. Zudem wurde eine standardisierte Nachrichtenstruktur eingeführt und die dynamische Auswahl von Planning-Strategien ermöglicht.

Die Integration des Planners Fast Downward diente als Fallstudie, um die erweiterte Architektur zu validieren. Dabei zeigte sich, dass trotz der Nutzung eines gemeinsamen Eingabeformats wie Planning Domain Description Language (PDDL) eine vollständige Wiederverwendbarkeit von Parsern durch werkzeugspezifische Annahmen und Vorverarbeitungsschritte eingeschränkt ist. Während Converting und Generierung erfolgreich modularisiert werden konnten, bleiben Parsing-Komponenten schwer übertragbar. Diese Ergebnisse verdeutlichen, dass echte Composability nicht allein durch standardisierte Schnittstellen erreicht werden kann, sondern ein tiefes Verständnis der internen Struktur von Planning-Tools erfordert.

Die Arbeit liefert konkrete Entwurfsmuster zur Verbesserung von Maintainability, Extensibility und Integration in modularen AI Planning-Systemen und leistet damit einen Beitrag zur praktischen Umsetzung moderner Architekturprinzipien im Bereich der AI Engineering.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

List of Abbreviations

AI Artificial Intelligence. 3, 5, 7

CBSE Component-Based Software Engineering. 3, 5, 25

CoDMAP Competition of Distributed and Multiagent Planners. 30

EUROPA Extensible Universal Remote Operations Planning Architecture. 29

FD Fast Downward. 13

IR Intermediate Representation. 57

ML Modelling Language. 35

MOM Message-Oriented Middleware. 20

NDDL New Domain Definition Language. 30

OMPS Open Multi-component Planner and Scheduler. 29

PANDA Planning and Acting in a Network Decomposition Architecture. 31

PDDL Planning Domain Description Language. 3, 5, 11

PDDL4J Planning Domain Description Library for Java. 30

SOA Service-Oriented Architecture. 3, 5, 25, 26

1 Introduction

AI planning is a fundamental area of AI that focuses on generating sequences of actions to achieve specific goals in diverse domains, such as robotics, logistics, and autonomous systems.

The relevance of AI planning has grown significantly with the emergence of advanced AI-driven applications. For instance, AI planning techniques have been applied in smart environments, such as using AI Planning for energy smart buildings [GSA23], and in long-term robot task planning using large language models [LPK+24]. These applications highlight the necessity for flexible and scalable planning frameworks capable of handling real-world uncertainties.

With the increasing complexity of modern applications, AI planning systems must be both adaptable and efficient. However, many current planning frameworks struggle with interoperability, reusability, and modularity, limiting their ability to scale and integrate into dynamic environments [Geo23a]. Existing AI planning frameworks, such as EUROPA [BBD+12], F4Plan [ADN+12], and CoDMAP [ŠKK16], have significantly contributed to the field. However, these frameworks struggle with interoperability at both the semantic and connectivity levels, which seems to be a general challenge in AI planning (see Barrier 5 in [Geo24]). The functional complexity and entanglement of existing planning tools create obstacles in defining clear functional boundaries, which in turn hinders the modularization and reusability of individual planning functionalities.

A key example of this challenge is the common parsing functionality used to ensure the compliance of encoded planning problems with standardized syntaxes such as PDDL [AHK+98]. This parsing process is often intertwined with object conversion and data preprocessing, making it difficult to extract, reuse, and integrate as a standalone functionality. Consequently, many planning systems implement their own parsing mechanisms, even when supporting a common syntax, resulting in redundant development efforts and limited interoperability across different planning tools[GPAA24].

To address the challenges of interoperability, flexibility, and maintainability in AI planning systems, adopting a component-based architecture is essential. By designing independent and reusable components, AI planning frameworks can better integrate diverse planning tools and adapt to evolving requirements. This approach not only improves scalability but also reduces dependencies, allowing for a more maintainable system. However, the practical realization of such an architecture remains challenging, as existing AI planning frameworks often exhibit strong coupling between components, making modularization non-trivial.

One prominent initiative that aims to address these limitations is the *PlanX Toolbox* [Geo23a]. While PlanX already employs a modular approach, its existing implementation presents challenges in terms of its compatibility with component-based architectures. The current architecture exhibits dependencies between services that may hinder full modularity, raising questions about how effectively it aligns with component-based design principles. Furthermore, integrating new planning tools into the toolbox requires a systematic assessment of how well the framework supports interoperability and reusability.

1.1 The PlanX Toolbox

The *PlanX Toolbox* is a modular framework designed to facilitate the construction and integration of AI planning systems. It provides a service-oriented approach to AI planning, enabling developers to compose and expand planning functionalities efficiently. By framing planning functionalities as loosely coupled services, PlanX aims to enhance flexibility and extensibility in AI planning system development.

The architecture of the PlanX Toolbox follows a microservices paradigm, where distinct planning functionalities are encapsulated as independent services. These services include parsing, encoding, solving, and validation, which together enable the full operational cycle of an AI planning system. This decomposition aims to improve maintainability and foster easy integration with external components.

PlanX is implemented using modern software engineering principles, employing technologies such as *Spring Boot* and *Kotlin* for backend services and *Angular* for frontend interfaces. Its backend services are structured around Message-Oriented Middleware (MOM) using *RabbitMQ*, ensuring decoupled communication between components. However, despite this modular structure, challenges remain in achieving full interoperability and flexible composition of AI planning services.

The primary goal of PlanX is to overcome the limitations posed by monolithic AI planning architectures. By refining its modularization and ensuring greater alignment with component-based software engineering principles, the toolbox can serve as a more scalable and maintainable platform for AI planning research and development. Given these objectives, a key question is whether PlanX's current architecture sufficiently realizes the benefits of modularization and how it can be further improved.

1.2 Research Question

This thesis investigates whether a component-based architecture can enhance the flexibility and maintainability of AI planning frameworks, using the PlanX Toolbox as a case study. Specifically, the research is guided by the following questions:

RQ1. How compatible is the PlanX Toolbox with component-based architectures, and how can it be enhanced to align more effectively with component-based design principles?

RQ2. How has the redesign of the PlanX Toolbox contributed to key insights in AI planning, and what lessons can be applied to future AI planning system architectures?

To address these questions, the thesis follows a structured methodology:

1. **Literature Review**: Analyzing existing AI planning frameworks, focusing on modularity, interoperability, and reusability.
2. **PlanX Toolbox Analysis**: Examining the structure of the PlanX Toolbox and identifying areas for modularization.
3. **Component-Based Architecture Design**: Developing a modular architecture with clear functional boundaries.
4. **Implementation**: Implementing the proposed architecture using Spring Boot and comparing it against the previous approach.

1.3 Contributions

This thesis makes several significant contributions to the field of AI planning by demonstrating how a component-based design enhances the flexibility, maintainability, and modularity of AI planning systems. Firstly we present a critical analysis of existing AI planning frameworks in Chapter 3, examining their modularity, interoperability, scalability and resilience. This analysis provides new insights into the structural limitations of current systems and highlights the need for a more structured approach to modular AI planning architectures. By identifying key shortcomings, the study lays a foundation for improving the architectural design of AI planning systems.

Building upon this analysis, the thesis proposes a refined architecture for the PlanX Toolbox that aligns more closely with component-based design principles. The redesigned framework establishes clearer functional boundaries, reduces dependencies between system components, and enhances the toolbox's ability to integrate diverse planning tools. The modular restructuring not only improves adaptability but also facilitates the efficient replacement or extension of individual components without affecting the overall system. This work extends previous research advocating for service-oriented AI planning [Geo23a] by systematically applying and evaluating component-based software engineering principles within an AI planning context.

Furthermore, this research offers practical guidelines for developing modular AI planning systems. The findings of this study contribute to the broader field of AI engineering by demonstrating how service-oriented and component-based software engineering principles can be effectively leveraged in AI planning. These insights serve as a reference for future work in designing scalable and maintainable AI planning frameworks, ensuring that AI planning systems remain adaptable to evolving technological and application demands.

This thesis is structured to provide a comprehensive exploration of the research topic. Chapter 2 presents the foundational concepts of AI planning, outlining its principles, common architectures, and the role of modularity in planning systems. Chapter 3 follows with an analysis of related work. Building on this foundation, Chapter 4 examines the current architecture of the PlanX Toolbox. Chapter 5 details the implementation of this redesigned architecture. The evaluation of the implemented system is presented in Chapter 6. Finally, Chapter 7 concludes the thesis by summarizing key findings and outlining potential directions for future research.

2 Background

2.1 Introduction to AI Planning

AI planning is a fundamental area in AI that focuses on generating sequences of actions to achieve a specific goal within a given environment. It is applied in diverse domains such as robotics, logistics, automated scheduling, and autonomous systems, where complex decision-making is required. Unlike decision-making approaches that rely on predefined rules, manually designed policies, or purely reactive mechanisms, AI planning explicitly models the problem as a search over a state space. Through systematic exploration of possible action sequences, planning systems identify solutions that satisfy well-defined goal conditions.

A key characteristic of AI planning is its use of formal problem representations, such as the PDDL, which defines the initial state, available actions, and goal conditions. While heuristics are employed within planning algorithms to improve efficiency, they are integrated within a structured model that distinguishes between declarative problem specification and the algorithmic methods used for plan generation. By explicitly encoding constraints, action effects, and domain dynamics, AI planning provides greater adaptability and ensures that generated plans adhere to formalized correctness criteria.

Formally, an AI planning problem is defined as a tuple $\mathcal{P} = \langle S, A, T, s_0, G \rangle$,

where:

- S represents the state space of the system.
- A is a set of available actions.
- $T : S \times A \rightarrow S$ defines the transition function.
- $s_0 \in S$ is the initial state.
- $G \subseteq S$ denotes the goal conditions.

The objective of AI planning is to find a sequence of actions $\pi = (a_1, a_2, \ldots, a_n)$ such that applying these actions to s_0 results in a final state satisfying G.

2.2 The AI Planning Process

Solving an AI planning problem follows a structured pipeline that aligns with the software development lifecycle for AI planning [Geo24], as illustrated in Figure 2.1.

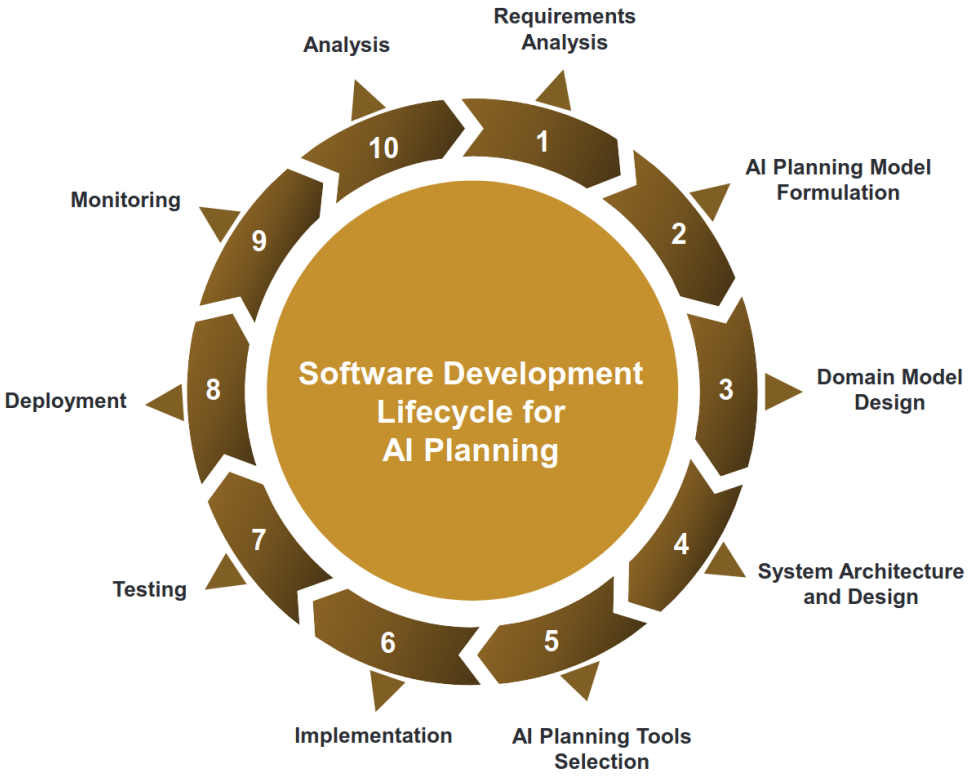

Figure 2.1: Software Development Lifecycle for AI Planning [Geo24]

This thesis focuses on three essential phases within this pipeline:

Phase 4: **System Architecture and Design:** Structuring the software components to ensure modularity and scalability.

Phase 5: **AI Planning Tools Selection:** Choosing solvers, frameworks, and libraries for the planning system.

Phase 6: **Implementation:** Developing modular components that generate AI plans from formalized inputs.

2.2.1 Formal Representation of AI Planning

A planning problem is often defined using the PDDL, which consists of:

- **Domain Definition:** Specifies predicates, actions, and their preconditions and effects.
- **Problem Definition:** Describes the initial state and goal conditions.

Table 2.1 presents a minimal PDDL domain and problem definition.

Domain Definition	Problem Definition

`(:action move`
`  :parameters (?from ?to)`
`  :precondition (at ?from)`
`  :effect (and (not (at ?from)) (at ?to))`
`)`
	`(:init (at A))`
	`(:goal (at B))`

Table 2.1: Example PDDL Domain and Problem Definitions

The left column defines a basic domain with the action `move`, which requires the agent to be at `?from` before execution. The action updates the world state by removing `(at ?from)` and adding `(at ?to)`. The right column specifies an initial state where the agent starts at location `A` and a goal condition requiring the agent to reach location `B`. Given this formulation, an AI planner generates a sequence of actions that transition the system from the initial state to the goal state.

2.3 Algorithmic Representation of AI Planning Execution

To effectively integrate AI planning into a modular system, it is essential to understand the structured execution pipeline that transforms a formal planning problem into an executable plan. Algorithm 2.1 provides a high-level representation of this process.

Algorithm 2.1 AI Planning Execution Pipeline

1: **Input:** AI planning problem $\mathcal{P}$
2: Parse $\mathcal{P}$ to obtain structured representation P_{parsed}
3: Convert P_{parsed} to planner-specific format $P_{encoded}$
4: Execute AI planner to generate plan $\pi = (a_1, a_2, ..., a_n)$
5: Return π as output

To get from the input problem $\mathcal{P}$ to the final plan π, the system undergoes several transformation stages. First, the input is parsed into a structured representation P_{parsed}, which captures the essential elements of the planning problem while ensuring that the input complies with the grammar rules of the chosen syntax (e.g., PDDL). This structured representation is then encoded into a format suitable for the selected AI planner, resulting in $P_{encoded}$. Finally, the AI planner executes on $P_{encoded}$ to generate the plan π. This plan represents a sequence of actions that transition the system from the initial state to the goal state.

These transformation stages can be implemented as independent modules, adhering to Component-Based Software Engineering (CBSE) principles. For example, a **Parsing Module** structures input data, while a **Planning Module** executes AI planning algorithms. This modularity ensures that components can be developed, tested, and replaced independently, aligning with scalable software design.

Understanding this execution pipeline provides the foundation for discussing software engineering paradigms that enable modular AI planning implementations. The next section explores how Service-Oriented Architecture (SOA) and CBSE principles facilitate the design of flexible, service-oriented AI planning architectures.

2.4 Software Engineering Foundations for AI Planning

To enable modular, scalable, and maintainable AI planning systems, modern software engineering paradigms are applied. A **component-based architecture** provides clear separation of concerns and can facilitate integration with various AI planning tools.

2.4.1 CBSE for AI Planning

CBSE promotes software reuse, maintainability, and modularity by structuring software into independent, interchangeable components with well-defined interfaces. A key principle of CBSE is **separation of concerns**, ensuring that each module is responsible for a specific function and can be developed, updated, or replaced independently. Another fundamental aspect is **explicit interfaces**, which define clear interaction points between components, allowing easy integration. Additionally, **composability** enables different modules to be assembled flexibly, depending on system requirements.

In this thesis, the AI planning system is designed following CBSE principles to enhance modularity and flexibility. The system consists of distinct modules, each encapsulating a specific functionality. Table 2.2 outlines the core components that support AI planning operations.

Component	Functionality
Parsing Module	Converts input files into structured representations
Encoding Module	Transforms structured representations into planner inputs
Planning Module	Executes AI planning algorithms

Table 2.2: Core Components in a Modular AI Planning System

These components serve as building blocks that can be assembled within a system, either as part of a monolithic architecture or within a **Service-Oriented Architecture (SOA)**, where services leverage modules to provide functionality.

2.4.2 Service-Oriented Architecture (SOA) for AI Planning

SOA structures software as a collection of **loosely coupled services** that interact through well-defined interfaces. Unlike software modules, which exist within a single application, services in SOA function as independent runtime entities that communicate over a network using APIs or message queues [LL09].

In the context of AI planning, an SOA-based system can consist of services that perform specific tasks, such as:

- **Parsing Service:** Receives user-defined planning problems and converts them into a structured format.
- **Conversion Service:** Transforms the structured representation into a solver-specific format.
- **Planning Service:** Executes AI planning algorithms and generates action sequences.

Each service operates independently, ensuring flexibility and scalability. Importantly, these services do not perform their tasks in isolation — they rely on **one or more CBSE modules** to execute their functionalities. For instance, the `Parsing Service` might internally use the `Parsing Module` from Table 2.2 to process input data. Similarly, the `Planning Service` could employ the `Planning Module` to execute AI planning algorithms.

This separation of concerns allows `modules to be reused across different services` while enabling services to function as independent, deployable units. The distinction between **SOA** and **CBSE** is crucial: while CBSE focuses on designing modular, reusable software components within a system, SOA organizes these components into distributed, networked services. Thus, SOA-based services act as **orchestrators**, invoking the appropriate **modules** to accomplish their objectives.

Figure 2.2 illustrates this distinction: two microservices (representing SOA-based services) each encapsulate multiple modules (representing CBSE components). The microservices communicate with each other, while reusing shared standalone modules. This demonstrates how modules can be embedded within services while still supporting reusability and separation of concerns across the system.

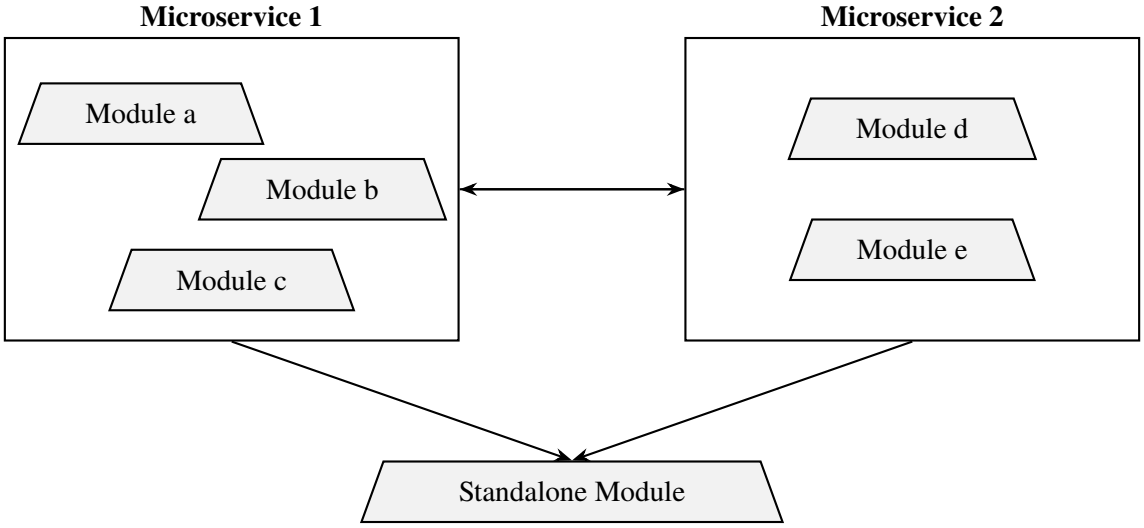

Figure 2.2: Service-Oriented Architecture with Modules

3 Related Work

We now present a discussion of related work concerning the design of existing AI planning frameworks and system architectures. In particular, we first describe each related work, and then we provide a qualitative analysis, highlighting the strengths and weaknesses of existing frameworks and architectures concerning modularity, flexibility, scalability, and resilience.

In the following section, we present the related work that has been done in that field. We take a look at different approaches of implementing modularity in AI Planning Systems in form of different framworks and languages in chronological order.

3.1 Overview

Cesta et al. [CFP08] introduce Open Multi-component Planner and Scheduler (OMPS), a component-based planning and scheduling system that integrates control theory and constraint-based reasoning. OMPS's modular architecture, where domain entities are modeled as components with behaviors and constraints, offers flexibility and adaptability. The system combines planning and scheduling using timelines, which allows for effective handling of complex temporal and resource constraints. A key feature is the decision network for constraint management, facilitating robust and flexible plan generation. However, the lack of comparative performance benchmarks and reliance on a single timeline extraction approach limit its evaluation. Furthermore, while the system addresses space mission planning, the solving process's complexity and missed opportunities for parallel processing could restrict its scalability. Despite these limitations, OMPS contributes significantly to flexible, modular AI planning and scheduling, especially in specialized domains.

André et al. [ADN+12] introduce F4Plan, a framework designed to integrate diverse planning algorithms into dynamic adaptation systems (DAS), enhancing the efficiency and coherence of adaptation plans, especially in distributed environments. The modular nature of F4Plan allows for the interchangeability of planning algorithms, aligning with the goals of component-based AI planning. Additionally, the use of translators to facilitate interoperability between decision algorithms and planning components is a notable strength. However, the lack of implementation details and empirical evaluation limits the framework's practical assessment. F4Plan also does not address the adaptability beyond algorithm selection or incorporate fault tolerance mechanisms, which are critical for robust real-world applications. Despite these limitations, the framework's focus on modularity and algorithmic flexibility offers valuable insights for enhancing flexibility in component-based planning systems.

Barreiro et al. [BBD+12] introduce Extensible Universal Remote Operations Planning Architecture (EUROPA), a C++ framework that supports constraint-based AI planning and scheduling through a modular, extensible design. The framework enables developers to create customized solutions by

combining pre-built components, with notable applications in NASA missions. EUROPA's strengths lie in its flexibility, with modules such as the Constraint Engine and Plan Database that can be adapted for diverse planning needs. Additionally, its open-source nature fosters collaboration. However, the paper lacks detailed explanations of component interactions and internal processes, limiting insight into its efficiency and effectiveness. Performance metrics and benchmarks are also underexplored, and the focus on New Domain Definition Language (NDDL) could restrict broader accessibility. Despite these limitations, EUROPA's modularity and extensibility make it a promising tool for enhancing flexibility in automated planning.

Štolba et al. [ŠKK16] present Competition of Distributed and Multiagent Planners (CoDMAP), a competition designed to advance research in distributed and multi-agent planning. The competition evaluates planners based on their ability to address privacy concerns, distributed computation, and plan quality. Its strengths lie in exploring various planning architectures, including centralized, distributed, and hybrid systems, as well as its focus on privacy-preserving techniques such as obfuscation and goal assignment strategies, which are relevant to component-based planning. CoDMAP's benchmarking methodology offers valuable comparative insights, but it lacks focus on modularity at the component level and does not address standardized communication between planners. Additionally, it primarily focuses on static planning problems, overlooking the dynamic adaptability required in real-world systems. Despite these gaps, CoDMAP provides important empirical insights that can inform the design and evaluation of flexible, modular AI planning systems.

Pellier and Fiorino [PF18] introduce Planning Domain Description Library for Java (PDDL4J), a versatile Java-based library designed to support AI planning with the PDDL. The library's strength lies in its support for multiple PDDL versions and its integration with various search algorithms, making it adaptable for a wide range of planning problems, including robotics and business process verification. Its open-source nature and platform independence further enhance its accessibility. However, the paper provides limited information on the library's internal architecture, leaving questions about its modularity and adaptability for component-based planning. Additionally, the lack of performance benchmarks hinders a full assessment of its computational efficiency. While PDDL4J's practical applications are evident, a deeper exploration of its structure and empirical evaluation would better position it for modular, component-based AI planning.

Georgievski and Breitenbücher [GB21] propose a service-oriented approach to AI planning that emphasizes modularity, composability, and automated deployment. Their framework allows users to integrate and compose existing planning tools into adaptable, deployable units, supporting the creation of flexible planning systems. The four-step model transformation process simplifies the transition from logical compositions to deployable systems, while the use of MOM ensures interoperability between services. Despite its strengths, the paper lacks specific guidance on service granularity and composition, which may hinder practical implementation. Additionally, the abstraction of planning tool complexities could limit informed decision-making, and the discussion on error handling and fault tolerance is underdeveloped. Addressing these gaps would enhance the adaptability, transparency, and reliability of service-oriented AI planning systems.

Höller et al. [HBBB21] present the Planning and Acting in a Network Decomposition Architecture (PANDA) framework, a comprehensive platform for hierarchical planning that integrates functionalities like automated planning, plan repair, and goal recognition. PANDA's modular design supports various planning techniques, making it adaptable for different planning scenarios. Its open-source nature promotes transparency and encourages empirical research. However, the paper lacks detailed insights into component interaction, communication protocols, and performance analysis, hindering a full understanding of its efficiency and scalability. Additionally, PANDA's reliance on grounded models and focus on hierarchical planning may limit its applicability to more dynamic or non-hierarchical planning domains.

Muise et al. [MPSK22] introduce PLANUTILS, a package-based library designed to simplify the setup and execution of AI planners. Its modularity and use of virtualization technologies like Docker and Singularity enhance portability and reproducibility, making it valuable for experimental AI planning research. However, the library's focus on package-level modularity limits fine-grained decomposition within planners, restricting flexibility. Additionally, the absence of detailed inter-component communication and performance analysis makes it difficult to assess the efficiency and extensibility of the system.

Georgievski's [Geo23b] paper offers a SOA approach to structuring AI planning functionalities, emphasizing modularity, interoperability, and deployability. By breaking down planning tasks into independent services, it fosters flexibility and reusability, using established software engineering practices like messaging middleware and containerization. However, the paper lacks detailed guidance on service granularity and composition, and abstracts away some of the complexities of specific planning tools. Additionally, the absence of performance evaluation and fault tolerance mechanisms raises questions about the practical application of the proposed approach in real-world settings.

Georgievski's PlanX Toolbox [Geo23a] presents a promising service-oriented toolbox for AI planning systems, featuring modular services that enable developers to customize and extend their planning systems. The architecture's use of Docker for deployment and JSON for interoperability enhances scalability and portability. While the modular approach is advantageous for flexibility, the paper lacks performance evaluation and detailed error handling strategies, raising concerns about the system's scalability, efficiency, and reliability in real-world applications.

Micheli's Unified Planning library [MBR+24] offers a promising foundation for integrating diverse planning components through its standardized interface, facilitating interoperability across various planning engines. Its Python implementation enhances accessibility and enables integration with other AI tools. However, the paper lacks detailed architectural insights, particularly concerning how the library handles data flow and component interaction, which are crucial for building dynamic, modular planning systems. The absence of performance analysis also makes it difficult to evaluate the efficiency of the library in real-world applications.

3.2 Analysis

After taking a look at the related work, we can see that there are different approaches to implement modularity in AI Planning Systems. We ranked these approaches in the categories *Modularity & Interoperability* and *Scalability & Resilience* in regards to how well they can inform the development of improved modularity in AI planning systems.

Rating *Modularity & Interoperability* shows how modularity influences a system's ability to communicate and integrate with other systems, revealing potential barriers or advantages.

Rating *Scalability & Resilience* highlights the system's ability to handle increased demands and maintain robustness under stress. When *Scalability* is considered alongside *Resilience*, this combination illustrates the balance between handling increased demands and maintaining robustness under stress.

3.2.1 Modularity & Interoperability

Modularity: The degree to which a system's components can be separated and recombined. In this context, it refers to how the system is structured to allow flexibility and integration with others.

Interoperability: The ability of different systems or components to work together, sharing data or services without compatibility issues.

The results of the evaluation of **Modularity** and **Interoperability** are plotted in the Figure 3.1 and Figure 3.2.

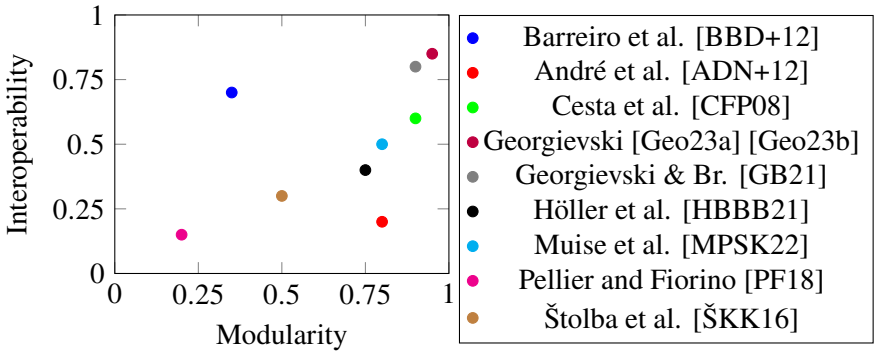

Figure 3.1: Rating of Papers in regards of Modularity and Interoperability

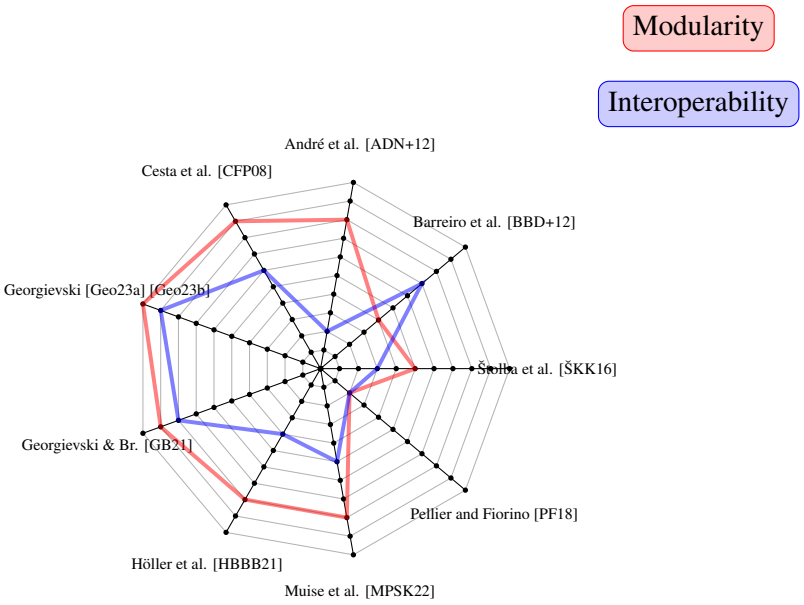

Figure 3.2: Spiderweb Diagram of Modularity and Interoperability ratings

The evaluation of **Modularity** and **Interoperability** highlights several key insights. The work of **Georgievski [Geo23a; Geo23b]**, **Georgievski & Br. [GB21]**, and **Muise et al. [MPSK22]** stand out for their excellent balance, scoring high in both dimensions.

3.2.2 Scalability & Resilience

Scalability: The ability of a system to handle growing amounts of work or its potential to be expanded to accommodate growth.

Resilience: The ability of a system to recover from failures or handle stressful situations (e.g., hardware failures, high load) without losing functionality or performance.

The results of the evaluation of **Scalability** and **Resilience** are plotted in the Figure 3.3 and Figure 3.4.

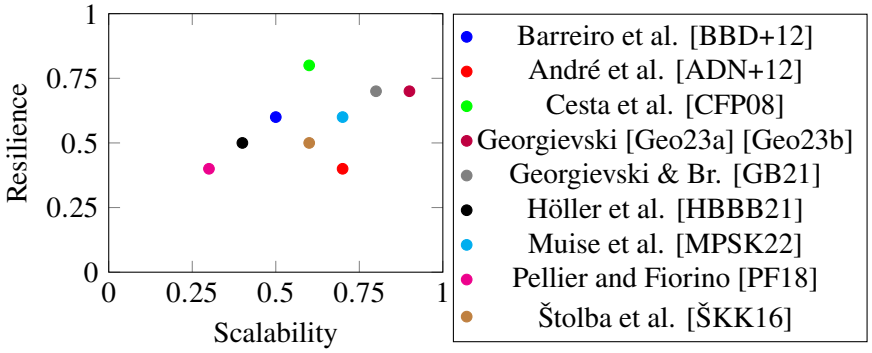

Figure 3.3: Rating of Papers in regards of Scalability and Resilience

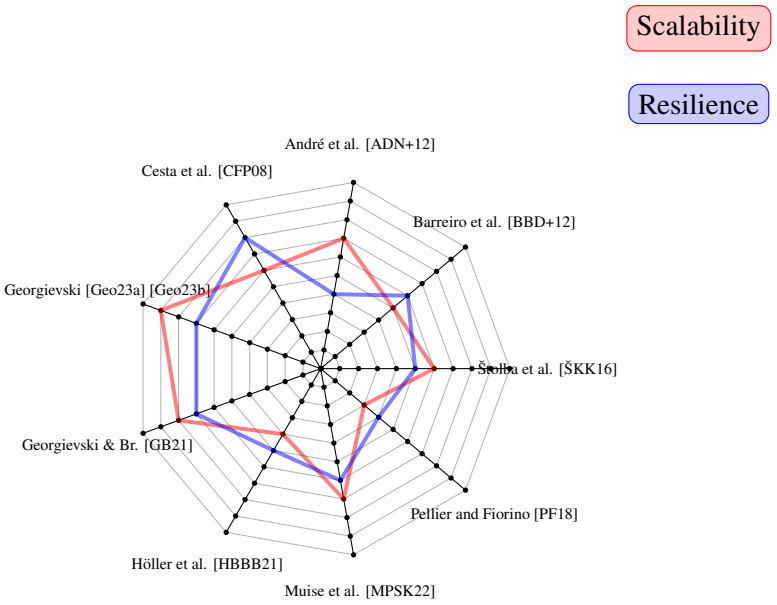

Figure 3.4: Spiderweb Diagram of Scalability and Resilience ratings

The evaluation reveals that all papers considered provide limited details on their handling of the categories of scalability and resilience. Among them, **Georgievski & Br. [GB21]** and **Cesta et al. [CFP08]** stand out as the highest-rated contributions, but their performance in these areas remains comparatively low compared to the other categories analyzed. Similarly, works like **Cesta et al. [CFP08]** exhibit strengths in either scalability or resilience but still show room for significant improvement.

4 Analysis and Redesign of PlanX Services

Improving the modularity of the services in the PlanX Toolbox requires a clear understanding of their current architecture and design. This chapter examines the most relevant services within the toolbox, focusing on their structure and evolution.

Section 4.1 analyzes the existing architecture, identifying potential weaknesses. Section 4.2 explores possible enhancements, and Section 4.3 presents the redesigned architecture.

4.1 Analysis of Existing Design

In this section, we dive into the previous architecture and design of the PlanX Toolbox AI Planning System. This perspective provides valuable insights into the system's evolution, highlighting the initial design choices, and the challenges encountered. The following subsections detail the components, workflows, and design patterns that characterized the current version of the PlanX Toolbox.

4.1.1 A System Pipeline with the Toolbox's Services

The PlanX Toolbox AI Planning System pipeline, considered for this work, consists of several services that work together to process user requests and generate plans. The system is designed to be modular and extensible, allowing for the addition of new services and functionalities. The pipeline is divided into several steps, each of which is handled by a specific service. The system uses MOM to communicate between services, enabling asynchronous message passing and decoupling the components. Each Service reveices a call stack that contains next steps to be executed and the service to be called next. This pipeline is illustrated in Figure 4.1 and described in detail below.

Step 1: **Frontend Services:** Users interact with the system by creating a domain and a problem, and selecting a planner to solve the problem.

Step 2: **Management Service:** The managing service initializes a call stack, appending itself (for result retrieval) followed by the solving service. This call stack is then dispatched into a message queue that the solving service monitors.

Step 3: **Solving Service Initialization:** Upon receiving the message, the solving service processes the top element of the stack. It then appends itself to the stack as a reply handler and adds the parsing service to the stack before sending the updated call stack into the parsing queue.

Step 4: **Parsing Service:** The parsing service decodes the domain and problem definitions and parses them using the Modelling Language (ML) specified in the encoded message (e.g., PDDL). Once parsed, the parsing service removes itself from the stack and forwards the parsed content to the converting service via another queue.

Step 5: Converting Service: The converting service transforms the parsed content into objects representing the domain and problem. It then sends these objects back to the solving service via the queue, including the updated call stack.

Step 6: Solving Service: The solving service uses the domain and problem objects to compute a plan. Upon successfully generating the plan, it removes itself from the stack and forwards the result to the managing service.

Step 7: Result Retrieval: The managing service collects the generated plan and delivers it to the frontend for user interaction.

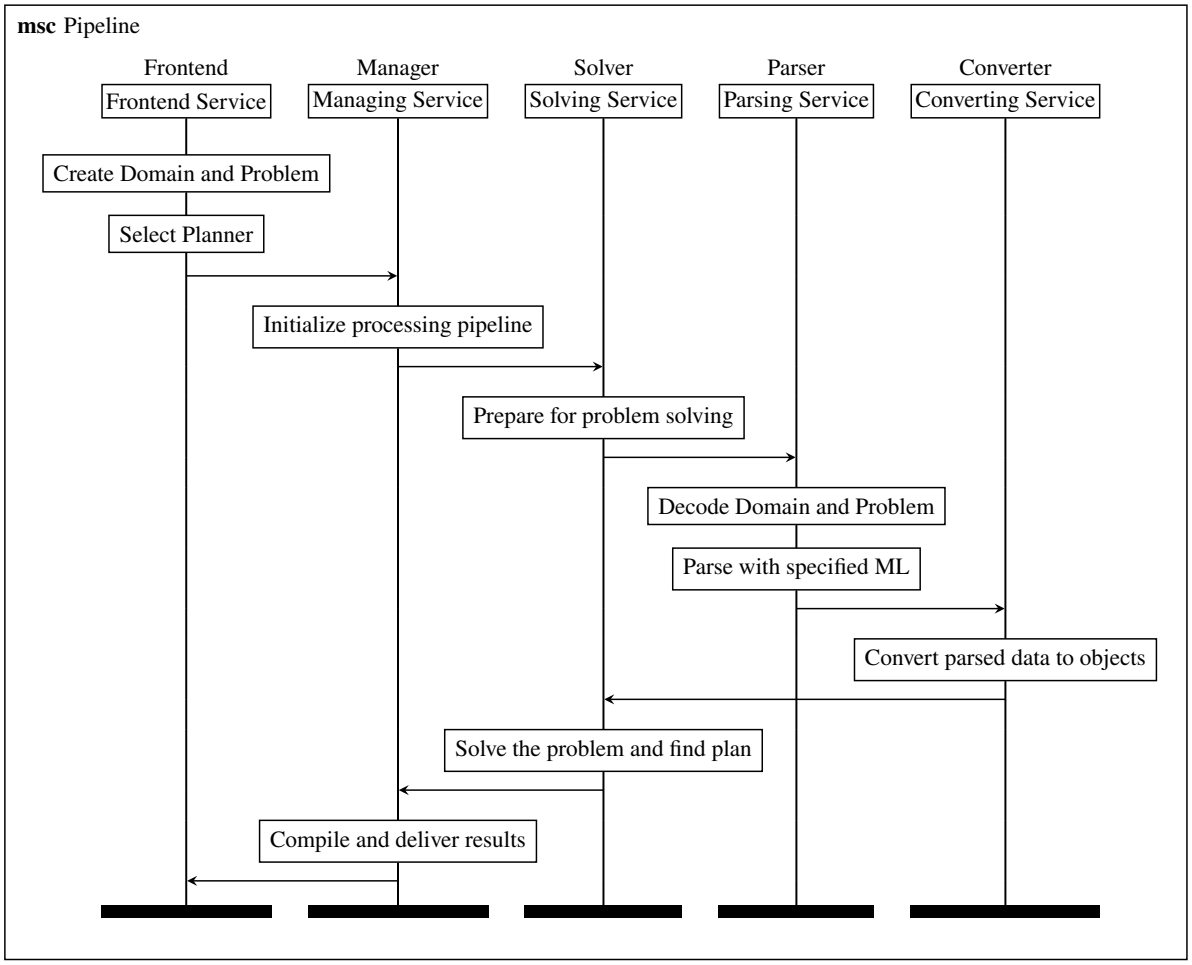

Figure 4.1: AI Planning System Pipeline

4.1.2 Design Pattern Analysis

Each service in the PlanX Toolbox AI Planning System pipeline is designed to handle a specific task, such as parsing, converting, or solving planning problems. In the following sections, we analyze the design patterns used in each service to determine how they contribute to the system's modularity, flexibility, and maintainability.

Parser Module

The `Parser` module, which includes the `Receiver` class, implements the **Strategy** design pattern. The module adapts incoming messages from the queue and delegates the actual parsing work to the `ParsingService`. The `Receiver` acts as a context, selecting the correct parsing strategy based on the message type (e.g., `PddlParsingRequestBody`).

The `ParsingService`, in turn, uses a strategy pattern to choose the appropriate parser for different input languages. In the method `parseProblemAndDomain`, it dynamically selects between different parsers (`PddlParser`, `Shop2Parser`, `HpdlParser`) based on the specified `language` parameter. This enables the system to easily extend support for new parsing strategies without modifying the core logic of message reception or parsing.

Strategy Pattern in Action The key strategy implementation is within the `ParsingService`, where the parser to be used is determined at runtime. The `when` block in `parseProblemAndDomain` selects the appropriate parser class based on the `language` parameter. This allows for interchangeable parsing algorithms, encapsulated within distinct classes, depending on the type of problem and domain being processed.

The `Receiver` class ensures that the correct parsing strategy is chosen and delegates the processing to the `ParsingService`, which handles the parsing logic. The modularity introduced by the Strategy pattern allows new parsers to be added easily by simply extending `ParsingService`, without needing to modify the `Receiver`.

Extendability and Modularity This design allows the system to scale and adapt over time:

- New parsing strategies can be introduced in `ParsingService` without modifying the `Receiver`.
- The `Receiver` only handles message reception and delegates all parsing to the `ParsingService`, ensuring separation of concerns and clean code.

Conclusion The `Parser` module, using both the `Receiver` and `ParsingService`, clearly implements the Strategy pattern. The `Receiver` selects the correct strategy for processing the message, while the `ParsingService` dynamically chooses the appropriate parsing strategy based on the message content, promoting extendability and flexibility within the system.

Converting Module

The `Converting` module, which includes the `Receiver` class, does not implement a specific design pattern like the `Parser` module. Instead, it is specifically tailored for handling PDDL encoding requests. However, this limitation is further amplified by the module's reliance on PDDL4J, a dedicated Java library for parsing and manipulating PDDL. As a result, the module is not just constrained to PDDL in general but is specifically bound to the PDDL4J framework. This dependency significantly restricts its usability for other planning-related frameworks and adds further limitations to its extendability.

Why No Specific Design Pattern Unlike the `Parser` module, which uses the Strategy pattern to allow flexibility in parsing various languages, the `Converting` module is highly specific in its purpose. The `handleMessage` method only processes `PddlEncodingRequestBody` messages, which limits the system's extensibility to other modelling languages. In the method, the `ConvertingService` is called to encode the PDDL problem and domain. However, since this service is designed around PDDL4J, there is no mechanism to support alternative encoding approaches or different modelling languages, reinforcing the module's inflexibility.

This tight coupling to PDDL4J means that this module does not utilize a design pattern like Strategy or Adapter, as it is built around a single library rather than an abstract conversion process. This rigid approach makes it difficult to introduce support for future modifications, such as integrating additional modelling languages beyond PDDL.

Extendability Limitations The `Converting` module's reliance on PDDL4J creates additional extendability challenges:

- The module is not only restricted to PDDL but is further confined to PDDL4J, making integration with other planning frameworks complex.
- New modelling languages cannot be incorporated without substantial modifications due to the PDDL4J-specific implementation.
- The `Receiver` and `ConvertingService` would both require significant refactoring to accommodate additional languages, unlike the `Parser` module, which can introduce new parsing strategies with minimal changes to existing code.

Conclusion The `Converting` module is designed specifically for handling PDDL data using PDDL4J, which imposes additional constraints on its flexibility. Since the module does not incorporate adaptable design patterns like Strategy or Adapter, it remains rigid in structure and difficult to extend for other modelling languages. The strong dependency on PDDL4J makes it impractical for use in other planning-related frameworks without major modifications, further limiting its adaptability and potential for future enhancements.

Solving Service (PDDL4J)

The `Solving Service PDDL4J` module, including the `ObjectReceiver` class, is specifically tailored for the PDDL4J framework, which provides a structured environment for solving planning problems in PDDL. However, this module offers flexibility through the use of different planners. This flexibility is achieved through the use of the `Planner` enum, which allows the system to choose between different planners at runtime, making it adaptable within the PDDL4J context but still limiting to PDDL-based problem-solving.

Why No Specific Design Pattern The `ObjectReceiver` class does not strictly implement a design pattern like Strategy or Adapter, as it is more directly focused on utilizing the PDDL4J framework with different planners. Instead of using a general-purpose design pattern to handle a variety of planning algorithms, the module uses a hardcoded decision-making process in the `when` block to select one of the predefined planners based on the incoming message data. This approach is still flexible within the specific PDDL context but lacks the general extensibility seen in the more abstract design patterns like Strategy.

The decision is made based on the state of the current element in the `callStack`, and the planner is selected accordingly. If the planner value is not recognized, it defaults to `Planner.HSP`. This is a simple form of flexibility, but it is still confined within the boundaries of the PDDL4J framework, as the planners are preconfigured and cannot be easily extended to other frameworks or planning languages.

Extendability and Modularity While the system supports different planners, it is still tightly bound to the PDDL4J framework and its associated planners. Therefore, extending the system to incorporate new planning algorithms outside of PDDL4J or to support a wider range of planners would require significant modification. Unlike the `Parser` or `Converting` modules, which are constrained by the specific languages they handle, the `Solving Service` allows for some flexibility in choosing the planner, but it does not abstract the planner selection process in a way that would easily allow for integration with external systems or languages.

- The use of different planners makes the module flexible within the PDDL4J ecosystem but still limits its applicability to other frameworks or languages.
- New planners could be added by modifying the `when` block or extending the `Planner` enum, but introducing completely new solving strategies beyond PDDL4J would be challenging.

Conclusion The `Solving Service PDDL4J` module offers flexibility within the PDDL4J framework by allowing the selection of different planners at runtime. However, this flexibility is constrained to the PDDL4J context, and the system does not implement a flexible design pattern like Strategy or Adapter that would allow it to easily integrate with other planning frameworks or languages. While new planners can be added, the system remains rigid within its scope, and the planner selection is hardcoded based on predefined values.

4.2 Limitations & New Requirements

The existing architecture of the PlanX Toolbox AI Planning System has several limitations that need to be addressed to improve the system's modularity, scalability, and maintainability. The following sections discusses the flaws in the existing architecture and propose improvements to overcome these limitations.

4.2.1 Componentize Functionalities

The existing version of the Toolbox contains separate components that are tightly coupled and could benefit from being combined into one cohesive unit. Specifically, the *Converting Module* and *Solving Module* components are tightly coupled and could be merged into a single *Generation* component.

Although this merging appears to reduce modularity at first glance, it in fact supports a cleaner separation of concerns and eases integration of new planners. This is because the current architecture requires the integration of a new planner to interact separately with both the Converting and Solving components, introducing unnecessary complexity and redundancy. Combining these components allows the planner interface to focus solely on receiving parsed input and returning a plan, delegating the internal coordination between conversion and solving to a single responsibility unit. By restructuring these functionalities into a dedicated component, we can enhance system maintainability, reduce unnecessary dependencies between services, and eliminate the need for queues used for communication between tightly coupled components.

This approach is likely beneficial beyond just the case of the PlanX Toolbox, as reducing tight coupling and unnecessary interdependencies generally leads to more modular, maintainable, and scalable software architectures. [1]

Proposed Solution and Benefits

To address this issue, we propose the integration of the *Converter* and *Solver* components into a single *Generation* component. This new component will handle the generation of plans from parsed problem instances, thereby streamlining the process and reducing complexity in the system architecture. By merging these functionalities, the system can operate more efficiently without the overhead of message passing between distinct yet interdependent services.

This restructuring offers several key advantages. Firstly, it reduces complexity by simplifying the system architecture and minimizing dependencies between services. Secondly, it enhances maintainability by allowing modifications to be made within a single, cohesive component rather than across multiple interdependent services. Lastly, performance improvements are anticipated as the elimination of communication overhead between separate components enables faster execution and reduced latency.

[1] In software design, reducing tight coupling improves modularity, simplifies debugging and updates, and enhances overall system flexibility.

4.2.2 Uniform Design of Components

The existing architecture lacks a uniform design pattern for components, leading to inconsistencies in the implementation and maintenance of services. By adopting a consistent design pattern for all components, we can improve the system's modularity and maintainability.

Proposed Solution and Benefits

To ensure uniformity, we propose the adoption of a standardized design pattern, such as the *Strategy Pattern*, across all components. This will enforce consistency in implementation, making it easier to develop, maintain, and extend the system. Additionally, standard interfaces will be defined for each component, specifying the required methods and properties necessary for communication between services.

Implementing a uniform design pattern provides several advantages. It ensures a consistent implementation, making the system easier to understand and maintain. Furthermore, it improves modularity by allowing components to interact in a standardized manner, thereby enhancing the overall extensibility of the architecture.

4.3 Refining the Architecture

As described in section 4.2, the old architecture of the PlanX Toolbox has several limitations that need to be addressed to improve the system's modularity, scalability, and maintainability. The following sections will explain how we improved the old architecture to overcome these limitations.

4.3.1 Creating the *Generation-Unit*

As described in paragraph 4.2.1, the *Converter* and *Solver* components were tightly coupled and have now been restructured into a more modular and extensible design. This leads to the introduction of the *ComposingService*, which orchestrates the conversion and plan generation processes by selecting the appropriate strategies dynamically based on the requested planning strategy.

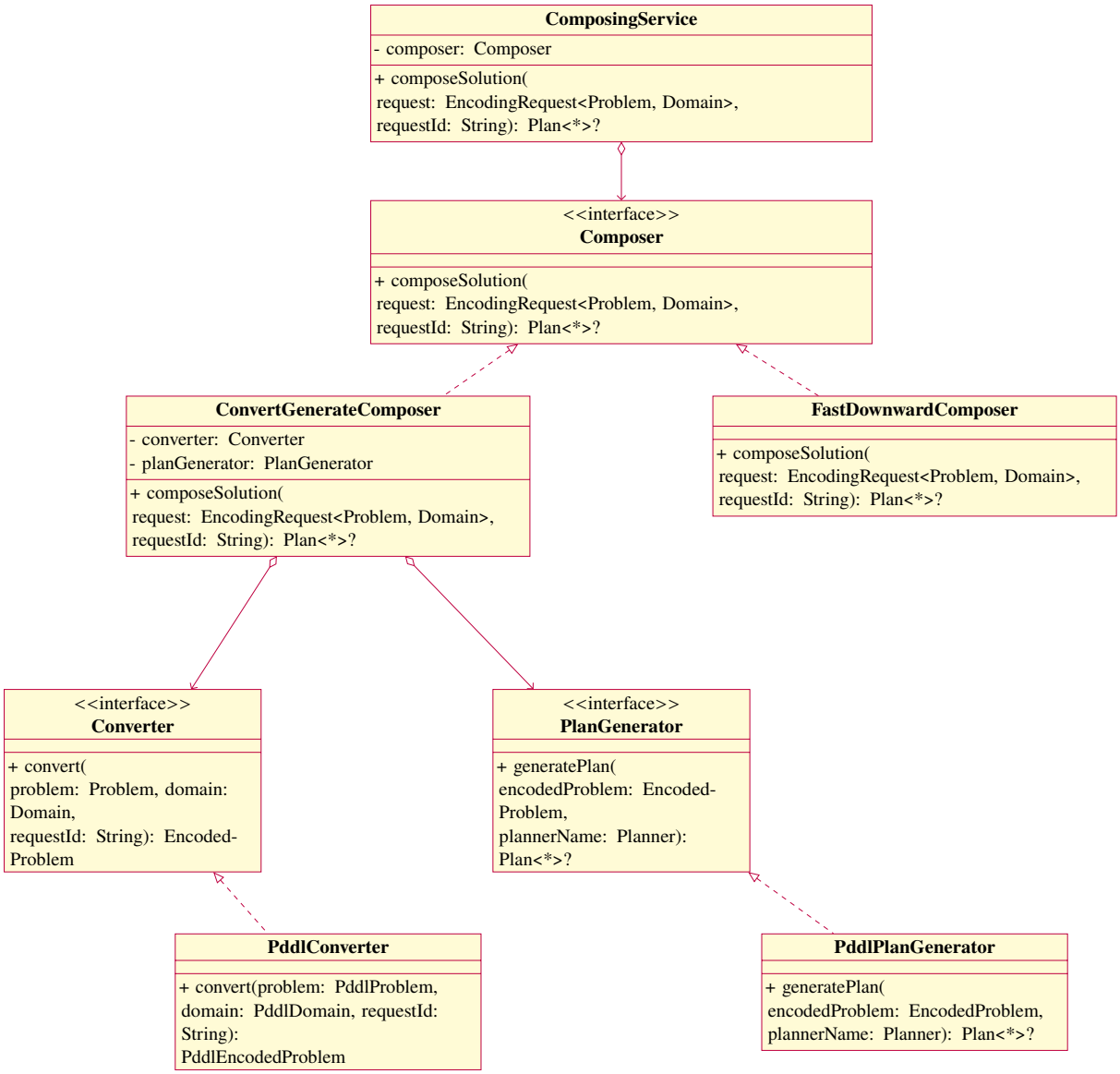

Figure 4.2: Composing Service Structure

To achieve this, we struct the *Generation-Unit* around a flexible composition mechanism that can be seen in Figure 4.2. The core of this design is the *ComposingService*, which delegates the solution composition to a specific *Composer*. The *ConvertGenerateComposer* is one such implementation that integrates a *Converter* and a *PlanGenerator* to transform parsed data into an executable plan. Additionally, alternative compositions can be added by implementing the *Composer* interface. This allows a *Composer* to have different composition strategies, which directly generate a plan without converting the input data.

Input and output data are passed to the *Generation-Unit* in a standardized format (see Section 4.3.3). The unit receives the parsed data from the *Parsing-Service* and returns the generated plan to the *Managing-Service*. This separation ensures that the *Generation-Unit* remains replaceable and extendable without disrupting other system components.

Strategy-Based Composition Selection

A key improvement in the new design is the use of the *Strategy Pattern* to dynamically select the appropriate composition of services. The *ComposingService* determines which *Composer* to use based on the planner selected by the user, allowing for flexibility in handling different problem domains. For example, the *FastDownwardComposer* is used when the user selects the FD planner, while the *ConvertGenerateComposer* is used for other planners. You can see this in figure 4.3 where the user can select the planner they want to use. Other languages can be supported by adding corresponding *Converters* and *PlanGenerators* and associating them with a suitable *Composer*. This design ensures that the system can easily adapt to new requirements without requiring major architectural changes.

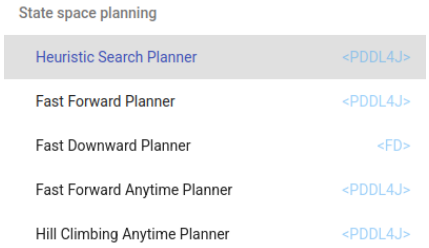

Figure 4.3: Planner Selection in the frontend

Service Structure and Implementation

Each *Converter* and *PlanGenerator* follows the *Template Method Pattern*, ensuring that the structure of their respective processes is maintained while allowing different implementations for specific languages. The *ConvertGenerateComposer* invokes the appropriate *Converter*, such as *Pddl4JConverter*, to transform the problem and domain into an encoded format. This encoded problem is then passed to the appropriate *PlanGenerator*, in that case the *Pddl4JPlanGenerator*, to generate a plan.

This modular structure ensures that new languages can be easily added by implementing a new *Converter* and *PlanGenerator* and integrating them into the *ConvertGenerateComposer*, or simply by creating a new *Composer* that uses the new components. The system's flexibility and extensibility make it robust to future expansions without requiring major architectural changes.

4.3.2 Improving the pipeline

To improve the pipeline, we focused on minimizing the message flow between services. We can see for example in figure 4.1 the *Solving-Service* is already early on in the pipeline but it does nothing but pass the request on to the *Parsing-Service*. This is a waste of resources and can be improved by directly routing the message from the *Managing-Service* to the *Parsing-Service*.

In Figure 4.4 we can see the improved pipeline of the Planning System. We can see that the *Solving-Service* is replaced by the *Generation Unit* (as described in 4.3.1) and the *Managing-Service* now directly routes the message to the *Parsing-Service*.

4.3.3 Standardizing the message structure

Subsection 4.2.2 defines a standard message structure used throughout the system. The structure includes all necessary information for each service to process the message and is extensible to accommodate different message types and data formats. Standard message types ensure that every service can process any message adhering to the standard structure.

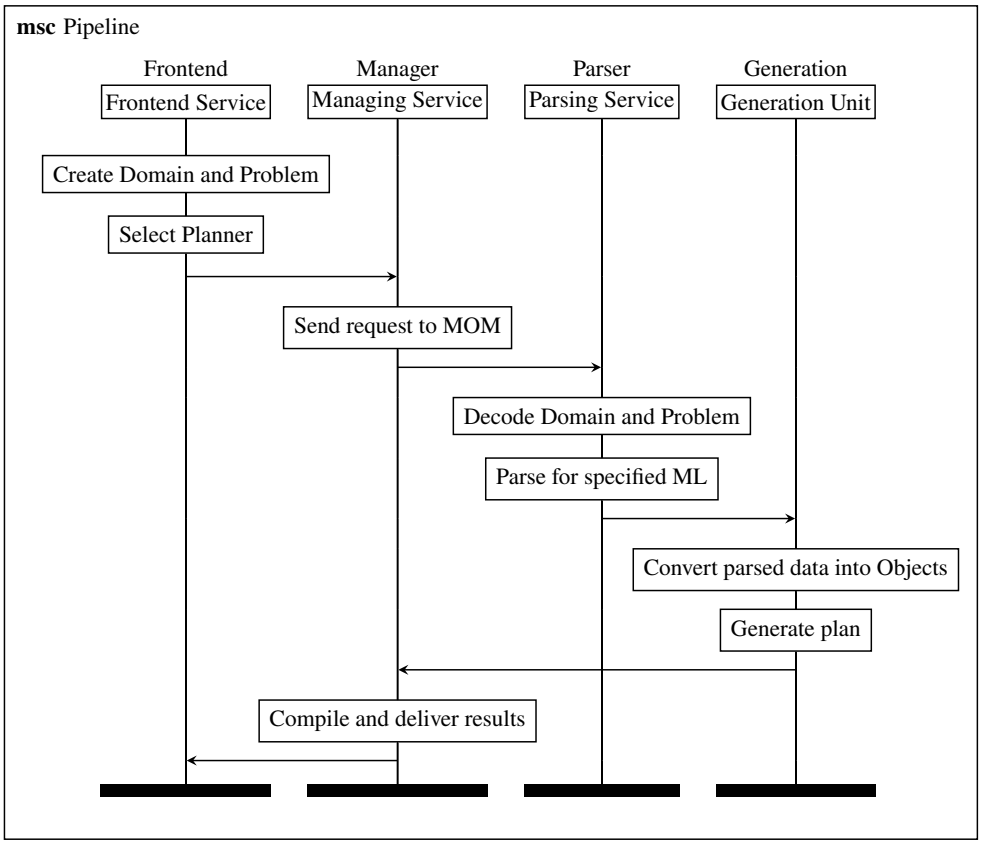

Figure 4.4: Improved AI Planning System Pipeline

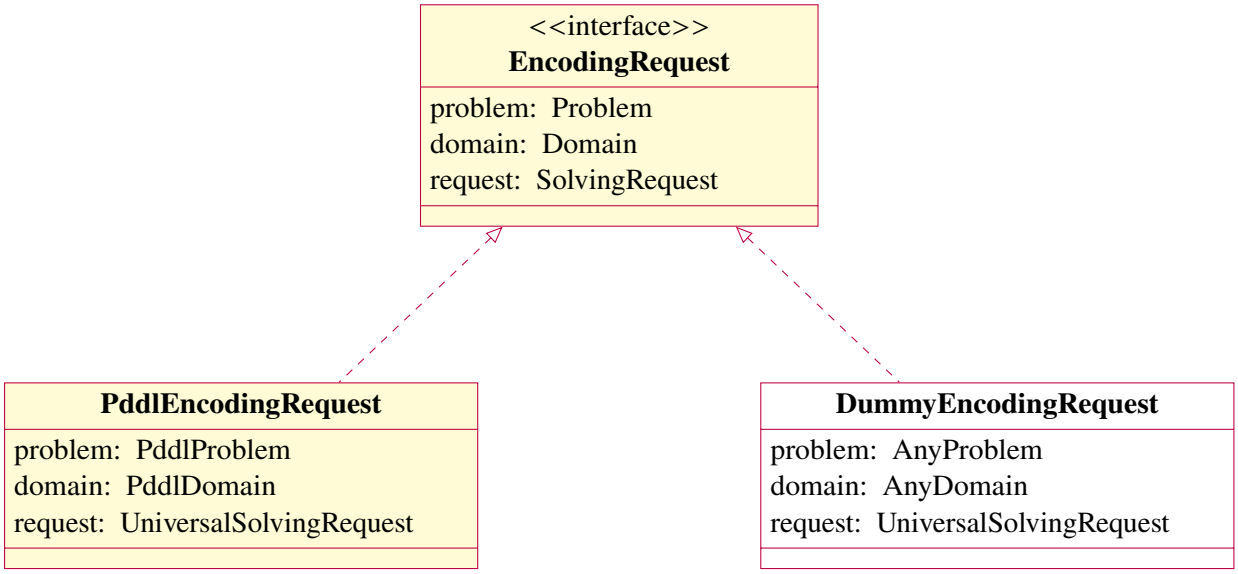

Figure 4.5: Standardized Message Structure

Figure 4.5 shows two message types conforming to the *EncodingRequest* interface. The *GenerationUnit* processes any message conforming to this interface without requiring knowledge of the specific message type.

5 Implementation

After establishing the requirements and designing the architecture of the PlanX Toolbox, the next phase is the implementation. This phase involves translating the conceptual design into a working system by developing modular, maintainable, and scalable software components.

The design and implementation of the PlanX Toolbox build upon the efforts of a master's thesis [Gra20] and are thoroughly described in [Geo23a]. The toolbox adopts a microservice-based architecture, and was designed with usability, interoperability, and reusability in mind. The system is implemented using Kotlin and Spring Boot, with RabbitMQ serving as the MOM. Additionally, Docker is employed for containerization, ensuring modularity and ease of deployment. The toolbox primarily supports PDDL but is structured for extensibility.

Building upon this foundation, this work enhances the modularity of the PlanX Toolbox by refining the architecture, optimizing service interactions, and improving scalability. While the core technologies remain consistent with the original prototype, modifications have been made to better support additional functionalities, improve maintainability, and facilitate future expansions. The focus of this implementation is on improving inter-service communication, refining message routing strategies, and introducing a clearer separation of concerns within the microservices.

Beyond these refinements, several key additions have been made. The call stack has been removed from the pipeline to enhance performance and streamline processing. Additionally, a new component, the Generation Unit, has been implemented. This unit combines the converting and solving functionalities, as detailed in Section 4.3.1. Furthermore, efforts have been made to integrate the FD Planner alongside existing planners such as PDDL4J, leveraging PDDL for compatibility. The objective of integrating FD was to evaluate the feasibility of adding a new planner to the toolbox, testing the ease of integration and the reusability of the existing and improved components.

The following sections detail the technologies used in the implementation (Section 5.1), and refine the service interactions (Section 5.2). The subsequent sections describe the removal of the call stack from the pipeline (Section 5.3), the implementation of the Generation Unit (Section 5.4), and the integration of the FD Planner (Section 5.5).

5.1 Technologies

This section provides an overview of the technologies used in the implementation of the PlanX Toolbox. While the implementation builds upon the foundation established by [Gra20], certain modifications and extensions have been made to improve modularity, maintainability, and extensibility.

The core implementation of the microservices is written in Kotlin, utilizing Spring Boot for service development. No additional Spring modules are used beyond the necessary configurations for RabbitMQ as the MOM. The decision to use Kotlin and Spring Boot follows the foundation set by Graef's prototype, leveraging their strong support for microservices architecture and maintainability.

RabbitMQ is employed as the MOM facilitating communication between the various microservices. This design choice was inherited from the original implementation by [Gra20], as RabbitMQ provides a straightforward mechanism for implementing asynchronous communication patterns. Each service publishes and subscribes to specific message queues, enabling loosely coupled service interactions.

All services are containerized using Docker, ensuring consistency across different environments. Docker Compose is used to define and manage multi-container deployments, streamlining the startup process of the system. This approach allows all services to be launched together with minimal configuration effort, following the same deployment methodology as the initial prototype.

The frontend of the system is implemented as an Angular web application. It serves as the main user interface for modeling planning problems. The frontend leverages the Angular Material library for UI components and runs as a Node.js application served via an NGINX web server. Since no modifications were made to the frontend in this implementation, the original structure and functionality remain unchanged.

The system includes an administration interface located under the `/admin` route, providing an overview of the microservices and their statuses. This monitoring feature, integrated into the frontend, includes:

- A system overview displaying the microservices and their MOM routes.
- A real-time system status page connected to the MOM, showing active services and their health status.
- An embedded administration interface for RabbitMQ, enabling queue monitoring and configuration.

These features were part of Graef's original implementation and remain functional in this iteration of the PlanX Toolbox.

The next sections will explore the system architecture (Section 5.2) and specific implementation details, including the removal of the call stack, the development of the Generation Unit (Section 5.4), and the integration of the FD Planner.

5.2 Refinement of Service Interactions

In this section, we analyze the improvements made to the architecture, specifically focusing on modularity, message routing, standardization, and integration of components. These refinements were necessary to enhance flexibility, maintainability, and overall system efficiency.

The previous architecture lacked modularity due to tightly coupled interfaces that were tailored to specific data structures, making it difficult to extend or modify services without affecting other components. With the introduction of the *Generation-Unit*, as described in Section 4.3.1, and the overhaul of data structures (Section 4.2.2), components are now designed to be more independent and interchangeable. Standardized interfaces ensure that new planners can be integrated with minimal changes, following the *Strategy Pattern*.

Previously, the service callstack introduced inefficiencies by accessing an additional service. In the refined design, as outlined in Section 4.3.2, the solving service is now invoked before the parser, reducing unnecessary overhead. This change streamlines the workflow and eliminates redundant service dependencies.

The revised architecture does not introduce new technologies, middleware, or caching mechanisms. The focus has been on restructuring existing components rather than incorporating additional dependencies. However, the standardized message structure (Section 4.3.3) ensures consistent communication across services.

At present, the system does not implement fallback strategies or error-handling mechanisms for service interactions. This remains an area for potential improvement in future iterations.

All services now adhere to the *Strategy Pattern*, significantly improving the ease of integration for new planners. The uniform design facilitates code maintenance and enhances overall system consistency.

Section 4.3.3 explains the importance of defining a standardized message structure used throughout the system. Each service can now process any message that adheres to this structure, making it extensible for various message types and data formats. Unlike the previous approach, where services rejected messages that did not fit specific criteria, the new design ensures that all services can interpret and respond to messages in a standardized manner.

No additional security measures such as authentication, encryption, or data validation have been introduced as part of the refinement process. These aspects may be considered in future iterations.

The refined architecture does not directly impact scalability. However, due to the use of a queue-based MOM, horizontal and vertical scaling remain feasible. Any service subscribed to the queue can process incoming requests independently, allowing for distributed processing.

5.3 Eliminating the Call Stack

The original architecture of the PlanX Toolbox relied on a call stack mechanism that introduced tight coupling between services, making modifications and extensions difficult. The call stack was deeply integrated into the message-passing logic, reducing the flexibility of the system. To improve modularity and maintainability, we replaced the call stack with a more dynamic and decoupled message-passing system.

Since the PlanX Toolbox already utilizes a MOM, message coordination is now handled through configurable queues. Each service listens to predefined queues and processes incoming messages independently, eliminating the need for a rigid call stack. By leveraging this approach, services operate autonomously, reducing dependencies and improving modularity.

The redirection of requests between services is managed through configuration properties. Each component has a dedicated request and response queue, ensuring structured and predictable message flow. For example, in the parser configuration shown in Listing 5.1, the request queue is defined as `v1.transforming.parsing-service.in`, while the response queue is set to `v1.endpoint.generation-unit`.

This setup allows each service to direct responses dynamically to the appropriate next step in the processing pipeline. By defining input and output queues per service, the system achieves a clear separation of concerns and greater flexibility in workflow execution.

Listing 5.1 Parser Configuration

```
# request queue
planx.queues.request.name=v1.transforming.parsing-service.in
planx.queues.request.key=toolbox.parsing
# response queue
planx.queues.response.name=v1.endpoint.generation-unit
planx.queues.response.key=toolbox.generating
```

To ensure that all services can interpret and process messages consistently, a standardized message format has been established (see Section 4.3.3). This structure is designed to be extensible, allowing future enhancements without significant architectural changes. It includes metadata that enables efficient message routing and processing, supporting the extensibility and adaptability of the system.

Eliminating the call stack and adopting a queue-based messaging approach provides several advantages. The system's modularity is significantly improved, as services are now loosely coupled and independently maintainable. The simplified architecture reduces overall complexity, making it easier to develop and extend the system. Additionally, scalability is enhanced, as multiple services can process messages concurrently, improving performance and responsiveness. Finally, the configuration-driven routing mechanism allows for dynamic message redirection, enabling flexible system reconfiguration and expansion.

5.4 Implementation of the Generation Unit

The implementation of the *Generation-Unit* is designed to provide a modular and extensible approach to plan generation. It follows the Strategy Pattern to dynamically select the appropriate composition strategy based on the requested planner. Additionally, it integrates with RabbitMQ to handle message-based communication between components.

Service Composition

The core of the *Generation-Unit* is the `ComposingService`, which acts as the central coordination unit. It receives encoded problem requests from the *Parsing-Service*, determines the appropriate `Composer`, and delegates the composition task. The selection of the composer is handled dynamically using a configuration-driven strategy mapping.

This approach ensures that each planner has a corresponding `Composer`, which is injected via a configuration mapping, as shown in Listing 5.3.

Implementation Details

The `ConvertGenerateComposer` follows a similar strategy pattern to select the appropriate `Converter` and `PlanGenerator` based on the input language. This enables flexibility in supporting multiple input languages and planners without modifying core logic.

Listing 5.2 Composing Service

```
        @Service
        class ComposingService(
            private val composers: Map<Planner, Composer>,
            val sender: Sender
        ) {
            private val logger = getLoggerFor<ComposingService>()

            fun composeSolution(request: EncodingRequest<Problem, Domain>, requestId: String) {
                val planner = request.request.planner!!
                val language = request.request.language!!
                logger.info("Composing solution with planner '$planner' for language: $language")
                val plan: Any?
                try {
                    val composer = composers[planner]
                        ?: throw NotImplementedError("No composer implemented for planner: $planner"
)
                    plan = composer.composeSolution(request, requestId)
                } catch (e: Exception) {
                    logger.error("Error encoding problem for planner $planner", e)
                    throw ConvertorError(requestId, e.message ?: "Unknown error")
                }

                sender.sendPlan(plan, requestId)
            }
        }
```

The selection of converters and generators is also handled dynamically through a configuration class:

The interfaces `Converter` and `PlanGenerator` define a contract that all future implementations must conform to, ensuring extensibility, as shown in Listings 5.6 and 5.7.

5.5 Integration of the Fast Downward Planner

FD is integrated into the system as a standalone Docker container, orchestrated alongside other components using Docker Compose. A lightweight Flask-based REST service facilitates communication with FD, exposing endpoints for problem transformation and solution computation.

Encapsulating FD within a Docker container ensures a controlled and reproducible execution environment. The planner is accessed via a Python Flask service that manages PDDL problem transformations and plan computations through the following endpoints:

- `/transform`: Converts a given PDDL domain and problem into an intermediate SAS representation.
- `/solve`: Computes a plan using the transformed problem representation.

Listing 5.3 Composer Configuration

```
@Configuration
class ComposerConfig {

    @Bean
    fun composers(
        convertGenerateComposer: ConvertGenerateComposer,
        fdComposer: FdComposer
    ): Map<Planner, Composer> {
        return mapOf(
            Planner.FD to fdComposer,
            Planner.HSP to convertGenerateComposer,
            Planner.FFAnytime to convertGenerateComposer,
            Planner.FF to convertGenerateComposer,
            Planner.HCAnytime to convertGenerateComposer
        )
    }
}
```

This architecture allows interaction with FD through REST API calls, bridging the gap between the system's internal representation (Java/Kotlin) and the planner's requirements (Python).

The system follows a structured workflow for interaction with FD:

1. The parser submits a PDDL domain and problem to the `/transform` endpoint.
2. The transformed representation is returned and passed on to the generation unit.
3. The generation unit forwards the transformed problem to the `/solve` endpoint.
4. The computed solution is returned as a JSON response and parsed into the system's internal representation.

The communication between the generation unit and FD is illustrated in Listing 5.8.

This abstraction simplifies FD interaction while enabling flexible deployment via containerized execution.

The parser initially provides FD with raw PDDL domain and problem descriptions. However, FD requires a transformed input format within the generation unit. The system ensures proper processing by invoking the transformation step before requesting a solution.

The container is based on Python 3.9 and includes dependencies required for FD. The Dockerfile in Listing 5.9 demonstrates the setup.

This setup enables efficient REST-based communication for translating PDDL problems and computing solutions.

If a parsing error occurs, an exception is thrown and displayed on the frontend. The error output provides details such as failure reasons and relevant logs, that can be seen in Listing 5.10.

Listing 5.4 ConvertGenerateComposer

```
        @Component
        class ConvertGenerateComposer(
            private val configurations: Map<Language, Pair<Converter, PlanGenerator>>
        ): Composer  {
            override fun composeSolution(request: EncodingRequest<Problem, Domain>, requestId:
String):  Plan<*>? {
                val language = request.request.language!!
                val (converter, planGenerator) = configurations[language] ?: throw
IllegalArgumentException("No configuration found for $language")

                val problem = converter.convert(
                    problem = request.problem!!,
                    domain = request.domain!!,
                    requestId = requestId
                )
                return planGenerator.generatePlan(
                    encodedProblem = problem,
                    plannerName = request.request.planner!!
                )
            }
        }
```

Listing 5.5 ConvertGenerateComposer Configuration

```
        @Configuration
        class UnifiedConfig {

            @Bean
            fun composeStrategyConfigurations(
                pddl4JConverter: Pddl4JConverter,
                pddl4JPlanGenerator: Pddl4JPlanGenerator
            ): Map<Language, Pair<Converter, PlanGenerator>> {
                val composeCombination = mapOf(
                    Language.PDDL to Pair(pddl4JConverter, pddl4JPlanGenerator)
                )

                return composeCombination
            }
        }
```

Currently, the system does not implement a fallback mechanism upon failure. Errors are logged and reported, but no alternative solutions are attempted.

Listing 5.6 Converter Interface

```
interface Converter {
    fun convert(problem: Problem, domain: Domain, requestId: String): EncodedProblem
}
```

Listing 5.7 PlanGenerator Interface

```
interface PlanGenerator {
    fun <P> generatePlan(
        encodedProblem: EncodedProblem,
        plannerName: Planner
    ): P? where P : Plan<*>
}
```

Listing 5.8 REST communication with FD.

```
suspend fun solvePlanningProblem(transformed: String): String {
    val jsonBody = JSONObject().also {
        it["transformed"] = transformed
    }.toJSONString()

    val fastDownwardHost = composerConfig.fdHost

    val request = HttpRequest.newBuilder()
        .uri(URI.create("$fastDownwardHost/solve"))
        .header("Content-Type", "application/json")
        .POST(HttpRequest.BodyPublishers.ofString(jsonBody, StandardCharsets.UTF_8))
        .build()

    return withContext(Dispatchers.IO) {
        client.send(request, HttpResponse.BodyHandlers.ofString()).body()
    }
}
```

Listing 5.9 Dockerfile for FD container.

```
FROM python:3.9

# Install dependencies
RUN apt-get update && apt-get install -y \
    build-essential cmake g++ git && \
    rm -rf /var/lib/apt/lists/*

# Clone and build Fast Downward
WORKDIR /app
RUN git clone https://github.com/aibasel/downward.git && \
    cd downward && \
    ./build.py

# Install Flask for API handling
RUN pip install flask

# Copy the API script
COPY server.py /app/server.py

# Expose the port
EXPOSE 5000

# Start the API
CMD ["python3", "/app/server.py"]
```

Listing 5.10 Example parsing error from FD.

```
        Error while transforming PDDL using FD:
        Translation failed

        log:
        INFO     planner time limit: None
        INFO     planner memory limit: None
        INFO     Running translator.
        INFO     translator stdin: None
        INFO     translator time limit: None
        INFO     translator memory limit: None
        INFO     translator command line string: /usr/local/bin/python3 /app/downward/builds/release
/bin/translate/translate.py /app/domain.pddl /app/problem.pddl --sas-file output.sas
        Parsing...
        Error: Could not parse domain file: /app/domain.pddl
        Reason: Missing ')'
        translate exit code: 31

        Driver aborting after translate
        INFO     Planner time: 0.08s

        domain:
        *some domain content*

        problem:
        *some problem content*
```

6 Discussion

This chapter reflects on the key insights gained during the redesign and implementation of the PlanX Toolbox, highlighting fundamental lessons learned and discussing their broader implications for automated planning. Additionally, it provides concrete recommendations for future AI planning system architectures based on the findings of this research.

6.1 Reflection

The refinement of the PlanX Toolbox and the integration of additional planning components have provided significant insights into both the benefits and challenges of a component-based AI planning architecture. We now want to answer the research questions RQ1 and RQ2 in sections 6.1.1 and 6.1.2, respectively.

6.1.1 Challenges in Implementing a Component-Based Architecture in PlanX

The transition from the previous structure to a more modular, component-based architecture in PlanX introduced several critical challenges that required substantial consideration. One of the most significant issues was *interoperability*, as different planners and encoding mechanisms employ distinct internal representations. Even when utilizing standardized formats such as PDDL, additional translation layers would be necessary to bridge the variations in interpretation and ensure compatibility between components.

Another major obstacle involved *message standardization*. The absence of universally accepted messaging conventions for AI planning systems complicated integration between modular services. Without a structured approach to defining input-output formats, components were difficult to be extended or even replaced. To address this, a standardized message structure was introduced, though further refinement is needed to support broader adaptability.

Furthermore, the *tight coupling of components* in the previous architecture created substantial barriers to modularization. The interdependencies between solvers, parsers, and encoding mechanisms required extensive restructuring to ensure that components could operate independently while still maintaining system coherence. Decoupling these elements led to an increase in *pipeline complexity*, as additional communication channels were needed to coordinate the flow of data between services.

Despite these challenges, the feasibility of a modular architecture within the existing PlanX framework was successfully demonstrated. The introduction of the *Generation Unit* as an intermediary service for encoding and solving marked a significant step towards achieving modularity. While this redesign has proven effective, further enhancements are necessary to optimize interoperability across diverse planners and reduce overhead in system communication, especially in the context of the parsing process.

6.1.2 Impact of Component-Based Implementation on Maintainability and Adaptability

The adoption of a component-based architecture in PlanX has resulted in notable improvements in several key areas. One of the primary benefits is the enhancement of *maintainability*. By separating concerns across distinct, well-defined modules, individual components can now be modified and tested independently. This minimizes the risk of unintended side effects when updating specific functionalities, streamlining the development process and improving system stability.

Additionally, the modular structure has significantly increased *adaptability*. The decoupling of services facilitates the integration of new planners with minimal modifications. This advantage was exemplified by the successful incorporation of the FD planner, demonstrating that the new architecture supports the expansion of PlanX's capabilities without extensive reengineering.

Scalability remained largely unaffected by the redesign, as both the previous and the new version of the system utilize MOM, which is inherently efficient in terms of both vertical and horizontal scalability. MOM facilitates asynchronous communication between loosely coupled services, allowing the system to dynamically distribute workloads across multiple instances without performance bottlenecks. This ensures that scaling the system, whether by increasing processing power on individual nodes (vertical scalability) or by adding more service instances (horizontal scalability), remains efficient and effective.

6.2 Lessons Learned and Limitations

The redesign of PlanX has significantly enhanced its modularity, yet several limitations remain that need further refinement. A major challenge stems from the complexity of parsing, which was initially envisioned as a separate process from encoding and solving. However, this separation proved more complicated than anticipated due to the fact that many parsing implementations inherently integrate problem transformations, which are critical for the planners' operation. The interdependence between parsing and encoding complicates the development of a modular system where these components can be freely interchanged.

Another fundamental limitation pertains to semantic interoperability. Despite the adoption of standardized formats such as PDDL, variations in how different planners interpret constraints and optimization parameters hinder full interchangeability. Even within a standardized framework, planners may diverge in their assumptions and internal processing, necessitating additional adaptation layers to ensure compatibility.

Moreover, the reliance on predefined message queues introduces constraints in dynamically adapting to workload variations. While message-oriented middleware facilitates decoupled communication between services, predefined message flows can limit the system's ability to optimize resource allocation dynamically. More flexible mechanisms for workload distribution and adaptive queue management could improve responsiveness and scalability.

A further complexity arises in error handling and debugging. The modular design of the system, while beneficial for flexibility and maintainability, increases the difficulty of tracing issues across multiple interacting services. Debugging in a distributed environment necessitates comprehensive logging and monitoring mechanisms to efficiently diagnose errors that may originate from multiple components.

6.3 Recommendations and Future Work

Based on the findings of this research, the following recommendations outline strategies for improving the design and implementation of future AI planning systems. These focus on enhancing parser reusability, evaluating the cost-benefit trade-offs of modularity, and improving the efficiency of message handling.

6.3.1 Intermediate Representations for Parser Reusability

One potential approach to mitigating interoperability issues is the introduction of an Intermediate Representation (IR) that serves as a universal intermediary between problem encodings and solver-specific formats. In theory, this would allow planners to share a common parsed representation, reducing the need for custom parser implementations.

However, the feasibility of such an approach is questionable. Many planners perform optimizations during parsing, meaning that an IR would have to be minimalistic enough to remain compatible across solvers while still supporting useful transformations. This would likely result in the loss of solver-specific enhancements that improve efficiency.

The trade-off between parser reusability and solver-specific optimizations should be carefully analyzed. If an IR is too restrictive, it may hinder performance improvements. On the other hand, if it allows for planner-specific extensions, it risks reintroducing the same compatibility issues that it aims to solve. Future research should focus on evaluating whether an IR can provide tangible benefits without sacrificing planner efficiency.

6.3.2 Evaluating the Cost-Benefit of Increased Modularity

A highly modular AI planning architecture theoretically offers greater flexibility, but implementing modularity at all levels comes with a cost. The more abstract and generic a system becomes, the more effort is required to ensure that modules remain compatible while maintaining high performance.

The introduction of the *Generation Unit* in PlanX demonstrated that modularity simplifies planner integration. However, the question remains whether increasing modularity further—such as by designing a universally applicable IR—would justify the development overhead. There is also a risk that excessive modularity could introduce additional processing steps, reducing system efficiency.

Future work should explore quantitative analyses to determine the impact of modularity on system performance. This would help establish whether the benefits of flexibility and ease of extension outweigh the potential performance trade-offs.

6.3.3 Optimizing Message-Oriented Middleware and Queue Usage

A key limitation of the current implementation is its reliance on a *dedicated queue per service* model, where each component listens to specific request and response queues. While this ensures structured message flow, it introduces constraints in dynamically adapting to workload variations and new planner integrations. Additionally, predefined routing configurations require manual adjustments when extending the system.

Future AI planning systems should explore more flexible message-oriented middleware architectures that support dynamic queue creation and routing. By leveraging advanced message brokers such as RabbitMQ, planners could be added or removed without manual configuration changes. This would enhance system scalability and adaptability, enabling the integration of new components.

7 Conclusion

This thesis investigated the redesign of the PlanX Toolbox, focusing on advancing modularity, flexibility, and maintainability in AI planning systems. The primary objective was to realign the toolbox's architecture with principles from component-based and service-oriented software engineering, thereby enabling easier integration of diverse planning tools and reducing architectural complexity.

The redesigned architecture emphasized a clear separation of concerns among key planning functionalities—parsing, converting, and generating—while also restructuring their interactions to reduce coupling and improve extensibility. A major architectural enhancement was the fusion of converting and generating functionalities into a unified *Generation Unit*. This decision was based on the observation that conversion is always a prerequisite for generation, whereas parsing can be required independently. This decoupling reflects a deliberate design choice to accommodate varied usage scenarios, such as syntax validation without plan generation, thus promoting reuse and configurability.

A significant challenge addressed during the redesign was the reuse of parsing components across different planners. While many planners accept standardized representations like PDDL, their internal parsers diverge in terms of assumptions, preprocessing routines, and interpretation of constraints. This limits the extent to which parsing functionalities can be abstracted or shared. The findings indicate that parser reusability is not infeasible, but subject to substantial constraints stemming from tool-specific internal representations and tightly integrated processing stages.

The integration of the FD planner served as a concrete test case for evaluating interoperability in practice. Although FD accepts PDDL as input, it internally transforms the input into a proprietary SAS+ format. This transformation requires an additional intermediate step, which complicates the planner's integration into toolchains that operate on higher-level representations. The case study illustrates that adherence to common standards such as PDDL does not guarantee smooth tool interoperability, as differences in internal architectures and assumptions persist.

Throughout the redesign and implementation, several lessons were learned that are of broader relevance to the AI planning community. The importance of clearly defined component interfaces, standardized message structures, and minimal coupling between services emerged as critical for building flexible and maintainable systems. Additionally, the practical difficulties of tool interoperability — even in seemingly standardized ecosystems — underscore the need for future research into intermediate representations and reusable abstractions that can bridge gaps between planning tools.

Based on these insights, this thesis proposes several directions for future work. These include exploring the feasibility of lightweight intermediate representations to improve parser reusability, investigating dynamic message routing strategies to enhance scalability and flexibility, and conducting a quantitative evaluation of the trade-offs introduced by increasing modularity. Such efforts may contribute to making AI planning systems more composable, deployable, and usable across diverse domains.

This work demonstrates that a component-based redesign of AI planning systems such as PlanX is both technically viable and architecturally beneficial. The outcomes of this research contribute to the field of AI engineering by showing how modular design principles can be applied in the context of AI planning to improve system extensibility, maintainability, and future growth.

Bibliography

[ADN+12] F. André, E. Daubert, G. Nain, B. Morin, O. Barais. “F4Plan: An Approach to Build Efficient Adaptation Plans”. In: *Mobile and Ubiquitous Systems: Computing, Networking, and Services*. Ed. by P. Sénac, M. Ott, A. Seneviratne. Vol. 73. Series Title: Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 386–392. ISBN: 978-3-642-29153-1 978-3-642-29154-8. DOI: 10.1007/978-3-642-29154-8_47. URL: http://link.springer.com/10.1007/978-3-642-29154-8_47 (visited on 11/05/2024) (cit. on pp. 19, 29, 32–34).

[AHK+98] C. Aeronautiques, A. Howe, C. Knoblock, I. D. McDermott, A. Ram, M. Veloso, D. Weld, D. W. Sri, A. Barrett, D. Christianson. “Pddl| the planning domain definition language”. In: *Technical Report, Tech. Rep.* (1998). URL: https://www.researchgate.net/profile/Craig-Knoblock/publication/2278933_PDDL_-_The_Planning_Domain_Definition_Language/links/0912f50c0c99385e19000000/PDDL-The-Planning-Domain-Definition-Language.pdf (visited on 03/12/2025) (cit. on p. 19).

[BBD+12] J. Barreiro, M. Boyce, M. Do, J. Frank, M. Iatauro, T. Kichkaylo, P. Morris, J. Ong, E. Remolina, T. Smith, D. Smith. “EUROPA: A Platform for AI Planning, Scheduling, Constraint Programming, and Optimization”. In: (2012) (cit. on pp. 19, 29, 32–34).

[CFP08] A. Cesta, S. Fratini, F. Pecora. “Unifying planning and scheduling as timelines in a component-based perspective”. In: *Archives of Control Science* 18.2 (2008). Publisher: Versita, pp. 231–271. URL: https://urn.kb.se/resolve?urn=urn:nbn:se:oru:diva-24033 (visited on 10/10/2024) (cit. on pp. 29, 32–34).

[GB21] I. Georgievski, U. Breitenbücher. “A Vision for Composing, Integrating, and Deploying AI Planning Functionalities”. In: *2021 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. 2021 IEEE International Conference on Service-Oriented System Engineering (SOSE). ISSN: 2642-6587. Aug. 2021, pp. 166–171. DOI: 10.1109/SOSE52839.2021.00025. URL: https://ieeexplore.ieee.org/abstract/document/9564364 (visited on 10/21/2024) (cit. on pp. 30, 32–34).

[Geo23a] I. Georgievski. “PlanX: A Toolbox for Building and Integrating AI Planning Systems”. In: *2023 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. 2023 IEEE International Conference on Service-Oriented System Engineering (SOSE). ISSN: 2642-6587. July 2023, pp. 130–134. DOI: 10.1109/SOSE58276.2023.00022. URL: https://ieeexplore.ieee.org/abstract/document/10254773 (visited on 10/10/2024) (cit. on pp. 19, 21, 31–34, 45).

[Geo23b] I. Georgievski. “Towards Engineering AI Planning Functionalities as Services”. In: *Service-Oriented Computing – ICSOC 2022 Workshops*. Ed. by J. Troya, R. Mirandola, E. Navarro, A. Delgado, S. Segura, G. Ortiz, C. Pautasso, C. Zirpins, P. Fernández, A. Ruiz-Cortés. Cham: Springer Nature Switzerland, 2023, pp. 225–236. ISBN: 978-3-031-26507-5. DOI: 10.1007/978-3-031-26507-5_18 (cit. on pp. 31–34).

[Geo24] I. Georgievski. “Engineering AI Planning Systems”. In: (Apr. 2024) (cit. on pp. 19, 23, 24).

[GPAA24] I. Georgievski, A. V. Palghadmal, E. Alnazer, M. Aiello. “SH: Service-oriented system for HTN planning in real-world domains”. In: *SoftwareX* 27 (Sept. 1, 2024), p. 101779. ISSN: 2352-7110. DOI: 10.1016/j.softx.2024.101779. URL: https://www.sciencedirect.com/science/article/pii/S235271102400150X (visited on 03/10/2025) (cit. on p. 19).

[Gra20] S. Graef. “Designing and implementing usable, interoperable, and reusable services of AI planning capabilities”. In: (2020). URL: http://nbn-resolving.de/urn:nbn:de:bsz:93-opus-ds-112753 (visited on 02/20/2025) (cit. on pp. 45, 46).

[GSA23] I. Georgievski, M. Z. Shahid, M. Aiello. “AI temporal planning for energy smart buildings”. In: *Energy Informatics* 6.1 (Oct. 19, 2023), p. 18. ISSN: 2520-8942. DOI: 10.1186/s42162-023-00289-w. URL: https://doi.org/10.1186/s42162-023-00289-w (visited on 03/10/2025) (cit. on p. 19).

[HBBB21] D. Höller, G. Behnke, P. Bercher, S. Biundo. “The PANDA Framework for Hierarchical Planning”. In: *KI - Künstliche Intelligenz* 35.3 (Nov. 1, 2021), pp. 391–396. ISSN: 1610-1987. DOI: 10.1007/s13218-020-00699-y. URL: https://doi.org/10.1007/s13218-020-00699-y (visited on 10/10/2024) (cit. on pp. 31–34).

[LL09] K. B. Laskey, K. Laskey. “Service oriented architecture”. In: *WIREs Computational Statistics* 1.1 (2009). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/wics.8, pp. 101–105. ISSN: 1939-0068. DOI: 10.1002/wics.8. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/wics.8 (visited on 03/12/2025) (cit. on p. 26).

[LPK+24] Y. Liu, L. Palmieri, S. Koch, I. Georgievski, M. Aiello. *DELTA: Decomposed Efficient Long-Term Robot Task Planning using Large Language Models*. Sept. 13, 2024. DOI: 10.48550/arXiv.2404.03275. arXiv: 2404.03275[cs]. URL: http://arxiv.org/abs/2404.03275 (visited on 03/10/2025) (cit. on p. 19).

[MBR+24] A. Micheli, A. Bit-Monnot, G. Roger, E. Scala, A. Valentini, A. Rovetta, A. Trapasso, L. Bonassi, A. Gerevini, L. Iocchi, F. Ingrand, U. Kockemann, F. Patrizi, A. Saetti, I. Serina, S. Stock. “Unified Planning: Modeling, Manipulating and Solving AI Planning Problems in Python”. In: (2024) (cit. on p. 31).

[MPSK22] C. Muise, F. Pommerening, J. Seipp, M. Katz. *Planutils: Bringing Planning to the Masses*. 32nd International Conference on Automated Planning and Scheduling (ICAPS 2022). System Demonstrations. Publisher: AAAI Press. 2022. URL: https://edoc.unibas.ch/93500/ (visited on 10/28/2024) (cit. on pp. 31–34).

[PF18] D. Pellier, H. Fiorino. “PDDL4J: a planning domain description library for java”. In: *Journal of Experimental & Theoretical Artificial Intelligence* 30.1 (Jan. 2, 2018). Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/0952813X.2017.1409278, pp. 143–176. ISSN: 0952-813X. DOI: 10.1080/0952813X.2017.1409278. URL: https://doi.org/10.1080/0952813X.2017.1409278 (visited on 11/05/2024) (cit. on pp. 30, 32–34).

[ŠKK16] M. Štolba, A. Komenda, D. Kovacs. “Competition of Distributed and Multiagent Planners (CoDMAP)”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 30.1 (Mar. 5, 2016). ISSN: 2374-3468, 2159-5399. DOI: 10.1609/aaai.v30i1.9870. URL: https://ojs.aaai.org/index.php/AAAI/article/view/9870 (visited on 10/28/2024) (cit. on pp. 19, 30, 32–34).

All links were last followed on March 27, 2025.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature