Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master Thesis

Exploring the Challenges of Engineering AI Planning-Based Applications

Swjatoslaw Pastuchov

Course of Study:	Software Engineering
Examiner:	Dr. Ilche Georgievski
Supervisor:	Dr. Ilche Georgievski
Commenced:	December 1, 2022
Completed:	June 1, 2023

Abstract

Developing AI planning-based applications is a challenging task, which requires expertise both in planning and software engineering. It demands proficiency in architecture design, integration of planning functionalities, utilization of service-oriented technologies, deployment solutions, etc.

To uncover the challenges of building systems and applications involving AI planning, three separate software projects with varying scenarios are implemented and analyzed. Each of the projects is built with a different architecture: monolithic, Service-Oriented Architecture and microservices architecture, and integrates several planning tools. Subsequently, such findings of this practical research as: the importance of patterns in architecture design, the difficulties of planner selection, the utilization of service and deployment technologies are discussed and summed up.

Additionally, this thesis contains an attempt to formulate guidelines for planning technology selection. They cover determining whether a planner is up to date, its availability, compatibility, and usability.

Kurzfassung

Die Entwicklung planungsbasierter KI-Anwendungen ist eine anspruchsvolle Aufgabe, die das Fachwissen sowohl in der Planung als auch in der Softwareentwicklung benötigt. Sie erfordert Kenntnisse im Architekturdesign, der Integration von Planungsfunktionen, der Nutzung serviceorientierter Technologien, den Lösungen zur Softwarebereitstellung usw.

Um die Herausforderungen beim Aufbau von Systemen und Anwendungen mit KI-Planung aufzudecken, werden drei separate Softwareprojekte mit unterschiedlichen Szenarien implementiert und analysiert. Jedes der Projekte basiert auf einer anderen Architektur: monolithischer, serviceorientierter Architektur und Microservices-Architektur und integriert mehrere Planungswerkzeuge. Anschließend werden solche Erkenntnisse dieser praktischen Forschung wie die Bedeutung von Patterns im Architekturdesign, die Schwierigkeiten bei der Planerauswahl, die Nutzung von Service- und Softwarebereitstellungstechnologien diskutiert und zusammengefasst.

Darüber hinaus enthält diese Arbeit einen Versuch, Richtlinien für die Planung der Technologieauswahl zu formulieren. Sie umfassen die Feststellung, ob ein Planer auf dem neuesten Stand ist, seine Verfügbarkeit, Kompatibilität und Benutzerfreundlichkeit.

Contents

List of Figures

List of Tables

Listings

Acronyms

AI Artificial Intelligence. 3, 19, 20, 21, 22, 25, 26, 27, 33, 34, 37, 44

API Application Programming Interface. 25, 27, 67, 69, 70, 71, 72, 74, 78, 79, 80, 83

AWS Amazon Web Services. 44, 48, 56, 57, 58, 59, 61, 63, 64, 78, 79, 83, 84

CLI Command-Line Interface. 28, 34, 38, 39, 40, 41, 48, 49, 53, 60, 64, 71, 72, 75, 81, 83

CORS Cross-Origin Resource Sharing. 70, 79, 81

CPEF Continuous Planning and Execution Framework. 28

CSS Cascading Style Sheets. 43, 46

DLQ Dead-Letter Queue. 58, 59

EC2 Elastic Compute Cloud. 44, 48, 49, 50, 53, 56, 57, 58, 63, 83, 84

ESB Enterprise Service Bus. 77

GUI Graphical User Interface. 31, 43, 46, 47, 48, 61, 69, 70

HDDL Hierarchical Domain Definition Language. 23, 36, 44, 47, 53, 55, 56, 63, 69, 75

HTN Hierarchical Task Network. 23, 28, 36, 53

HTTP Hypertext Transfer Protocol. 81

IAM Identity and Access Management. 58

ICAPS International Conference on Automated Planning and Scheduling. 29

ID Identifier. 46, 48, 49

IDE Integrated Development Environment. 43, 67

IoT Internet of Things. 25

JAR Java Archive. 49, 50, 56, 63, 71, 74, 97

JDK Java Development Kit. 49, 50

JSON JavaScript Object Notation. 46, 48, 49, 52, 53, 55, 70, 71, 72, 74

KEPS Knowledge Engineering in Planning and Scheduling. 19

LPG Local search for Planning Graphs. 98

OPTIC Optimizing Preferences and TIme dependent Costs. 99

OS operating system. 30, 36, 39, 63, 97

PDDL Planning Domain Definition Language. 15, 22, 23, 28, 29, 33, 34, 35, 36, 37, 38, 39, 41, 44, 46, 47, 48, 49, 50, 52, 53, 55, 69, 71, 72, 74, 78, 97, 98, 100

PRP Planner for Relevant Policies. 100

PyPI Python Package Index. 34

REST Representational State Transfer. 67, 70, 77, 78, 79, 80, 81, 83, 90

SDK Software Development Kit. 48, 49, 50, 55, 78

SNS Simple Notification Service. 59

SOA Service-Oriented Architecture. 3, 19, 20, 28, 43, 56, 61, 81, 83, 89

SOA-PE Service Oriented Architecture for Planning and Execution. 25

SQS Simple Queue Service. 44, 46, 48, 49, 50, 52, 53, 55, 56, 57, 58, 59, 60, 61, 78, 83

SSH Secure Shell Protocol. 57

STRIPS STanford Research Institute Problem Solver. 22, 97

UI User Interface. 37, 38

URL Uniform Resource Locator. 70, 72, 74, 79, 80, 81

UUID Universally Unique Identifier. 46, 48, 52, 55, 59

1 Introduction

The first chapter of this thesis explains the necessity for investigation of challenges which developers are confronted with in the field of Artificial Intelligence (AI) planning applications. It also provides the objectives that the thesis is meant to achieve, and a short summary of the following chapters.

1.1 Motivation

AI is one of the newer fields both in science and technology. Beginning with the first work generally recognized as AI, a model of artificial neurons proposed in 1943, the field grew and is utilized today in many different areas, like robotics, speech recognition, game playing, planning, etc. [RN10]

Building applications based on AI planning is a challenging task. It requires not only an expertise in Knowledge Engineering in Planning and Scheduling (KEPS) [VK20] but also in software engineering. A developer must be familiar with the implementation and integration of planning functionalities, architecture design, employment of service-oriented technologies, cloud computing, and deployment solutions [GB21]. As result, it is important to understand the types of challenges that are faced by developers of applications which make use of AI planning.

To name a few fields involving AI-planning applications: smart energy systems [AFG21], for taking advantage of the automation and optimization opportunities given by the digitalization, autonomous driving [AGPA22], for increasing the driver's trust in the vehicle's capabilities, and office activity recognition [Geo22a], for dealing with all the characteristics of the office activity recognition problem.

1.2 Goals

The main goal of this thesis is the exploration of challenges encountered when engineering planning-based applications. The exploration is concerned with several implementation stages: starting with design, continuing with building and integration, and ending with deployment of an application.

For this purpose, scenarios for different implementations of planning-based applications are designed, which are based either on the existing literature or on popular architectural styles, e.g., Service-Oriented Architecture (SOA). The scenarios are also meant to be diverse, with the intent being the coverage of different types of planning functionalities, different types of interaction and varying complexity of applications. The resulting applications include both simple planners and composite applications with multiple components and planning functionalities.

Once the work on the implementation of scenarios is completed, the challenges encountered during each scenario are analyzed. Design choices, necessary preparations, implementation decisions, limitations, encountered issues and general experiences are documented and discussed. The discussion ends by providing key observations and key recommendations.

Additionally, as a consequence of requiring planning technologies for the development of planning-based applications, this thesis investigates the process of planner selection. It attempts to formulate guidelines for determining whether a planning technology is suitable for integration into an application or a system.

1.3 Outline

The thesis contains the following chapters:

Chapter 2 - Background provides the background information on Artificial Intelligence, AI planning and scheduling and planning types.

Chapter 3 - State of the Art lists several related papers on the subject of planning-based applications and a few software examples offering multiple planning capabilities.

Chapter 4 - Which Planning Tools to Use? investigates the difficulties of planning technology selection, proposes adjustments to a software development cycle reliant only on available planning tools and demonstrates the planning tools used in the following chapters.

Chapter 5 - Project 1: Monolithic Architecture describes and analyzes a project based on a monolithic architecture.

Chapter 6 - Project 2: Service-Oriented Architecture describes and analyzes a project based on a Service-Oriented Architecture.

Chapter 7 - Project 3: Microservice Architecture describes and analyzes a project based on a microservices architecture.

Chapter 8 - Conclusion wraps up the thesis by providing a summary of the done work and an outlook.

2 Background

Chapter 2 provides the general details about Artificial Intelligence, planning and scheduling, planning types and planning languages. This background information is meant to lay out the basics of planning for non-experts.

2.1 Artificial Intelligence

According to Russel and Norvig [RN10] there are historically four approaches to AI:

- Acting humanly: the Turing Test approach.

  In 1950 Alan Turing designed the “Turing Test”, which purpose is to provide an operational definition of intelligence. If a person cannot distinguish between a computer and another person, the computer passes the test. For this, a computer would require four disciplines: natural language processing, knowledge representation, automated reasoning and machine learning.

- Thinking humanly: the cognitive modeling approach.

  This approach requires determining how humans think before being able to express it as a computer program. The three ways to model cognition are: introspection, psychological experiments and brain imaging.

- Thinking rationally: the “laws of thought” approach.

  The laws of thought were an attempt by Aristotle to codify the reasoning process, the study of those laws is the field called “logic”. There are two problems with this approach: first, transforming informal knowledge into formal terms, and second, solving a problem “in principle” vs. solving it in practice.

- Acting rationally: the rational agent approach.

  A rational agent is a computer that can act to achieve the best outcome. For this purpose, it needs to be able to operate autonomously, perceive its environment, persist over a prolonged time period, adapt to change, and create and pursue goals.

Today AI is utilized in many fields: robotics, robotic vehicles, autonomous planning and scheduling, logistics planning, game playing, speech recognition, machine translation and more. [RN10]

2.2 AI planning and scheduling

Planning is a synthesis task, involving formulating a course of action to reach some desired objective. One example of such an objective in a planning problem is achieving a defined set of goals. In practical terms, the form of a plan is often a simple sequence of orderings or actions. [SFJ00]

AI planning entails generating a plan by solving a given problem and providing a possible solution. An AI planning system expects a twofold input: actions, usually described as preconditions and effects, and a problem, containing an initial state description and a goal. The generated plan is a collection of actions and is a solution to the input problem. [HTD90]

Scheduling involves defining resource and temporal constraints, which are applied to activities meant to achieve a particular goal [VK20]. According to Smith et al. [SFJ00] there are three important things about scheduling to consider:

- The core of scheduling problems is the reasoning about time and resources.
- Scheduling problems are usually optimization problems, concerned not just with finding a solution, but with finding an optimal solution.
- Scheduling problems involve choices, with potentially several alternative resources with different costs and/or durations being available.

Where scheduling is often focused on developing optimized techniques for specific classes of scheduling problems, the field of AI scheduling is more focused on general representations and techniques that cover a range of different types of scheduling problems. [SFJ00]

2.3 Planning types

There are multiple different ways to solve a given problem in order to achieve a goal, resulting in different planning types. Section 2.3 provides information on a few of them.

2.3.1 Classical planning

In classical planning the objective is to achieve a given set of goals. The goals, as well as the initial state of the world, are expressed as a set of positive and negative literals [SFJ00]. Those literals are usually described with the help of a language called the Planning Domain Definition Language (PDDL).

PDDL is based on the concepts introduced in the STRIPS (STanford Research Institute Problem Solver) problem-solving program, proposed in 1971 by Fikes and Nilsson [FN71]. It is used to define the properties of a domain (see Appendix B.1) by defining predicates, possible actions, and the preconditions and effects of those actions [GKW+98]. It is also used to define the properties of a problem (see Appendix B.2), which are an initial state and a goal, as mentioned previously.

To date, the original PDDL 1.2 language has been extended with additional features, resulting in multiple versions: PDDL 2.1 [FL03], PDDL+ [FL02], PDDL 2.2 [Ede04], PDDL 3.0 [GL05] and PDDL 3.1 [Hel08][1].

2.3.2 Hierarchical planning

Hierarchical Task Network (HTN) planning differs from classical planning in regard to the problem solving approach. While it also requires a description of an initial state and the objective to be achieved, instead of a set of actions it deals with a task network consisting of primitive and compound tasks. A problem is solved not by arranging actions to reach certain goals, but by reducing the compound tasks of the initial task network into primitive tasks – a hierarchical decomposition. [GA15]

The Hierarchical Domain Definition Language (HDDL) is an extension to the PDDL language. It was proposed in 2020 by Höller et al. [HBB+20] and it serves as a common input language for hierarchical planning problems. It is based on the input language of PANDA (see Appendix C.2). An example of an HDDL domain can be found in Appendix B.3 and an HDDL problem example in Appendix B.4.

2.3.3 Other planning types

There are more planning types than classical and hierarchical, handling different aspects of planning models. As they are less relevant for this thesis, this section contains only an short list of other planning types with short clarifications. Those types include, but are not limited to:

- Temporal planning: models must specify what actions and events are caused at various points during an action's performance. [GNT16; Rin07]
- Non-deterministic planning: models can predict alternative options, meaning that an action applied in a state can result in multiple possible states. [GNT16; MMB12; MR15]
- Probabilistic planning: models are necessarily incomplete, as the future is not entirely predictable, and the possible action outcomes are not equally likely. [GNT16]
- Decision-theoretic planning: models have actions with uncertain effects, and different factors lead to solutions of varying quality. [BDH99]

2.4 Software engineering

In this thesis it is assumed that the reader is at least somewhat familiar with such software engineering aspects as architectural styles and communication technologies and principles. However, should that not be the case, in Appendix A one can find short definitions on a few of those subjects, which are used in later chapters.

[1]https://ipc08.icaps-conference.org/deterministic/PddlExtension.html

3 State of the Art

With the goal of this thesis being the exploration of challenges which are encountered during designing, building, integrating, and deploying planning-based applications, Chapter 3 lists several scientific papers concerned with applications and architectures making use of AI planning, and provides a few software examples with multiple planning capabilities.

3.1 Related papers

"SOA-PE: A service-oriented architecture for Planning and Execution in cyber-physical systems"

[FMJB15] proposes a Service Oriented Architecture for Planning and Execution (SOA-PE), to provide separation between domain modeling, planning, execution, monitoring and evaluation services. The functions supported by each service are as follows:

A. The Domain Management Service manages domains and provides an Application Programming Interface (API) for designing domain models. It allows to check them for consistency, correctness and completeness.

B. The Planning Service takes domain and problem as input, generates a plan as output. The API offers such functions as selection of planning engine and parameters, selection of output format, translation of output plans, refactoring of existing plans, decomposition and composition of plans.

C. The Execution Service executes the generated plan by parsing it and scheduling the individual plan steps.

D. The Domain Sensing Service builds the state of the world as per the domain model, which requires interfacing with Internet of Things (IoT) Services.

E. The Domain Actuation Service is invoked to execute plan steps.

F. The Monitor Service supervises the state of the plan and the state of world during plan execution.

G. The Application Controller Service handles external requests to incorporate new goals, enforces specified policies for planning, execution and actuation services.

“A Vision for Composing, Integrating, and Deploying AI Planning Functionalities”

In [GB21] a vision for designing and deploying integrated and composed AI planning systems is presented. It includes 3 steps:

1. Creation of a Logical Composition Model by a user, consisting of a graph with nodes representing planning tools. Their functionalities are specified by unique names, and the nodes are interconnected with weighted edges representing composition relationships.
2. Automated generation of a Logical Composition Model, which is the updated graph from step 1 with additional nodes. The new model is based on messaging technology with the added nodes signifying such components as queues, Message Routers, Message Translators, etc.
3. Automated transformation into an Executable Deployment Model, which an updated previous graph. The newly added nodes represent services to be deployed, with the edges representing relationships between those services.

Figure 3.1 shows the diagram of a messaging-based system employing planning technology described in step 2.

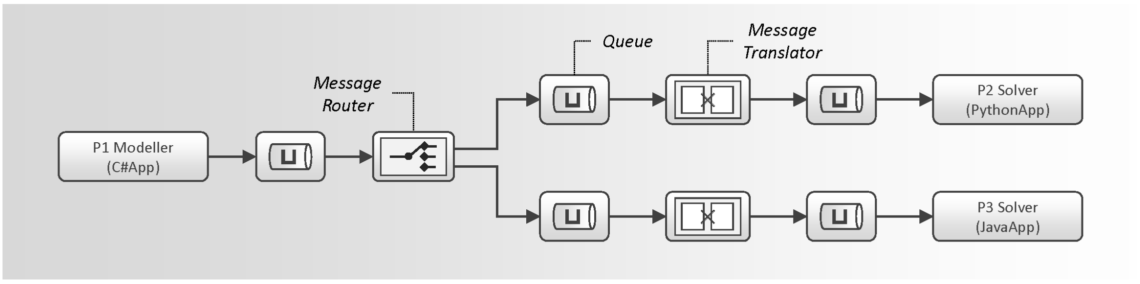


Figure 3.1: Excerpt: diagram of a system with planning services. [GB21]

“Towards Engineering AI Planning Functionalities as Services”

[Geo22b] provides an overview of the challenges related to engineering and use of AI planning tools:

- Architecture:
  - Component Complexity: existing planners are often complicated with intertwined algorithms.
  - Interoperability: a common syntax of input of different planners does not guarantee that they are able to communicate with each other.
  - Reuse: planners employ functionalities, e.g., parsing, that could be reused between different tools, which is difficult to accomplish due to the planning tools’ complexity.

- Development and Deployment:
  - Integration: different planners use different data models, algorithms and domain models, while lacking APIs for controlling interactions.
  - Composition: the mechanisms for enabling the composition of custom planning systems or extending existing ones have not been investigated.
  - Deployment: the mechanisms for deployment, including versioning, performance measurement and raising warnings have not been established.
- Process:
  - Heterogeneity: the diversity of planning functionalities, the multitude of existing planning tools, and the various classes of planning makes identification, selection and management of planning functionalities difficult.
  - Set Up: figuring out the requirements, managing dependencies and lackluster instructions makes setting up planners into a challenge.
  - Monitoring: deployed planners require performance monitoring, event logging and error tracking.

"Conceptualising Software Development Lifecycle for Engineering AI Planning Systems"

In [Geo23] a development lifecycle with ten phases for AI planning software is proposed:

a) Requirements Analysis: identification of requirements by relevant stakeholders. The requirements are functional, non-functional, goal-oriented, and user-related.

b) Planning Model Selection: definition of a planning model is based on the requirements from hase a) and requires expertise and experience in planning.

c) Domain Model Design: domain information is derived from phase a) requirements and then formalized in a planning syntax.

d) Architecture and Design: the phase b) planning model imposes conditions on the interaction model between planning components.

e) Planning Technology Selection: exploration of existing planning tools, selection if available.

f) Implementation: the planning system is implemented following the architecture design. Components without available planning technology are developed as well.

g) Testing: the developed planning system must fulfil the requirements from phase a).

h) Deployment: manual or automated deployment to make the planning system ready for execution.

i) Monitoring: collecting data specific to planning and storing it for analysis.

j) Analysis: identification of problems and generation of relevant insights for planning system improvement.

3.2 Related software

planutils

planutils[1] [MPSK22] is a library for developing, running, and evaluating planners. While it is not a system that enables interaction between planning components, it allows installation of and access to multiple existing planners from a single Command-Line Interface (CLI). Examples of *planutils'* utilization can be found in Appendix C.4.

Planning.Domains

Planning.Domains[2] [ML20] is a browser tool providing domain modeling and plan generating capabilities, among other things. It consists of four principal components:

1. "A programmatic interface[3] to all existing planning problems."
2. "An open and extendable interface[4] to a planner-in-the-cloud service."
3. "A fully featured editor[5] for creating and modifying PDDL."
4. "A central source[6] for educational resources for planning."

CPEF

Continuous Planning and Execution Framework (*CPEF*) [Mye99] is a system with multiple planning capabilities. It is a composition of multiple components and can be applied to solve planning problems to generate plans, execute plans, monitor plan execution, and repair complex plans. As it was developed in 1999, it does not employ a SOA architecture.

PELEA

PELEA [GAP+12] is an online planning architecture presented in 2012. It is a domain-independent and component-based architecture, offering modeling, solving, execution, monitoring, repairing, and learning capabilities. It employs a continuous planning approach, combining planning and execution, and works both with PDDL- and HTN-based planning.

[1] https://github.com/AI-Planning/planutils
[2] http://planning.domains
[3] http://api.planning.domains
[4] http://solver.planning.domains
[5] http://editor.planning.domains
[6] http://education.planning.domains

4 Which Planning Tools to Use?

When designing a system with planning capabilities one may want to make use of one or more planning tools to avoid having to implement own ones. Doing so without making sure that the required tools are available beforehand is not the best course of action, which may lead to multiple issues later. The following sections contain the guidelines, which have emerged while researching and testing available planning tools, and an adjusted software development cycle for applications based only on available planners. In addition to that, the chapter lists the details of several tested planning tools, which are used in the three projects described in the following chapters.

4.1 Selecting planning tools

During the research of planning technologies one should ask themself the following questions:

1. What type of planning is required?

   As described previously in Section 2.3 on page 22, there are many types of planning, e.g., classical, hierarchical, temporal. Deciding on the planning type best suited for a specific purpose is the prerequisite of all following steps.

2. What tools are available for the selected planning type?

   Over the years a multitude of planning and scheduling tools, e.g., at International Conference on Automated Planning and Scheduling (ICAPS) competitions[1], have been developed. For example, a variety of tools exist for classical planning: *Fast-Forward* (see Appendix C.1), *Fast Downward* (see Section 4.4.2), *LAMA* (see Appendix C.4.1), *PRP* (see Appendix C.4.5), etc. Having a wide selection of tools available increases the chances of finding one best suited for one's purposes.

3. Are selected tools compatible?

   When selecting multiple tools to work together, their compatibility should be verified. For example, even if two or more chosen tools are employing classical planning, they may still be using different PDDL versions. If the intent is to make use of features of later PDDL versions, one should make sure they are supported by all employed tools.

4. Is the selected tool up to date?

   Any software can become outdated, and the planning tools are no exception. Before investing any time into designing a system depending on a specific planning tool, it is advisable to ensure that it and its dependencies are not out of date.

[1]https://www.icaps-conference.org/competitions/

4.1 How old is the repository/executable?

The most obvious sign of an outdated or abandoned tool is the age of its repository or the release date of its executable. Before deciding to settle for a planning tool developed over a decade ago, it may be worth it to research more recent alternatives. Even if an old planning tool still provides viable solutions, an improved implementation of the same/similar algorithm can exist somewhere else.

4.2 Is any documentation available?

The importance of documentation cannot be overstated. Having instructions on required dependencies, building, execution, expected parameters, input format restrictions, expected output, etc., reduces developer's workload and stress. Missing documentation is also another sign of an outdated or abandoned tool.

4.3 Is the documentation sufficiently detailed?

If documentation is available, it is worthwhile to ensure that it is sufficiently detailed, e.g., having nothing but installation instructions may not be enough for proper operating of the planning tool. Missing guidelines can cost development time when solving an unexpected tool behavior. A rule of thumb: the more extensive the documentation is, the better are the chances that the tool is still maintained.

4.4 What is the required operating system (OS) architecture?

It is possible that an older tool is still reliant on an OS with an outdated architecture. Developing a system working in a 64-bit environment could be problematic if a planning tool works only in a 32-bit environment.

4.5 How old are the dependencies?

Similar to the planning tool's repository age should the age of the required dependencies be considered. Older versions of dependencies may be unavailable or have security vulnerabilities, which have been patched in later versions. If an application is not intended for personal use, integrating a planning tool requiring such dependencies should be reconsidered.

4.6 Are the dependencies compatible with my application?

While a planning tool can function without issues on its own, its dependencies can come in conflict with the application the tool is supposed to be integrated with. For example, PANDA (Appendix C.2) requires the environment to run Java 8[2], which will cause compatibility issues if one's application is being developed using Java 19[3].

5. Is the selected tool operable?

Once it has been determined that the selected planning tool is not outdated, it is a good idea to test it hands-on: cloning the source code, building or downloading the tool's executable, and running the tool. After all, it does not matter if the tool is new or well documented if it refuses to function.

[2]https://www.oracle.com/java/technologies/javase/javase8-archive-downloads.html
[3]https://www.oracle.com/java/technologies/javase/jdk19-archive-downloads.html

5.1 Is the tool's executable available?

To state the obvious, before a tool can be executed, its executable must be acquired. If the planning tool's repository offers a release executable, it can just be downloaded, otherwise one may need to clone the tool's source code and build it manually. In the latter case, engaging the build scripts can uncover multiple issues, e.g., errors in the source code, broken build scripts, missing or wrong dependencies, or unspecified build arguments not mentioned in the documentation.

5.2 Can the tool be executed?

It would be a mistake to assume that just because an executable is available, the tool is in a working condition. The executable may not have been built correctly, its download could be corrupted, or it could show problems during runtime. Alternatively, if the executable is functioning, it could be difficult to run it properly if the documentation omits some critical information, e.g., expected arguments.

5.3 Is the tool working correctly?

Once the tool can be executed successfully, one needs to determine whether it also delivers correct results. For this purpose, it is advisable to use inputs which deliver a known output, similar to writing software tests.

5.4 Does the tool work with varied inputs?

It should not be assumed that a planning tool is working perfectly after one successful test. Before relying on a tool, it should be tested with multiple different inputs, e.g., testing a problem solver by using simple and complex problems requiring different processing times, resulting in both short and long plans. It may very well be the case that a tool works fine with simpler problems but freezes when calculating larger or more complex ones.

6. What is the tool's output?

   After determining that a tool is in a functioning condition and is performing as well as expected, one should consider the type of output it produces. A tool may simply write its output in the terminal, show it in a GUI (Graphical User Interface), produce new files, etc. Depending on the application, which will have to make use of the planning tool's output, a specific output type can cause difficulties or be of an advantage.

7. Is changing the tool's output type possible?

   The output can require format transformation before it can be used further. For example, if all output of a process ends up in the terminal, both logs and results, then the necessary information will have to be extracted manually. Such output parsing can be unreliable, especially if the output format varies or is not well documented. It could prove worthwhile investigating the possibility of altering the tool's source code to change its output type, instead of having to accommodate for it later.

8. How large is the tool?

   Lastly, the planning tool's size should be taken into consideration. The amount of space a tool requires can vary significantly, e.g., *LPG-td* (see Appendix C.3) needs less than 2 MB, while *Fast Downward* requires almost 800 MB. A larger size can prove problematic, e.g., when adding the tool to a Docker image.

9. Can the tool be downsized?

   Should a tool's size prove too large for an intended purpose, then perhaps it can be reduced. An application may not necessarily require all functionality of a tool, instead making use only of one of its features. If a tool has multiple executables or many additional resources, then removing them without affecting the necessary functionality might be possible.

4.2 Discussion

The following paragraphs contain additional guidelines and findings from a non-expert point of view, which have arisen while researching planning tools.

4.2.1 Knowing when to move on

A situation can occur when one is faced with an older planning tool, which simply refuses to execute, despite following the documentation instructions to the letter. Even after multiple attempts an executable fails to build, does not run, or just throws cryptic error messages. In such a case one may be tempted to continue tinkering with the tool in the attempt to locate and fix the error.

With new planners being developed constantly there are currently over a hundred of different planning tools available (an incomplete list can be found on the unofficial Planning Wiki[4]). This means that there is a good chance that a suitable alternative to the broken tool is available. Unless one needs that specific planning tool, it may be best to stop wasting time and effort and move on to a substitute.

4.2.2 Documenting progress

If one elects to abandon a planning tool, as mentioned above, it is still worth documenting any proceedings and/or results. The number of available planners is large, but it is not unlimited, meaning that some specific use cases may not have a large variety of tools available. There is a possibility in which one may have no choice but to revisit a previously rejected planner, in which case having notes will make that easier.

[4]https://planning.wiki/ref/planners/atoz

4.2.3 Automated problem generation

During the preparation for Project 1 Scenario 1 (see Section 5.2 on page 38) it was initially assumed that some sort of automated generation of problem PDDL files would take place. While there are problem generators[5] [6] for specific domains available, no domain-independent generators could be found. The question arises whether there is a reason for it or whether it is a gap in software solutions, which could be filled.

When looking at the basic PDDL 1.2 format one can see that besides the problem- and the corresponding domain-name, a problem PDDL needs only 3 types of arguments:

- "objects" can be constructed from domain's "types".
- "init" can be combined from domain's "predicates" and problem's "objects".
- "goal" can be combined in the same manner as problem's "init".

Of course, most problems generated in such a way would most likely have no valid plan solutions. But it can be assumed that they could still find a use, e.g., for some sort of automated fuzz testing of PDDL solvers.

4.3 Software development cycle reliant on available planning tools

As described previously in Section 3.1, an AI planning software development cycle is proposed in [Geo23]. This development cycle requires not only system engineering expertise, since the phase *f) Implementation* also "includes developing components for planning functionalities for which no available technology can be found". Development of planning algorithms and functionalities is not a trivial task, requiring proficiency in planning technologies or a planning expert. If one were to develop a system relying only on available planning technologies, of which there is a large number available, the development cycle would be as follows:

a) Requirements Analysis
b) Planning Model Selection
c) Planning Technology Selection and Adjustment
d) Domain Model Design
e) Architecture and Design
f) Implementation
g) Testing
h) Deployment
i) Monitoring

[5]https://github.com/AI-Planning/pddl-generators
[6]https://fai.cs.uni-saarland.de/hoffmann/ff-domains.html

j) Analysis

The change is twofold: first, it involves moving the *Planning Technology Selection* from phase *e)* to phase *c)*. A developer, who is a non-expert in the field of AI planning, would not be familiar with planning technologies. It makes sense to research available planners which are suitable for the selected planning model first, before beginning the *Domain Model Design* and *Architecture and Design* phases. Discovering that a planner is unavailable afterwards could entail both architecture and domain model redesign. Additionally, a planning tool could end up imposing rules or limitations on the architecture design.

Second, the *Planning Technology Selection* phase is extended to include a potential modification of the selected planning tool, becoming *Planning Technology Selection and Adjustment*. As discussed in Section 4.1, the output of the selected planner may be delivered in an unreliable manner. In that case, the possibility of adjusting the output delivery method could be worth investigating.

4.4 Planners used in Projects 1, 2 and 3

Section 4.4 contains the description of the four planning tools used in Project 1, Project 2 and Project 3 (see Chapter 5, Chapter 6 and Chapter 7). The tools were tested in a VirtualBox[7] instance running a Debian[8] 11 64-bit (amd64) environment. A list of other planners that were investigated can be found in Appendix C.

4.4.1 pddl

The *pddl* tool is a parser supporting the features of PDDL 3.1, its source code[9] is obtainable from GitHub[10]. It can be used to model PDDL domains and problems programmatically by employing it as a library in a Python[11] script. Additionally, it can also be used as a CLI tool to validate and format PDDL domain and problem files. The installation[12] of *pddl* is done through Python Package Index (PyPI)[13] and may require adding the tool's executable to the PATH variable.

When utilizing the tool for modeling purposes, it is used to import the required classes, as shown in Listing 4.1. Listing 4.2 contains an example of using the tool as a validator of domain and problem PDDL files.

[7]https://www.virtualbox.org
[8]https://www.debian.org
[9]https://github.com/AI-Planning/pddl
[10]https://github.com
[11]https://www.python.org
[12]https://pypi.org/project/pddl/
[13]https://pypi.org

```
from pddl.logic import Predicate, constants, variables
from pddl.core import Domain, Problem, Action, Requirements
from pddl.formatter import domain_to_string, problem_to_string
...
```

Listing 4.1: Using *pddl* as a library.

```
$ pddl domain domain.pddl
...
$ pddl problem problem.pddl
```

Listing 4.2: Using *pddl* as a CLI validator.

Whether modeling or validating domains and problems, the tool's output ends up in the terminal. Since the Python script, which is used for modeling, is edited by the user, it is possible to insert markers into the output to make it easier to extract results.

4.4.2 Fast Downward

Fast Downward [Hel11] is a domain independent classical planning system, with the source code[14] and several releases[15] available on GitHub, and with an extensive Wiki[16]. It supports PDDL 2.2 and the feature ":action-costs" from PDDL 3.1, with a few limitations[17].

To obtain[18] the tool, the following dependencies are required: git, cmake, make, g++ and python3. After cloning the source code from GitHub, the tool can be built by executing the "build.py" script located in the root directory.

An example of the utilization of *Fast Downward* to solve a problem and generate a plan is provided in Listing 4.3. The [--search] option is used to select a search engine[19], without it no plan will be created.

```
$ ./fast-downward.py domain.pddl problem.pddl --search "astar(blind())"
```

Listing 4.3: Executing *Fast Downward* to generate a plan.

The output of the solver is a text document named "sas_plan", generated at the location from where the tool's executable was called. Should the solving process fail, an intermediate file named "output.sas" can be found at the location instead.

[14]https://github.com/aibasel/downward
[15]https://github.com/aibasel/downward/releases
[16]https://www.fast-downward.org
[17]https://www.fast-downward.org/PddlSupport
[18]https://www.fast-downward.org/ObtainingAndRunningFastDownward
[19]https://www.fast-downward.org/Doc/SearchEngine

4.4.3 VAL

VAL [HLF04] is a plan validation system, for plans generated from PDDL 2.2 domains and problems. Its source code[20] is available on GitHub and requires the following dependencies for compilation and execution: cmake, make, g++, mingw-w64, flex and bison.

Once the dependencies are installed, the source code is cloned and the tool is compiled using an appropriate build script, e.g., "build_linux64.sh" for an installation on a Linux OS. As can be seen in Listing 4.4, the build script requires two input arguments: a rule for the build target and the build directory name. The documentation omits this information for an unknown reason, and it had to be discovered manually.

```
$ ./build_linux64.sh all release
```

Listing 4.4: Compiling *VAL* to the "release" directory.

The tool's validation functionality is used by running the "Validate" executable, located in the generated build directory. An example is provided in Listing 4.5.

```
$ ./Validate domain.pddl problem.pddl plan.txt
```

Listing 4.5: Executing *VAL* to validate a plan.

For output, the tool determines whether a plan could be executed successfully and whether its goal was satisfied and writes the results to the terminal. As for errors in the provided files: a mistake in the domain definition will be detected, whereas a mistake in the problem definition will cause a segmentation fault.

4.4.4 Lilotane

Lilotane (Lifted Logic for Task Networks) [Sch21] is a planner for totally-ordered HTN planning problems. Its source code[21] [22] can be found on GitHub or GitLab[23] and the tool requires the dependencies cmake and make for building an executable.

Listing 4.6 provides an example of utilizing *Lilotane* to generate a plan from domain and problem HDDL files.

```
$ ./lilotane domain.hddl problem.hddl
```

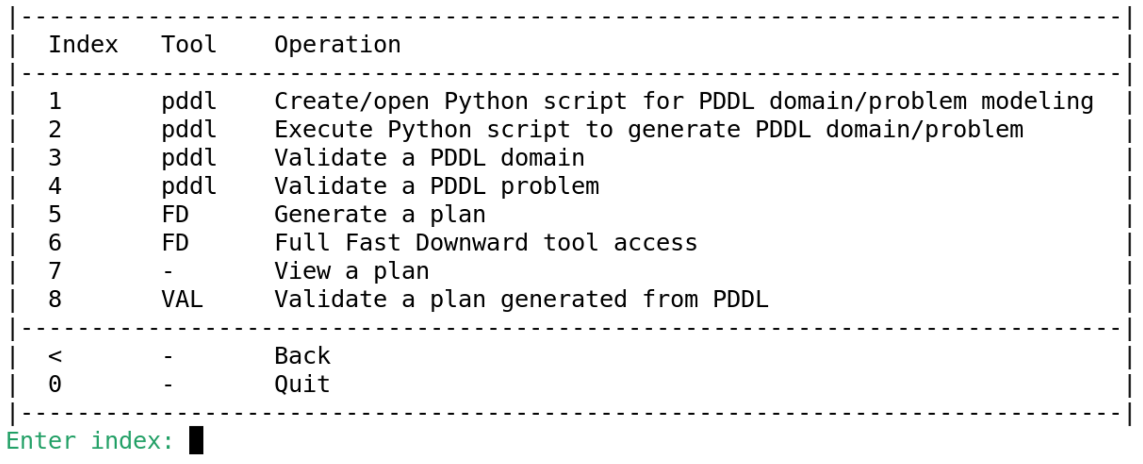
Listing 4.6: Executing *Lilotane* to generate a plan.

If a plan is created, it is output in the terminal. The tool detects mistakes in both domain and problem HDDL files during parsing.

[20]https://github.com/KCL-Planning/VAL
[21]https://github.com/domschrei/lilotane
[22]https://gitlab.anu.edu.au/u1092535/htn2020-competitor-1
[23]https://about.gitlab.com

5 Project 1: Monolithic Architecture

For the first project a monolith application is selected for implementation, the reasoning being that the monolithic architecture is one of the most basic and traditional use cases in software development. This application combines several planning tools and permits the user to operate them from a single interface. The user can create a PDDL domain and problem, then solve the problem to generate a plan, and finally to validate that plan. Figure 5.1 shows a screenshot of the application's main menu UI (User Interface).



```
|-----------------------------------------------------------------------|
|  Index   Tool    Operation                                            |
|-----------------------------------------------------------------------|
|  1       pddl    Create/open Python script for PDDL domain/problem modeling  |
|  2       pddl    Execute Python script to generate PDDL domain/problem      |
|  3       pddl    Validate a PDDL domain                                |
|  4       pddl    Validate a PDDL problem                               |
|  5       FD      Generate a plan                                       |
|  6       FD      Full Fast Downward tool access                        |
|  7       -       View a plan                                           |
|  8       VAL     Validate a plan generated from PDDL                   |
|-----------------------------------------------------------------------|
|  <       -       Back                                                  |
|  0       -       Quit                                                  |
|-----------------------------------------------------------------------|
Enter index: █
```

Figure 5.1: Project 1: UI screenshot.

5.1 Tools

The three AI planning tools, which are integrated within the monolith application and their purposes are listed in Table 5.1.

Planner name	Purpose	Description
pddl	PDDL domain and problem generation and validation.	Section 4.4.1
Fast Downward	Plan generation from PDDL files.	Section 4.4.2
VAL	Plan validation with PDDL files.	Section 4.4.3

Table 5.1: Project 1: integrated planning tools.

Since every selected planning tool is a CLI tool, the decision is made to implement Project 1 in the same manner, using Bash scripting. Bash[1] is a command language freely available in many Linux distributions, fitting the "basic use case" theme of the monolith application.

The application deployment is realized through Docker[2], ensuring that the application can run in different environments. While the development is being performed in a Debian 11 64-bit environment, the resulting Docker image is tested in a Windows[3] 10 64-bit environment.

5.2 Scenario 1: domain and problem modeling

Scenario 1 consists of setting up the *pddl* tool, used both as a modeler and a validator, and implementing a wrapper using the Bash command language. Through a text-based UI (see Figure 5.1) a user can create a custom modeler Python script and then to run that script, generating a domain and a problem PDDL files. Additionally, the user has the option to validate the previously generated domain and problem files, by employing the validation functionality of the *pddl* tool.

To make it easier for the user to create a custom modeler script the user is provided with a stub file, containing an example of working modeling instructions. Revisiting the earlier created custom modeler scripts for any editing purposes is also possible.

Once the user has completed a modeler script, it can be executed by providing its name, resulting in generated PDDL domain and/or problem files, depending on the instructions in the script. Since the output of the Python script ends up in the terminal, the results cannot be simply saved as files. The output needs to first be captured and then parsed using regular expressions.

The secondary use of the *pddl* tool is domain/problem validation. After accessing this functionality through the application's UI, the user needs to provide the corresponding file name to initiate that file's validation.

5.3 Scenario 2: plan generation

For Scenario 2 a second planning tool – *Fast Downward* – is set up and integrated into the application. This tool adds the problem-solving functionality for plan generation.

When accessing that functionality, the user is first shown a list of existing PDDL domain and problem files to choose from. Then the user is prompted to input a name for the future plan file and to select a search parameter. With all the execution requirements satisfied, the tool attempts to generate a plan. If the execution fails, any leftover temporary files are cleaned up, otherwise a successfully produced plan is saved under the previously provided name.

Additionally, the user is also provided with the means to access *Fast Downward's* full functionality. Through this alternative access it is possible to input more options and parameters than described above.

[1]https://www.gnu.org/software/bash/

[2]https://www.docker.com

[3]https://www.microsoft.com/de-de/windows

5.4 Scenario 3: plan validation

Scenario 3 consists of setting up and integrating the third planning tool – *VAL* – and deploying the application by using Docker. Through *VAL* the application provides the ability to validate plans.

To execute the validation functionality the user needs to input the names of PDDL domain, problem, and plan files. All files which the application has access to are displayed to allow for easier selection. Once the validation process is completed, the user is informed whether a plan executes successfully, fails to execute or if a provided file contains an error.

For the application deployment via Docker, a Dockerfile is created which includes:

- Setting up a Debian environment.
- Installing dependencies for building and running the application, the *pddl* tool, the *Fast Downward* tool and the *VAL* tool.
- Cloning and building the three planning tools.
- Removing the no longer necessary build dependencies.
- Copying the application's source code and example files to the Docker image.
- Specifying the entry point.

5.5 Discussion

The following sections contain design decisions, necessary preparations and encountered issues of Project 1.

5.5.1 Bash scripting

Writing an application as a CLI tool using Bash requires both a proper environment and some experience with Bash scripting. Should a Linux environment be unavailable, due to one's workstation running a Windows or a Mac[4] OS, then setting up a VirtualBox instance with, e.g., Debian, Ubuntu[5], Arch[6], is sufficient.

As for using Bash for writing code, fortunately, there is plenty of documentation[7], tutorials and code examples on that matter available, with Bash being a popular tool with widespread use. Additionally, having even a basic understanding of Linux terminal commands is advantageous.

[4]https://www.apple.com/macos/ventura/
[5]https://ubuntu.com
[6]https://archlinux.org
[7]https://www.gnu.org/software/bash/manual

5.5.2 CLI planner output

As expected from a CLI tool, the integrated planning tools display their output in the terminal. As such output often contains progress and other superfluous information, extracting the necessary result, e.g., domain, problem, plan, can pose a challenge. That challenge can be mitigated if there is a way to alter the output, e.g., when writing a modeler Python script for the *pddl* tool it is possible to insert markers to be shown in the output. Being able to mark the beginning and the end of both the domain and the problem content makes parsing that output a non-issue.

However, other tools, like *VAL*, do not have a simple way of altering output and such a change would require updating the tool's source code. Depending on the difficulty of output parsing, such an endeavor may be worth considering.

5.5.3 Docker image

The utilization of Docker for application deployment requires expertise in engineering a Dockerfile, used to produce a Docker image. Should one lack experience in that area, the extensive documentation[8] should provide sufficient help with that issue.

When writing and executing the Dockerfile, one encountered issue was the larger than expected size of the resulting image, caused by some Dockerfile idiosyncrasies. For example, writing "RUN" instructions separately causes the Dockerfile to create additional layers in the image, requiring more disk space. Instead, one should attempt to put as many instructions as possible in a single command.[9]

Another reason for the increased Docker image size are the planning tools. As the application requires the tools' executables to employ the said tools, they are cloned and built for the image, as mentioned in the Section 5.4 on the preceding page. The *Fast Downward* tool, for example, requires close to 800 MB of space "out of the box", significantly inflating the image's size. To mitigate that problem, one would need to determine whether a planning tool can be somehow downsized, since only a part of its functionality may be required in the first place.

5.5.4 Docker persistent storage

Once the Docker image was up and running, one issue that became apparent was the lack of persistent storage for any newly created files. One solution is to utilize a directory stored locally in the image's current environment. This permits easy access to any files produced by the tool and allows manual insertion of new files in the directory. Another solution is to use the container of the Docker image as file storage. However, in that case, one needs to make sure that the container is not removed between application's uses, but instead reused each time.

[8]https://docs.docker.com/engine/reference/builder

[9]https://docs.docker.com/develop/develop-images/dockerfile_best-practices/#minimize-the-number-of-layers

5.6 Key observations

Employed technologies

The two technologies used for building and running the monolith application are Bash scripting and Docker. Even if one is lacking the experience in utilizing those technologies, because of their documentation and tutorials being widely available online, learning those technologies does not pose much of a challenge.

Planner integration

Integration of a CLI planning tool within an application is no different than using any other external executable. In the end, it is just a process that must be initiated, possibly with a specific input, and which then provides some form of output.

5.7 Key recommendations

Integration of multiple planners

When integrating multiple planning tools, which are supposed to be working with each other, it is important to ensure their compatibility. For example, even if employing only PDDL tools, they may be using different PDDL versions, limiting each other's capabilities.

Planner selection

Before settling on a specific planner, its other characteristics besides its main function should be investigated. The tool's size, its dependencies, the format of the required arguments, the output type, etc., can make a tool's integration difficult.

6 Project 2: Service-Oriented Architecture

Project 2 consists of building a loose-coupled (see Appendix A.1) application using the SOA (see Appendix A.2) architectural style. The main idea is to employ the same planners that were used in the first project, the monolith application (see Chapter 5), but this time as a distributed system. Afterwards, the system is extended with a new planning tool providing new functionality.

The utilized planners are integrated in services, which are built using the patterns described in "Enterprise Integration Patterns" [HW04]. The communication between the frontend and the services is routed through a message middleware using message queues, ensuring that it is asynchronous. The frontend application, the message middleware and the services are hosted in separate and independent environments.

6.1 Tools

The services containing the planning tools and the message middleware are implemented with the Java[1] programming language and Gradle[2] for build automation, using the IntelliJ[3] IDE (Integrated Development Environment) with the free student license[4]. The frontend application is implemented through the React[5] framework with the use of the TypeScript[6] programming language and npm[7] package manager. The application's GUI design is improved with a light use of CSS (Cascading Style Sheets)[8], and the work on the frontend is carried out in the WebStorm[9] IDE, also with the free student license. GitLab[10] is used for development and version control of all system components.

[1] https://www.java.com/en/
[2] https://gradle.org
[3] https://www.jetbrains.com/idea/
[4] https://www.jetbrains.com/community/education/#students
[5] https://react.dev
[6] https://www.typescriptlang.org
[7] https://www.npmjs.com
[8] https://developer.mozilla.org/en-US/docs/Learn/CSS
[9] https://www.jetbrains.com/webstorm/
[10] https://about.gitlab.com

The message communication between the services and the frontend functions via the Simple Queue Service (SQS)[11] of Amazon Web Services (AWS). This is a service for hosting message queues which allows for secure and asynchronous message exchange between systems or components[12]. The pricing scheme AWS Free Tier[13] includes one million requests over SQS each month free of charge[14].

For deployment, each service and the frontend are hosted in independent Amazon Elastic Compute Cloud (EC2)[15] instances. EC2 offers virtual computing environments with various preconfigured templates, persistent storage volumes, scaling and other features[16]. The AWS Free Tier includes 750 hours of instances runtime monthly free of charge[17].

Table 6.1 lists the four AI planning tools, which are integrated in the services of Project 2.

Planner name	Purpose	Description
pddl	PDDL domain and problem generation.	Section 4.4.1
Fast Downward	Plan generation from PDDL files.	Section 4.4.2
VAL	Plan validation with PDDL files.	Section 4.4.3
Lilotane	Plan generation from HDDL files.	Section 4.4.4

Table 6.1: Project 2: integrated planning tools.

6.2 Scenario 1

The first scenario of the second project consists of the implementation of multiple components: the frontend application, the message middleware and the solver service. Initially, the frontend application was planned to include the *pddl* tool, used as a modeler of PDDL domains and problems, to allow the user an easier and faster way to create them. But for reasons discussed in the following sections the modeling functionality was decoupled into a separate modeler service.

Scenario 1 also includes the creation of a message format and SQS message queues for communication, as well as the deployment of the implemented services and the frontend application on EC2 instances. Figure 6.1 shows the corresponding system diagram.

[11] https://aws.amazon.com/sqs/
[12] https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/welcome.html
[13] https://aws.amazon.com/free/
[14] https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-pricing.html
[15] https://aws.amazon.com/ec2
[16] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html
[17] https://aws.amazon.com/ec2/pricing

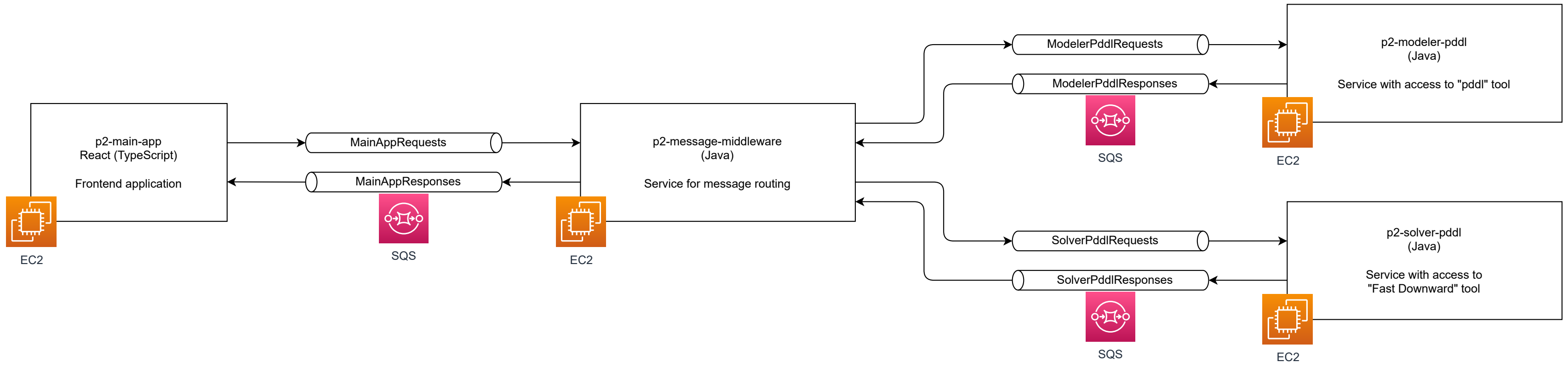


Figure 6.1: Project 2, scenario 1: system diagram.

6.2.1 Message format

The SQS message format consists of message attributes and a message body. Since all messages between the frontend application and the services are routed through the message middleware, custom message attributes are used to indicate which services should receive each message. Those attributes contain “true” or “false” values, but as the SQS message attribute data types do not support Booleans[18], those values are saved as strings. Additional attributes are created for the message ID (Identifier) and error occurrence indication. Table 6.2 lists the custom created message attributes.

Attribute Name	Type	Value
MESSAGE_ID	string	UUID
ERROR	string	true/false
GOTO_MODELER_PDDL	string	true/false
GOTO_SOLVER_PDDL	string	true/false

Table 6.2: Project 2, scenario 1: SQS message attributes.

The message body is a JSON (JavaScript Object Notation)[19] containing the following fields: “ERROR_MESSAGE” in case of an error occurrence, “MODELER_PDDL_SCRIPT” for a Python modeler script, “DOMAIN” for a PDDL domain, “PROBLEM” for a PDDL problem, and “PLAN” for a generated plan. All information is encoded as Base64[20] strings. The custom message body is shown in Listing 6.1.

```json
{
  "ERROR_MESSAGE": "<Base64 string>",
  "MODELER_PDDL_SCRIPT": "<Base64 string>",
  "DOMAIN": "<Base64 string>",
  "PROBLEM": "<Base64 string>",
  "PLAN": "<Base64 string>"
}
```

Listing 6.1: SQS message body, scenario 1.

6.2.2 Frontend application

The frontend application consists of a GUI with text fields for user input and the SQS communication logic for message creation and processing. The GUI design is kept simplistic, employing a small measure of CSS for a few visual improvements. A screenshot of the GUI of the application is shown in Figure 6.2.

[18]https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-message-metadata.html
[19]https://www.json.org/json-en.html
[20]https://developer.mozilla.org/en-US/docs/Glossary/Base64

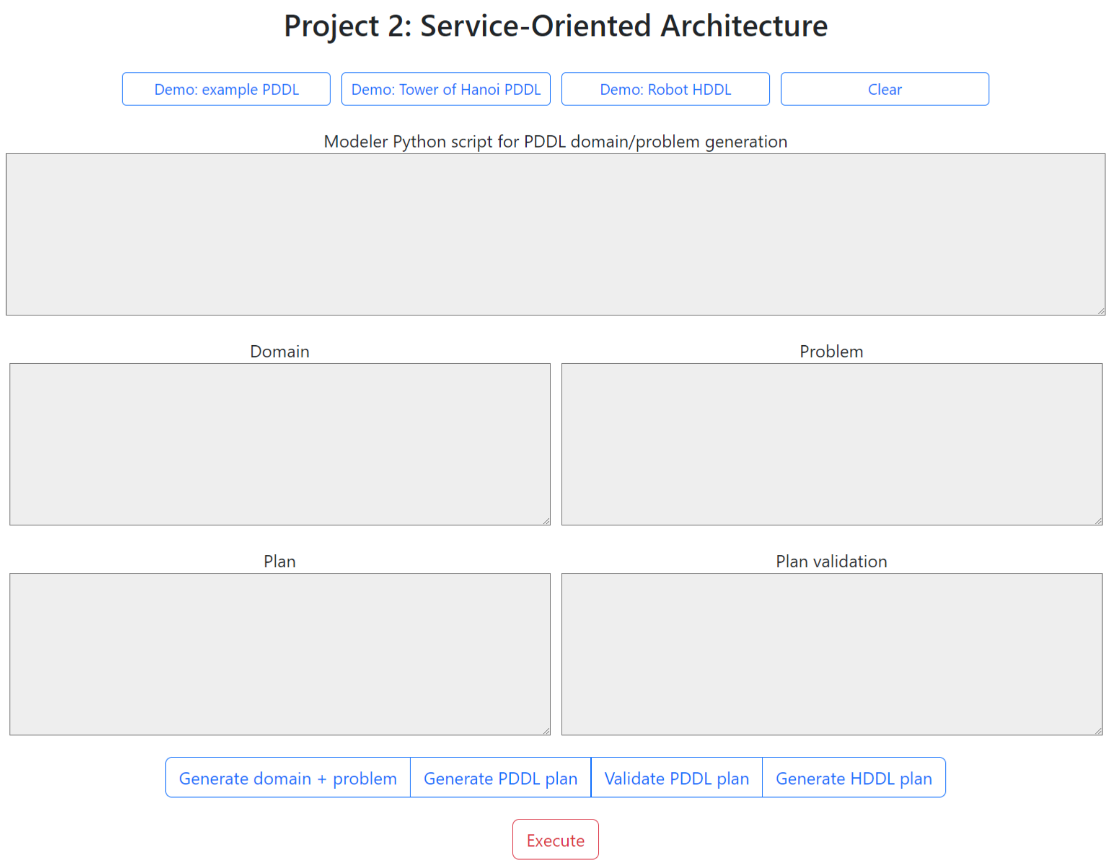

Project 2: Service-Oriented Architecture
Demo: example PDDL
Demo: Tower of Hanoi PDDL
Demo: Robot HDDL
Clear
Modeler Python script for PDDL domain/problem generation
Domain
Problem
Plan
Plan validation
Generate domain + problem
Generate PDDL plan
Validate PDDL plan
Generate HDDL plan
Execute

Figure 6.2: Project 2: GUI screenshot.

Through the GUI the user has the ability to select services to which the frontend application will send messages, which can be either one or multiple services. In the former case, for example, the user might want to just generate a PDDL domain and problem from a modeler Python script, or to just generate a plan from the provided PDDL domain and problem. Meanwhile in the latter case the user might want to engage the whole chain of services with one action, for example: creating a PDDL domain and problem and then immediately generating a plan from them, before the frontend receives a response message.

Also included in the GUI are a few prepared inputs for demonstration purposes:

- A Python modeler script stub, its PDDL domain and problem, the generated plan and plan validation.
- A Python modeler script, its PDDL domain (see Appendix B.1) and problem (see Appendix B.2) for the “Tower of Hanoi” puzzle, a generated plan and plan validation.
- An HDDL domain (see Appendix B.3) and problem (see Appendix B.4) for a delivery robot, and a generated plan.

The SQS messages for queue communication are constructed based on user input. Sending and receiving messages works via the two custom-created SQS queues, one for request and the other for response messages.

The sequence for frontend message communication is as follows: first, the user needs to provide an input and select at least one service. Then, a unique ID which is required for response message identification – a UUID (Universally Unique Identifier) – is generated. Of course, SQS provides its own message identifier mechanism[21], but since messages are consumed and recreated multiple times throughout the communication, using that ID here is not viable. The information of the selected service and the UUID are converted into the message attribute values, meanwhile the provided input is encoded as Base64 string/s, from which the message body is built, thus resulting in a new SQS message. This message is then put into the request queue and is expected to be received by the message middleware.

Afterwards, the frontend application begins to poll the response queue for messages with corresponding UUID. Once such a message is found, its payload is extracted, decoded, and then displayed for the user in the GUI. Only then is the processed response message deleted from the response queue to ensure that messages are not lost, e.g., in case of an application crash.

The application is deployed on a custom created EC2 instance running a Debian 11 64-bit environment. The instance contains AWS credentials to permit SQS access, and application dependencies: Node.js[22] and npm[23]. Once the application build is uploaded to the instance, the PM2[24] process manager is used to run it in background permanently as a daemon process.

6.2.3 PDDL modeler service

The PDDL modeler service contains access to the planning tool *pddl*. Its purpose is to use a Python modeler script, provided through an SQS message, to generate a domain and a problem PDDL.

As mentioned previously, the planning tool was initially supposed to be a part of the frontend application to give the user a quicker access to its modeling capabilities without the SQS communication waiting times. But instead, the tool has been exported into a separate service, with the main reason for this being that the React application would be running in the browser and thus would not be able to access the terminal to run the CLI tool. Additionally, the separation also makes sense in terms of producing cleaner code and results in an independent service, which can be reused or scaled if necessary.

For communication the service employs SQS, using the SQS Software Development Kit (SDK), and two SQS queues, one for requests the the other for responses. The SQS request queue is continuously polled for messages. Once a message is received, it is processed: the message attributes and the message JSON body are extracted, the Base64 string containing the Python modeler script is

[21]https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-queue-message-identifiers.html
[22]https://nodejs.org/en
[23]https://www.npmjs.com
[24]https://pm2.keymetrics.io

decoded and saved locally as a ".py" file. This script file is then executed, producing a domain and a problem PDDL as output. Since the output is provided only in the terminal, the service has to catch and parse it.

Afterwards, a new SQS message is created using the received message attributes and body JSON. The request message attributes need to be preserved, so that the subsequent message flow is not disrupted. The one exception is the "GOTO_MODELER_PDDL" attribute, which is consumed, meaning its value is set to "false", thus preventing the message being sent back to the PDDL modeler service. Meanwhile, the body is updated with the generated domain and problem, encoded as Base64 strings. The newly created response SQS message is then put into the response SQS queue, to be received by the message middleware. At this point the request message can be safely deleted from the request queue and the remaining local file, the Python modeler script, cleaned up.

In case of an error at any point in the process after receiving a request message, an error message is created. It contains the request message ID and the error attribute with the value "true", but all other attributes are set to "false" to prevent the message being sent to any other services and ensure it is sent to the fronted application instead. The exception message is encoded as a Base64 string and saved in the JSON body.

As with the frontend application, to host the PDDL modeler service an EC2 instance is created, which runs a Debian 11 64-bit environment. The instance contains credentials for SQS access, a Java Development Kit (JDK) for the service itself and the *pddl* tool dependency pip[25]. Once the dependencies are taken care of, the *pddl* tool is cloned and built, and the service Java Archive (JAR) is uploaded to the EC2 instance. The service is then run as a background daemon process using the PM2 process manager.

6.2.4 PDDL solver service

The purpose of the PDDL solver service is to generate a plan from the provided domain and problem PDDL files. It contains access to the *Fast Downward* planner.

Same as the PDDL modeler service, this service uses SQS for communication via the SQS SDK and two custom SQS message queues. It polls the request queue, then processes received messages by extracting the payload, a domain and a problem PDDL, and then decodes and saves them locally as ".pddl" files. The files are necessary, because the *Fast Downward* CLI solver requires files as input arguments. Using the saved files, the tool is executed, generating a plan which the tool automatically saves locally as a new file.

An SQS response message is constructed with its JSON body containing the newly generated plan. The attribute "GOTO_SOLVER_PDDL" is consumed while the other message attributes are preserved. Once the response message is put into the SQS response queue, the request message is safely deleted from the request queue. The local files: domain, problem, and plan are no longer necessary and are deleted as well.

[25]https://pypi.org/project/pip/

Error handling in the PDDL solver service functions the same way as for the PDDL modeler service, as does deployment: an EC2 instance with a Debian 11 environment. The only difference in this case are the dependencies required by *Fast Downward*.

6.2.5 Message middleware

The message middleware is a service for routing all messages between the frontend application and the other services. Similar to the other services, it uses an SQS SDK for message polling, sending and deletion. The message middleware has no own custom SQS queues, instead it polls the response queues of other services and sends messages to their request queues. The communication with the frontend application is slightly different, here the middleware polls the request queue and sends messages to the response queue.

To decide where a received message should be sent, the middleware acts as a message router employing the Content-Based Router pattern[26] [HW04]. The message recipient is determined by the message “GOTO_[...]” attributes.

In the case that the values of multiple attributes are set to “true”, meaning that multiple services are intended to receive the message, the message middleware needs to regulate with which service to communicate first. For that purpose, it employs a service priority ranking, which is determined logically, e.g., domain and problem generation should come before plan generation, which requires those domain and problem.

Should a message be received with its “ERROR” attribute set to “true”, it is returned directly to the frontend application. To ensure this behavior, all other attributes indicating which services were supposed to receive that message are ignored.

The message middleware is hosted in its own EC2 instance with a Debian 11 64-bit environment. The only required dependency is a JDK for the service JAR and the service process is daemonized through the PM2 manager.

6.3 Scenario 2

Scenario 2 consists of the implementation and integration of a new service, which grants access to the plan validator tool, *VAL*. This scenario’s purpose is to discover challenges occurring during integration of a new service into the established system of planning tools. The updated system diagram is shown in Figure 6.3.

[26]https://www.enterpriseintegrationpatterns.com/patterns/messaging/ContentBasedRouter.html

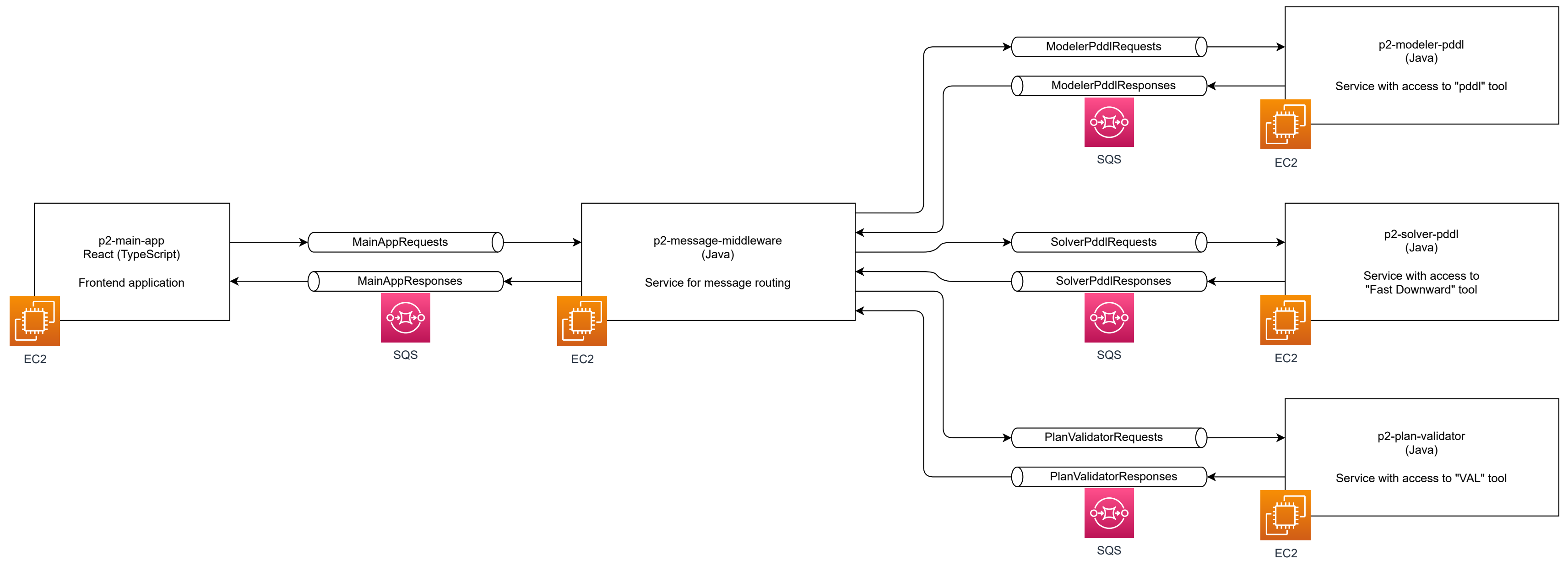


Figure 6.3: Project 2, scenario 2: system diagram.

6.3.1 Message format

The message format used for SQS communication is updated to accommodate for the new plan validation service. A new message attribute, “GOTO_PLAN_VALIDATOR”, is introduced, used to signify whether the message needs to be sent to that service. Same as the other attributes with a similar purpose, it holds a “true”/“false” value of type string. Table 6.3 displays the updated message attributes.

Attribute Name	Type	Value
MESSAGE_ID	string	UUID
ERROR	string	true/false
GOTO_MODELER_PDDL	string	true/false
GOTO_SOLVER_PDDL	string	true/false
GOTO_PLAN_VALIDATOR	string	true/false

Table 6.3: Project 2, scenario 2: SQS message attributes.

Since the message payload now needs to include plan validation data, the JSON body is extended with a new field, “PLAN_VALIDATION”. It, too, holds information encoded as a Base64 string. The updated body format is displayed in Listing 6.2.

```
{
  "ERROR_MESSAGE": "<Base64 string>",
  "MODELER_PDDL_SCRIPT": "<Base64 string>",
  "DOMAIN": "<Base64 string>",
  "PROBLEM": "<Base64 string>",
  "PLAN": "<Base64 string",
  "PLAN_VALIDATION": "<Base64 string>"
}
```

Listing 6.2: SQS message body, scenario 2.

Once the message format changes are determined, the messaging logic must be updated in the frontend application and in all services. The service priority ranking in the message middleware, which determines the order of services when routing messages intended for multiple services, is updated as well. As the plan evaluation tool requires a PDDL domain, a PDDL problem and the resulting plan, the PDDL modeler service and the PDDL solver service from Scenario 1 need to have a higher priority. Thus, the new service is assigned the lowest priority.

6.3.2 Plan validation service

The plan validation service is implemented using the Java programming language, same as the other services. It uses two new SQS queues for communication: polling, sending, and deleting messages in the same manner.

The PDDL plan validator requires a plan, as well as the corresponding domain and problem PDDL, to determine whether that plan is a valid solution. Those are provided by the SQS message payload, they need to be extracted, decoded and saved locally as files, since the validator CLI tool expects files as input arguments. After the tool's execution, the terminal output is parsed, and the resulting validation information is saved in the JSON body of the SQS response message.

Afterwards, the plan validation service proceeds in the same manner as the other services, sending the response message and cleaning up both the request message and the locally saved files: domain, problem and plan. The consumed message attribute in this case is "GOTO_PLAN_VALIDATOR".

Error handling and service deployment is realized as stated in the sections describing the other services. Setting up the *VAL* tool, however, has to proceed differently, as the tool's build scripts refuse to build the executable in the EC2 environment, for reasons which could not be determined. Because of this, the tool is built in a test environment – a VirtualBox instance, also running Debian 11 64-bit – and the executables are then copied to the EC2 instance.

6.4 Scenario 3

Scenario 3 consists of several tasks. First, the extraction of all SQS communication logic required by a service into a separate messaging endpoint, as per the Message Endpoint[27] [HW04] pattern. The endpoint also employs the Messaging Gateway[28] [HW04] pattern to encapsulate the SQS messaging-specific method calls. Additionally, the conversion of the objects into the message format is encapsulated using the Messaging Mapper[29] [HW04] pattern. The endpoint is implemented as a Java library.

Second, the implementation and integration of a new service, the HDDL solver service, into the established system. This service grants access to the planning tool *Lilotane* and is required to make use of the new message endpoint. An HTN tool is chosen for the purpose of discovering challenges which may occur when integrating a tool which requires ".hddl" files into a system previously working only with ".pddl" files.

For the final task, the message endpoint is integrated into the other services: the message middleware, the PDDL modeler service, the PDDL solver service and the plan validator service. The updated system diagram is shown in Figure 6.4.

[27]https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageEndpoint.html
[28]https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessagingGateway.html
[29]https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessagingMapper.html

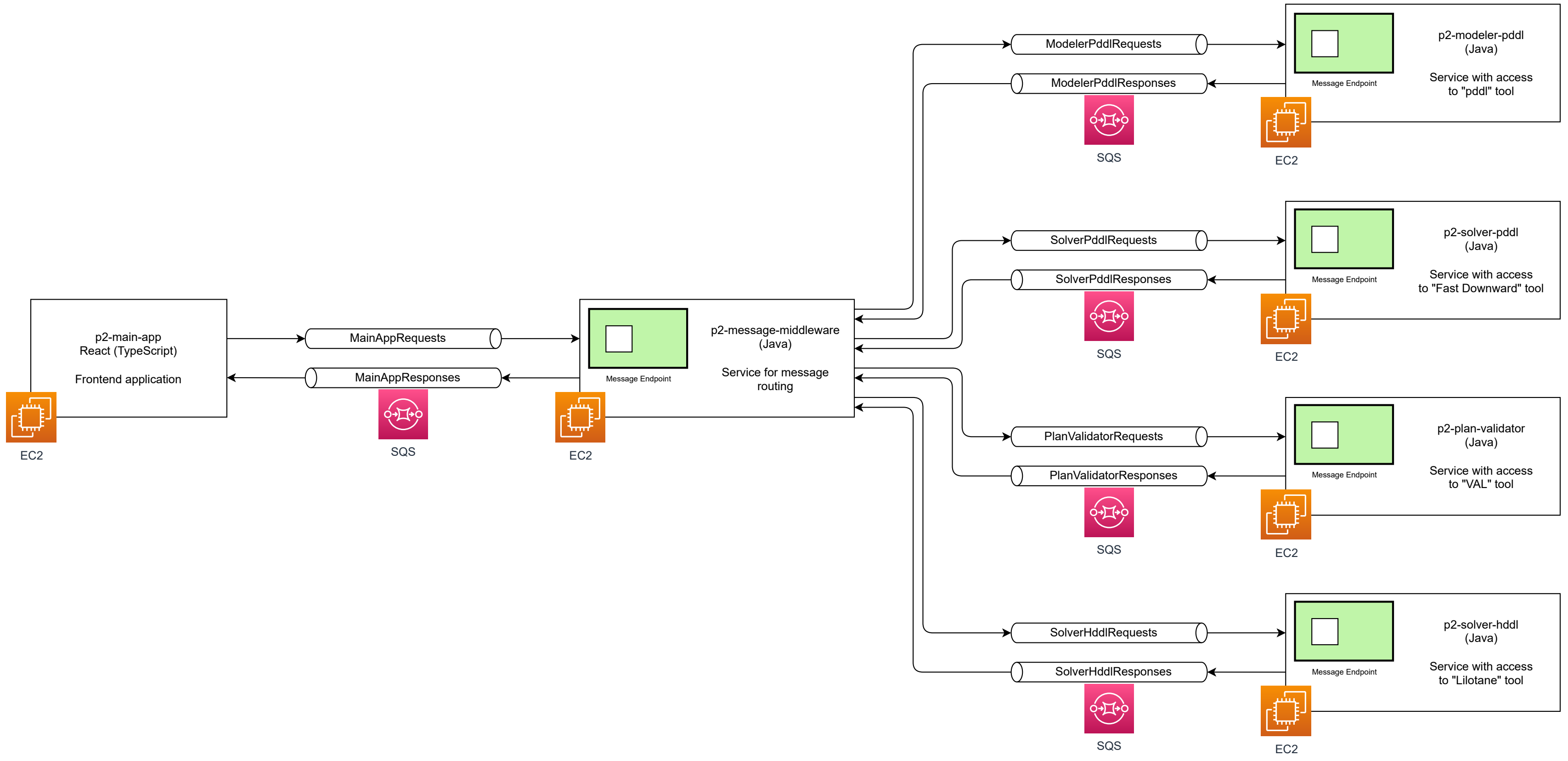


Figure 6.4: Project 2, scenario 3: system diagram.

6.4.1 Message format

To accommodate for the new service, the message format is updated with a new attribute – “GOTO_SOLVER_HDDL”. It is used by the message middleware to determine whether the message needs to be routed to the HDDL solver service. The updated and final version of the message attributes is shown in Table 6.4.

Attribute Name	Type	Value
MESSAGE_ID	string	UUID
ERROR	string	true/false
GOTO_MODELER_PDDL	string	true/false
GOTO_SOLVER_PDDL	string	true/false
GOTO_PLAN_VALIDATOR	string	true/false
GOTO_SOLVER_HDDL	string	true/false

Table 6.4: Project 2, scenario 3: SQS message attributes.

As for the message body, it is kept the same as in Listing 6.2. Since the new HDDL solver service requires a domain and a problem file, same as the PDDL solver service, no new fields in the body JSON are necessary. Instead, the already established “DOMAIN” and “PROBLEM” fields are reused.

6.4.2 Message endpoint

The message endpoint is a library implemented with the Java programming language. It contains all the communication code that the services require to make use of the SQS queues.

Utilizing the SQS SDK, the endpoint builds an SQS client and enables message polling, sending, and deleting. It also handles both the payload extraction of the received request messages and the construction of the attributes and the body of the response messages. The error handling process and the construction of SQS error messages is implemented in the message endpoint as well.

Once implemented, the message endpoint library is added to every service, as can be seen in Figure 6.4. Although making changes to the already working and tested software requires additional time, both for the implementation and more testing, it results in a significant reduction of code, which previously was duplicated throughout the services.

6.4.3 HDDL solver service

The final service of Project 2, the HDDL solver service, is implemented using Java, same as the other services. Similarly, it employs a custom SQS queue for requests and another for responses. Unlike the other services, it makes use of the message endpoint library from the beginning. That means the service relies on the endpoint for all its SQS communication needs: handling request, response and error messages.

Once a request message is received and processed, its payload, containing an HDDL domain and problem, is saved locally. The *Lilotane* planning tool, similar to the other CLI planning tools, requires its input arguments to be files. Afterwards, the resulting terminal output needs parsing to extract the generated plan.

With a plan available, the message endpoint takes over again to create the response message and consume the "GOTO_HDDL_SOLVER" message attribute. After the response message is put into the SQS response queue, the local files are cleaned up and the HDDL solver service is ready to process the next request message.

The deployment of the server proceeds in the same manner as the other services. It is hosted on a custom EC2 instance in a Debian 11 environment, containing the *Lilotane* tool executable and the service JAR file managed by PM2.

6.5 Discussion

Section 6.5 contains the reasoning behind design choices, implementation decisions, encountered difficulties, and experiences the were made during the realization of Project 2.

6.5.1 Amazon Web Services

Before work could begin on the SOA-based Project 2, it was necessary to select a fitting infrastructure for the application's needs. With the Amazon Web Services offering solutions for queue-based communication and application hosting, the decision was made to make use of it.

The following are a few additional reasons in favor of that decision:

- Popularity: similarly to Project 1, one of the goals was to carry out any implementation using only freely available, proven tools and means. With AWS being available since 2006[30] and being used by multiple companies[31], that goal was satisfied.
- Documentation: solutions, their features, and the specifics of their utilization are extensively documented[32] with code examples in multiple programming languages[33].
- Pricing: when planning Project 2, another goal was to avoid any hosting or service expenses. As the AWS Free Tier offers 750 hours of EC2 usage and up to 1,000,000 SQS requests each month free of charge, this goal could be achieved. Additionally, the transparency of the pricing scheme also spoke favorably for AWS.
- Scaling: although the application is not intended to be offered to any users, one of the goals was for it to be potentially scalable. Hosting the frontend application and its services on EC2 instances makes that possible[34].

[30]https://web.archive.org/web/20121005123855/http://aws.amazon.com/about-aws/
[31]https://aws.amazon.com/ec2/customers/
[32]https://docs.aws.amazon.com
[33]https://github.com/awsdocs/aws-doc-sdk-examples
[34]https://aws.amazon.com/ec2/autoscaling/

- Experience: getting familiar with a large service like AWS requires a significant time investment. However, since some experience with the AWS solutions was already acquired from the lecture “Cloud Computing: Concepts and Technologies”[35], that time investment was reduced.

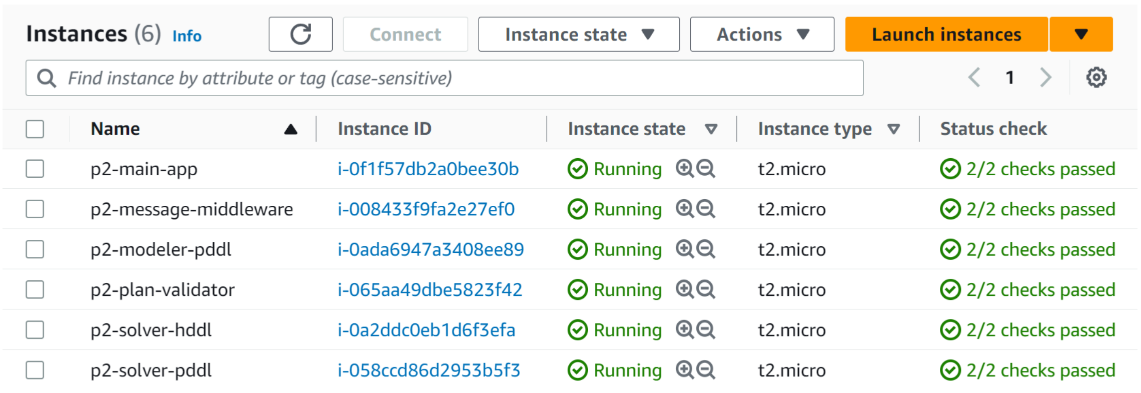
Once the technologies for the project were selected, it was necessary to acquire both theoretical knowledge and practical skills with them. That means reading the AWS SQS and the AWS EC2 documentation, learning the capabilities and requirements of those services and studying the available code examples in the project specific programming language.

After accumulating enough theoretical background knowledge, it needed to be put into practice. That includes creating a new AWS account, getting familiar with the AWS console, figuring out security roles and acquiring security credentials, setting up EC2 instances with selected environments and SSH (Secure Shell Protocol) access, setting up SQS queues and creating some SQS messages for testing, and more.

6.5.2 AWS billing

As mentioned previously, with the AWS Free Tier it is possible to make use of multiple solutions without any monetary expenses. It is important, however, to remember the limits of the free offer.

In particular, the 750 hours of free monthly use of EC2, which translates to just over 31 days, is enough to run one EC2 instance permanently for one month. But when working on Project 2 it was not considered that multiple instances are required: one for the frontend application and one for each service, as showcased in Figure 6.5. And since those instances were not stopped between tests, the 750 hours limit was reached within a week. This led to incurred costs of approximately $10.



Name	Instance ID	Instance state	Instance type	Status check
p2-main-app	i-0f1f57db2a0bee30b	Running	t2.micro	2/2 checks passed
p2-message-middleware	i-008433f9fa2e27ef0	Running	t2.micro	2/2 checks passed
p2-modeler-pddl	i-0ada6947a3408ee89	Running	t2.micro	2/2 checks passed
p2-plan-validator	i-065aa49dbe5823f42	Running	t2.micro	2/2 checks passed
p2-solver-hddl	i-0a2ddc0eb1d6f3efa	Running	t2.micro	2/2 checks passed
p2-solver-pddl	i-058ccd86d2953b5f3	Running	t2.micro	2/2 checks passed

Figure 6.5: Project 2: EC2 instances screenshot.

Alternatively, the amount of SQS requests accumulated during testing was only about 25,000 in total, which is still well under the 1,000,000 requests limit. However, it is strongly advisable to not lose track of such limits when working with AWS solutions.

[35]https://www.iaas.uni-stuttgart.de/en/teaching/lectures/2021_ws/cc/

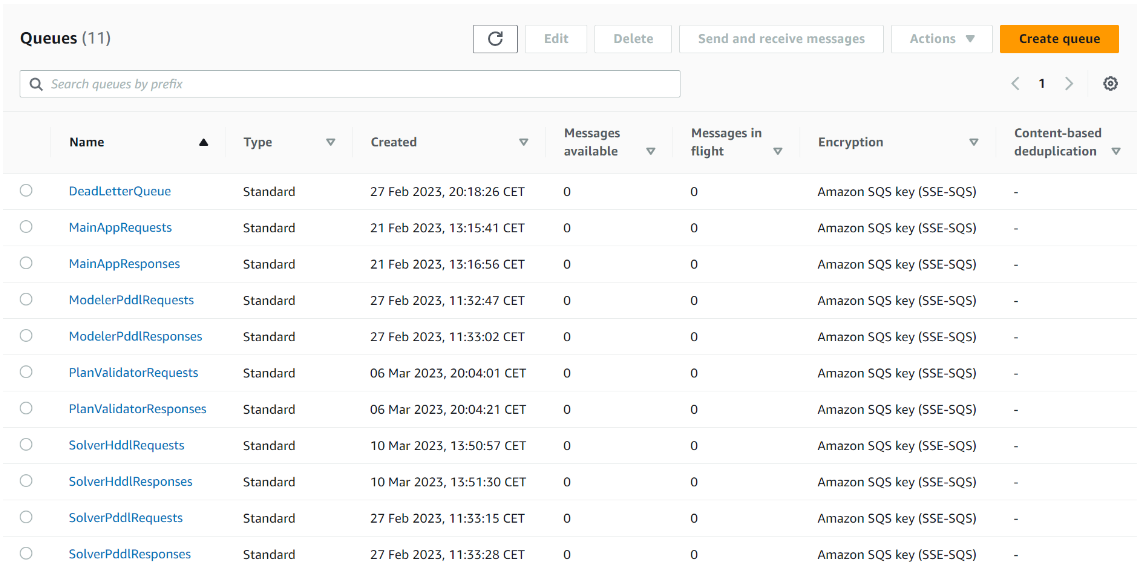
6.5.3 AWS security

Services running in EC2 instances require security credentials to send, receive and delete messages to/from SQS queues. To achieve that, an Identity and Access Management (IAM) user with limited access, namely just the SQS access, was created. Those user credentials were then uploaded to each EC2 instance, permitting their SQS access.

While there may be better security practices[36], a deeper dive into the AWS security peculiarities was deemed to be outside the project scope. Additionally, such research would have pushed the project over its allotted time budget.

6.5.4 Dead-Letter Queue

One useful tool both in testing and production when using SQS is the Dead-Letter Queue (DLQ)[37]. The way it works is that a queue is designed a DLQ and the other queues are provided with its address. SQS allows specifying the maximum number of times a message can be polled, and once this amount is exceeded the message is automatically transported from its original queue to the DLQ. There it remains for a certain period, which can be specified in the queue's settings, before being purged. The queues created for Project 2, including the DLQ, can be seen in Figure 6.6.



Name	Type	Created	Messages available	Messages in flight	Encryption	Content-based deduplication
DeadLetterQueue	Standard	27 Feb 2023, 20:18:26 CET	0	0	Amazon SQS key (SSE-SQS)	-
MainAppRequests	Standard	21 Feb 2023, 13:15:41 CET	0	0	Amazon SQS key (SSE-SQS)	-
MainAppResponses	Standard	21 Feb 2023, 13:16:56 CET	0	0	Amazon SQS key (SSE-SQS)	-
ModelerPddlRequests	Standard	27 Feb 2023, 11:32:47 CET	0	0	Amazon SQS key (SSE-SQS)	-
ModelerPddlResponses	Standard	27 Feb 2023, 11:33:02 CET	0	0	Amazon SQS key (SSE-SQS)	-
PlanValidatorRequests	Standard	06 Mar 2023, 20:04:01 CET	0	0	Amazon SQS key (SSE-SQS)	-
PlanValidatorResponses	Standard	06 Mar 2023, 20:04:21 CET	0	0	Amazon SQS key (SSE-SQS)	-
SolverHddlRequests	Standard	10 Mar 2023, 13:50:57 CET	0	0	Amazon SQS key (SSE-SQS)	-
SolverHddlResponses	Standard	10 Mar 2023, 13:51:30 CET	0	0	Amazon SQS key (SSE-SQS)	-
SolverPddlRequests	Standard	27 Feb 2023, 11:33:15 CET	0	0	Amazon SQS key (SSE-SQS)	-
SolverPddlResponses	Standard	27 Feb 2023, 11:33:28 CET	0	0	Amazon SQS key (SSE-SQS)	-

Figure 6.6: Project 2: SQS queues screenshot.

The DLQ ensures that when a message cannot be processed by a recipient, that message does not end up clogging the queue, being unnecessarily polled, and attempted to be processed repeatedly. Simultaneously, transporting the erroneous message to a special queue, instead of outright deleting it, preserves it for debugging purposes.

[36]https://docs.aws.amazon.com/IAM/latest/UserGuide/best-practices.html
[37]https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-dead-letter-queues.html

However, it is important to make sure not to remove messages from the normal message flow too early. When multiple consumers poll a queue for messages, SQS employes the Competing Consumers[38] [HW04] pattern. As an example, that means that there could be multiple instances of the frontend application, all polling the same queue, awaiting a response message with a specific UUID. This can lead to a message being received and rejected by several application instances, reaching its polling limit and ending up in the DLQ, without ever being polled by the one instance which possesses its UUID.

Another case, when a valid response message may end up in the DLQ, is if the frontend application crashes. Since the UUID is not preserved, e.g., in a database, it would be lost. This means that when the frontend application comes back online, it can no longer retrieve the response, as it no longer possesses the response's UUID.

6.5.5 Message polling

Continuous message polling produces a lot of unnecessary requests which return nothing if there are few messages present in the queues. Not only does this waste processing power, it also counts towards the SQS request limit of the AWS Free Tier.

A better solution for a system with fewer messages would be to employ the Event-Driven Consumer[39] [HW04] pattern instead of the Polling Consumer[40] [HW04] pattern. However, the SQS documentation does not mention any event-driven solutions, instead, the answer would be to employ Amazon Simple Notification Service (SNS)[41] or Amazon EventBridge[42].

6.5.6 Error handling

There are multiple types of errors a service can encounter under workload. Errors can occur during message handling, e.g., when polling/processing/creating/sending/deleting messages, or during the execution of the planning tool integrated in that service, e.g., when providing input arguments or parsing the tool's output. But whichever is the case, the question arises how to handle those errors.

In terms of the Project 2 application, it has been elected to cancel the current flow of messages in case of an error and return an error message to the frontend application. For example, a message could be intended to be sent to the solver service first, to generate a plan, and then sent to a validation service, to make sure that the plan is valid. If an error occurs in the solver service, then it makes sense to abort this message flow instead of proceeding to the validation service, because there is no plan to validate.

[38] https://www.enterpriseintegrationpatterns.com/patterns/messaging/CompetingConsumers.html

[39] https://www.enterpriseintegrationpatterns.com/patterns/messaging/EventDrivenConsumer.html

[40] https://www.enterpriseintegrationpatterns.com/patterns/messaging/PollingConsumer.html

[41] https://aws.amazon.com/sns/

[42] https://aws.amazon.com/eventbridge/

However, in another system, e.g., with multiple different solver services, aborting the message flow could be counterproductive. In such a scenario, a message could be intended to go through multiple solver services, generating several (different) plans, before proceeding to a validation service. This would require a more advanced mechanism of error type recognition, to keep track of each failed solver service and to stop the message flow only if all have failed.

It gets even more complicated when one considers that the previously mentioned solver services can fail not due to the integrated planner, but because of some unrelated internal error. In this case it might make sense to let a service attempt to process a message more than once before moving on. And since each service is meant to be stateless, all that error information would need to be contained within the message, either in the attributes or in the payload, or in a database.

6.5.7 CLI planner output

One difficulty which became apparent when working with CLI planning tools was the necessity to process their output, as mentioned previously in Section 5.5.2 on page 40. While such tools like *Fast Downward* save their output in a new file, making it simple to access and process it, others do not make it so easy. The output of such tools is only accessible directly through the terminal, meaning it needs to be caught and parsed first. Unfortunately, building a reliable parser can be difficult, as not all possible output cases are known and must be discovered manually.

As an alternative to output parsing, one may consider updating the planning tool to change its output type. While this would involve accessing and altering the tool's source code, it would increase the reliability of the service which provides access to that planning tool.

6.5.8 Loose coupling of components

As mentioned in the description of Project 2, the system is meant to be loosely coupled (see Appendix A.1 on page 89). The following points describe how the loose coupling of components has been handled.

- Reference Autonomy: neither the frontend application nor the services know the addresses of their communication partners or are aware where they are hosted. The only known addresses are those of the SQS queues, thus fulfilling the autonomy aspect's requirement.
- Time Autonomy: this autonomy aspect is respected, since the communication through messaging is asynchronous. The system components poll SQS queues and process messages at their own pace, meaning that they do not need to be online at the same time.
- Platform Autonomy: the frontend application is implemented with TypeScript, whereas the services are implemented with Java. This fact, however, does not influence communication in any way, leading to platform autonomy.
- Format Autonomy: although this aspect requires the system components to use different formats of exchanged data, the decision was made to utilize only one message format instead. Because of this, the format autonomy aspect is not respected in this project, resulting in a tighter coupling of components. The reasons that led to this decision are explained in the next section.

6.5.9 Single message format

One of the features implemented in Project 2 is the ability to communicate with a chain of services by initiating only a single request. Those services are selected with the toggle buttons in the frontend's GUI and the request is initiated with the "Execute" button. Having each service use the same message format containing a predetermined set of attributes and body fields is a simple mechanism to help messaging multiple services in a row. Without it, tracking which services are to take part in a communication sequence would have to be done in a different manner than saving that information in the message attributes, e.g., by coupling the message middleware with a database and storing the information there.

Another reason for using a single message format is that each component of the SOA application was developed by a single developer. In a real-world scenario, a system may be worked on by multiple developers or even by multiple teams, meaning that components can be in development independently from each other. This leads to the source code of those components being unknown or inaccessible to other developers. In this case, it is unrealistic to expect every component to use the same message format and then to be able to quickly adapt to any changes. One solution for such an issue is to implement message translators using the Message Translator[43] [HW04] pattern. However, since there was only one developer working on the Project 2 application, having access to the source code of each component would have made introducing message translators just unnecessarily increase the overall system complexity.

Using the single message format approach instead of message translators means having to add new attributes and/or body fields to the message format for each new service, as mentioned in Scenario 2 and Scenario 3 of Project 2. This also means that whenever a new service is added to the system, the code handling the communication in each previous service and the frontend application requires an update. But it is worth noting that utilizing message translators would not necessarily reduce that workload. Introducing a new service would mean that the translators need to be updated instead of the services, but in the end, updating is required in either case. The amount of work needed for updating the services was mitigated through the use of the message endpoint library, since all communication code is contained within.

The final reason in favor of a single message format is the reduced number of messages that need to be created during one communication sequence. As can be observed in the example in Figure 6.7, a communication sequence is initiated with the intent to first produce domain and problem files, then a plan and then to validate said plan. For that purpose, after receiving a request from the frontend application, the message middleware needs to communicate with three services: the modeler service, the solver service and the validation service, before returning a response to the frontend application. This results in $2n + 2$ number of messages for n services.

In comparison to that number, the implementation of message translators for every service would possibly require the creation of additional SQS queues to decouple the translators from the services. This would lead to two more queues being located between each translator and its corresponding service. Those additional queues would result in $4n + 2$ number of messages for n services, increasing the overall AWS SQS costs and the communication latency.

[43]https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageTranslator.html

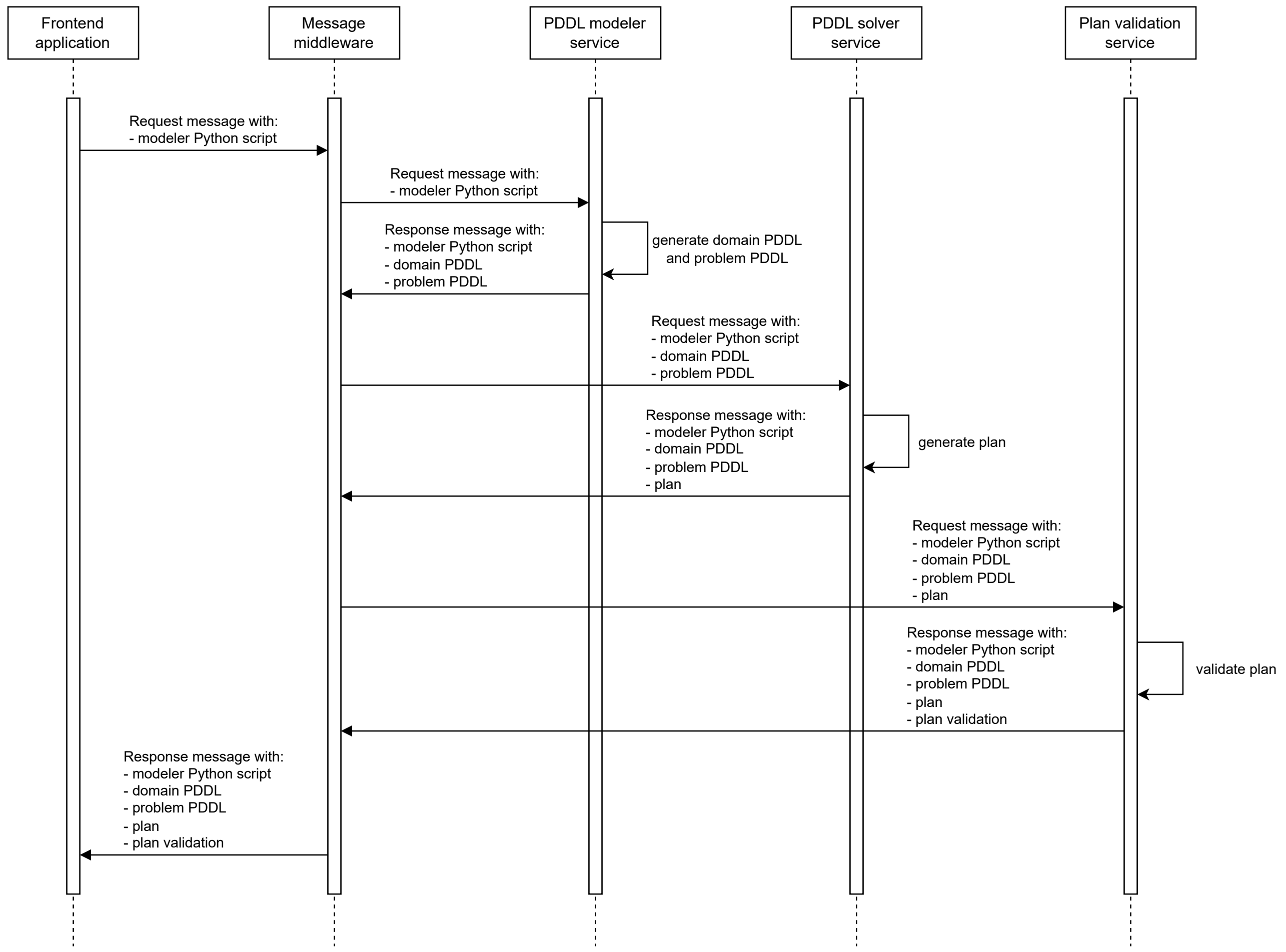


Figure 6.7: Project 2: communication sequence example.

6.5.10 Code changes

In Scenario 3, the implementation of the message endpoint library and, consequently, the need to update the existing services with it led to many code changes. Those changes resulted in code quality improvements, like the extraction of all communication code into a separate component and the reduction of code duplication. However, they also required a significant rework of the software, which was already functioning and tested, potentially breaking some components.

Several days were spent on error fixing and re-testing, making sure the updated services functioned as well as previously. Such effort and time investment can be avoided if the changes are instead part of the initial system design. While this may be an obvious recommendation, it is important to remember to give the system design plenty of thought from the beginning.

6.5.11 Deployment on EC2

During the deployment of services with planners in EC2 instances it was necessary to provide the corresponding planning tool executables. One possibility is to clone the planner and build it directly in the instance, another is to build the planner in another environment and then to upload the executable to the instance.

The first approach, although simpler, requires installing build dependencies in the EC2 instance. Additionally, the planning tool may simply refuse to build, as mentioned previously in Section 6.3.2 on page 52. As a reminder, the issue was that the *VAL* planning tool would fail to build within the EC2 instance and had to be built separately, leading to the second approach.

Building a planner executable separately requires an environment identical to the EC2 environment. Setting up a VirtualBox instance running Debian 11 64-bit allowed building the necessary *VAL* executable. Another benefit of this additional effort was that the installation of any build dependencies on the EC2 instance could be omitted.

6.5.12 Testing

The importance of testing is well known and doesn't need advocating. But when dealing with AWS it is highly advisable to thoroughly test one's application before and after deploying it in an EC2 environment.

Testing the software in a VirtualBox instance with the same environment as the future EC2 instance, as mentioned in the previous section, will prevent unnecessarily using up the resources of the AWS Free Tier. Additionally, testing locally can save a lot of time, which would otherwise be wasted, e.g., on uploading a JAR file to the EC2 instance, testing it, applying a fix in the source code, and then having to generate a new JAR and upload it to the instance again.

Inversely, just because the software functions in its test environment does not necessarily mean it will do so in its production environment. As an example, the service containing the HDDL solver *Lilotane* would have no problem generating plans from provided domain and problem files in a VirtualBox instance. But when that service was hosted in an EC2 instance, which ran the same OS and used the same dependencies, suddenly there were cases when the service would not produce a plan. Instead, the solving process would simply get stuck and never complete. The exact reason for this behavior could not be determined, perhaps the instance was lacking in memory or processing power.

6.6 Key observations

AWS pricing

While building an application making use of the AWS solutions one should be aware of their costs. If one is employing the AWS Free Tier, it is important to keep track of its limits, otherwise the free offer can run out unexpectedly. Alternatively, if using an AWS pricing scheme, the future expenses should be carefully estimated using the pricing information provided by Amazon[44].

Planner integration

Integrating CLI planners into services is very similar to integrating them into a monolithic application. One still has to provide input arguments in the expected format, while catching and parsing output in the terminal.

Importance of testing

When deploying an planning component in a production instance it would be a mistake to assume it will be functioning exactly as it did in its test environment. As discussed in Section 6.5.12, testing in a local environment first saves time, while testing in the production environment afterwards ensures the component is still working as intended.

6.7 Key recommendations

System design

When working on a larger project, especially one with multiple separate components, it is important to spare no effort on the system design. Attempting to save time on this phase might result in having to make significant changes later, even to the components that were already functioning. As shown in Section 6.5.10 on the preceding page, a thought-out system design is well worth the effort.

[44]https://calculator.aws/#/?nc2=pr

Pattern utilization

Many of the problems one can find themself facing when designing system architecture may already have been solved by others. It is highly advisable to research applicable design patterns before attempting solutions which only end up “reinventing the wheel”.

“If it ain’t broken, don’t fix it”

While viewing an established system, one may feel tempted to try and improve it. But, when considering any big changes to the system’s architecture, it is advisable to weigh out the advantages as well as the disadvantages carefully. Every big change can lead to unexpected behaviors, errors, system downtime and the necessity to re-test many, if not all, system components. An attempt to improve what is already working can end up breaking it.

Message handling

When implementing a distributed system, which employs messaging, one should make an effort to determine the messaging system’s intricacies early on. The design of message routing, format, transformation, etc., should not be done as an afterthought. A wrong decision in this area can lead to cumbersome and error-prone updates, or unnecessarily increased expenses, as discussed in Section 6.5.9 on page 61.

7 Project 3: Microservice Architecture

Project 3 consists of the implementation of a distributed application based on the microservices architecture (see Appendix A.3). Similar to Project 1 (see Chapter 5) and Project 2 (see Chapter 6) the idea is to create a system employing the same planning tools as before but utilizing microservices. The system is using REST (see Appendix A.4) as a lightweight communication mechanism.

The system components include: a frontend application, for initializing requests and displaying response results, an API gateway, for routing requests and controlling the communication flow, a service discovery, for registration and discovery of microservices, and several microservices, offering different planning capabilities.

7.1 Tools

The frontend application is developed using the React framework, the TypeScript programming language, the npm package manager and the WebStorm IDE. Other components of the system, namely the gateway, the service discovery, and the microservices, are implemented using the Spring Boot[1] framework, the Java programming language, and the IntelliJ IDE. Initialization of those components is assisted by the online tool Spring Initializr[2] and build automation is done using Maven[3] for some components and Gradle for others. The project's version control is managed by using GitLab.

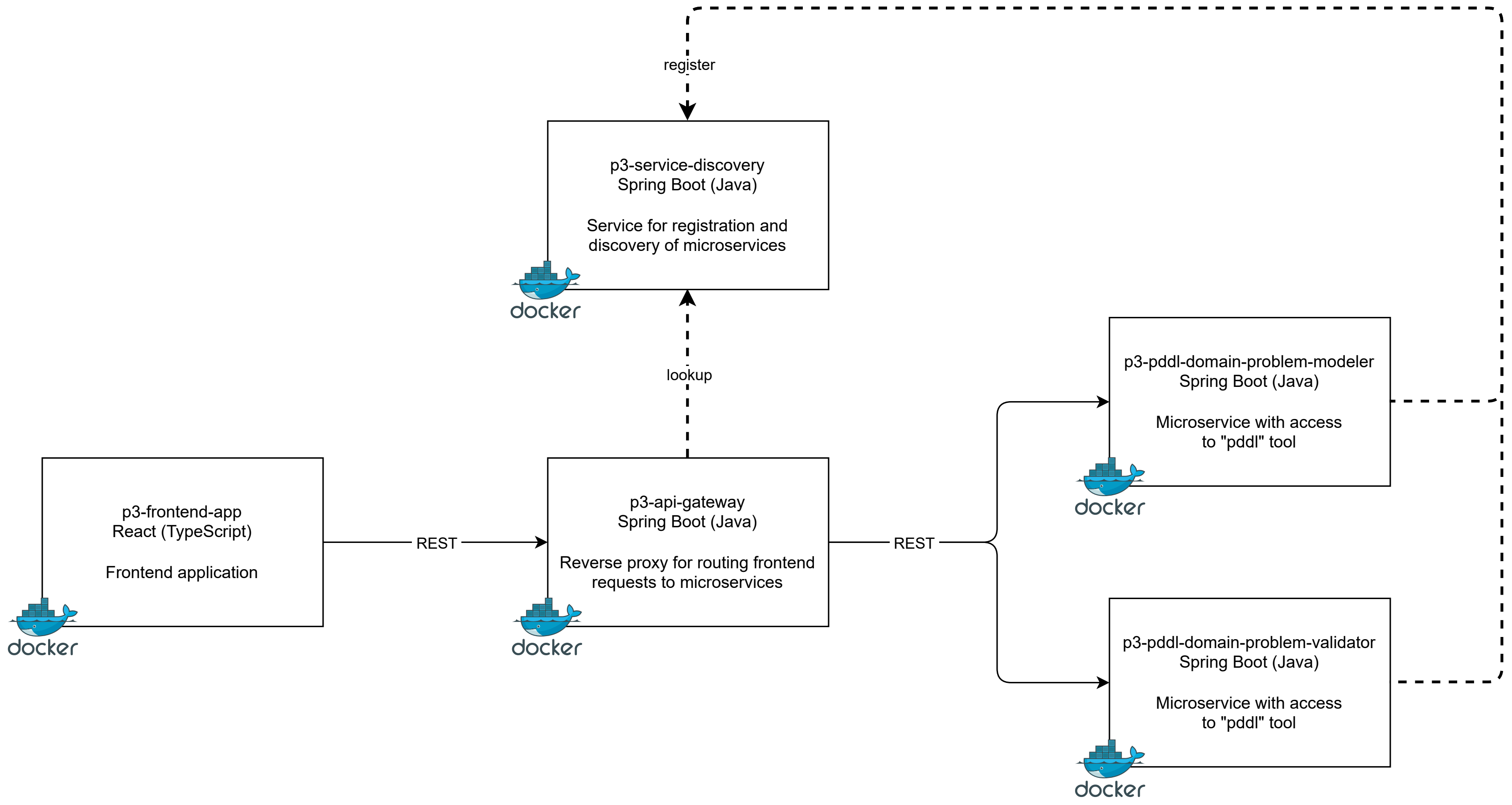
As a lightweight communication mechanism, REST is used for sending requests from the frontend application to the microservices. Each system component is deployed by hosting in a Docker container, with the initialization of all Docker images and containers controlled by a single Docker Compose[4] file.

The planning tools, which are integrated into the microservices, are listed in Table 7.1.

7.2 Scenario 1

Scenario 1 begins with the design of the overall system architecture and communication mechanisms. It includes the implementation of the frontend application, the API gateway, the service discovery, and two microservices, both of which make use of the *pddl* planning tool. Afterwards, the system components are deployed using Docker. Figure 7.1 showcases the system diagram of Scenario 1.

[1] https://spring.io
[2] https://start.spring.io
[3] https://maven.apache.org
[4] https://docs.docker.com/compose/



register
p3-service-discovery
Spring Boot (Java)
Service for registration and discovery of microservices
lookup
p3-frontend-app
React (TypeScript)
Frontend application
REST
p3-api-gateway
Spring Boot (Java)
Reverse proxy for routing frontend requests to microservices
REST
p3-pddl-domain-problem-modeler
Spring Boot (Java)
Microservice with access to "pddl" tool
p3-pddl-domain-problem-validator
Spring Boot (Java)
Microservice with access to "pddl" tool
docker

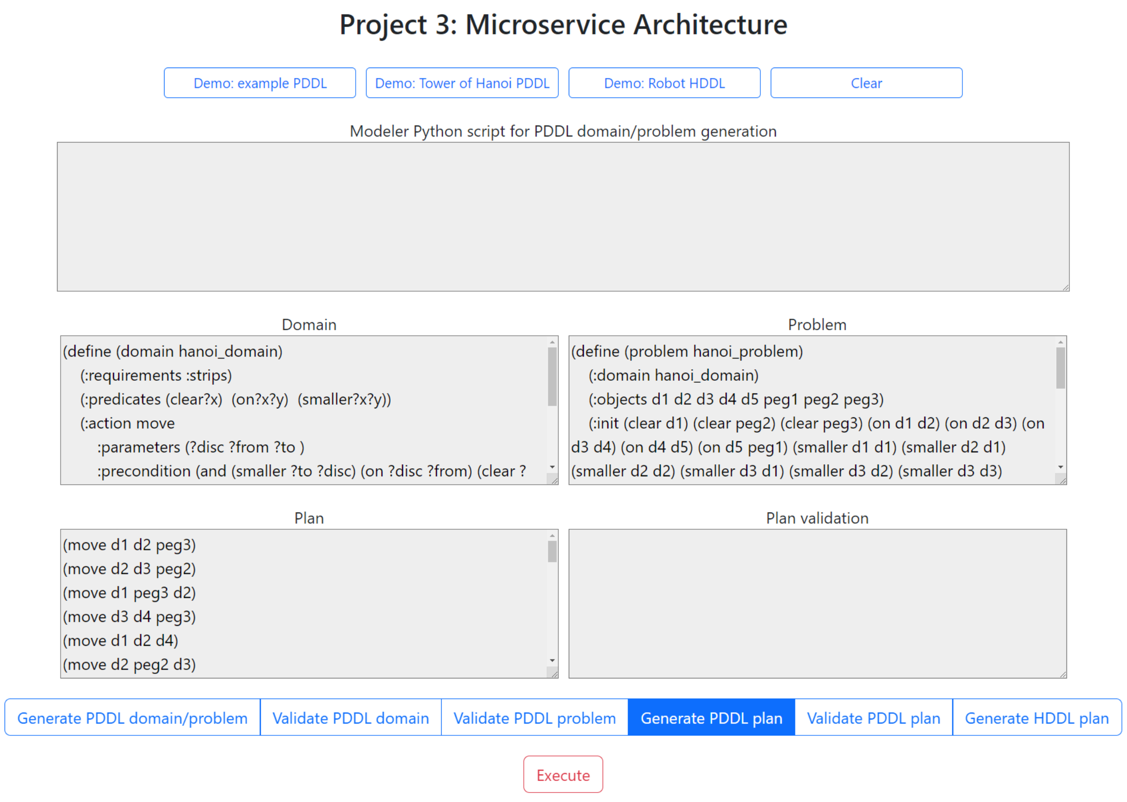
Figure 7.1: Project 3, scenario 1: system diagram.

Planner name	Purpose	Description
pddl	PDDL domain and problem generation and validation.	Section 4.4.1
Fast Downward	Plan generation from PDDL files.	Section 4.4.2
VAL	Plan validation with PDDL files.	Section 4.4.3
Lilotane	Plan generation from HDDL files.	Section 4.4.4

Table 7.1: Project 3: integrated planning tools.

7.2.1 Frontend application

The frontend application is, for the most part, based on Project 2 frontend. As can be seen in the screenshot in Figure 7.2, it employs a very similar GUI, with the difference being that the execution options are selected with radio buttons instead of toggle buttons. The intention behind this change is to prevent the API gateway being blocked by multiple consequent requests, since the communication is synchronous.



Domain:

```
(define (domain hanoi_domain)
    (:requirements :strips)
    (:predicates (clear?x)  (on?x?y)  (smaller?x?y))
    (:action move
        :parameters (?disc ?from ?to )
        :precondition (and (smaller ?to ?disc) (on ?disc ?from) (clear ?
```

Problem:

```
(define (problem hanoi_problem)
    (:domain hanoi_domain)
    (:objects d1 d2 d3 d4 d5 peg1 peg2 peg3)
    (:init (clear d1) (clear peg2) (clear peg3) (on d1 d2) (on d2 d3) (on
d3 d4) (on d4 d5) (on d5 peg1) (smaller d1 d1) (smaller d2 d1)
(smaller d2 d2) (smaller d3 d1) (smaller d3 d2) (smaller d3 d3)
```

Plan:

```
(move d1 d2 peg3)
(move d2 d3 peg2)
(move d1 peg3 d2)
(move d3 d4 peg3)
(move d1 d2 d4)
(move d2 peg2 d3)
```

Figure 7.2: Project 3: GUI screenshot.

To communicate via REST, Fetch API[5] is utilized for creation of request and response objects. The communication process is as follows: user input is gathered from the GUI's text fields, the payload is encoded as Base64 strings and inserted into the JSON request body. Then a request is created and sent to the API gateway with the "fetch()" method, and a response with a Base64 encoded payload is received and decoded. Finally, the response data or an error message is displayed through the GUI.

The API gateway address and the URL (Uniform Resource Locator) paths of the microservices are contained in a separate "config.json" file. This allows for easy changes of addresses without affecting the rest of the code, e.g., if the gateway is hosted using a domain name instead.

The application's dependencies and deployment are handled by Docker through a custom Dockerfile. By using the serve[6] npm package, the frontend is served as a static server inside its Docker container.

7.2.2 Service discovery

The service discovery is a service for registration and discovery of microservices. Its purpose within the project is to assign ports to the microservices, preventing fixed addresses. The API gateway utilizes the discovery registry to look up those addresses during request routing. Other than that, the service does not take part in the communications of the rest of the system.

The registry and discovery functionalities are provided by Netflix Eureka Server[7]. Aside from the usual main method and the "@EnableEurekaServer" Spring framework annotation, no additional Java code is required. Unlike the microservices, the registry runs on the fixed 8761 port, which is the default Eureka Server port. The service is deployed via Docker.

7.2.3 API gateway

The API gateway is a reverse proxy[8] based on the Gateway Routing[9] pattern, located between the frontend and the microservices. It receives every request sent from the frontend application and uses the registry of the service discovery to look up the port assigned to the required microservice. Using that port and a predefined URL path, the gateway forwards the requests to the corresponding microservice.

In this project, the API gateway runs on port 8080, since the frontend application needs to know where to send its requests. It also handles Cross-Origin Resource Sharing (CORS)[10] permissions for request origins, headers and methods, which allows accepting cross-origin requests from the frontend application. Like the other services, the gateway is deployed in its own Docker container.

[5] https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API
[6] https://www.npmjs.com/package/serve
[7] https://cloud.spring.io/spring-cloud-netflix/reference/html/#spring-cloud-eureka-server
[8] https://developer.mozilla.org/en-US/docs/Web/HTTP/Proxy_servers_and_tunneling#reverse_proxies
[9] https://learn.microsoft.com/en-us/azure/architecture/patterns/gateway-routing
[10] https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

7.2.4 Tools library

The tools library contains tools which are used by the microservices with an integrated planner in Project 3. Its purpose is to reduce code duplication by offering functionalities required by multiple services. It is imported into the microservices as an external JAR.

The library includes such tools as: Base64 request decoder and response encoder, file creator for saving files for CLI tools and file remover for cleaning up processed files, custom exceptions and their payload format.

7.2.5 PDDL domain-problem modeler microservice

The first implemented microservice – the PDDL domain-problem modeler – provides access to the modeling functionality of the *pddl* planning tool for PDDL domains and problems. The microservice does not possess a fixed port, instead it registers with the service discovery and is assigned a port. The Spring framework handles the registration automatically, it requires only the Netflix Eureka Client[11] dependency and an "@EnableDiscoveryClient" Spring framework annotation in the code.

During communication the microservice receives a "PUT" request containing a Python modeler script in its JSON body, encoded as a Base64 string. That script is extracted, decoded, and saved locally as a file, which is then executed via a terminal command. The resulting output is caught and parsed, with the parser expecting specific markers to recognize the PDDL domain and/or problem within the output: "DOMAIN START" and "DOMAIN END", "PROBLEM START" and "PROBLEM END". Afterwards, a response JSON is created, containing the Base64 encoded domain and/or problem, depending on whether one or both were present in the output. The response to the "PUT" request is then returned to the frontend through the API gateway and the locally saved modeler script is deleted.

Should an error occur during request processing, an error response is returned instead of the usual response. The error handling is taken care of by the Spring framework through the use of the "@ExceptionHandler" Spring framework annotation. The microservice is hosted in a Docker container as specified by its Dockerfile.

7.2.6 PDDL domain-problem validator microservice

The PDDL domain-problem validator microservice grants access to the *pddl* planning tool, similar to the PDDL domain-problem modeler microservice, but with a few differences. First, the service employs a separate functionality of the planning tool, namely its capability to validate PDDL domains and programs. Second, during request processing the *pddl* tool is not used as a library in a script but is accessed directly through a console command. Third, the microservice provides two separate endpoints, one for domain validating, the other for problem validating.

[11] https://cloud.spring.io/spring-cloud-netflix/reference/html/#service-discovery-eureka-clients

The communication process is as follows: a “PUT” request is received either on the domain or the problem endpoint. Its decoded payload is saved as a local file and the CLI *pddl* tool is executed with the corresponding domain or problem argument. The output is checked for either a successful validation or a validation error and a response JSON with a Base64 encoded payload is created. Once the response is returned to the frontend, the locally saved file is cleaned up.

The service registration and port assignment are enabled by the Netflix Eureka Client dependency and the service discovery. The microservice is hosted in a Docker container.

7.2.7 Docker Compose

Once every component of Scenario 1 is implemented and hosted via Docker, five separate Docker containers require managing. For automating the process of building the Docker images and containers from the Dockerfile of each system component, a custom Docker Compose file is created.

The “docker-compose.yml” omits the ports mapping in the definitions of microservices, which allows the service discovery registry to assign ports to the microservices and their containers. This also enables potential horizontal scaling of microservices, as multiple containers running the same microservice but with different ports would not block each other.

Lastly, the “docker-compose.yml” enables communication between Docker containers. For this purpose, a custom network is defined within it.

7.3 Scenario 2

Scenario 2 adds PDDL plan generation and validation capabilities to the current system. This entails implementing, integrating, and deploying two new microservices – the PDDL plan solver and the PDDL plan validator. Additionally, the new microservices require updating the API gateway and the frontend application with new URL paths. The system diagram of Scenario 2 is displayed in Figure 7.3.

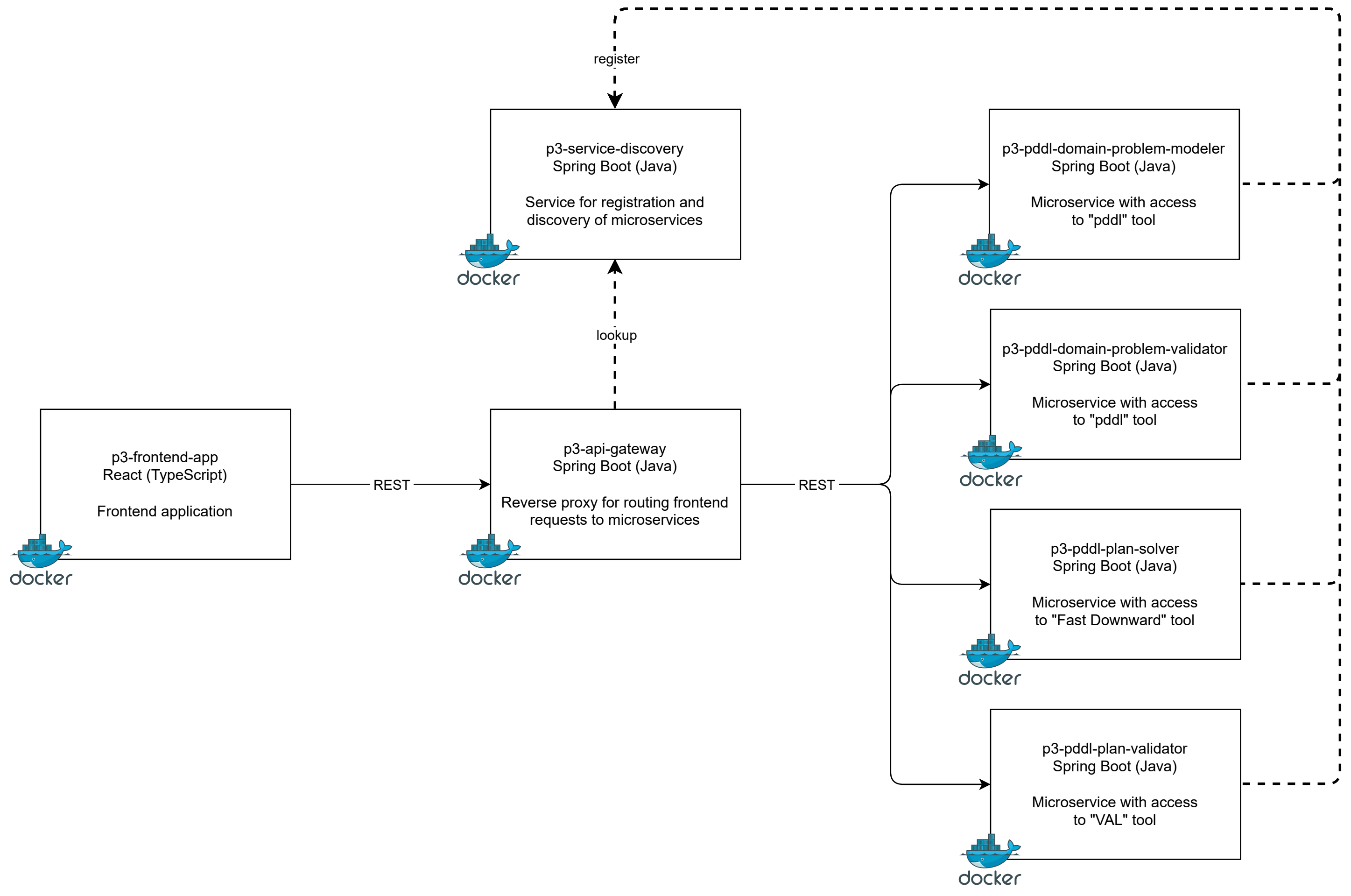

register
p3-service-discovery
Spring Boot (Java)
Service for registration and discovery of microservices
docker
lookup
p3-frontend-app
React (TypeScript)
Frontend application
docker
REST
p3-api-gateway
Spring Boot (Java)
Reverse proxy for routing frontend requests to microservices
docker
REST
p3-pddl-domain-problem-modeler
Spring Boot (Java)
Microservice with access to "pddl" tool
docker
p3-pddl-domain-problem-validator
Spring Boot (Java)
Microservice with access to "pddl" tool
docker
p3-pddl-plan-solver
Spring Boot (Java)
Microservice with access to "Fast Downward" tool
docker
p3-pddl-plan-validator
Spring Boot (Java)
Microservice with access to "VAL" tool
docker

Figure 7.3: Project 3, scenario 2: system diagram.

7.3.1 PDDL plan solver microservice

The PDDL plan solver microservice utilizes the *Fast Downward* planning tool to generate plans from the provided PDDL data. It registers with the service discovery and is assigned a port, same as the microservices implemented in Scenario 1. It also makes use of the tools library JAR for request processing.

In a received “PUT” request’s JSON body, the microservice expects Base64 encoded PDDL domain and problem. The decoded and locally saved domain and problem files are used as arguments to execute the CLI *Fast Downward* tool. As mentioned in Project 2, this tool saves its output plan as a file on its own, meaning that no output parsing is necessary. The content of the plan file is encoded as a Base64 string and inserted into the response JSON body, which is then returned to the frontend. At this point the request has been processed and the domain, problem and plan files can be safely removed.

Error handling is taken care of by the Spring framework with the help of the “@ExceptionHandler” Spring framework annotation and the microservice is deployed in a Docker container, same as the other system components.

7.3.2 PDDL plan validator microservice

The validation of plans, generated from PDDL domains and problems, is enabled by the PDDL plan validator microservice with the integrated *VAL* planning tool. Any received “PUT” request is expected to provide the following data: PDDL domain, PDDL problem and the resulting plan. Decoding and saving the request payload as three separate files allows executing the *VAL* tool via a terminal command with the required arguments. The tool’s process can result in several output cases, depending on the provided input: valid plan, invalid plan, invalid domain, or invalid problem. In either case, the output is parsed, the result is sent back to the frontend application in the response JSON body encoded as a Base64 string, and the processed local files are deleted.

Other details of the microservice are handled as described previously in the sections on the other microservices: registration and port assignment through the service discovery, error handling assisted by the Spring framework, deployment through Docker.

7.3.3 System update

Once the two new microservices are implemented, their integration into the system requires a few additional changes. Their URL paths are added to the API gateway and the frontend application is updated to handle new requests/responses for PDDL plan generation and validation. Additionally, the Docker Compose file is extended to automate the creation of two new Docker images and to give the new containers access to the communication network.

7.4 Scenario 3

Scenario 3 consists of providing the system with the functionality to generate plans from HDDL domains and problems. Afterwards, the system components are updated to be able to communicate with the new microservice. The finalized system diagram can be seen in Figure 7.4.

7.4.1 HDDL plan solver microservice

The fifth and final microservice, the HDDL plan solver, gives access to the *Lilotane* planning tool. The microservice provides an endpoint for a "PUT" request with HDDL domain and problem as its payload. The domain and problem are then prepared to be used as CLI tool arguments, by being decoded from Base64 strings and saved as local files. After *Lilotane's* execution, the resulting plan needs to be extracted from the terminal output, which is assisted by the planning tool labeling the beginning and the end of the plan with "==>" and "<==" strings. The generated plan is then returned to the frontend and the locally saved HDDL domain and problem files are removed in preparation for the next request.

Other aspects of the microservice, like the registration at service discovery, utilization of the tools library, and Docker deployment, are implemented in the same manner as with the other microservices.

7.4.2 System update

To allow routing of requests from the frontend to the new microservice, its URL path is added to the API gateway. Likewise, the frontend application is extended with a new "PUT" request factory. Finally, the Docker Compose file is updated to include building the Docker image of the HDDL plan solver microservice.

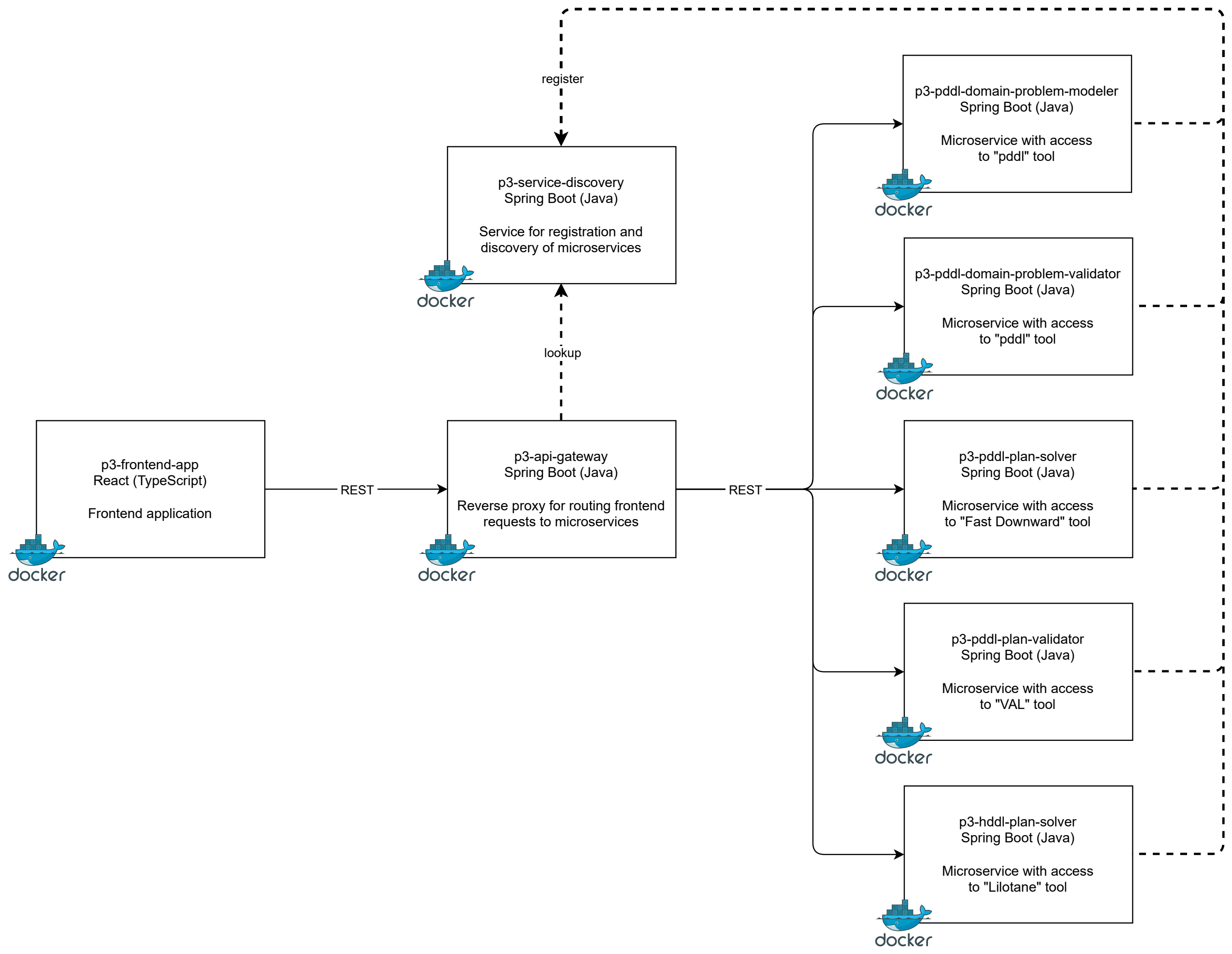

register
p3-service-discovery
Spring Boot (Java)
Service for registration and discovery of microservices
lookup
p3-frontend-app
React (TypeScript)
Frontend application
REST
p3-api-gateway
Spring Boot (Java)
Reverse proxy for routing frontend requests to microservices
REST
p3-pddl-domain-problem-modeler
Spring Boot (Java)
Microservice with access to "pddl" tool
p3-pddl-domain-problem-validator
Spring Boot (Java)
Microservice with access to "pddl" tool
p3-pddl-plan-solver
Spring Boot (Java)
Microservice with access to "Fast Downward" tool
p3-pddl-plan-validator
Spring Boot (Java)
Microservice with access to "VAL" tool
p3-hddl-plan-solver
Spring Boot (Java)
Microservice with access to "Lilotane" tool
docker

Figure 7.4: Project 3, scenario 3: system diagram.

7.5 Discussion

The following sections describe system properties, design difficulties and decisions made during the implementation of Project 3.

7.5.1 Adherence to the microservices properties

As per the definition of microservices in Appendix A.3, they are expected to have certain properties. But how well does the Project 3 application adhere to those properties?

- "*a single application as a suite of small services*": the application is built as a distributed system, with several services providing functionalities of integrated planning tools.
- "*each [service is] running in its own process*": through the use of separate Docker containers each microservice is running independently, unaffected by other containers.
- "*[each service is] communicating with lightweight mechanisms*": communication is kept lightweight by using REST requests instead of setting up and employing the Enterprise Service Bus [Cha04].
- "*services are built around business capabilities*": each microservice is built around a single planner, offering only one of its functionalities through one (or two) endpoint/s.
- "*[services are] independently deployable by automated deployment machinery*": although the microservices are deployed by using a "docker-compose.yml" file, each can be deployed separately through its own Dockerfile. As for the automated deployment, that functionality was not implemented in the project, but it would still be possible to add it.
- "*bare minimum of centralized management of [...] services*": excepting the service discovery registry, the microservices do not need additional management.
- "*[services] may be written in different programming languages*": all microservices in Project 3 are implemented with Java and Spring Boot, but another programming language and/or framework would also have been acceptable. The language only needs to be able to use REST, execute commands in the terminal and access a Eureka Client library.
- "*[services] may use different data storage technologies*": while none of the microservices utilize any storage technologies, their addition is possible.

7.5.2 Microservice granularity

As mentioned in the previous section, microservices are meant to be kept small. Consequently, this leads to an additional design challenge, namely deciding on the proper granularity of the microservices being implemented.

For example, the planner *pddl* offers both modeling and validating of PDDL domains and problems. Although it is a single tool, it was decided that its two provided functionalities are different enough to warrant the creation of two separate microservices: the PDDL domain-problem modeler and the PDDL domain-problem validator. The reasoning was that those functionalities might as well have been provided by two different planners and should not be merged into one microservice.

Continuing with the PDDL domain-problem validator microservice, it offers two endpoints for requests: one for validating PDDL domains and the other for validating PDDL problems. While those are technically two different functionalities, they only differ in their execution commands: "pddl domain domain.pddl" and "pddl problem problem.pddl". Which is why it was decided that creating two microservices, with the only real difference between them being the "domain" or the "problem" argument string, was not worth it.

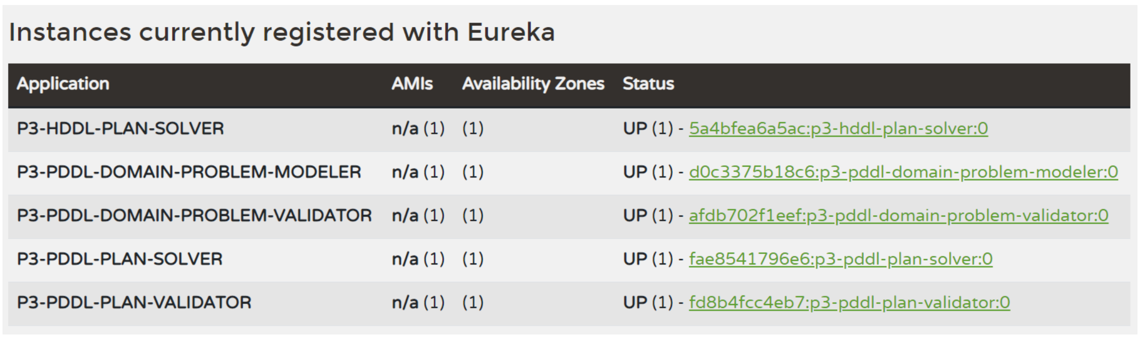
7.5.3 Loosening the tight coupling of microservices

Although the services built in compliance with the microservice architectural style can become tightly coupled, it is possible to take steps towards mitigating that. The following points explore whether the autonomy aspects of loose coupling (see Appendix A.1 on page 89) are respected in Project 3.

- Reference autonomy: the autonomy aspect is respected, since the microservices possess no fixed addresses. They register with the service discovery and are assigned ports, as displayed in Figure 7.5, which are then looked up by the API gateway. The frontend is aware only of the gateway's address.
- Format autonomy: unlike the Project 2 application, there is no reason to keep to a single message format. As result, each microservice expects requests and produces responses containing only specific information.
- Platform autonomy: with the frontend being implemented in TypeScript and the backend in Java, with additional microservices being possible in other programming languages, the platform autonomy is respected.
- Time autonomy: contrary to messaging, the communication via REST is synchronous, in violation of the time autonomy. However, the microservices are horizontally scalable, meaning that at least it is possible to prevent requests getting blocked during multiple simultaneous communication exchanges.

7.5.4 Setting up communication

Speaking in practical terms, although communication via REST is more lightweight than communication via messaging, e.g., AWS SQS, setting it up is not without difficulties. In the case of SQS, it involves getting familiar with the documentation and AWS SDK code examples, handling security, creating messaging queues and enabling the system components to interact with them. This endeavor requires a meaningful time investment to get communication working properly.


Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
P3-HDDL-PLAN-SOLVER	n/a (1)	(1)	UP (1) - 5a4bfea6a5ac:p3-hddl-plan-solver:0
P3-PDDL-DOMAIN-PROBLEM-MODELER	n/a (1)	(1)	UP (1) - d0c3375b18c6:p3-pddl-domain-problem-modeler:0
P3-PDDL-DOMAIN-PROBLEM-VALIDATOR	n/a (1)	(1)	UP (1) - afdb702f1eef:p3-pddl-domain-problem-validator:0
P3-PDDL-PLAN-SOLVER	n/a (1)	(1)	UP (1) - fae8541796e6:p3-pddl-plan-solver:0
P3-PDDL-PLAN-VALIDATOR	n/a (1)	(1)	UP (1) - fd8b4fcc4eb7:p3-pddl-plan-validator:0

Figure 7.5: Project 3: service discovery registry screenshot.

Compared to that, handling URL paths, routing, REST request and response mechanisms may not require reading an extensive documentation or employing any AWS solutions, but it can still be time consuming and error prone. Setting up custom networking is not easy and involves avoiding several pitfalls, e.g., correctly handling microservices port assignment, enabling CORS preflight requests[12] and creating a Docker container network.

7.5.5 Sidecar pattern

Through the use of the Sidecar[13] pattern, the microservice endpoint logic, e.g., configuration, routing, message transformation, can be decoupled into a separate component. The sidecar component runs as a separate process in its own container, while sharing the lifecycle of its parent component. Making the sidecar be responsible for communication of the parent component results in reduced coupling.

The decision not to employ the Sidecar pattern for Project 3 was made for several reasons. First, there is no need for extra code for the implementation of service discovery, as the registration of the microservice is done automatically through the “@EnableDiscoveryClient” Spring framework annotation. Similarly, no additional message routing is required, since every REST response is sent back to the API gateway. In this regard there is no logic to decouple.

Second, sidecar components are useful when the source code of the parent component is inaccessible, which is not the case in Project 3. Employing sidecar components would mean running extra processes and handling communication between the sidecar and the parent components. Not only would this unnecessarily increase the system complexity but also the communication latency.

Finally, the tools required for request payload handling and transformation are implemented as a Java library instead. As the library turned out to be quite small, decoupling it in a separate component was not worth the effort. In addition to that, since all microservices are built with Java in this project, the library can be used in each of them, without requiring a separate component.

[12]https://developer.mozilla.org/en-US/docs/Glossary/Preflight_request
[13]https://learn.microsoft.com/en-us/azure/architecture/patterns/sidecar

7.5.6 Gateway

The purpose of a gateway in a system architecture is to permit communication with multiple services from a single address and route incoming requests to the appropriate services. A gateway can be combined with a load balancer, e.g., Nginx[14], to manage traffic and handle horizontally scaled services. Other uses of gateways include encryption, decryption and validation of content, authentication, authorization, logging, monitoring, etc.

In the microservices architecture design[15], the gateway is usually the only backend component that is known to the frontend. In Project 3, the API gateway runs on the fixed 8080 port to allow the frontend application to send requests through it. However, in a real-world scenario the gateway might instead use a domain name, e.g., “www.example.com”.

One advantage of using a domain name is that it makes it possible for the system to run multiple API gateway instances on different ports. If the gateway is not horizontally scaled, it becomes a single point of failure and in case of its crash the whole system would come to a standstill. Additionally, since the system employs REST communication, which is synchronous, multiple gateway instances would allow multiple frontend application instances to communicate with the backend simultaneously.

Speaking of synchronous communication, one disadvantage of employing a gateway is that it increases latency. As the API gateway is an additional component located between the frontend application and the microservices, routing communication through it requires additional requests and responses.

7.5.7 Integration of new microservices

As the system design respects (for the most part) the loose coupling autonomy aspects (see Section 7.5.3 on page 78), integrating new microservices into the system does not influence the existing ones. However, even if the previous microservices do not need to be updated, the API gateway and the front application do. The gateway is extended with the URL paths of the new microservices and the frontend application is added the ability to send requests to them.

7.6 Key observations

Employing a gateway

When designing a system with the microservices architectural style, one may contemplate omitting building a gateway component. But based on the experiences made during the work on Project 3 (and Project 2), the advantages of a gateway outweigh the increased latency and the danger of introducing a single point of failure.

[14]https://www.nginx.com

[15]https://learn.microsoft.com/en-us/azure/architecture/microservices

Designing fine-grained microservices

According to the microservices definition, they are meant to be kept small and built around business capabilities. Designing a fine-grained microservice, however, is not easy. On the one hand, it should not become bloated with features, and on the other hand, dividing similar features between multiple microservices can lead to unnecessary code duplication and overcomplication of the resulting system.

CLI planners integration

As with the monolithic application in Project 1 and the services in Project 2, the integration of CLI planners into microservices is faced with the same difficulties. Catching and parsing their output, without adjusting the planner's source code, remains unreliable.

Microservices' integration

In a properly designed system, the integration of new microservices is simple and does not require many adjustments. Using a lightweight communication mechanism means that once the gateway is extended with the new URL path/s for routing, the frontend is free to communicate with the microservice.

7.7 Key recommendations

Researching microservices architecture

Before attempting to design a system using the microservices architectural style, one should get familiar with its properties and limitations. Otherwise, for example, one may end up reinventing SOA when attempting to improve communication by turning it into asynchronous messaging.

Learning communication intricacies

While REST is a more lightweight communication mechanism than messaging queues, it can be difficult to implement correctly. It is advisable to make sure one understands the intricacies of HTTP (Hypertext Transfer Protocol) communication, e.g., CORS.

Reducing tight coupling

As a result of using synchronous communication, the microservices have tighter coupling. But it is worth loosening those couplings, at least in other areas than communication, like port assignment, as it enables the scaling of microservices.

Determining pattern usefulness

Even though there are many advantages to using patterns, one should determine whether they suit the system's needs before making use of them. As shown with the Sidecar pattern example in Section 7.5.5 on page 79, even though the pattern is useful in theory, in given circumstances it was determined not to be a good fit for the system.

8 Conclusion

The final chapter of this thesis concludes with a summary of the done research, implemented projects and findings. Additionally, it provides a perspective on possible future work.

8.1 Summary

The goal of the thesis is to uncover the challenges of planning-based application engineering in regard to design, integration and deployment. For that purpose, three projects with several scenarios have been implemented, partaking in each of the mentioned engineering aspects.

To summarize:

- The three projects have been designed in different architectural styles: monolithic, SOA and microservices architecture.
- Integration of planning components has been carried out using different technologies: message queueing and REST.
- The project components have been deployed via EC2 and Docker.

In terms of design, the importance of a robust architecture design cannot be overstated. In this regard it helps greatly to adhere to proven patterns and guidelines, instead of committing design faux pas. To be able to employ patterns, e.g., Messaging Patterns[1], a suitable platform is required, demanding a certain level of know-how. Learning and setting up technologies, e.g., AWS SQS, Spring Boot and custom networking, may not be trivial, but is still feasible after a sufficient time investment. The biggest challenge during the design phase involving planning tools is their selection process. This endeavor includes researching available planners, their set up, determining their usability and ensuring their compatibility, ending up requiring more effort than initially predicted.

Integrating a planning tool CLI executable in an application or a system component is similar to using any other external executable: it requires a certain input and provides some output. However, as the tool does not offer an API, the only way to provide said input is through command line commands with arguments. Since the output is supplied through the terminal as well, it is often interlaced with additional logging information, making it difficult to discern the actual process results. In the implementation of the three projects, the output is caught and parsed manually by using regular expressions. This method has proven itself to be unreliable, as the employed planning tools do not conform to a singular output format. This leads to the necessity to cover all possible

[1]https://www.enterpriseintegrationpatterns.com/patterns/messaging/

process results, e.g., success, failure, problems with input or execution, and unexpected planning tool crash. As the planner documentation rarely covers every possible form of output, manual tool testing is required to be able to anticipate possible outputs.

Concerning deployment, a large part of the challenge involves learning the correct use of deployment technologies, e.g., AWS EC2 and Docker. Other than that, an additional problem arises when managing the dependencies of the utilized planning technologies. Although which dependencies are necessary should become clear during planner set up in the design phase, ensuring that they are functioning, e.g., in a Dockerfile, sometimes requires additional tweaking.

8.2 Future work

Each of the three implemented projects includes a logical sequence of planning technologies:

- Domain and problem modeling through *pddl*.
- Domain and problem validation through *pddl*.
- Problem solving for plan generation through *Fast Downward*.
- Plan validation through *VAL*.

This chain of planners can be extended through a plan execution tool, an execution monitoring tool, etc. Although, judging by the experiences made during project implementation, the integration of those tools would not proceed in a different manner than with the previous tools, it is possible that some new insights may be uncovered.

In addition to the employed classical planning tools, a hierarchical planner *Lilotane* was added to two projects. Its purpose was to determine whether such an addition would require different integration techniques or pose different challenges. Although that was not the case, integrating tools with other planning types, e.g., temporal, could result in different findings.

As mentioned in Section 4.2.3, no tools for automated problem generation could be found. The reasons for their absence and the usefulness of a potential implementation may warrant investigation.

Bibliography

[AFG21] M. Aiello, L. Fiorini, I. Georgievski. “Software Engineering Smart Energy Systems”. In: *Handbook of Smart Energy Systems*. Cham: Springer International Publishing, Mar. 2021, pp. 1–29. ISBN: 978-3-030-72322-4. DOI: 10.1007/978-3-030-72322-4_21-1. URL: https://doi.org/10.1007/978-3-030-72322-4_21-1.

[AGPA22] E. Alnazer, I. Georgievski, N. Prakash, M. Aiello. “A Role for HTN Planning in Increasing Trust in Autonomous Driving”. In: *2022 IEEE International Smart Cities Conference (ISC2)*. Sept. 2022, pp. 1–7. DOI: 10.1109/ISC255366.2022.9922427.

[All10] S. Allamaraju. *RESTful Web Services Cookbook*. O’Reilly Media, 2010. ISBN: 978-0-596-80168-7.

[BCC12] J. Benton, A. Coles, A. Coles. “Temporal Planning with Preferences and Time-Dependent Continuous Costs”. In: *ICAPS 2012 - Proceedings of the 22nd International Conference on Automated Planning and Scheduling* 22 (May 2012). DOI: 10.1609/icaps.v22i1.13509.

[BDH99] C. Boutilier, T. Dean, S. Hanks. “Decision-Theoretic Planning: Structural Assumptions and Computational Leverage”. In: *The Journal of Artificial Intelligence Research (JAIR)* 11 (July 1999), pp. 1–94. DOI: 10.1613/jair.575.

[Cha04] D. A. Chappell. *Enterprise Service Bus*. O’Reilly, 2004, pp. 1–247. ISBN: 978-0-596-00675-4. URL: https://www.oreilly.com/library/view/enterprise-service-bus/0596006756/.

[Ede04] S. Edelkamp. “PDDL2.2: The language for the Classical Part of the 4th International Planning Competition”. In: (Jan. 2004), pp. 1–21.

[FL02] M. Fox, D. Long. “PDDL+ : Modelling Continuous Time-dependent Effects”. In: (Apr. 2002), pp. 34–44.

[FL03] M. Fox, D. Long. “PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains”. In: *J. Artif. Intell. Res. (JAIR)* 20 (Dec. 2003), pp. 61–124. DOI: 10.1613/jair.1129.

[FMJB15] A. Feljan, S. Mohalik, M. B. Jayaraman, R. Badrinath. “SOA-PE: A service-oriented architecture for Planning and Execution in cyber-physical systems”. In: Dec. 2015, pp. 1–6. DOI: 10.1109/SMARTSENS.2015.7873602.

[FN71] R. Fikes, N. Nilsson. “STRIPS: A new approach to the application of theorem proving to problem solving”. In: *Artificial Intelligence* 2 (Dec. 1971), pp. 189–208. DOI: 10.1016/0004-3702(71)90010-5.

[GA15] I. Georgievski, M. Aiello. “HTN planning: Overview, comparison, and beyond”. In: *Artificial Intelligence* 222 (Feb. 2015), pp. 124–156. DOI: 10.1016/j.artint.2015.02.002.

[GAP+12] C. Guzmán Alvarez, V. Alcázar, D. Prior, E. Onaindia, D. Borrajo, J. Fdez-Olivares, E. Quintero. "PELEA: a Domain-Independent Architecture for Planning, Execution and Learning". In: June 2012.

[GB21] I. Georgievski, U. Breitenbücher. "A Vision for Composing, Integrating, and Deploying AI Planning Functionalities". In: Aug. 2021, pp. 166–171. DOI: 10.1109/SOSE52839.2021.00025.

[Geo22a] I. Georgievski. "Office Activity Recognition Using HTN Planning". In: *2022 16th International Conference on Signal-Image Technology & Internet-Based Systems (SITIS)*. Oct. 2022, pp. 70–77. DOI: 10.1109/SITIS57111.2022.00019.

[Geo22b] 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, 2022, pp. 225–236. ISBN: 978-3-031-26507-5.

[Geo23] I. Georgievski. "Conceptualising Software Development Lifecycle for Engineering AI Planning Systems". In: *IEEE/ACM International Conference on AI Engineering – Software Engineering for AI*. To appear. 2023.

[GKW+98] M. Ghallab, C. Knoblock, D. Wilkins, A. Barrett, D. Christianson, M. Friedman, C. Kwok, K. Golden, S. Penberthy, D. Smith, Y. Sun, D. Weld. "PDDL - The Planning Domain Definition Language". In: (Aug. 1998), pp. 1–27.

[GL05] A. Gerevini, D. Long. "Plan Constraints and Preferences in PDDL3 The Language of the Fifth International Planning Competition". In: *ICAPS 2006* (Jan. 2005), pp. 1–12.

[GNT16] M. Ghallab, D. Nau, P. Traverso. *Automated Planning and Acting*. July 2016. ISBN: 9781107037274. DOI: 10.1017/CBO9781139583923.

[GS02] A. Gerevini, I. Serina. "LPG: A Planner Based on Local Search for Planning Graphs with Action Costs." In: Jan. 2002, pp. 13–22.

[HBB+20] D. Höller, G. Behnke, P. Bercher, S. Biundo, H. Fiorino, D. Pellier, R. Alford. "HDDL: An Extension to PDDL for Expressing Hierarchical Planning Problems". In: vol. 34. Apr. 2020, pp. 9883–9891. DOI: 10.1609/aaai.v34i06.6542.

[HBBB21] D. Höller, G. Behnke, P. Bercher, S. Biundo. "The PANDA Framework for Hierarchical Planning". In: *KI - Künstliche Intelligenz* 35 (Jan. 2021). DOI: 10.1007/s13218-020-00699-y.

[Hel08] M. Helmert. "Changes in PDDL 3.1". In: *Unpublished summary from the IPC-2008 website* (2008).

[Hel11] M. Helmert. "The Fast Downward Planning System". In: *Journal of Artificial Intelligence Research - JAIR* 26 (Sept. 2011). DOI: 10.1613/jair.1705.

[HLF04] R. Howey, D. Long, M. Fox. "VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL". In: Dec. 2004, pp. 294–301. ISBN: 0-7695-2236-X. DOI: 10.1109/ICTAI.2004.120.

[HN11] J. Hoffmann, B. Nebel. "The FF Planning System: Fast Plan Generation Through Heuristic Search". In: *Journal of Artificial Intelligence Research - JAIR* 14 (June 2011). DOI: 10.1613/jair.855.

[HTD90] J. Hendler, A. Tate, M. Drummond. “AI Planning: Systems and Techniques”. In: *AI Magazine* 11 (June 1990), pp. 61–77. DOI: 10.1609/aimag.v11i2.833.

[HW04] G. Hohpe, B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Ed. by G. Hohpe, B. Woolf, K. Brown, C. F. D’Cruz, S. Neville, M. J. Rettig, J. Simon. The Addison-Wesley signature series. Boston; San Francisco; New York; Toronto: Addison-Wesley, 2004, pp. 1–683. ISBN: 9780321200686.

[Mas11] M. Masse. *REST API Design Rulebook*. O’Reilly Media, 2011. ISBN: 9781449319908.

[ML20] C. Muise, N. Lipovetzky. “KEPS Book: Planning.Domains”. In: Mar. 2020, pp. 91–105. ISBN: 978-3-030-38560-6. DOI: 10.1007/978-3-030-38561-3_5.

[MMB12] C. Muise, S. Mcilraith, J. Beck. “Improved Non-Deterministic Planning by Exploiting State Relevance”. In: *ICAPS 2012 - Proceedings of the 22nd International Conference on Automated Planning and Scheduling* 22 (May 2012), pp. 172–180. DOI: 10.1609/icaps.v22i1.13520.

[MPSK22] C. Muise, F. Pommerening, J. Seipp, M. Katz. “PLANUTILS: Bringing Planning to the Masses”. In: 2022.

[MR15] G. Markou, I. Refanidis. “Non-deterministic planning methods for automated web service composition”. In: *Artificial Intelligence Research* 5 (Sept. 2015). DOI: 10.5430/air.v5n1p14.

[Mye99] K. Myers. “CPEF - A continuous planning and execution framework”. In: *Ai Magazine* 20 (Dec. 1999), pp. 63–69.

[New15] S. Newman. *Building Microservices: Designing Fine-Grained Systems*. 1st. O’Reilly Media, Feb. 2015, p. 280. ISBN: 978-1-4919-5035-7.

[Pap12] M. P. Papazoglou. *Web Services & SOA: Principles and Technology*. Pearson Education, 2012. ISBN: 978-0273732167.

[Ric15] M. Richards. *Microservices vs. Service-Oriented Architecture*. O’Reilly Media, 2015. ISBN: 9781491975657.

[Rin07] J. Rintanen. “Complexity of Concurrent Temporal Planning.” In: Jan. 2007, pp. 280–287.

[RN10] S. Russel, P. Norvig. “Artificial Intelligence: A Modern Approach”. In: *Prentice Hall, Englewood Cliffs, NJ* (Jan. 2010).

[RW10] S. Richter, M. Westphal. “The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks”. In: *J. Artif. Intell. Res. (JAIR)* 39 (Sept. 2010), pp. 127–177. DOI: 10.1613/jair.2972.

[Sch21] D. Schreiber. “Lilotane: A Lifted SAT-based Approach to Hierarchical Planning”. In: *Journal of Artificial Intelligence Research* 70 (Mar. 2021), pp. 1117–1181. DOI: 10.1613/jair.1.12520.

[SFJ00] D. Smith, J. Frank, A. Jonsson. “Bridging the Gap Between Planning and Scheduling”. In: *The Knowledge Engineering Review* 15 (Nov. 2000), pp. 47–83. DOI: 10.1017/S0269888900001089.

[SK15] V. Strobel, A. Kirsch. “Planning in the Wild: Modeling Tools for PDDL”. In: Nov. 2015, pp. 273–284. ISBN: 978-3-319-11205-3. DOI: 10.1007/978-3-319-11206-0_27.

[VK20] M. Vallati, D. Kitchin, eds. *Knowledge Engineering Tools and Techniques for AI Planning*. English. 1 Online-Ressource(VIII, 277 p. 97 illus., 53 illus. in color.) Cham, 2020. DOI: 10.1007/978-3-030-38561-3. URL: http://dx.doi.org/10.1007/978-3-030-38561-3.

[WRP10] J. Webber, I. Robinson, S. Parastatidis. *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly, 2010. ISBN: 978-0-596-80582-1.

All links were last followed on May 1, 2023.

A Definitions

A.1 Loose coupling

The loose coupling **core principle**: reduce number of assumptions two parties make about each other when they exchange information.

The loose coupling **autonomy aspects**:

- Reference Autonomy: Producers and consumers don't know each other.
- Time Autonomy: Producers and consumers access channel at their own pace.
- Format Autonomy: Producers and consumers may use different formats of data exchanged.
- Platform Autonomy: Producers and consumers may be in different environments, written in different languages, etc.

The information on loose coupling provided in this section is taken from the lecture "Loose Coupling and Message-based Applications"[1].

A.2 SOA

Service-Oriented Architecture (SOA) is an architectural style for realizing distributed computing by implementing business processes as distinct services, which are loosely coupled.

A **service** is a function which is provided at a network address, is always available and can be communicated with through various transports and formats.

The information on SOA provided in this section is taken from the lecture "Service Computing"[2].

Further reading: [Pap12].

[1]https://www.iaas.uni-stuttgart.de/en/teaching/lectures/2021_ws/lcm/

[2]https://www.iaas.uni-stuttgart.de/en/teaching/lectures/2021_ws/sc/

A.3 Microservices

“In short, the microservice architectural style is an approach to developing a single application as a **suite of small services**, each **running in its own process** and communicating with lightweight mechanisms, often an HTTP resource API. These services are **built around business capabilities** and **independently deployable** by fully automated deployment machinery. There is a **bare minimum of centralized management** of these services, which may be written in different programming languages and use different data storage technologies.”

– James Lewis and Martin Fowler, “Microservices” (2014) [3] [4]

Further reading: [New15; Ric15].

A.4 REST

Representational State Transfer (REST) is an architectural style for developing web services.

Key aspects of REST:

- Stateless interaction: requests do not depend on the state of the request’s recipient.
- Uniform interface: standard communication between client and server across platforms.

Fundamental methods of REST-based interfaces:

- GET: retrieve resource.
- POST: create resource.
- PUT: update resource.
- DELETE: delete resource.

The information on REST provided in this section is taken from the lecture “Service Computing”[5].

Further reading: [All10; Mas11; WRP10].

[3]https://martinfowler.com/microservices

[4]https://martinfowler.com/articles/microservices.html

[5]https://www.iaas.uni-stuttgart.de/en/teaching/lectures/2021_ws/sc/

B Domain and Problem Examples

B.1 PDDL domain

```
(define (domain hanoi_domain)
    (:requirements :strips)
    (:predicates (clear?x)  (on?x?y)  (smaller?x?y))
    (:action move
        :parameters (?disc ?from ?to)
        :precondition (and (smaller ?to ?disc) (on ?disc ?from) (clear ?disc) (clear ?to))
        :effect (and (clear ?from) (on ?disc ?to) (not (on ?disc ?from)) (not (clear ?to)))
    )
)
```

Listing B.1: Example of a PDDL domain[1] for the “Tower of Hanoi” puzzle.

[1]https://github.com/AI-Planning/classical-domains/tree/main/classical/hanoi

B.2 PDDL problem

```
(define (problem hanoi_problem)
    (:domain hanoi_domain)
    (:objects d1 d2 d3 d4 d5 peg1 peg2 peg3)
    (:init
        (clear d1)
        (clear peg2)
        (clear peg3)
        (on d1 d2)
        (on d2 d3)
        (on d3 d4)
        (on d4 d5)
        (on d5 peg1)
        (smaller d1 d1)
        (smaller d2 d1)
        (smaller d2 d2)
        (smaller d3 d1)
        (smaller d3 d2)
        (smaller d3 d3)
        (smaller d4 d1)
        (smaller d4 d2)
        (smaller d4 d3)
        (smaller d4 d4)
        (smaller d5 d1)
        (smaller d5 d2)
        (smaller d5 d3)
        (smaller d5 d4)
        (smaller peg1 d1)
        (smaller peg1 d2)
        (smaller peg1 d3)
        (smaller peg1 d4)
        (smaller peg1 d5)
        (smaller peg2 d1)
        (smaller peg2 d2)
        (smaller peg2 d3)
        (smaller peg2 d4)
        (smaller peg2 d5)
        (smaller peg3 d1)
        (smaller peg3 d2)
        (smaller peg3 d3)
        (smaller peg3 d4)
        (smaller peg3 d5)
    )
    (:goal (and (on d5 peg3) (on d4 d5) (on d3 d4) (on d2 d3) (on d1 d2)))
)
```

Listing B.2: Example of a PDDL problem[2] for the “Tower of Hanoi” puzzle.

[2] https://github.com/AI-Planning/classical-domains/tree/main/classical/hanoi

B.3 HDDL domain

```
(define (domain delivery_robot_domain)
    (:requirements
        :negative-preconditions
        :hierarchy
        :typing
        :universal-preconditions
        :method-preconditions
    )

    (:types PACKAGE ROOM ROOMDOOR)

    (:predicates
        (armempty)
        (rloc ?loc - ROOM)
        (in ?obj - PACKAGE ?loc - ROOM)
        (holding ?obj - PACKAGE)
        (closed ?d - ROOMDOOR)
        (door ?loc1 - ROOM ?loc2 - ROOM ?d - ROOMDOOR)
        (goal_in ?obj - PACKAGE ?loc - ROOM)
    )

    (:task achieve-goals :parameters ())
    (:task release :parameters ())
    (:task pickup_abstract :parameters (?obj - PACKAGE))
    (:task putdown_abstract :parameters ())
    (:task move_abstract :parameters ())
    (:task open_abstract :parameters ())

    (:method release-putdown_abstract
        :parameters (?loc - ROOM ?obj - PACKAGE)
        :task (release)
        :precondition (and (rloc ?loc) (holding ?obj) (goal_in ?obj ?loc))
        :ordered-tasks (and (putdown_abstract) (achieve-goals))
    )

    (:method release-move
        :parameters ()
        :task (release)
        :ordered-tasks (and (move_abstract) (release))
    )

    (:method release-open
        :parameters ()
        :task (release)
        :ordered-tasks (and (open_abstract) (release))
    )

    (:method achieve-goals-pickup
        :parameters (?loc - ROOM ?obj - PACKAGE)
        :task (achieve-goals)
        :precondition (and (rloc ?loc) (in ?obj ?loc) (not (goal_in ?obj ?loc)))
        :ordered-tasks (and (pickup_abstract ?obj) (release))
    )

    (:method achieve-goals-move
        :parameters ()
        :task (achieve-goals)
        :ordered-tasks (and (move_abstract) (achieve-goals))
    )

    (:method achieve-goals-open
        :parameters ()
        :task (achieve-goals)
        :ordered-tasks (and (open_abstract) (achieve-goals))
    )

    (:method finished
        :parameters ()
        :task (achieve-goals)
        :ordered-subtasks (and)
    )

    (:method newMethod22
        :parameters (?obj - PACKAGE ?loc - ROOM)
        :task (pickup_abstract  ?obj)
        :ordered-subtasks (pickup ?obj ?loc)
    )

    (:method newMethod23
        :parameters (?obj - PACKAGE ?loc - ROOM)
        :task (putdown_abstract )
        :ordered-subtasks (putdown ?obj ?loc)
    )

    (:method newMethod24
        :parameters (?loc1 - ROOM ?loc2 - ROOM ?d - ROOMDOOR)
        :task (move_abstract )
        :ordered-subtasks (move ?loc1 ?loc2 ?d)
    )

    (:method newMethod25
        :parameters (?loc1 - ROOM ?loc2 - ROOM ?d - ROOMDOOR)
        :task (open_abstract )
        :ordered-subtasks (open ?loc1 ?loc2 ?d)
    )

    (:action pickup
        :parameters (?obj - PACKAGE ?loc - ROOM)
        :precondition (and (armempty) (rloc ?loc) (in ?obj ?loc))
        :effect (and (not (in ?obj ?loc)) (not (armempty)) (holding ?obj))
    )

    (:action putdown
        :parameters (?obj - PACKAGE ?loc - ROOM)
        :precondition (and (rloc ?loc) (holding ?obj) (goal_in ?obj ?loc))
        :effect (and (not (holding ?obj)) (armempty) (in ?obj ?loc))
    )

    (:action move
        :parameters (?loc1 - ROOM ?loc2 - ROOM ?d - ROOMDOOR)
        :precondition (and (rloc ?loc1) (door ?loc1 ?loc2 ?d) (not (closed ?d)))
        :effect (and (rloc ?loc2) (not (rloc ?loc1)))
    )

    (:action open
        :parameters (?loc1 - ROOM ?loc2 - ROOM ?d - ROOMDOOR)
        :precondition (and (rloc ?loc1) (door ?loc1 ?loc2 ?d) (closed ?d))
        :effect (and (not (closed ?d)))
    )
)
```

Listing B.3: Example of an HDDL domain[3] for a delivery robot.

[3]https://github.com/panda-planner-dev/ipc2020-domains/tree/master/total-order/Robot

B.4 HDDL problem

```
(define (problem delivery_robot_problem)
    (:domain delivery_robot_domain)
    (:objects o1 o2 o3 o4 o5 - PACKAGE c r1 r2 r3 - ROOM d01 d13 d12 - ROOMDOOR)
    (:htn
        :ordered-tasks (and (task0 (achieve-goals)))
    )
    (:init
        (rloc c)
        (armempty)
        (door c r1 d01)
        (door r1 c d01)
        (door r1 r2 d12)
        (door r1 r3 d13)
        (door r2 r1 d12)
        (door r3 r1 d13)
        (closed d01)
        (in o1 r3)
        (in o2 r3)
        (in o3 r1)
        (in o4 r2)
        (in o5 r2)
        (goal_in o1 r3)
        (goal_in o2 r3)
        (goal_in o3 r3)
        (goal_in o4 r1)
        (goal_in o5 r3)
    )
    (:goal (and (in o1 r3) (in o2 r3) (in o3 r3) (in o4 r1) (in o5 r3)))
)
```

Listing B.4: Example of an HDDL problem[4] for a delivery robot.

[4]https://github.com/panda-planner-dev/ipc2020-domains/tree/master/total-order/Robot

C Tested Planning Tools

The following are some of the planners that were tested during the work on this thesis. Most of the tools were tested in a Debian 11 64-bit (amd64) environment, with a few exceptions.

C.1 Fast-Forward

Fast-Forward[1] [HN11] is a domain independent planning system, that can handle classical STRIPS planning tasks specified in PDDL.

The homepage offers several versions of its source code, in this case the patched version of FF-v2.3 by Robert Goldman is used. The downloaded archive contains a ready executable, which requires no additional dependencies, but since it is a 32-bit executable, it needs either a 32-bit OS or a 64-bit OS with added 32-bit architecture. Listing C.1 shows an example of using the tool to solve a problem and generate a plan.

```
$ ./ff -p path/to/directory/ -o domain.pddl -f problem.pddl
```

Listing C.1: Executing *Fast-Forward* to generate a plan.

The steps of the generated plan are output in the terminal, with some additional information.

C.2 PANDA

The *PANDA*[2] [HBBB21] project consists of multiple components. The *PANDA* planner, which is presented in this section, is a hierarchical planning system for solving hierarchical planning problems.

The homepage provides the source code, also found on GitHub[3], and a JAR executable, which is used here for testing. The JAR executable requires no additional dependencies and was tested on Debian 11 64-bit and Windows 10 64-bit. However, during testing it would produce plans only when using Java 8 and refused to work properly when using Java 11 or Java 19.

Information on the tool can be acquired by running the executable with the [-help] option. Listing C.2 shows the command for plan generation.

[1]https://fai.cs.uni-saarland.de/hoffmann/ff.html
[2]https://www.uni-ulm.de/en/in/ki/research/software/panda/
[3]https://github.com/galvusdamor/panda3core

```
$ java -jar PANDA.jar domain.hddl problem.hddl
```

Listing C.2: Executing *PANDA* to generate a plan.

The output is provided through the terminal, with the plan steps beginning after the "SOLUTION SEQUENCE" string. Besides the plan, the output contains additional information on copyright, configurations, parsing and compilation.

C.3 LPG-td

LPG-td[4] [GS02] is a planner that can solve both plan generation and plan adaptation problems. It is an extension of *LPG* (Local search for Planning Graphs) for handling the features of PDDL 2.2, while the previous tool version handles PDDL 2.1.

The executable code of *LPG-td* is found on the tool's homepage, with a 1.4 release compiled for Linux Ubuntu and a 1.0 release compiled for Linux Debian 3.0 and Windows 2000. The executable requires no dependencies, an example execution command is shown in Listing C.3. The [-n] option denotes the desired number of solutions.

```
$ ./lpg-td -o domain.pddl -f problem.pddl -n 1
```

Listing C.3: Executing *LPG-td* to generate a plan.

The tool displays problem solving process information in the terminal and the generated plan is saved in a ".SOL" file in the directory of the domain and problem files, which are used as arguments.

C.4 planutils

planutils[5] [MPSK22] is a library for developing, running, and evaluating planners. It can be utilized through Docker or installed and accessed through the terminal. The installation can be done via the PIP package manager[6], after which the system must be rebooted. Afterwards, an installation of apptainer[7] is required as well.

The list of planning tools, that can be installed with *planutils* can be found in its GitHub archive[8]. The installation command is shown in Listing C.4:

```
$ planutils install <planner_name>
```

Listing C.4: Executing *planutils* to install a planner.

A few of the planning tools, which were tested through *planutils*, are described in the following sections.

[4]https://lpg.unibs.it/lpg/
[5]https://github.com/AI-Planning/planutils
[6]https://pypi.org/project/planutils/
[7]https://github.com/apptainer/apptainer/releases
[8]https://github.com/AI-Planning/planutils/tree/main/planutils/packages

C.4.1 LAMA

LAMA[9] [RW10] is a classical planning system, built on the *Fast Downward* planning system. To execute *LAMA* via *planutils* requires the command in Listing C.5:

```
$ planutils run lama domain.pddl problem.pddl
```

Listing C.5: Executing *LAMA* through *planutils* to generate a plan.

The generated plan is output in the terminal, among other logging information, and saved as a "sas_plan.1" file at the location of the provided input files.

C.4.2 Fast Downward

Fast Downward is described in Section 4.4.2. Unlike *LAMA*, which employs it for plan generation, using *Fast Downward* directly through *planutils* runs into a problem. Besides the "planutils run downward" command, *planutils* allows only two arguments: for domain and problem files. *Fast Downward*, however, also expects a [-- search] option with a value, e.g., "astar(cegar())". Without those additional arguments no plan is generated.

C.4.3 Fast-Forward

A description of *Fast-Forward* can be found in Appendix C.1. The tool is executed through *planutils* with the command found in Listing C.6.

```
$ planutils run ff domain.pddl problem.pddl
```

Listing C.6: Executing *Fast-Forward* through *planutils* to generate a plan.

The generated plan is output in the terminal.

C.4.4 OPTIC

OPTIC (*Optimizing Preferences and TIme dependent Costs*)[10] [BCC12] is a temporal planner. It solves problems which involve plan costs determined by preferences or time-dependent goal-collection costs. The command to execute the tool through *planutils* can be found in Listing C.7.

```
$ planutils run optic domain.pddl problem.pddl
```

Listing C.7: Executing *OPTIC* through *planutils* to generate a plan.

Even though the tool generates a plan and outputs it in the terminal, the solving process does not terminate and must be interrupted manually. Whether the problem lies with the planner itself or with *planutils* has not been investigated.

[9] https://github.com/rock-planning/planning-lama
[10] https://nms.kcl.ac.uk/planning/software/optic.html

C.4.5 PRP

PRP (*Planner for Relevant Policies*)[11] [MMB12] is a fully observable non-deterministic planner. Information on the tool's usage, changes, relevant papers, etc., can be found its Wiki[12]. The command to run the tool via *planutils* is listed in Listing C.8.

```
$ planutils run prp domain.pddl problem.pddl
```

Listing C.8: Executing *PRP* through *planutils* to generate a plan.

During testing, the tool solved given problems and provided generated plans through the terminal, together with process logs.

C.5 myPDDL

myPDDL[13] [SK15] is a modular toolkit for developing and manipulating PDDL domains and problems. Using the tool requires installing Sublime Text[14].

Once the tool and Sublime Text are set up, PDDL stubs can be created by switching the "Syntax" setting inside the "View" dropdown to "PDDL". Afterwards, one needs to start typing, e.g., "domain", and to confirm the appearing suggesstion by pressing the *tab* key. Listing C.9 shows a PDDL domain stub generated by *myPDDL*.

```
(define (domain domain-name)
  (:requirements
    :typing
  )

  (:types
    subtype1 subtype2 subtype3 - object
  )

  (:predicates
    (predicateName ?x - object ?y - object)
  )

  (:action action-name
    :parameters ()
    :precondition ()
    :effect ())

)
```

Listing C.9: Example of a PDDL domain stub generated by *myPDDL*.

[11] https://github.com/QuMuLab/planner-for-relevant-policies
[12] https://github.com/QuMuLab/planner-for-relevant-policies/wiki
[13] https://github.com/Pold87/myPDDL
[14] https://www.sublimetext.com/docs/linux_repositories.html