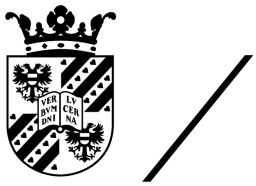
Automated Planning of Data Processing Pipelines

Key words: AI, Big Data, Data mining, Distributed Systems,

Mo Assaf, **supervisors:** Dr. Viktoriya Degeler, Mostafa Hadadian,



university of groningen

faculty of science and engineering

Bachelor thesis

Mo Assaf
Presenting for the degree of
Bachelor Computing Science

Faculty of science and engineering
University of Groningen
Netherlands
July 18, 2022

Abstract:

Large scale Big Data processing has become imperative as more data either structured or unstructured is being collected especially with the rise of IoT technologies and analytics. This brings demands for processing large chunks of data which brings many challenges such as real-time feedback. Distributed system computing is commonly applied to solve such challenges and often uses data processing pipelines to manipulate the data. Data processing pipelines are sequence of operations and transformations applied to data to achieve a desired task. Planning data processing pipelines is crucial because if not planned properly, it can increase risks and errors that can have consequences such as downtime. However, such planning is a non-trivial labour intensive task that requires consideration of many factors and constraints such as latency and computational resources. In this thesis, I analyse different aspects of data processing and AI planning to construct a theoretical framework that defines different orientations of planning as a scoping strategy. I also leverage the concept of Object-relational mapping (ORM) to easily generate Planning Domain Definition Language (PDDL) planning problems. I use the framework to develop an AI-powered data processing pipeline planner that utilises the Fast Downward planning system and PDDL ORM. The planner can be used to construct valid pipelines that achieve desired data tasks. Unlike other data pipeline planners, this planner can be easily extended to support dynamic aspects of the system making the planning of complex aspects such as feedback loops system possible, it can reduce errors by checking consistency and can improve the effectiveness of an organisation. The planner was tested against hard and simple problems producing results showcasing feasible performance for moderately complex pipelines.

Contents

1 Introduction & Motivation

Organisations have realised the importance of data and began utilising information systems and data management techniques for spotting and predicting trends, create data-driven decisions, and descriptive reporting [1]. Such information systems are deeply embedded within an organisation and create many challenges due to integration costs, managerial challenges, large scope with high risks, and complexity. However, successful integration can be rewarding and crucial for developing a competitive advantage as these challenges make it difficult to imitate [2]. Therefore, various data are frequently collected which brings challenges and demand for large-scale data processing. Major challenges (see section 4.2) constitute data heterogeneity, storage, integration, resource management, real-time demand, and many more [3]. For example, traditional Online Transaction Processing (OLTP) databases have failed to provide analytical capabilities and so Online Analytical Processing (OLAP) environments were developed to satisfy the demand for analytics. However, such analytical platforms require expensive integration techniques such usage of Extract-Transform-Load (ETL) pipelines as they utilise different data management techniques to serve their purpose [4].

When addressing such challenges, appropriate data processing techniques are required. Distributed computing is a common way to provide scalability and performance. A mainframe approach is less cost-effective and so the data is distributed to many machines to be processed in parallel. Such processing techniques commonly involve usage of distributed data processing pipelines [5]. Data pipelines are chains of generic steps of transformations and operations sequenced from a source (e.g., OLTP database) to a sink (e.g., datawarehouses, visualisation platforms, etc.). Data pipelines aim to automate flow of data, reduce human error, and improve performance [6].

One of the common issues with distributed data processing is the fixed nature of the data processing platforms. A data pipeline may need to be updated due to dynamic external factors such as changes in the business goals, changes in the environment, parameter tuning, or structural changes in data. Therefore, two scenarios may occur. The first scenario is terminating the pipeline and re-instantiating the process causing downtime which is not always acceptable. The other scenario involves instantiating a new process and replacing the existing process once it has reached the desired progress. However, this scenario requires the precondition that computational resources are available and time is affordable. Furthermore, the dynamic changes must be verified and checked for consistency [5], [7]. Hence, planning data processing pipelines is non-trivial and must consider many factors which can be external or internal (e.g. latency, putthrough, etc.).

The planning constitutes finding an optimal data pipeline that supports transformation of heterogeneous sources of data with minimal resources, lowest latency, and highest scaling potential. Each chained component within a data pipeline manipulates the data passed through it which can be regarded as an action that causes a deterministic state change or an effect in classical planning terms [8]. Hence, the ultimate goal is to find an optimal chain of components that will yield the final desired state. This can be done by defining a domain of the data manipulations and rules that will be used by the planner. The final state can be formulated by given criteria such as max latency, data format, max storage, etc. Planning is a broad fundamental problem of AI [9] and there are many planning techniques (see section 4.1) that can be used to evaluate such problem formulations based on logical definitions. To address this problem, I propose a theoretical framework that combines AI planning concepts and data pipeline concepts. I also use this framework to develop the automated pipeline planner. I test the planner and showcase a real-life use case where machine learning inference pipelines are automated to perform certain machine learning tasks.

2 Contribution

Currently, automation of data pipeline planning is a relatively new topic and is underdeveloped. I create an AI-powered data processing pipeline planner that can generate valid pipelines. Such a planner can be used to prevent downtime, reduce human error, and improve the organisation's effectiveness by enabling such parties to be more flexible with external or internal dynamic changes. It can be used to help automatically deploy pipelines without relying on experts or certain internal organisational dynamics. A previous planner [5] was developed that is used to automatically reconfigure running data pipelines but does not consider dynamic aspects such as feedback loops. However, unlike this planner, I consider dynamic aspects of the system that create complex interactions such as feedback loops. In addition, I created a theoretical framework based on the relevant background that connects concepts from AI planning with concepts from data processing pipelines. The framework pins different orientations of planning such as resource allocation and achieving certain data tasks. Finally, I was inspired by the concept of Object-relational mapping (ORM) where objects within an object-oriented programming language are used to map data between two systems [10]. I leverage this concept to create mappings between Planning Domain Definition Language PDDL specifications and object-oriented programs. This creates synergy since the benefits of object-oriented programming languages can be utilised to provide useful abstractions of PDDL making the development of domain-specific planners less tedious and more straightforward.

3 Research problem

As mentioned, planning data processing pipelines is crucial and can be labour intensive since there are many challenges that come with processing and managing data (see section 4.2) [6]. Therefore, the planner must be capable of dealing with constraints that are useful to tackle some of these challenges. In addition, AI planning requires the domain to be well-defined and needs to be descriptive of the problem and the relevant constraints with respect to such challenges. Lastly, such problems are dynamic since resources need to be allocated to the pipeline and many variables could be affected as the pipeline is executed. Temporal dynamics play a role since pipelines contain components that run for periods of time and such. Therefore, our research questions are:

- How can data processing pipelines be planned automatically with respect to given constraints?
- How can data pipeline problems be represented in an AI planning domain?
- How can we plan complex dynamics such as feedback loops?

4 Background

To consider how data processing pipelines can be planned, the definitions of data processing and AI planning and the relevant background must be taken into account. The following background forms the backbone of the proposed theoretical framework.

4.1 AI Planning

Planning is one of the most challenging but fundamental areas of research in AI. Planning occurs within some context or domain that has rules which shape the attainability of different states and applicability of different actions. An agent might desire to achieve a specific goal within a given domain but does not know what specific actions are available and in what sequence they might be applied. Hence, planning requires knowledge of the domain (e.g., available actions, when the actions can be applied, the rules of the context, etc.) and the capability of looking ahead by foreseeing the effects of actions.

Planning occurs at a given initial state within a given domain and the goal state might be attainable by applying a sequence of actions from the initial state. Therefore, the gap between the initial state and the goal state can be regarded as the **planning problem** and the sequence of actions leading to the goal as the final plan. Furthermore, planning can be considered as a problem of search of actions that satisfy the goal from the initial state or an optimisation problem if criteria or metrics were considered [11]. Such problems are seen as hard problems due to their exponential nature and often many search heuristics are applied and in many cases, memory storing visited states and information to prune the search [8], [12].

Actions in a domain can only be applied if the current state allows the action within the domain, meaning an action has a precondition that must be satisfied for the action to be applicable [8]. Once an agent applies an action the preceding state is affected and is transitioned into a new state. The difference between the new and old states is called an effect and it can be fully deterministic, or non-deterministic [9].

There are different types of domains and natures of planning. Classical planning uses first-order-logic FOL that describes entities within a domain using predicates and logic. Effects in such domains are fully deterministic and constitute add and remove lists of predicates. A predicate is false if it does not exist in the domain and so the initial state is a list of positive predicates [8]. In addition, a domain allowing numeric fluents may contain numeric variables which can be affected by actions or events in either a linear or non-linear fashion. Such domains are dynamic and can model more complex transitions. Temporal domains consider temporal logic meaning predicates can be true for a certain period, actions can have durations, and different events might occur at specific or relative moments [9], [13].

In classical planning, a **planning domain** constitutes a set of propositions $L = \{p_1, p_2, ..., p_n\}$ with a state-transition system defined as $\Sigma(S, A, \gamma)$, where $S \subseteq 2^L$, A is a set of available actions, and γ is the state-transition function that given an applicable action $a \in A$ and a state $s \in S$, $\gamma(s,a) = (s - effects^-(a)) \cup effects^+(a)$ [13]. A negative effect or a remove-list ($effects^-(a)$) can be represented as a list of propositions that will be removed from the state when the action a is applied. The positive effect or an add-list ($effects^+(a)$) is essentially the same as the negative effect but instead, the list of propositions are added to the state. Therefore, applying an action a results in the transition from the state from s to $(s - effects^-(a)) \cup effects^+(a)$. However, an action a represented as a tuple $(precond(a), effects^-(a), effects^+(a))$ can only be applied if the precondition of the action holds within the current state s, in other words, $precond(a) \in s$. Otherwise, if not applicable, then $\gamma(s,a)$ is not defined, and such a transition is not allowed. The **planning problem** as mentioned above, is the gap between the initial state and the goal. The classical planning terms, a planning problem can be defined as $\mathscr{P} = (\Sigma, s_0, g)$ where $s_0 \in S$ is the initial state and g is a set of propositions such that the state must hold for it to be the goal state [13].

4.1.1 PDDL

Planning Domain Definition Language (PDDL) is widely used to describe the domain and the planning problem within that domain. PDDL can be used to define predicates, objects, actions, functions, metrics, and many other domain features. There are different versions of PDDL that model different aspects of domain definition (e.g., numeric fluents, temporal, etc.)[11].

PDDL1.0 considers classical planning and does not consider any temporal or dynamic aspects. Propositions can be defined with predicates and formulas. The domain can be specified as follows:

```
(define (domain <name>)
  (:requirements :<req0> :<req1> ... :<reqn>)
  (:types <t0> <t1> ... <tn>)
  (:predicates
    (<pred0> <arg00> <arg01> ... )
    ...
    (<predn> <argn0> <argn1> ... )
  )
  (:action <action0>
    :parameters (?a0 - t0 ?a1 - t1 ...)
    :precondition (<formula>)
    :effect (<effect>)
  )
  ...
)
```

The planning problem can be specified by stating the initial state as a list of grounded predicates and specifying the goal as a formula. The objects are entities that exist within the domain and typically have a type.

```
(define (problem <name>)
  (:domain <name>)
  (:objects
    (obj1 - <type>)
    (obj2 - <type>)
    ...
    (objn - <type>)
  )
  (:init
    <grounded pred0>
    <grounded pred1>
    ...
    <grounded predn>
  )
  (:goal (<formula>))
)
```

PDDL2.0 introduces capability of modelling more complex dynamics, numerical aspects, temporal aspects, and scheduling. For example, it introduces functions which resemble variables that can be updated in a linear or non-linear way. They are also initialised with values in the initial state and are used to specify metrics. Thereby, this version of PDDL can be used to specify optimisation problems.

```
(define (domain <name>)
  ...
  (:functions
    (var0 ?a0 ?a1 ... ?an)
    ...
  )
  ...
)

(define (problem <name>)
```

```
  (:domain <name>)
  ...
  (:init
    (= (var0) <val0>)
    (= (var1) <val1>)
    ...
    (= (varn) <valn>)
  )
  ...
  (:metric <maximize|minimize> <numeric expression (e.g., var0 - varn)>)
)
```

Hence, if one wants to use PDDL to define planning problems, they must identify what aspects should be considered and whether PDDL supports such aspects because problems have different natures and if the given PDDL version does not support such natures then the problem cannot be represented. For example, if one wants to plan an optimal scheduling plan for airline takeoff times, then if the PDDL version does not support scheduling capability then such a problem can be represented using this version.

4.1.2 Available planners

Many domain-independent PDDL planners had been developed which support different versions of PDDL. The FF planner one of the most successful planners, estimates goal distance by omitting the delete lists produced by the actions [14]. There are other algorithms such as the GraphPlan algorithm which uses planning graphs that create useful heuristics [8]. Many of the planners rely on search algorithms such as A*, Nondeterministic or deterministic search, And/Or Search, alpha-beta pruning, and many more [8], [9]. SMTPlan+ supports the usage of events and processes [15] from PDDL3.0 therefore, it is a suitable planner if one wants to plan complex dynamics where events occur at given times and actions are completely dynamic and have a duration. Metric-FF [16] supports PDDL2.1 which allows numeric fluents and state variables and so is a great planner for solving optimisation problems or dynamic problems as mentioned in the previous section. Fast Downward [12] supports PDDL1.0 and action costs with a wide range of heuristics making it suitable for classical planning problems but is not capable of solving optimisation problems, scheduling problems, or problems where dynamic aspects are involved. Furthermore, it also does not support negative preconditions and so negations must be represented with predicates (e.g. lightTurnedOff and lightTurnedOn). In summarise, if one wants to pick a planner, they must find a planner that is efficient enough and also supports the appropriate version of PDDL so that the problem at hand can be represented and solved.

4.1.3 Bridging between Deep Learning and Planning

One of the problems of planning is that it requires the information to be formulated as logical models. For example, images can contain facts which can be translated to PDDL or predicates. Deep learning models can be used to extract features and facts which then can be translated to a form compatible with classical planning [17], [18].

4.2 Big data challenges

There are many challenges currently being faced which are relevant to consider when planning data processing pipelines [3], some examples are:

- **Data heterogeneity**: as more types of data are collected either structured, semi-structured, or unstructured data [19]. Deep learning (DL) and machine learning (ML) have played an important role in processing unstructured data. These techniques can handle unstructured data such as text and images well and can extract relevant features. For example, sentiment analysis is a common task that can be used to determine customer impressions on products [20], [21].
- **Data integration**: analytics environments utilise different data structures and techniques which require translation and integration from previous data storage environments. For instance, data warehouses that are developed for OLAP purposes, use star schema, data cubes, and special operations which require the data to be in a format that differs from the format of data stored in traditional OLTP environments [4].
- **Storage**: the pipelines for instance need the intermediate results in between. How can the data be stored reliably to allow quick retrieval?
- **Computational complexity**: operations that data pipelines perform are complex and expensive which can increase latency. Therefore, it is difficult to achieve real-time applications. Furthermore, appropriate infrastructure is also required to support such computations.

4.3 Data pipelines

A data pipeline is a chain of transformation where the output of a processing step becomes the input of another. Initially, the data typically originates from a source and at the end of the pipeline, the data is loaded into a sink or a destination typically a visualisation tool or storage [6]. Data processing pipelines have many use cases and can help with dealing with some of the data challenges. For instance, ETL pipelines extract, transform, and load data to data warehouses attempting to solve integration issues between Online Analytical Processing OLAP and Online Transaction Processing OLTP environments [4]. Data pipelines can also be used to deploy machine learning inference models and can help analyse large chunks of unstructured data [20]. Data pipelines can be complex and dynamic and can contain steps such as parallel computations and reduce operations to process large streams of data [22].

5 Theoretical framework

To approach the research problem, I constructed the following theoretical framework by analysing the concepts from the background.

5.1 Data processing pipelines as systems

Data processing pipelines (DPP) are sequences of operations and transformations applied to batches of data to achieve the desired data format from a data source to a sink (from section 4.3). Although this is a sound definition, it provides a narrow view and does not consider the entire context which is necessary for formulating planning problems. Thus, I propose to take a more broad view of data processing pipelines by regarding them as components interacting with other components in a system. A data processing pipeline can be viewed as a system composed of the following elements (see figure 5.2).

5.1.1 Processing step

A processing step shown in figure 5.1 is an abstract independent basic component with memory, allocated resources, and an internal state that accepts a data object as an input and outputs a new data object by manipulating the input object to achieve a certain task. A step can be invoked by the orchestrator and contains its state. A step when invoked runs for a dynamic period and dynamically mutates its internal state. For example, a processing step can be a process running an executable script stored on a local machine, taking input from a file that has been written into by another process and storing the output in a different file. It can also be an inference REST API parsing JSON input from its network buffer and writing the JSON response into its network card after the processing is completed. Such APIs are stateless and have empty state objects but can still update the global state. Hence, a sequence of steps in this context resembles a data pipeline following the definition from section 4.3.

5.1.2 Data object

A data object is an abstract basic component representing data that has a format. The data object may represent meta information for retrieval such as image URLs as well as internal variables. They can be instantiated by processing steps and can be passed from the output of one step to an input of another step.

5.1.3 State object

A state object is an abstract basic component representing a dynamic state that can be altered. A state object can be private to a step or shared globally by all steps. For instance, a process loaded into RAM has its local variables and private memory and cannot be accessed directly by other processes but it can access the global memory which is shared by other processes to communicate or access global information. For instance, a global state object can resemble the pipeline's configuration parameters. The state object could represent the status of the step (e.g. failed, terminated).

5.1.4 Resources

The steps need to have their resources allocated to them which might be available at a given moment. The resources constrain and shape different possibilities, for example, if there is only one compute node available then the orchestrator cannot apply different steps in parallel.

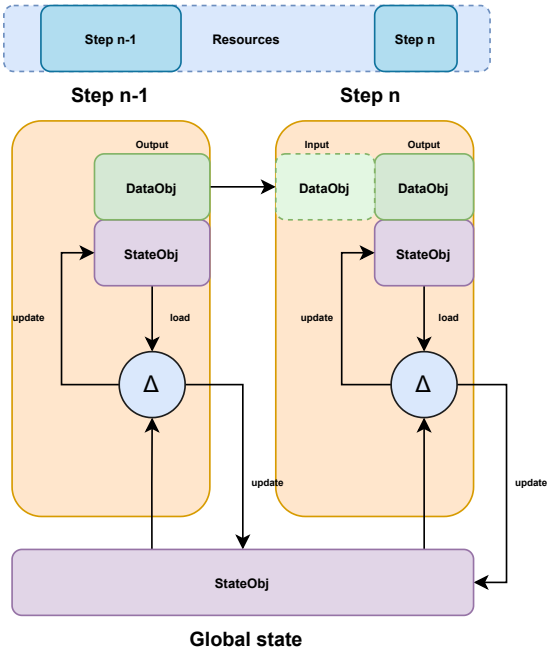


Figure 5.1: the processing step n has its own internal state and shares a global state with other steps. When step n runs it performs an operation Δ on its internal state and the global state and updates them; it takes in a data object from step $n-1$ as an input and writes the output data object into its output buffer. Lastly, all steps including step $n-1$ and step n have their own allocated resources.

5.1.5 Orchestrator

An orchestrator is an abstract basic component responsible for orchestrating the dynamics of the system such as the execution of steps, passage of data between different steps, allocation and management of resources, and parallelism. A pipeline orchestrator can run steps in parallel if applicable and can invoke any available step by loading a data object into its input memory (see figure 5.2). For example, processes can be dispatched by the operating system of the machine. The operating system is responsible for managing the file system and the resources of the processes. In this case, the operating system acts as the orchestrator and program processes as processing steps.

5.2 Planning data processing pipelines

Planning as mentioned involves filling the gap between the initial state and the goal and data processing are series of steps that achieve a certain task. Since we need to make a connection between data pipelines and AI planning, I make the analogy between a planning domain and a data processing pipeline system (see table 5.1).

5.2.1 Planning orientations & scoping

Furthermore, as mentioned in section 4.1, there are different natures of AI planning, thus, it is essential to assess the static and dynamic aspects of the system. The system has static aspects mainly the data objects being loaded through out the system from one module to another. For instance, a data object can only be loaded an input of a step if it follows the required input format. The system also has dynamic aspects mainly the available resources and the state objects. State objects are essentially a composition of dynamic variables updated throughout the events of the system. The resources are also composed of dynamic variables such as available memory, number of available compute nodes, etc. Furthermore, steps run for periods of time and if steps are run in parallel then it is likely that the latency is reduced but more resources are allocated in that period of time, and so temporal dynamics exist within such a system. Such a system can be complex and can make planning in such a domain difficult, so I define different orientations of planning as a scoping strategy (see table 5.2). When planning data pipelines, one can consider many aspects in the planning and two major aspects are the data tasks that need to be carried out and the resources that must be allocated to feasibly execute the pipeline. Hence, by identifying these two aspects, I define two orientations of data pipeline planning. *Task-oriented planning* mainly concerns achieving certain data tasks, the sole purpose is to obtain a pipeline that can perform the desired tasks and assumes that resources are available. In contrast, *resource-oriented* only concerns that resources that need to be allocated, it requires the pipeline to be known initially, and the sole purpose is to obtain a feasible execution plan for the given pipeline.

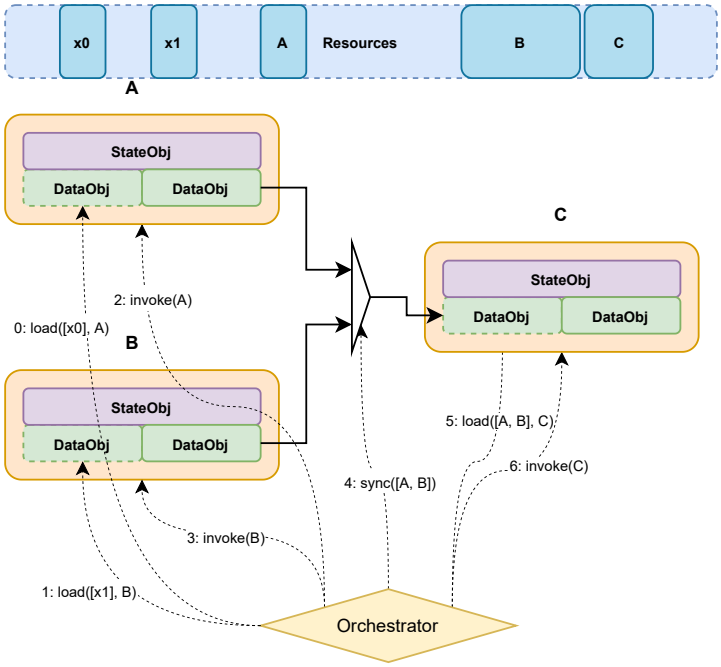


Figure 5.2: the orchestrator in this example loads the inputs of A and B from other steps x_0 and x_1. Then it invokes A and B in parallel then introduces a barrier for a barrier A and B and synchronises them. Finally, it loads the outputs from A and B to the input of C and invokes C.

Data processing pipeline system	AI Planning
Orchestrator	is the agent acting in the domain attempting to find a plan by executing its instructions.
Orchestration instructions	are the actions available in the domain applied by the orchestrator with certain preconditions. For example, the invoke instruction can only be applied when there is at least one compute node available.
Processing steps	are objects related to an action that accepts a certain data object with a required data format and results into an effect of loading a new data object with a format and the effect caused by Δ.
Data formats	are rules within the domain used to model the preconditions of processing step actions.
State object	is a set of variables the can be updated dynamically in either a linear or non-linear by effects or events. Such variables are updated dynamically in a linear or non-linear fashion (Δ operation of a step).
Resources	are rules that constrain the aforementioned actions and a set of variables that decide such constraints.
The data processing pipeline system	is the entire planning domain.
Data pipeline problem	is a goal resulting from applying the sequence of orchestration instructions starting from the initial state of the system. The initial state of the system can be composed of the available processing steps, available resources, the data objects already loaded in some steps, etc. The desired final state can constitute an achieved data format or task, or it can be an execution plan (see table 5.2).

Table 5.1: connecting concepts from AI planning and data processing pipeline systems.

	Resource-oriented planning	Task-oriented planning
Precondition	Pipeline known	Pipeline not known
Purpose	Creating a feasible execution plan of the pipeline satisfying the given criteria.	Creating a pipeline that satisfies the desired data tasks and formats.
Typical use case	Deploy pipeline X using infrastructure Y that satisfies criteria Z.	Find a pipeline that takes input from source X and performs task Y on the data and satisfies criteria Z.
Orchestration instructions	Parallel instructions (e.g., synchronise steps $i \in \{1,5,29\}$), resource allocation instructions (e.g., allocate x from memory M_i to step i), invocation instructions (e.g., run step i).	Data instructions (e.g., load output object from step $i-1$ to step i), invocation instructions (e.g., run step i).
Planning process	Evaluation of available resources, considers the dynamic and temporal aspects of the domain since resources are allocated for a period of time.	Evaluation of the data format and dynamic state preconditions of arbitrary steps before applying them.
Planning nature	Dynamic, temporal, static.	Dynamic, static.
Example	Deploy an existing pipeline using an existing compute infrastructure in such a way that it maximises putthrough, minimises the latency and power usage.	Find a pipeline that performs a car object detection on coloured images that minimises the total number of parameters and maximises the sum of accuracies equally weighted.

Table 5.2: scoping planning by resource and task orientations.

5.2.2 Modelling dynamic aspects

Dynamic aspects concern the non-static aspects of the domain. This entails the presence of variables within data objects and state objects either external or internal just as mentioned in section 5.1.3. Many dynamics that may be involved such as linear or non-linear updates to local variables in a step or feedback loops. Feedback loops are essentially data objects being loaded from the output of a step to the input of the same step. This implies that for such a feedback loop to occur, the input and output data formats must match and the step must not change the data format of the input, otherwise the formats will mismatch and the load action will not be possible. This appears to be useless as the step does not change the data format within the data objects, however, it does not say anything about the state objects. Therefore, such loops would only make sense when a step updates a state object because the effect will no longer be unique. To elaborate, consider a non-dynamic classical planning domain where a light can be turned on or turned off. The action of turning on the light will always yield the same effect, i.e., the light is on. Hence, applying this action multiple times will always yield the same effect of having the light on and so the action is redundant. In contrast, if we were to make this domain dynamic, an action to lower the power supply by a fixed amount can be added. Applying such action many times will eventually turn off the light as the power supply is no longer available for the light. This by analogy concludes that the same follows for DPPs where running a step is a dynamic action (e.g., lower supply).

For example, suppose one has a processing step that implements a language model that takes a word as an input and predicts the next word. The input and output have the same data format as they are both words but one might desire to generate N words. Therefore, the step must be applied N times to generate N words and also has to keep track of the numbers being generated, it also must take in the last word generated and not a random word. Thus, the variable *nwords* tracks the number of words generated so far, and the variable i belonging to a data object is used to check if it was the last generated word. The initial state is the state where the first word is available and the global variable regarded as the input parameter is set to zero (i.e., $nwords = 0$). The goal is that N words should be generated (i.e. $nwords == N$). This example can be shown in figure 5.3; note that the data types of the input and output data objects do not change and the variables i and *nwords* are dynamically updated.

Lastly, as mentioned in section 4, we must assess the nature of the problem. For example, if we were to model this system using PDDL1.0, then we cannot model such dynamic aspects as PDDL1.0 only considers classical planning. PDDL2.1 introduces functions that resemble variables that can be used to model the state objects and so a planner (e.g. MetricFF) that solves PDDL2.1 problems will be appropriate. Furthermore, PDDL2 introduces the capability to add duration to actions that can be used to model delays or latency. PDDL2.1 can also be used to create optimisation problems for data pipelines by using the metric directive, for example, in resource-oriented planning the expression $totalRAMused + latency + ncomputeNodesUsed$ can be minimised to use the least amount of resources. Consequently,

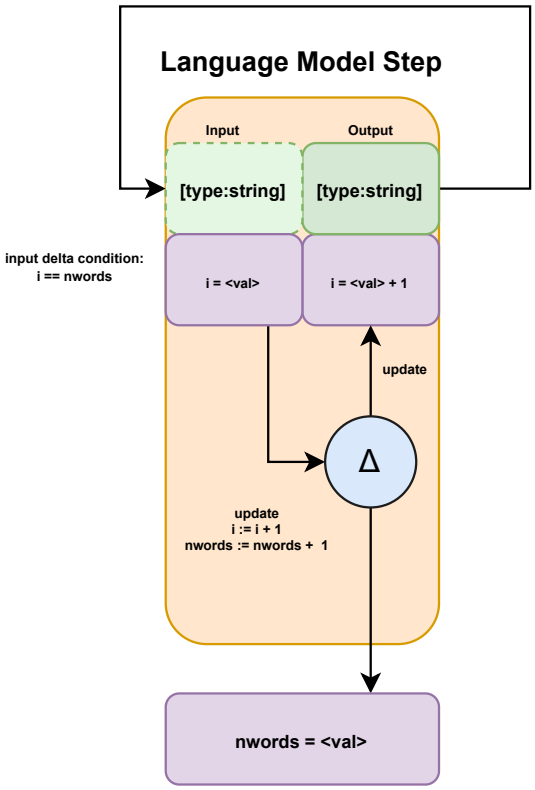


Figure 5.3: Example of a feedback loop where a step predicts the next word given a word. It updates the number of words and the variable *i*.

deciding what aspects are relevant for the use case helps with choosing the appropriate PDDL version.

6 Methodology & Design

I consider task-oriented planning and do not consider resource-oriented planning, I also do not model dynamic aspects because of time constraints. I created an algorithm for the system that carries out parallel opportunity discovery to provide suggestions for parallel computations. I use the theoretical framework as guidance for defining a descriptive planning domain and creating logical definitions. I leverage the concept of ORM where objects can be used to generate queries and create mappings between two type systems [23] (see figure 6.1). ORM is commonly used to map classes or objects within a programming language to database records where fields of a class represent fields in a table such as the ID of the entity. I was inspired by this concept to do the same but with classes mapped to PDDL specifications. I use an object-oriented programming language to generate PDDL problems and create one-to-one mappings between PDDL and objects (see section 7.1.2 for more elaboration). This technique of creating a layered system utilises the benefits of object-oriented programming and abstraction to make the implementation of the planning domain in this context less tedious. I use these PDDL objects to implement the logical definitions provided in this section. Finally, I use an external planner that accepts the created PDDL problems and translate the resulting actions back to orchestration instructions.

6.1 Defining a task-oriented planning domain

In this domain, I only consider data objects, processing steps, run instructions, and load instructions. I assume that data objects are objects composed of fields that can be addressed using a key. Each field has a signature that describes the data. For example, a data object containing an image field and a table can be described as follows:

```
{
  "image_field": {
     "key": "$.input.image",
     "signature": ["type:image", "image:nchannels:3"]
  },
  "table_field": {
```

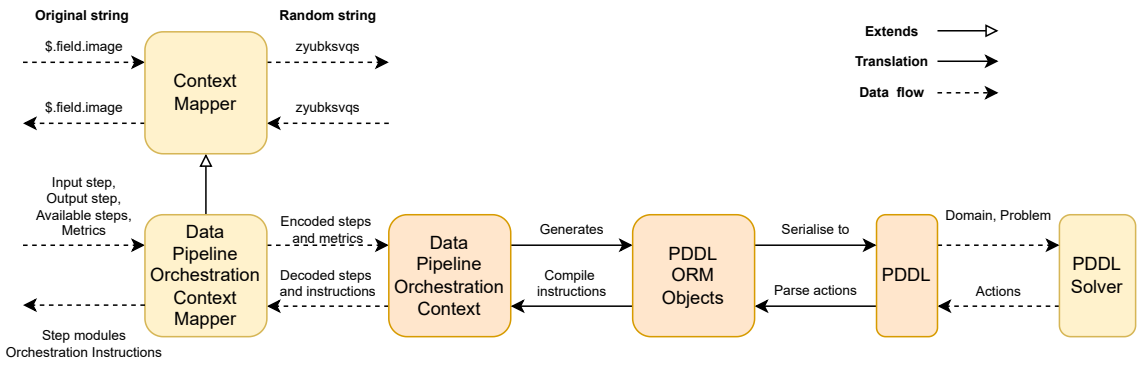


Figure 6.1: Demonstration of how PDDL ORM can be leveraged to construct layers of abstraction. Processing steps are specified to the data pipeline context which converts all processing steps into the relevant PDDL objects. These PDDL objects are used to generate PDDL problems that will be solved by external AI planners similar to how SQL ORM can be used to generate SQL queries to be executed by a DBMS. Finally, the process is reverted; actions are parsed into PDDL objects which contain the necessary information to compile orchestration instructions. The context mapper is used to make the data anonymous and remove special conflicting characters.

```
        "key": "/mnt/c/users/Mo/Result/extracted_users.csv",
        "signature": ["type:csv", "csv:ncols:2"]
    }
}
```

Keys express how the data can be accessed; in this example, the $ signifies the root of a JSON object. Signatures include static information that can describe the format of the data, the properties, adjectives, versions, and other attributes. Keys belong to certain steps and can be loaded to either an input or an output data object of some arbitrary step. A signature is a list of strings belonging to arbitrary keys each resembling a predicate. Hence, I define the following predicates:

Predicate	Description
has_key(s: step, k: key)	A step s has a key k.
key_is_input_of(s: step, k: key)	A key k in input of step s.
key_is_output_of(s: step, k: key)	A key k in output of step s.
key_loaded(s: step, k: key)	A step s has a key k loaded.
key_has_signature_s(k: key)	A step k has an arbitrary signature s.
is_step_s(t - step)	t equals s

6.1.1 Designing data signatures & considerations

The data signatures are defined by the application owner or the user and so the responsibility of defining appropriate signatures is delegated to them. Using arbitrary data signatures prevents the system from being application-specific allowing it to be usable in different domains.

Data objects are abstractions of instances of data. This implies that a signature must describe a certain domain of data instances and must not describe a unique instance unless needed. For example, an attribute signature such as "image:size:40mb" would restrict images to be exactly fourty megabytes. Thus, if a step was taking this data object as input, then it will not accept images with arbitrary sizes and only accept 40MB images. In addition, signatures must not be conflicting such as "image:type:grayscale" and "image:nchannels:3". In other words, the two signatures S_1 and S_2 are conflicting and cannot be used as a requirement for a field since:

$$\frac{S_1 \rightarrow C \wedge S_2 \rightarrow \neg C \quad (S_1 \wedge S_2) \equiv signature(field)}{\bot}$$

They also must not be redundant as it would hinder the performance and can make the signatures less descriptive and less specific, for

example, if $S_1 \implies S_2$ and the input requires a field with signature $[S_1, S_2]$:

$$\frac{S_1 \to S_2 \quad S_1}{(S_1 \wedge S_2) \equiv signature(field)}$$

Hence, the signature can be simply reduced to $S_1 \equiv signature(field)$.

Finally, the signatures must be specific enough for the application use case. For example, if a digit classifier only accepts gray-scale images, then the signature "type:image" alone is not enough. This will cause the step to fault when the plan is executed with a coloured image of three channels.

6.1.2 Action definitions

Lastly, the orchestration instructions (i.e., load instructions, and the run instruction) must be defined. Instructions as mentioned in section 5.2 are simply actions applied by the orchestrator agent. I make the following definitions to formulate the actions.

field_signature$(f : field, k : key) :=$
$And([$**key_has_signature_{s}**$(k) \| \forall s : s \in field.signature])$;

dataobj_belongs_to_step$(obj : dataobj, s : step) :=$
$And([$**has_key**$(s, field.key) | \forall field : field \in obj])$

dataobj_loaded_to_step$(obj : dataobj, s : step) :=$
$And([$**key_loaded**$(s, field.key) | \forall field : field \in obj])$

dataobj_input_of_step$(obj : dataobj, s : step) :=$
$And([$**key_is_input_of**$(s, field.key) | \forall field : field \in obj])$

dataobj_output_of_step$(obj : dataobj, s : step) :=$
$And([$**key_is_output_of**$(s, field.key) | \forall field : field \in obj])$

I assume that data objects act as buffers and are not simply passed around, but rather the contents of the data objects are copied form one step to another if the signatures of the fields match. Each step has its own load field and run actions, a field can only be loaded from one step to another if both steps own the keys of the fields and the output is available at the step being loaded from. Once a step is run, the input object is unloaded making sure that the step is not executed again with the same input, this also ensures that other steps get a chance to load their output into this step.

I represent a nameless action as follows:

$Action < [p_0 : t_0, p_1 : t_1, ..., p_n : t_n] > (precondition) => effect$

I define the following actions:

load_field_to_step$(s - step, f - field) :=$
$Action < [k_0 : key, k_1 : key, s_0 : step, s_1 : step] > ($
Output condition s0
has_key$(s_0, k_0) \wedge$
key_is_output_of$(k_0, s_0) \wedge$
key_loaded$(s_0, k_0) \wedge$
field_signature$(k_0, field) \wedge$
Input condition s_1
has_key$(s_1, k_1) \wedge$
key_is_input_of$(k_1, s_1) \wedge$
field_signature$(k_1, field) \wedge$
Action specific to this step
is_step_{step}(s_1)
) => (
key_loaded(s_1, k_1)
)

run_step(s - step) :=
$Action < [s_0 : step] \cup [k : key|$
$\forall k : k \in step.input.keys \cup step.output.keys] >$ (
Action specific to this step
is_step_{step}$(s_0) \wedge$
Input
dataobj_loaded_to_step$(step.input, step) \wedge$
dataobj_belongs_to_step$(step.input, step) \wedge$
dataobj_input_of_step$(step.input, step) \wedge$
Output
dataobj_belongs_to_step$(step.output, step) \wedge$
dataobj_output_of_step$(step.output, step) \wedge$
Delta condition
delta_condition$(step.condition, step)$
) => (
dataobj_loaded_to_step$(step.input, step) \wedge$
dataobj_loaded_to_step$(step.output, step) \wedge$
delta_effect$(step.delta, step)$
)

To define the planning problem, we need to define the initial state and the goal. The initial state constitutes the input step loaded with initial data objects in its input. Therefore, the initial state contains the following:

$And([$**field_signature**$(field, field.key)|\forall field : field \in input\_obj]) \wedge$
dataobj_belongs_to_step$(input\_obj, input\_step) \wedge$
dataobj_loaded_to_step$(input\_obj, input\_step) \wedge$
dataobj_input_of_step$(input\_obj, input\_step) \wedge$
dataobj_output_of_step$(input\_obj, input\_step)]$

To formulate the goal, we need the desired output object to be loaded into the output step. The output needs to obtain the desired output object from another step at the end of the pipeline. Therefore, the planner will have to find a sequence of orchestration instructions that will yield the signatures of the output object. The goal can be defined as follows:

$And([$**field_signature**$(field, field.key)|\forall field : field \in output\_obj]) \wedge$
dataobj_belongs_to_step$(output\_obj, output\_step) \wedge$
dataobj_loaded_to_step$(output\_obj, output\_step) \wedge$
dataobj_output_of_step$(output\_obj, output\_step)$

Finally, these formulations of actions and predicates are instantiated as PDDL objects which are then serialised to their corresponding PDDL equivalent definitions. Mapping is carried out between the input, output, and available steps to the required objects. The PDDL domain and problem are passed to a planner to be solved and thus obtaining lists of actions that are translated back by reverting the mapping process. Furthermore, the planners generally do not accept special characters like a dollar sign making specification of keys, signatures, and other attributes difficult. In addition to the previous problem, the planners are external and cannot be relied on to be secure. Hence, to solve these problems, I use a context mapper (see figure 6.2) that maps string values to new random codes. The context mapper makes the passed information anonymous and also does not use special keys. The mapper is used to convert back and forth between codes and the original values.

6.2 Discovery of parallel opportunities

The planner finds a list of instructions such as load and run instructions. However, the planner does not consider partially ordered plans. This entails that the planner does not consider parallel possibilities. Therefore, this responsibility is delegated to the orchestrator. The orchestrator recognises the independence of steps by considering the dependencies between them. For example, in figure 5.2, steps A and B need independent steps x_0 and x_1 and so can be executed in parallel. On the other hand, step C needs steps A and B and has to wait for both steps, so the orchestrator must synchronise steps A and B before C is invoked. I implemented the following algorithm that implicitly converts the instructions into a dependency graph based on the parameters of the instructions and constructs parallel instructions respectively. Both benchmarks were executed in a Jupyter notebook and are timed using a function that creates time stamps at the start and the end of the execution to compute the time.

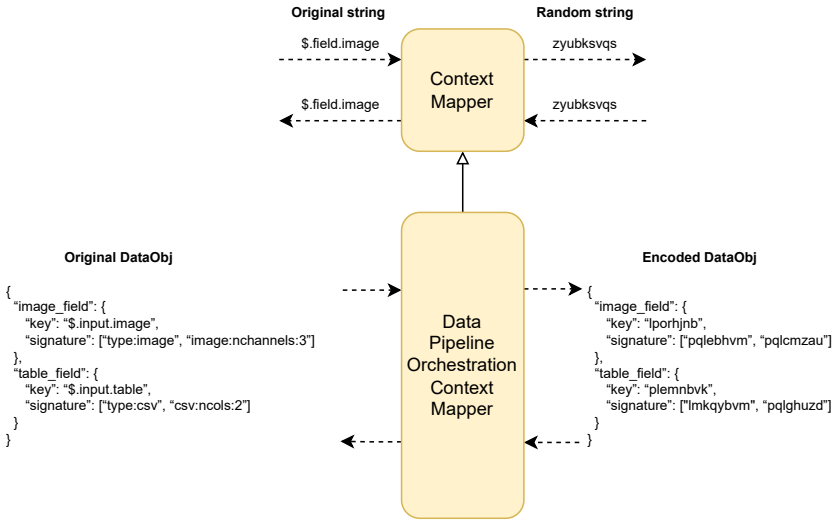


Figure 6.2: Demonstration of context mapping and how it is used to anonymise the data and to remove special characters

function discover_parallel_opportunities(instructions);
return a list of parallel instructions;
input: orchestration instructions;
$parallel\_instructions \leftarrow emptylist$;
$reversed\_instructions \leftarrow reverse(instructions)$;
$N \leftarrow length(instructions)$;
for $i \in [0, N)$ **do**
  if $revered\_instructions[i].type == run$ **then**
    $history \leftarrow reversed\_instructions[i, ..]$;
    $step \leftarrow revered\_instructions[i].params.step$;
    $sync\_list \leftarrow emptylist$;
    $M \leftarrow length(history)$;
    for $k \in [0, M)$ **then**
      if $history[k].type == run$ **then**
        if $step == history[k].params.step$ **then**
          break;
        end
      end
      // Check if step exists in load instruction
      if $history[k].type == load$ **then**
        $source \leftarrow history[k].params.source$;
        $destination \leftarrow history[k].params.destination$;
        // Check if the step has the source as a dependency
        if $step == destination$ **and not** $source \in sync\_list$ **then**
          $sync\_list.push(source)$;
        end
      end
    end
  $parallel\_instructions.push([step, sync\_list])$;
  end
end
$parallel\_instructions \leftarrow reverse(parallel\_instructions)$;
return parallel_instructions

6.3 Performance evaluation method

To evaluate the performance of the pipeline planner, I carryout benchmarks against two tasks. For each task, I generate an input of problem size N and I run the planner for three trials to find the minimum and average execution time and a 95% confidence interval is computed. The machine used for the benchmark has a core i9 processor with 64 GB of RAM. The following benchmarks evaluate the execution time including the translation to PDDL, solving the problem, and parsing back the instructions (see figure 6.1). They do no include the performance of the parallel opportunity discovery algorithm. I also assume that pipelines are typically short (e.g., $0 < N < 30$) and that all the steps in the system are used in the planning process.

6.3.1 Benchmark I: A simple pipeline

The purpose of this benchmark is to simulate common pipelines where a step requires only one step as input. The task used for this benchmark is created by generating N steps where a step s_{i+1} requires step s_i for $i \in [0, N)$. A step s_i is generated as follows:

```
{
    ...
    input: {
        "field": {
            "key": "{random_string()}",
            "signature": ["task:{i}"]
        }
    },
    output: {
        "field": {
            "key": "{random_string()}",
            "signature": ["task:{i+1}"]
        }
    }
}
```

Hence, the problem input is generated as follows:

```
{
    input: {
        "field": {
            "key": "{random_string()}",
            "signature": ["task:0"]
        }
    },
    output: {
        "field": {
            "key": "{random_string()}",
            "signature": ["task:{N}"]
        }
    }
}
```

6.3.2 Benchmark II: A complex pipeline

The purpose of this benchmark is to simulate complex pipelines where steps need inputs from many other steps. By assumption, this benchmark resembles an exaggerated example of a pipeline that is not common in practise. The task used for this benchmark is created by generating N steps where a step s_{i+1} requires steps s_k for $k \in [0, i]$ and for $i \in [0, N)$. A step s_i is generated as follows:

```
{
    ...
    input: {
        "field_0": {
            "key": "{random_string()}",
```

```
            "signature": ["task:0"]
        },
        "field_1": {
            "key": "{random_string()}",
            "signature": ["task:1"]
        },
        "field_2": {
            "key": "{random_string()}",
            "signature": ["task:2"]
        },
        ...
        "field_{i}": {
            "key": "{random_string()}",
            "signature": ["task:{i}"]
        },
    },
    output: {
        "field": {
            "key": "{random_string()}",
            "signature": ["task:{i+1}"]
        }
    }
}
```

Hence, the problem input is generated as follows:

```
{
    input: {
        "field": {
            "key": "{random_string()}",
            "signature": ["task:0"]
        }
    },
    output: {
        "field": {
            "key": "{random_string()}",
            "signature": ["task:{N}"]
        }
    }
}
```

7 Implementation

7.1 Architecture & design decisions

To go about implementing the system I designed and identified three different pinpoints which are the pipeline planner, PDDL ORM, and the PDDL solver. Hence, I divided the implementation into three parts and created three packages named Pipeline Planner (PiPlan), PY_PDDL, and TopPlan (tPlan). Furthermore, I chose to write the code in Python since it is a high programming language, it is well-supported, and it has a wide range of data processing tools. Therefore, I believe that Python provides higher utility through its powerful abstractions and its potential to be integrated with many other data processing tools developed by the large community (e.g., pandas, PyTorch, Tensorflow, etc.). I also follow a micro-architecture design since it isolates different responsibilities and adds more possibilities for scaling the system. Lastly, to prevent potability issues, all of the services provided by the system are containerised.

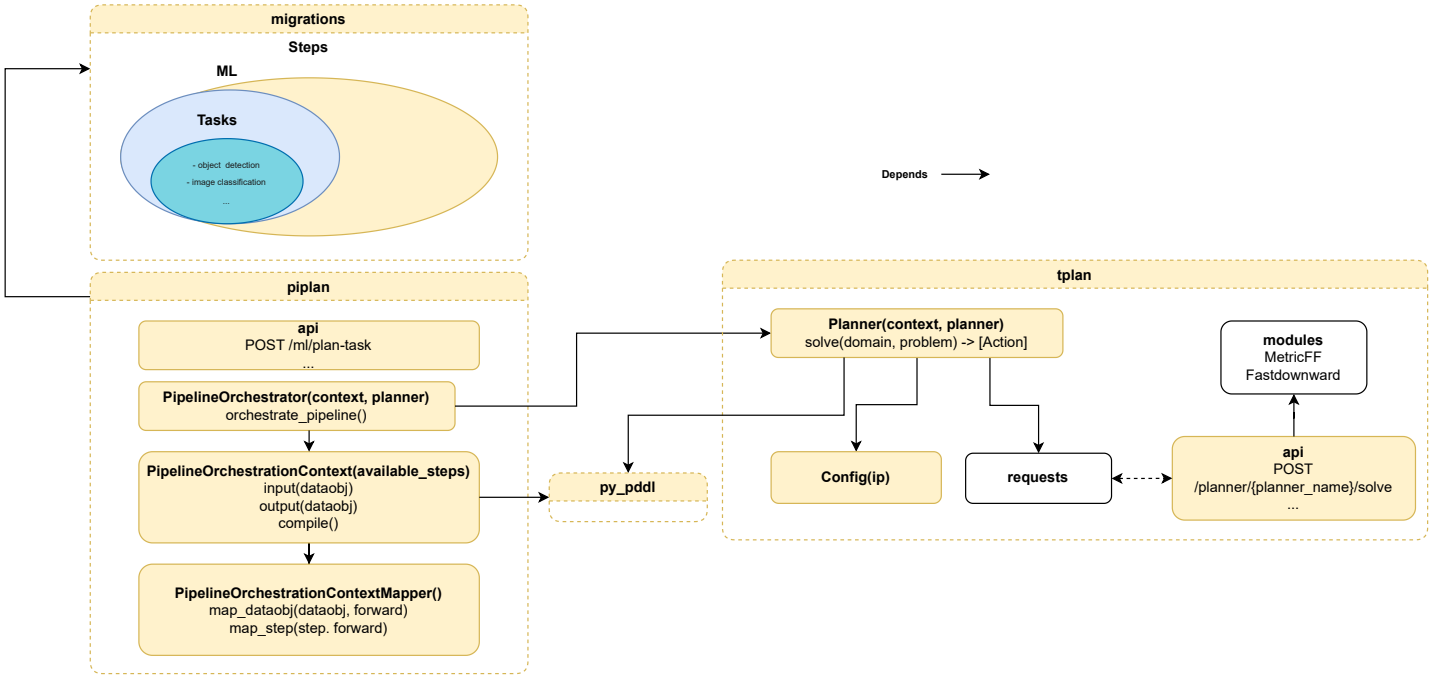


Figure 7.1: General overview of the architecture

7.1.1 Planning service (tPlan)

I use efficient planners Fast Downward [12] and Metric FF [16] which allows numeric fluents to serve the purpose of dynamic states. I created an API that accepts PDDL programs serialised to JSON and I translate the output of a planner to a standard JSON format. Isolating this service prevents the planning service from creating a performance bottleneck in the system. The API was built in Python using the API framework FastAPI that allows event-driven request handling. In addition, I wrote an SDK written in Python that has the Planner class that accepts PY_PDDL domain and problem objects and possibly returns lists of Actions.

7.1.2 PDDL Objects (PY_PDDL)

As mentioned, I create object mappings to PDDL that utilises the benefits of object-oriented-programming. Therefore, I created this package that has the necessary PDDL-serialisable classes (e.g. Domain, Action, Problem, Formula, etc.) to generate PDDL problems. ORM is commonly used to map classes or objects within a programming language to database records. By analogy, I create classes and object in Python that are responsible for mapping the data from the object to the PDDL equivalent. The following snippet demonstrates how the package can be used to specify variables, predicates, functions, types, etc.

```
from py_pddl import Predicate as P, Domain, Action, Parameter as arg, Formula, Requirement, Problem, Object, Type, Type
from py_pddl.core.effect import Inc
from py_pddl.core.formula import Not, And, Or
from py_pddl.core.metric import Metric, MetricOp

name = "analysis_sentiment"

type_data = Type("data")
dtype = Types(type_data, [])
data = arg("x", type_data)

# Variables
latency = var("latency")
storage = var("storage")
metric = Metric(MetricOp.MINIMIZE, latency + storage)

# Data types
```

```
db = P("list_entries", data("x"))
str_list = P("list_string", data("x"))
vec_list = P("list_vec", data("x"))
flt_list = P("list_flt", data("x"))

# Properties
rated = P("rated", data("x"))
concatenated = P("concatenated", data("x"), data("y"))

extract_text = Action(
    name="extract_text",
    precondition=db("x"),
    effect=And(
        str_list("x"),
        Inc(latency, 5),
        Inc(storage, 1024)
    ),
)
```

After definition, the specified objects can be serialised to PDDL. All objects in the package extend the class SERIALIZABLE which has an abstract method TO_PDDL(). This method is implemented by all the objects so that the objects can be serialised. The following snippet shows the implementation of the Action object.

```
class Action(Serializable):

    def __init__(self, name: str, precondition: Formula, effect: Effect, params: List[Parameter]=None):
        self.name = name
        self.precondition = precondition
        self.effect = effect
        if not params:
            self.params = extract_params_from_formulas([self.precondition, self.effect])
        else:
            self.params = params

    def to_pddl(self):
        return f"""
        (:action {self.name}
            :parameters ({serialize_pddls(self.params, " ")})
            :precondition {self.precondition.to_pddl(show_types=False)}
            :effect {self.effect.to_pddl(show_types=False)}
        )
        """
```

7.1.3 Pipeline planner service (PiPlan)

PiPlan is a Python package that can also be used as a web service where users can post requests to the API server and retrieve instructions with parallel suggestions (see figure 7.2). It implements the formulas and the methods mentioned in section 6 (see figure 7.3). The component PIPELINEORCHESTRATIONMAPPER extends CONTEXTMAPPER and is used by the component PIPELINEORCHESTRATIONCONTEXT to map data values as mentioned in figure 6.1. The PIPELINEORCHESTRATIONCONTEXT extends the PDDLPLANNINGCONTEXT that also extends the CONTEXT component. The context component is used to retrieve values using identifiers, and this component is extended by the PDDL planning context component which is used to specify and retrieve PY_PDDL object values using identifiers. The PIPELINEORCHESTROR component implements the parallel discovery algorithm and also uses the context to specify steps and solve pipeline problems using an external planner. PiPlan uses PY_PDDL to implement such formulas and utilises the planning service SDK to solve actions. Currently, PiPlan has JSON data migrations representing steps that carry out machine learning tasks. These migrations are used to demonstrate how PiPlan can be used to create inference pipelines (see section 8.2).

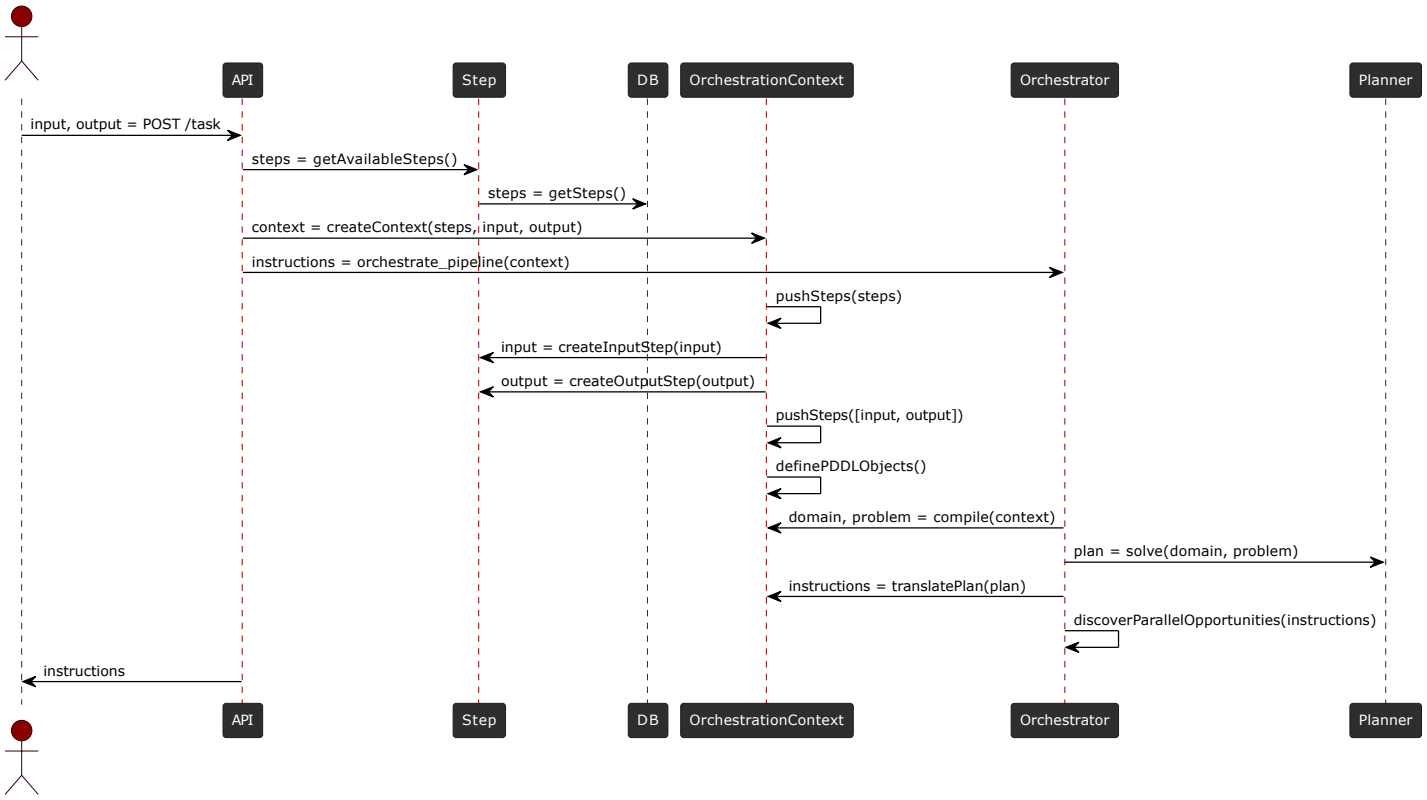


Figure 7.2: Sequence diagram demonstrating a use case where a user requests to create a pipeline that achieves a certain task

8 Results

8.1 Benchmark results

Since the system is using the Fast Downward system, I run the benchmarks but with different aliased configurations (i.e., lama-first, lama, seq-opt-bjolp, seq-opt-lmcut, seq-sat-fd-autotune-1, seq-sat-fd-autotune-2, and seq-sat-lama-2011). These are configurations of planners using the Fast Downward planning system from the International Planning Competition (IPC) 2011 [12].

8.1.1 Benchmark I

I carry out the benchmark in section 6.3.1 and it is apparent that the trend is exponential suggesting an exponential time complexity (see figure 8.1). This result is not surprising as planning problems typically have an exponential search space as mentioned in section 4.1. The best results are achieved with configurations in figures 8.1a, 8.1c, and 8.1d. Although the benchmark suggests poor performance for large problem sizes, the planner would be suitable for planning typical pipelines that do not require a huge number of steps (e.g., $n > 200$).

8.1.2 Benchmark II

I carry out the same method from benchmark I but with a more complex task (see section 6.3.2). Unfortunately, this is indeed a more complex task and so I was not able to run it for all configurations, nor could I run it for large values of N. Although the results in figure 8.2 suggest poor performance, this task is exaggerated to model worst-case scenarios. It is still feasible where in practise since data pipelines are typically not this complex but do not contain such long dependencies. Therefore, given the results of both benchmarks, the planner could be used in practise where pipelines are moderately complex.

8.2 Use case: automation of machine learning inference pipelines

This use case example demonstrates how the planner can be used to automate the creation of machine learning inference pipelines. It also demonstrates the generality of the planner and how it can be applied in new use cases. This use case falls underneath the definition

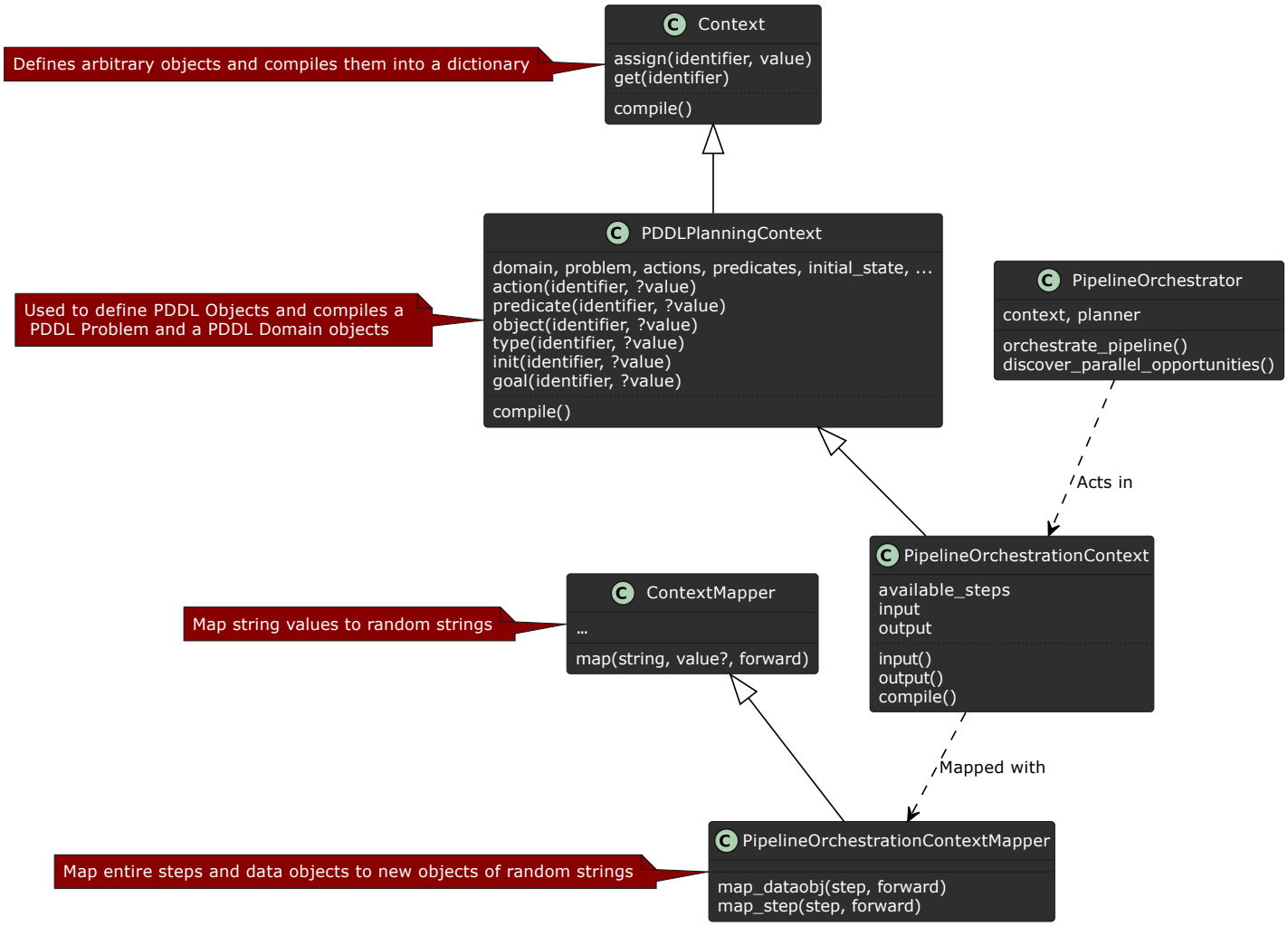


Figure 7.3: Class diagram showcasing the essential components of PiPlan

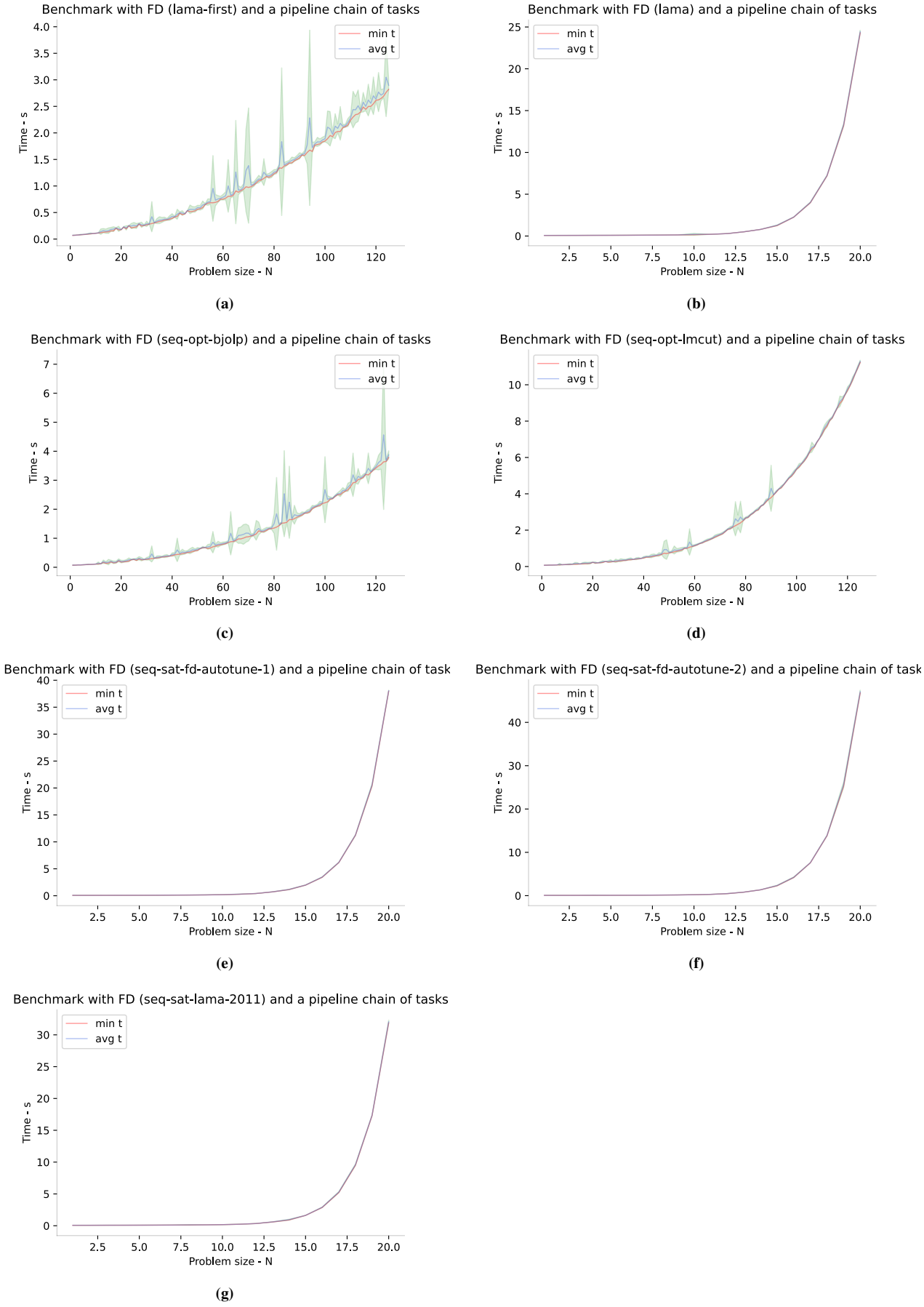


Figure 8.1: Benchmark I results with the Fast Downward (FD) planner configured with different aliases and computed confidence interval for the average time (green and blue).

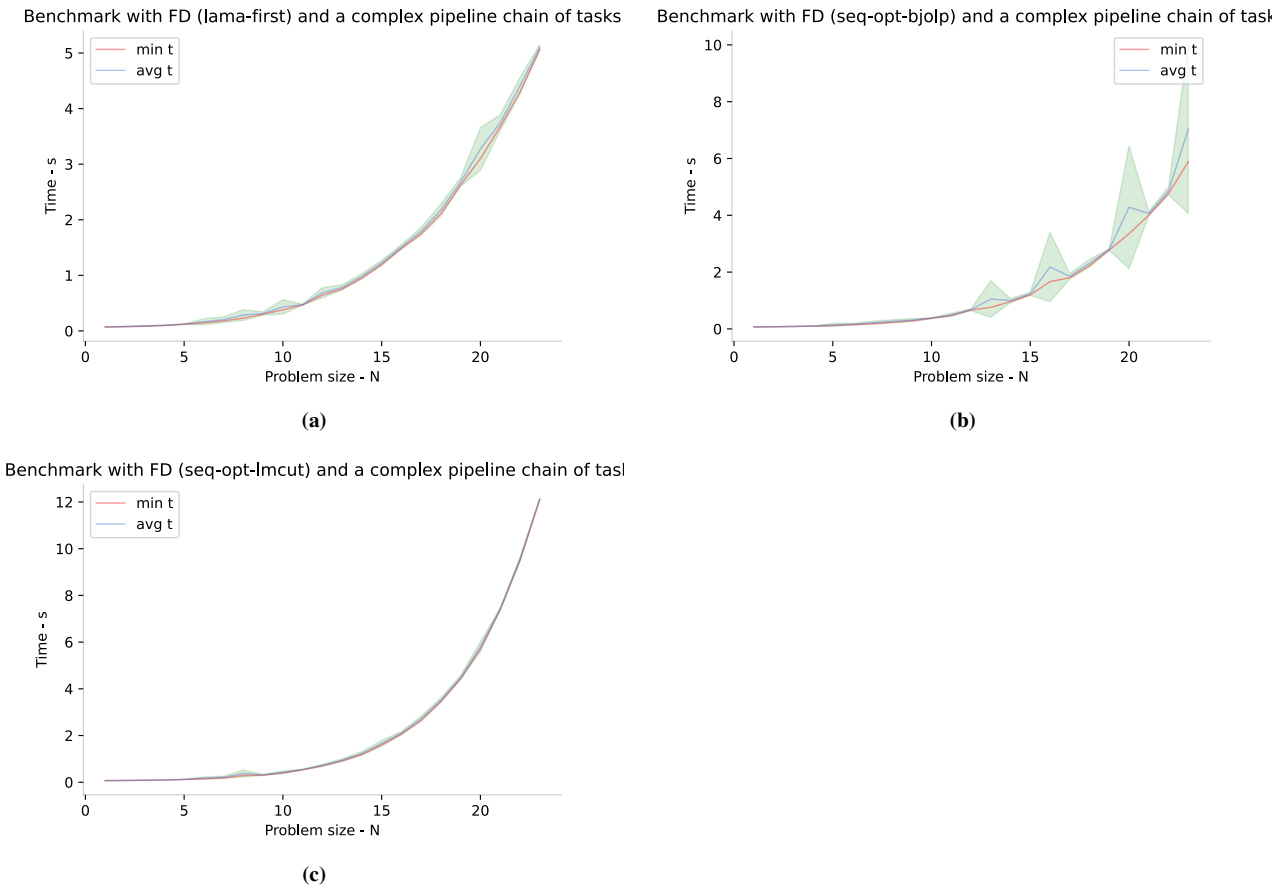


Figure 8.2: Benchmark II results with the Fast Downward (FD) planner configured with different aliases and computed confidence interval for the average time (green and blue).

of task oriented planning (see table 5.2) since the pipeline is not known and needs to be planned, and the purpose is to achieve a data task (i.e., a machine learning task).

8.2.1 Designing data signatures

The first step is to design the data signatures so that it suits the use case at hand. I structure the machine learning tasks in a hierarchy which is specified by delimiting the entries with a colon. I also use the signatures to define data types and add attributes.

```
{
 "dataobj": {
  "field": {
  "key": "some_key",
   "signature": [
    "task:<task_type>:<category>:<...>",
    "type:<data type>",
    "<data type>:<attribute>"
   ]
  }
 },
}
```

For example, a step running a medical AI model responsible for segmenting lungs and infections due to Covid-19 can be modelled as follows:

```
{
 "resource_id": "api:infere/covid19infection",
 "name": "Covid 19 lung infection segmentation",
 "input": {
  "image": {
   "key": "$.image",
   "signature": [
    "type:volume"
   ]
  }
 },
 "output": {
  "image": {
   "key": "$.segmented_image",
   "signature": [
    "type:volume",
    "task:image_segmentation:lung",
    "task:image_segmentation:lung_infection"
   ]
  }
 }
},
```

I can also describe a step carrying out an organ segmentation task as follows:

```
{
 "resource_id": "api:infere/organseg",
 "name": "Organ segmentation",
 "input": {
  "scan": {
   "key": "$.scan",
   "signature": [
    "type:volume"
   ]
```

```
    }
  },
  "output": {
    "segmentation": {
      "key": "$.segmented_image",
      "signature": [
        "type:volume",
        "task:image_segmentation:spleen",
        "task:image_segmentation:heart",
        "task:image_segmentation:lung",
        "task:image_segmentation:stomach"
      ]
    }
  }
}
```

A classic digit classifier step can be modelled as follows:

```
{
  "resource_id": "api:infere/mnist",
  "name": "MNIST",
  "input": {
    "image": {
      "key": "$.image",
      "signature": [
        "type:image",
        "image:nchannels:1"
      ]
    }
  },
  "output": {
    "predictions": {
      "key": "$.predictions",
      "signature": [
        "type:array",
        "array:length:10",
        "task:image_classification:digit"
      ]
    }
  }
},
```

I created examples to evaluate the results but not all tasks are included. I will use these defined examples to showcase the result in the following section.

8.2.2 Examples of requests

The end application or the user specifies the input of their data and the task they would like to achieve. They do not necessarily know the type of the output data, but they know what task they would like to achieve since the available tasks will be provided. For example, the set of available tasks could be:

```
{
  "available_tasks": [
    "task:image_classification:digit",
    "task:image_segmentation:lung",
    "task:image_segmentation:lung_infection",
    "task:image_segmentation:brain",
```

```
    "task:image_segmentation:tumor",
    "task:image_segmentation:spleen",
    "task:image_segmentation:heart",
    "task:image_segmentation:stomach",
    ...
  ]
}
```

Covid-19 diagnosis and organ segmentation example: One might want to diagnose a patient suspected to be infected with Covid-19 and segment the spleen in the volumetric data. Such a request can be made as follows:

```
{
 "input": {
  "scan": {
   "key": "$.myscan",
   "signature": [
    "type:volume"
   ]
  }
 },
 "output": {
  "infection_seg": {
   "key": "$.segmented_scan",
   "signature": [
    "task:image_segmentation:lung_infection"
   ]
  },
  "spleen_seg": {
   "key": "$.segmented_scan",
   "signature": [
    "task:image_segmentation:spleen"
   ]
  }
 }
}
```

After applying this request our planner suggests the following instructions:

```
{
 "parallel": [
  {
   "synchronize": [],
   "step": "input_buffer"
  },
  {
   "synchronize": [
    "input_buffer"
   ],
   "step": "api:infere/covid19infection"
  },
  {
   "synchronize": [
    "input_buffer"
   ],
   "step": "api:infere/organseg"
  },
```

```
  {
   "synchronize": [
    "api:infere/organseg",
    "api:infere/covid19infection"
   ],
   "step": "output_buffer"
  }
 ],
 "instructions": [
  {
   "type": "run",
   "params": {
    "key": [
     "$.myscan",
     "$.myscan"
    ],
    "step": [
     "input_buffer"
    ]
   }
  },
  {
   "type": "load",
   "params": {
    "key": [
     "$.myscan",
     "$.image"
    ],
    "step": [
     "input_buffer",
     "api:infere/covid19infection"
    ]
   }
  },
  {
   "type": "load",
   "params": {
    "key": [
     "$.myscan",
     "$.scan"
    ],
    "step": [
     "input_buffer",
     "api:infere/organseg"
    ]
   }
  },
  {
   "type": "run",
   "params": {
    "key": [
     "$.image",
     "$.segmented_image"
    ],
    "step": [
```

```
      "api:infere/covid19infection"
     ]
    }
   },
   {
    "type": "load",
    "params": {
     "key": [
      "$.segmented_image",
      "$.segmented_scan"
     ],
     "step": [
      "api:infere/covid19infection",
      "output_buffer"
     ]
    }
   },
   {
    "type": "run",
    "params": {
     "key": [
      "$.scan",
      "$.segmented_image"
     ],
     "step": [
      "api:infere/organseg"
     ]
    }
   },
   {
    "type": "load",
    "params": {
     "key": [
      "$.segmented_image",
      "$.segmented_scan"
     ],
     "step": [
      "api:infere/organseg",
      "output_buffer"
     ]
    }
   },
   {
    "type": "run",
    "params": {
     "key": [
      "$.segmented_scan",
      "$.segmented_scan",
      "$.segmented_scan",
      "$.segmented_scan"
     ],
     "step": [
      "output_buffer"
     ]
    }
```

```
  }
 ],
 "modules": [
  ...
 ]
}
```

There is no available step that does both of these tasks, hence, the planner suggests applying two different steps (i.e., the step that does Covid-19 diagnosis and the step that does organ segmentation). The planner also suggests that both of the steps require the input to be loaded before they are executed and the two steps can be executed in parallel since they carry out two independent tasks.

Digit classification example: A user wants to classify coloured images of digits. Therefore, a user creates the following request to the planner:

```
{
 "input": {
   "digit": {
    "key": "$.my_digit_image",
    "signature": [
     "type:image",
      "image:nchannels:3"
    ]
   }
  },
  "output": {
   "prediction": {
    "key": "$.my_digit_prediction",
    "signature": [
     "task:image_classification:digit"
    ]
   }
  }
}
```

The planner responds suggesting the following instructions:

```
{
 "parallel": [
  {
   "synchronize": [],
   "step": "input_buffer"
  },
  {
   "synchronize": [
    "input_buffer"
   ],
   "step": "script:transform/tograyscale"
  },
  {
   "synchronize": [
    "script:transform/tograyscale"
   ],
   "step": "api:infere/mnist"
  },
  {
   "synchronize": [
    "api:infere/mnist"
```

```
    ],
    "step": "output_buffer"
   }
  ],
  "instructions": [
   {
    "type": "run",
    "params": {
     "key": [
      "$.my_digit_image",
      "$.my_digit_image"
     ],
     "step": [
      "input_buffer"
     ]
    }
   },
   {
    "type": "load",
    "params": {
     "key": [
      "$.my_digit_image",
      "$.image"
     ],
     "step": [
      "input_buffer",
      "script:transform/tograyscale"
     ]
    }
   },
   {
    "type": "run",
    "params": {
     "key": [
      "$.image",
      "$.grayscale_image"
     ],
     "step": [
      "script:transform/tograyscale"
     ]
    }
   },
   {
    "type": "load",
    "params": {
     "key": [
      "$.grayscale_image",
      "$.image"
     ],
     "step": [
      "script:transform/tograyscale",
      "api:infere/mnist"
     ]
    }
   },
```

```
  {
   "type": "run",
   "params": {
    "key": [
     "$.image",
     "$.predictions"
    ],
    "step": [
     "api:infere/mnist"
    ]
   }
  },
  {
   "type": "load",
   "params": {
    "key": [
     "$.predictions",
     "$.my_digit_prediction"
    ],
    "step": [
     "api:infere/mnist",
     "output_buffer"
    ]
   }
  },
  {
   "type": "run",
   "params": {
    "key": [
     "$.my_digit_prediction",
     "$.my_digit_prediction"
    ],
    "step": [
     "output_buffer"
    ]
   }
  }
 ],
 "modules": [
  ...
 ]
}
```

The planner suggested to convert the coloured image to a gray-scale image because there are no available digit classifiers that accept coloured images. This example has been implemented in a UI using React as well which can be seen in appendix 12.1.

9 Conclusion

I gave background about AI planning and data processing. I analysed relevant literature and developed a theoretical framework as an approach to solving the research problem. In the theoretical framework, I took the approach of viewing data processing pipelines as components in a system, and also show how AI planning can be applied in such systems. I identified that data processing pipelines can be scoped by the purpose of the planning mainly related to resource allocation or achieving data tasks. Hence, I defined two orientations of planning for data processing pipelines, i.e., resource-oriented planning, and task-oriented planning. I also demonstrate why dynamic aspects such as feedback loops are important and how they can be modelled using the components defined in the theoretical framework. Then I showcased the methods to show how the planning domain is formulated by defining predicates and actions that model some of the

system components. I demonstrated how PDDL objects can be used as an abstraction method, how parallel opportunities are discovered through analysing dependencies, and how the performance of the data pipeline planner was evaluated. I cover the design aspects and the architecture and showcase how the system was implemented. The results show that the planner is suitable for moderately complex data pipelines and can be inefficient at very complex pipeline problems with $N > 20$. I also demonstrated how the planner can be used in different domains by giving flexibility in defining data signatures, and I showcase this idea by giving examples of automation of machine learning use cases. I give examples of different requests the planner is capable of handling and see that the planner is capable of analysing dependencies and giving suggestions for parallel execution.

10 Future work & Limitations

I was able to create a pipeline planner that is not application specific. Due to time constraints, I was not able to implement the dynamic aspects of the system (explained in section 5.2.2). Therefore, Δ state updates (explained in 5.1.1 are not taken into account. The planner only considers static aspects where data objects are tagged with data signatures and are loaded into steps. However, the current system has the potential to be extended and delta effects can be added but more sophisticated planners (e.g., SMTPlan+ [15], Metric-FF [16]) are needed to consider linear or non-linear action effects.

Moreover, I implemented a task-oriented planner but did not consider resource-oriented planning. Developing a resource planner adds more utility since it is complementary to the task planner. For example, one might generate a pipeline through the task planner and wishes to deploy it in a cloud computing infrastructure. Therefore, the resource planner can be a powerful tool to automatically deploy the resulting pipelines.

Indeed, the planning community offers many planners that have been well-developed and optimised and using such planners can be for the system can be fruitful. However, the planners are maintained by the community and appropriate planners may not always be found for specific PDDL features and PDDL may not support some specific features. For instance, most planners available do not support creation of new objects as effects of actions, this could have been useful for instantiating new data objects when invoking a step. The current system strictly relies on PDDL and the available PDDL solvers. Hence, it could be worthwhile to develop a context-specific planner for this system. This also gives more freedom in developing more specific heuristics and evaluation methods. For example, the signatures could be sorted and serialised to achieve more efficient signature matching. Developing such a planner removes this dependency and enhances the potential of extending the system. It makes the possibility of adding more functionality always possible since the application does not depend on the community.

11 Acknowledgement

I would like to thank our supervisors Mostafa Hadadian and Dr. Viktoriya Degeler for their kind support and guidance throughout the development of this project. I also appreciate the University of Groningen and the Faculty of Science and Engineering for providing the necessary resources and I express my gratitude and appreciation to my computer science professors who provided the essential education for this project.

References

[1] R. Sharda, D. Delen, E. Turban, and D. King, "Business intelligence: A managerial approach, global edition," in Pearson, 2017, ch. Chapter 1.

[2] J. Barney, "Firm resources and sustained competitive advantage," *Journal of Management*, vol. 17, no. 1, pp. 99–120, 1991. DOI: 10.1177/014920639101700108. eprint: https://doi.org/10.1177/014920639101700108. [Online]. Available: https://doi.org/10.1177/014920639101700108.

[3] M. D. Assunção, R. N. Calheiros, S. Bianchi, M. A. Netto, and R. Buyya, "Big data computing and clouds: Trends and future directions," *Journal of Parallel and Distributed Computing*, vol. 79-80, pp. 3–15, 2015, Special Issue on Scalable Systems for Big Data Management and Analytics, ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2014.08.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731514001452.

[4] S. Conn, "Oltp and olap data integration: A review of feasible implementation methods and architectures for real time data analysis," in *Proceedings. IEEE SoutheastCon, 2005.*, 2005, pp. 515–520. DOI: 10.1109/SECON.2005.1423297.

[5] T. Albers, E. Lazovik, M. Hadadian Nejad Yousefi, and A. Lazovik, "Adaptive on-the-fly changes in distributed processing pipelines," *Frontiers in Big Data*, vol. 4, 2021, ISSN: 2624-909X. DOI: 10.3389/fdata.2021.666174. [Online]. Available: https://www.frontiersin.org/article/10.3389/fdata.2021.666174.

[6] A. Raj, J. Bosch, H. H. Olsson, and T. J. Wang, "Modelling data pipelines," in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2020, pp. 13–20. DOI: 10.1109/SEAA51224.2020.00014.

[7] E. Lazovik, M. Medema, T. Albers, E. Langius, and A. Lazovik, "Runtime modifications of spark data processing pipelines," in *2017 International Conference on Cloud and Autonomic Computing (ICCAC)*, 2017, pp. 34–45. DOI: 10.1109/ICCAC.2017.11.

[8] S. J. Russell, P. Norvig, and E. Davis, in *Artificial Intelligence: A modern approach*. Pearson, 2022, ch. 10: Classical planning.

[9] M. Ghallab, D. S. Nau, and P. Traverso, *Automated planning and acting*. Cambridge University Press, 2016.

[10] *Objectrelational mapping*, Jul. 2022. [Online]. Available: https://en.wikipedia.org/wiki/Object%5C%E2%5C%80%5C%93relational_mapping.

[11] *The ai planning & pddl wiki*. [Online]. Available: https://planning.wiki/.

[12] M. Helmert, "The fast downward planning system," *Journal of Artificial Intelligence Research*, vol. 26, pp. 191–246, Jul. 2006. DOI: 10.1613/jair.1705. [Online]. Available: https://doi.org/10.1613%2Fjair.1705.

[13] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: Theory and Practice*, ser. The Morgan Kaufmann Series in Artificial Intelligence. Amsterdam: Morgan Kaufmann, 2004, ISBN: 978-1-55860-856-6. [Online]. Available: http://www.sciencedirect.com/science/book/9781558608566.

[14] J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," *CoRR*, vol. abs/1106.0675, 2011. arXiv: 1106.0675. [Online]. Available: http://arxiv.org/abs/1106.0675.

[15] M. Cashmore, M. Fox, D. Long, and D. Magazzeni, "A compilation of the full pddl+ language into smt," in *ICAPS*, 2016.

[16] J. Hoffmann, "The metric-ff planning system: Translating "ignoring delete lists" to numeric state variables," *Journal of Artificial Intelligence Research*, vol. 20, Jun. 2011. DOI: 10.1613/jair.1144.

[17] M. Asai and A. Fukunaga, "Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary," *CoRR*, vol. abs/1705.00154, 2017. arXiv: 1705.00154. [Online]. Available: http://arxiv.org/abs/1705.00154.

[18] M. Asai, "Unsupervised grounding of plannable first-order logic representation from images," *IBM Research*, 2019. [Online]. Available: https://arxiv.org/pdf/1902.08093.pdf.

[19] P. S. Diouf, A. Boly, and S. Ndiaye, "Variety of data in the etl processes in the cloud: State of the art," in *2018 IEEE International Conference on Innovative Research and Development (ICIRD)*, 2018, pp. 1–5. DOI: 10.1109/ICIRD.2018.8376308.

[20] G. Gautam and D. Yadav, "Sentiment analysis of twitter data using machine learning approaches and semantic analysis," in *2014 Seventh International Conference on Contemporary Computing (IC3)*, 2014, pp. 437–442. DOI: 10.1109/IC3.2014.6897213.

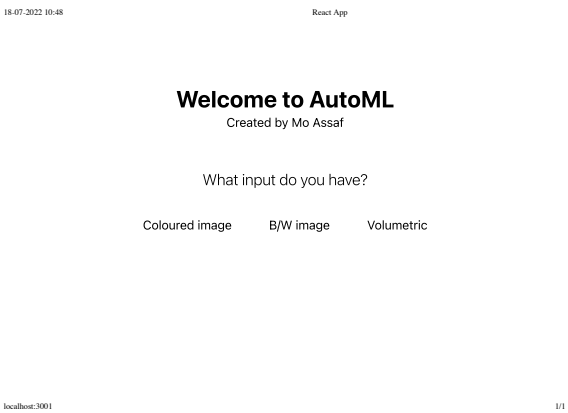
[21] S. M. S. Tanzil, W. Hoiles, and V. Krishnamurthy, "Adaptive scheme for caching youtube content in a cellular network: Machine learning approach," *IEEE Access*, vol. 5, pp. 5870–5881, 2017. DOI: 10.1109/ACCESS.2017.2678990.

[22] B. Franks, *Taming the big data tidal wave: Finding opportunities in huge data streams with advanced analytics*. Wiley, 2012.

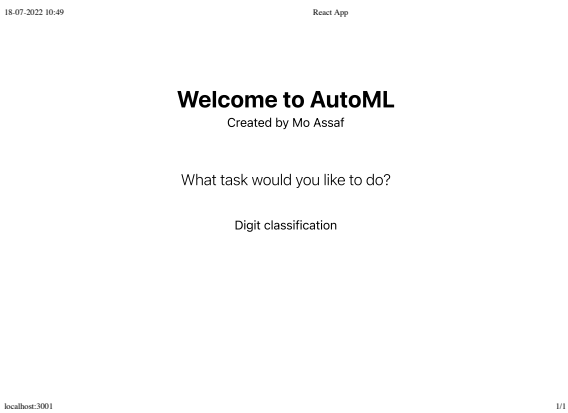
[23] M. Lorenz, J.-P. Rudolph, G. Hesse, M. Uflacker, and H. Plattner, "Object-relational mapping revisited - a quantitative study on the impact of database technology on o/r mapping strategies," in *HICSS*, 2017.

12 Appendix

12.1 AutoML example (UI)



(a)



(b)

(c)

Figure 12.1: An implemented UI for the example where machine learning pipelines are automated using PiPlan.