AIPlan4EU

Bringing AI Planning to the
European AI On-Demand Platform

D3.1. Infrastructure Description

June 30th 2021

Project funded by the European Commission within the Horizon 2020 Programme		
Dissemination Level		
PU	Public	X
CO	Confidential, only for members of the consortium (including the Commission Services)	□
CL	Classified, as referred to in Commission decision 2001/844/EC	□

Deliverable number:	D3.1
Deliverable name:	Infrastructure Description
Work package:	WP3: Unified Planning Framework
Lead WP:	FBK
Lead Task:	FBK

Contents

Document Revision History

Date	Issue	Author/Editor/Contributor	Summary of main change
20/06/2021	V1	Andrea Micheli (FBK)	Initial Draft
25/06/2021	V2	Andrea Micheli (FBK)	Added section on UPF proof-of-concept
28/06/2021	V3	WP3 Partners	Shared with WP3 partners
29/06/2021	V4	Andrea Micheli (FBK)	Final revision

Abstract

Automated Planning and Scheduling is a central research area in AI that has been studied since the inception of the field and where European research has been making strong contributions over decades. Planning is a decision-making technology that consists in reasoning on a predictive model of a system being controlled and deciding how and when to act in order to achieve a desired objective. It is a relevant technology for many application areas that need quick, automated and optimal decisions, like agile manufacturing, agrifood or logistics. Although there is a wealth of techniques that are mature in terms of science and systems, several obstacles hinder their adoption, thus preventing them from making the footprint on European industry that they should make. For example, it is hard for practitioners to find the right techniques for a given planning problem, there are no shared standards to use them, and there is no easy access to expertise on how to encode domain knowledge into a planner.

The AIPlan4EU project will bring AI planning as a first-class citizen in the European AI On-Demand (AI4EU) Platform by developing a uniform, user-centered framework to access the existing planning technology and by devising concrete guidelines for innovators and practitioners on how to use this technology. To do so, we will consider use-cases from diverse application areas that will drive the design and the development of the framework and include several available planning systems as engines that can be selected to solve practical problems. We will develop a general and planner-agnostic API that will both be served by the AI4EU platform and be available as a resource to be integrated into the users' systems. The framework will be validated on use-cases both from within the consortium and recruited by means of cascade funding; moreover, standard interfaces between the framework and common industrial technologies will be developed and made available.

Executive summary

This deliverable presents the general development infrastructure for the software components that will be developed within the AIPlan4EU project. In AIPlan4EU, we have three major software kinds, namely the Unified Planning Framework (UPF), the planning engines interfaces and the Technology-Specific Bridges (TSBs).

This document highlights the technological choices that were discussed in WP3 within task T3.1 and presents the rationale behind them as well as the proof-of-concepts for the UPF that we developed to assess different alternatives.

Moreover, we will highlight how such choices have been motivated by the use-cases elicited within WP2 and balanced with technological constraints from the partners. This document also discusses the licensing strategy that is also outlined in D1.3, the agile methodologies that will be adopted and that are experimented in the development of the UPF as well as the technologies that will be used to support this methodology.

The work summarized in this document was led by FBK that coordinated the task and provided the development effort, but all the partners involved in WP3 were involved in the decision-making; all the steps and decisions have been presented analyzed and discussed in WP-plenary bi-weekly meetings that were held starting from 24/02/2021.

1 Software Artifacts and Development Repository

The AIPlan4EU project will produce 4 different kinds of software outputs.

- The Unified Planning Framework (UPF), developed within WP3, is the main abstraction library that encompasses different kinds of planning technologies wrapping them in a convenient, high-level interface to be offered to client applications (TSBs).
- The Technology-Specific Bridges (TSBs), developed within WP5, are the interfaces between the UPF library and the client applications that contain the business logic to gather the needed data from the domain-specific technology, construct a suitable planning problem to be solved in the data structures offered by the UPF, invoke the UPF engines at the right time and translate back the answers to data structures and data usable by the client technology. As an example, in a logistics scenario the TSB is responsible for querying the Warehouse Management System (WMS) to assess the orders to be fulfilled as well as the status of the warehouse and its shelves, to construct a planning problem out of this data, to invoke a planner via the UPF abstraction API and to transform the plan produced by the engines into a suitable representation for the WMS to be put into execution.
- The Planner Interfaces, developed within WP4, are the interfacing code needed for the UPF to access each specific planner. The role of such interfaces is to abstract away the specifics of the planner and to describe the engine capabilities so that the UPF can automatically select suitable engines given a planning problem.
- The integration with the AI On-Demand Platform experiment infrastructure is the wrapping of the UPF and/or its components into containers that can be executed and integrated using the extended ACUMOS platform of AI4EU.

These outputs will be developed using an agile methodology and most of them will be released as open source and will follow an open development strategy. For this reason, we decided to use GitHub (https://github.com), the de-facto standard platform for open-source development. On the GitHub platform, we created an "organization" dedicated to the AIPlan4EU project[1].

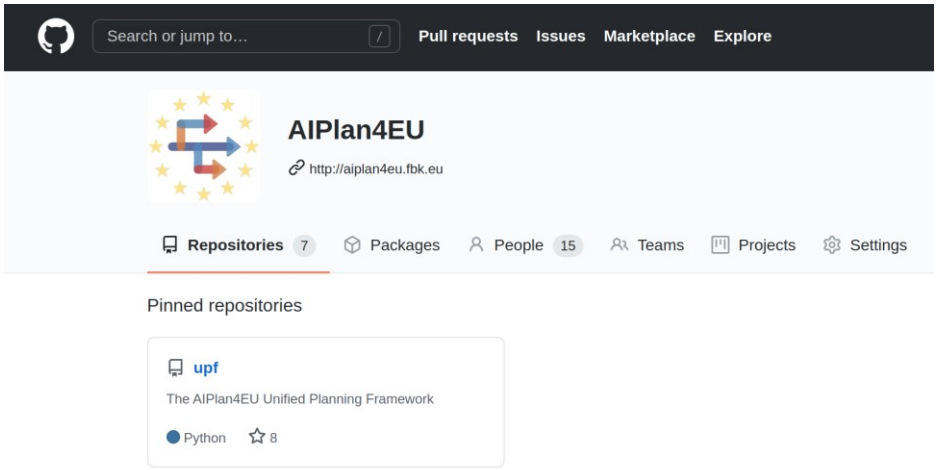

Figure 1 The AIPlan4EU organization on GitHub

An organization is a collection of different projects for the different software outputs will be created and archived. Each project has its own separate git repository and wiki as well as an issue-tracking system to facilitate and foster collaborative remote development. Inter-project coordination and linking is also possible in wikis and issues. An

[1] https://github.com/aiplan4eu

interesting feature is that it is possible to create both public and private repositories to serve the different licensing needs.

1.1 Software Licensing

As described in detail in the Data management Plan (D1.3), the project will adopt a diversified licensing schema to foster an "as open as possible" policy for its output taking into account the Exploitation Plan (D9.1) and the IPR needs of the companies involved. In particular, we will use the Apache 2.0 License for the UPF and for the Planner Interfaces and TSBs that are not covered by closed-source constraints. For the others, we will select a suitable license on a case-by-case basis.

2 Agile Development

The AIPlan4EU project embraces the Agile methodology and follows a User-Centered Design (UCD) approach. In particular, we will make use of "user-stories" to describe the requirements and user needs from the point of view of the project personas (a stereotypical and well-defined kind of user). To this end, in the project repository we created a template2 for the definition of this kind of issues and we use an agile kanban approach3 to monitor and track the development of the user stories. In addition, following the standard git development, each feature, bugfix and change is developed on a separate "branch", that is a separate copy4 of the repository that is then merged with the master branch (i.e., the stable version of the code) by means of a "pull-request". The latter is a methodological step supporting the evaluation of the developed code with tools to comment and clearly identify the changes that helps in maintaining a high-quality code.

2.1 User stories

Concretely, a user story is framed as follows.

```
User Story
----------

As a [USER] I want to [DO SOMETHING] to [PURPOSE].
Any other relevant description as needed.


Acceptance Criteria
-------------------

- Criterion 1
- Criterion 2

Additional Material
-------------------

- Links
- Snippets
- Papers

Attention Points
----------------

Any other relevant info for the developers
```

[2] https://github.com/aiplan4eu/upf/blob/master/.github/ISSUE_TEMPLATE/user-story.md

[3] https://github.com/aiplan4eu/upf/projects/1

[4] This is not exact; a branch is a collection of changes on top of an ancestor state of the master branch. But conceptually, one can think of a completely separate copy that is then merged in the reference code.

As an example, the user story below5 concerns the support of parallel solving within the UPF.

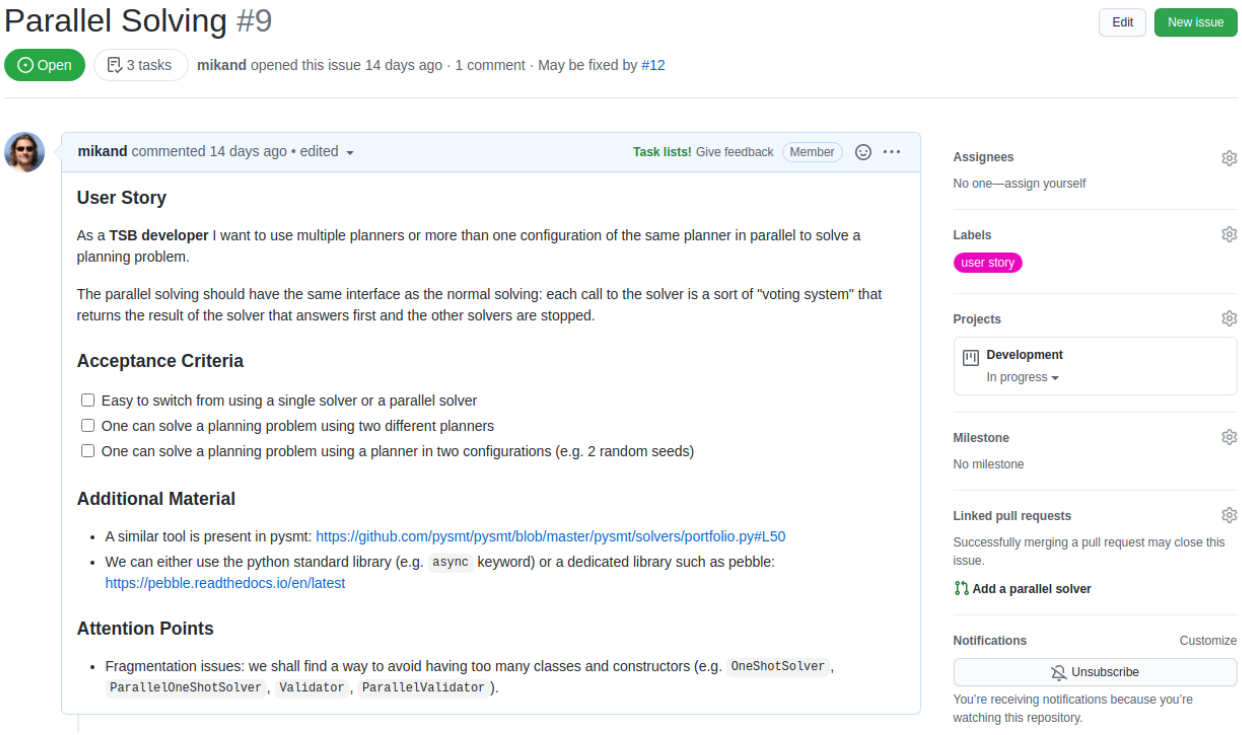

Figure 2 An example user story for the UPF

2.2 Development kanban

The project kanban simplifies the tracking of the status of the user stories and to easily assess the pace and the criticalities of the development. Each user-story is depicted as a card that can be moved between four columns:

- "To do" collects the stories that have been defined, but that have not started yet.
- "In progress" encompasses the user stories that are currently being developed.
- "In validation" lists the user stories that have been developed, but that are still in "pull-request", meaning that the code is still under evaluation and not yet available in the master branch.
- "Done" records the user stories that have been completed and are supported by the master branch of the project.

5 https://github.com/aiplan4eu/upf/issues/9

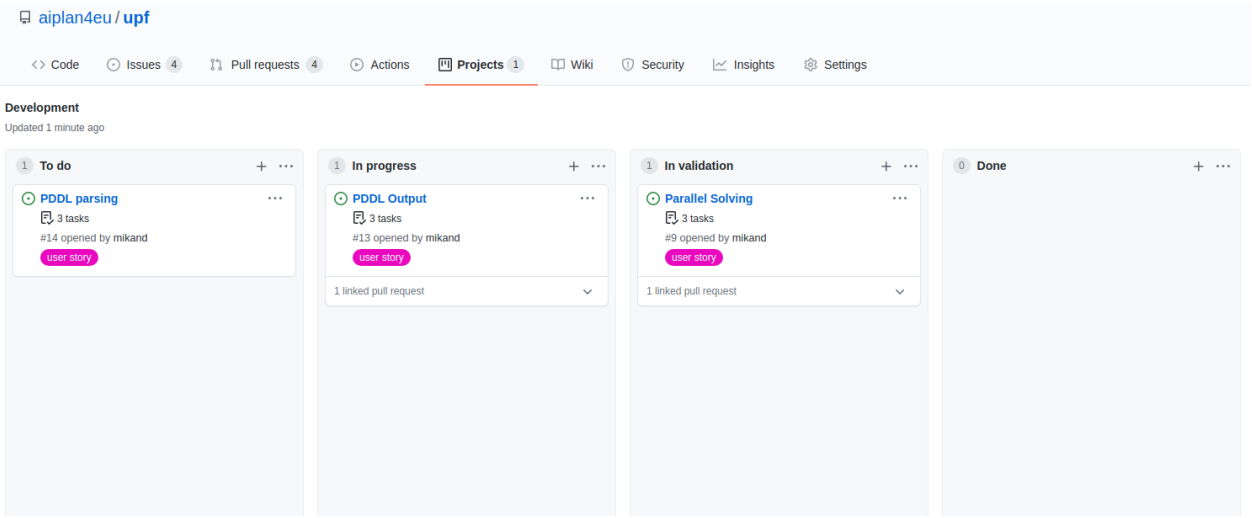

Figure 3 The UPF kanban view

2.3 Pull-requests

Pull-requests help in analyzing the code being contributed to the project and are a standard methodology supported by major git-based development frameworks (e.g. GitHub, GitLab, Bitbucket). A pull-request consists of a description attached to a set of changes and offers tools for reviewing the code, discuss about specific portions of the changes and reports the results of the continuous integration checks.

An example of pull-request can be seen at https://github.com/aiplan4eu/upf/pull/15. The first tab contains the description (in this case a link to the user story number 13) and the discussion (in this case a discussion with a user external to the project that was commenting and asking about the use of a library called “tarski”). The second tab collects the commits contributed to this pull-request; the third tab instead contains the results of the continuous integration checks (more on this in the next section). Finally, the fourth tab shows the code being contributed highlighting the added and removed lines of code so that a reviewer can comment and/or suggest changes.

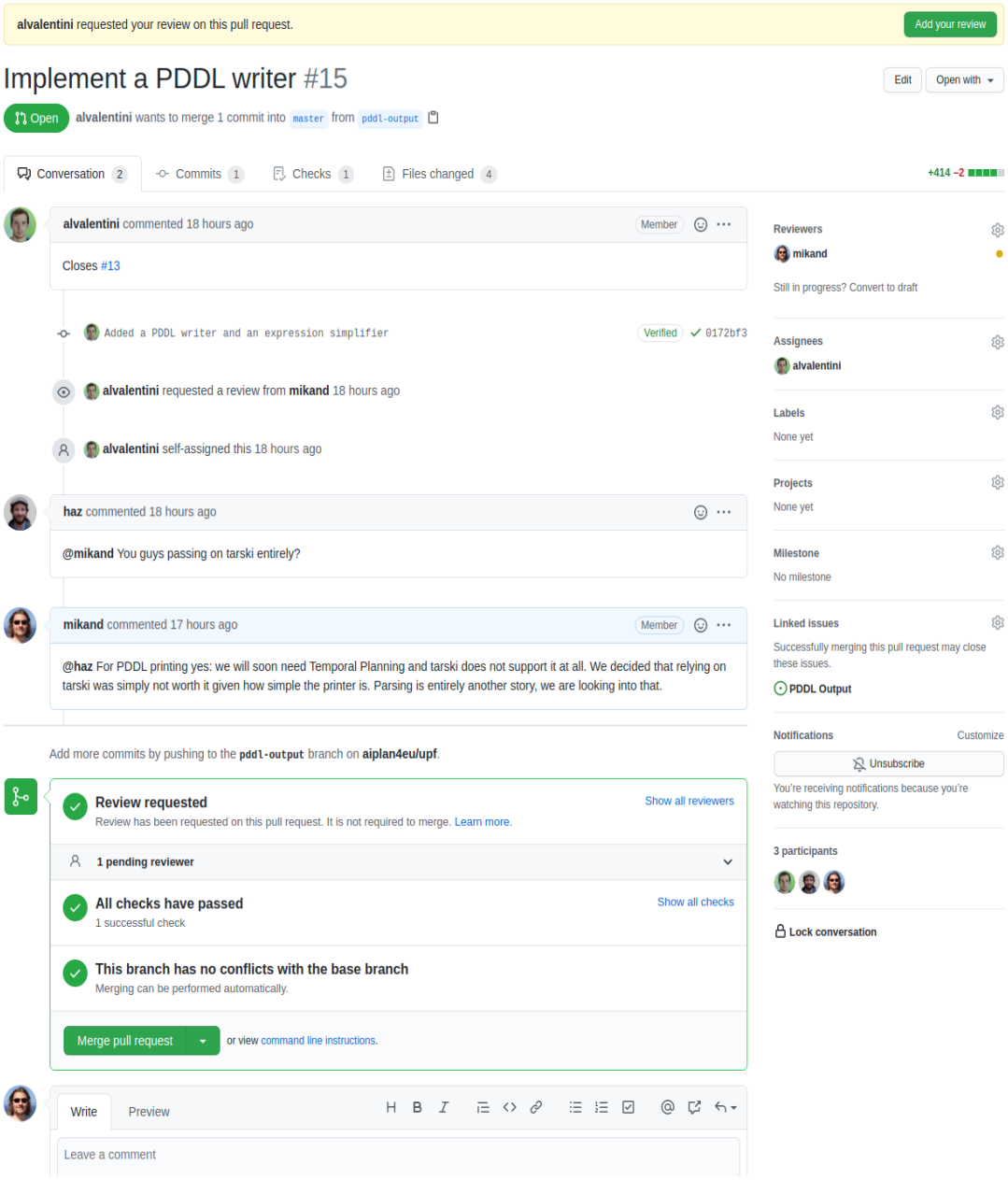

Figure 4 Discussion tab for an example pull-request from the UPF repository

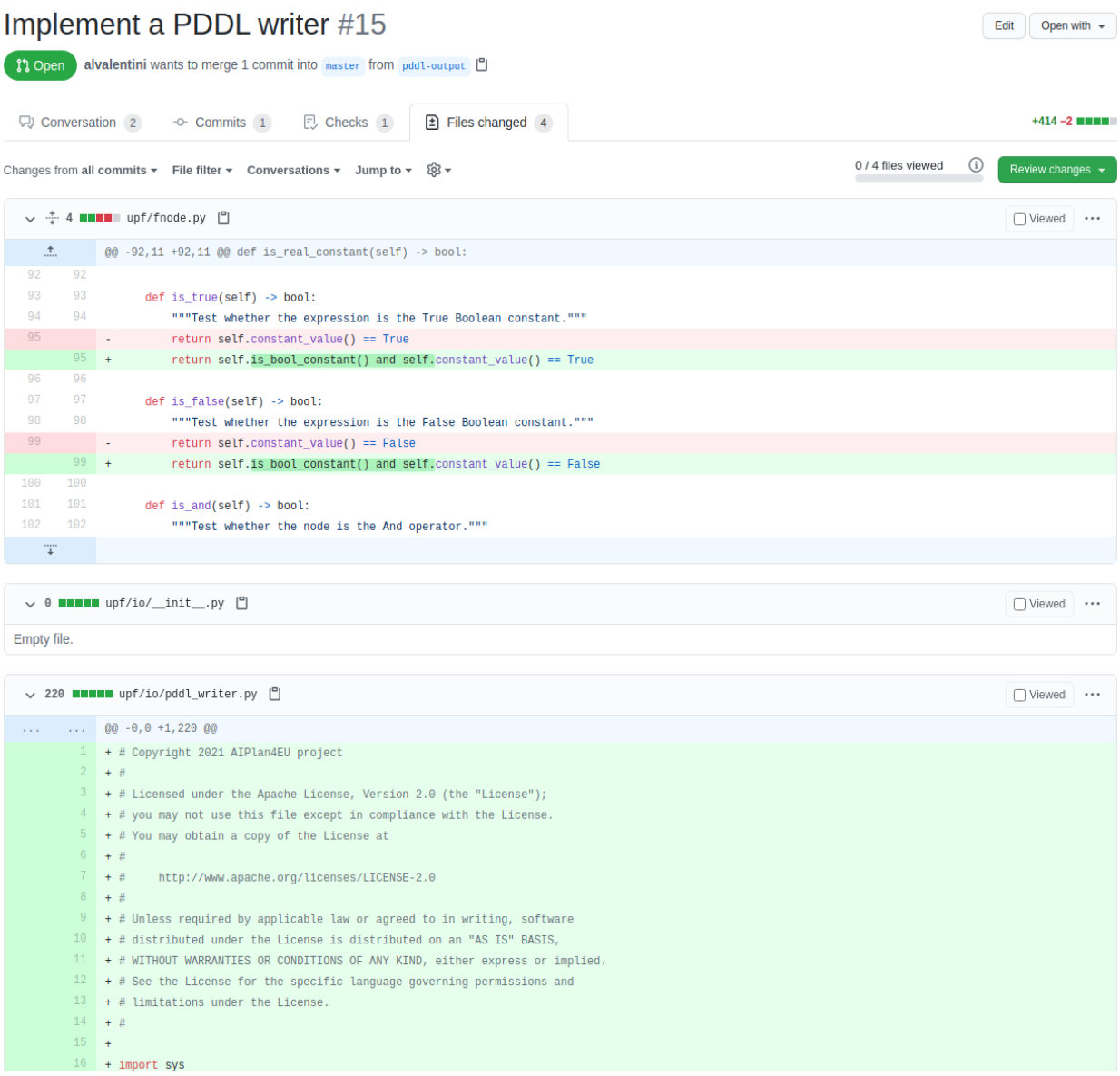

Figure 5 Code-review tab for a UPF pull-request

3 Continuous Integration

Continuous Integration (CI) is a technology-supported methodology consisting of automatically executing unit tests, integration tests and deployments at each commit. It is a must-have tool to monitor and maintain the quality of the software code, to avoid regression bugs and to simplify the review process of new code being contributed. GitHub integrates a CI solution that is free for open-source project called "GitHub Actions"[6], and AIPlan4EU takes advantage of this feature to automatically perform tests at each commit and for each pull-request.

[6] https://github.com/features/actions

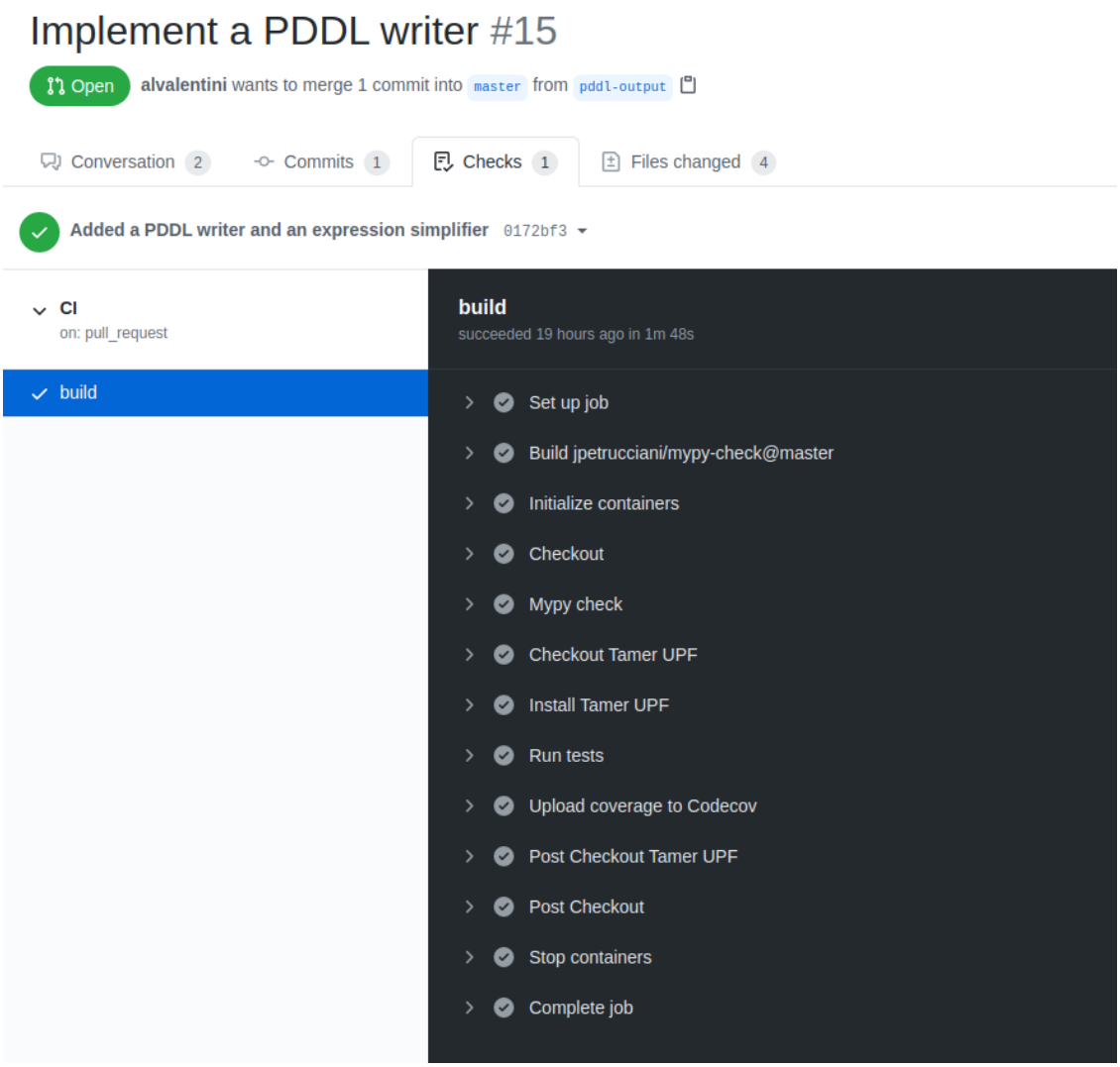

Figure 6 Automated checks perfromed by the UPF Continuos Integration

3.1 Unit-testing

In the AIPlan4EU software we will rely on unit-tests to ensure code quality and to avoid regressions during the development. At the time of writing, only the UPF software component is under active development, and it features a test suite based on the pytest[7] framework that is executed at every "push" automatically. In order to execute the tests, we created a containerized environment that uses the Docker[8] containerization technology. In particular, we maintain an ubuntu-based docker image[9] that is used to run the tests and contains the needed prerequisites. This creates a repeatable environment where problems can be locally reproduced, so that if a test fails the developers can easily access the environment where the problem occurred for debugging and analysis.

[7] https://pytest.org
[8] https://www.docker.com
[9] https://hub.docker.com/repository/docker/aiplan4eu/upf

3.2 Code coverage

Another useful feature that we adopted for the development of the UPF library is automatic code-coverage analysis. For this purpose, we use the codecov.io[10] service that is free of charge for open-source projects.

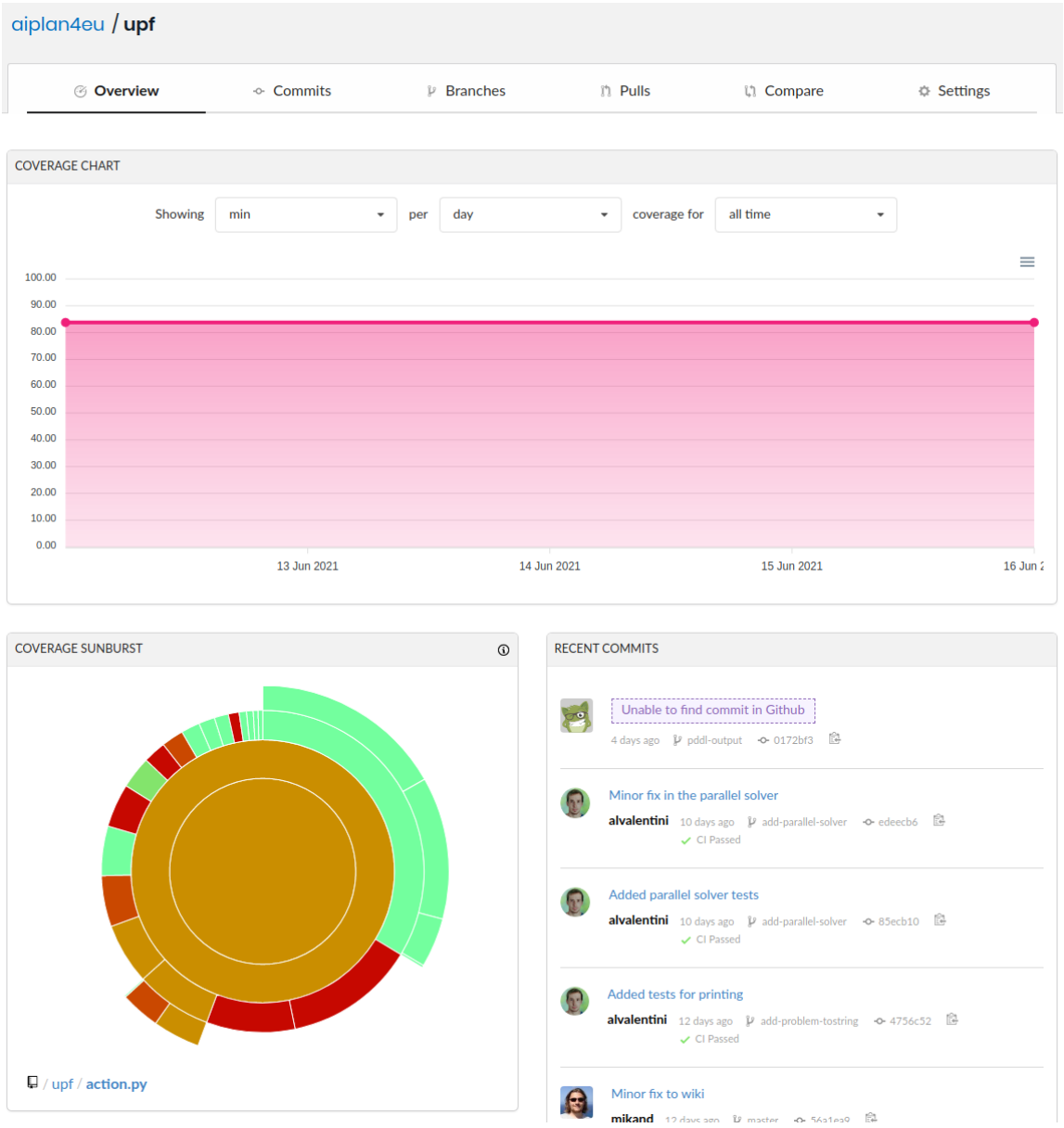

Figure 7 Codecov.io interface for the UPF project

The service automatically analyzes each pushed commit in the repository and prepares a report showing for each line of code how many times a test executes that specific line.

[10] https://codecov.io

```
26  1  class TypeChecker(walkers.DagWalker):
27  1      def __init__(self, env: 'upf.environment.Environment'):
28  1          walkers.DagWalker.__init__(self)
29  1          self.env = env
30
31  1      def get_type(self, expression: FNode) -> upf.typing.Type:
32          """ Returns the pysmt.types type of the expression """
33  1          res = self.walk(expression)
34  1          if res is None:
35              raise UPFTypeError("The expression '%s' is not well-formed" \
36                                 % str(expression))
37  1          return res
```

Figure 8 Codecov.io code analysis for the UPF code

This highlights portions of the code that are untested and helps in assessing new contributions. In fact, when a new pull request is opened, codecov.io automatically runs and reports a report of the changes in the coverage, helping the review process in assessing if the pull request features enough tests or not.

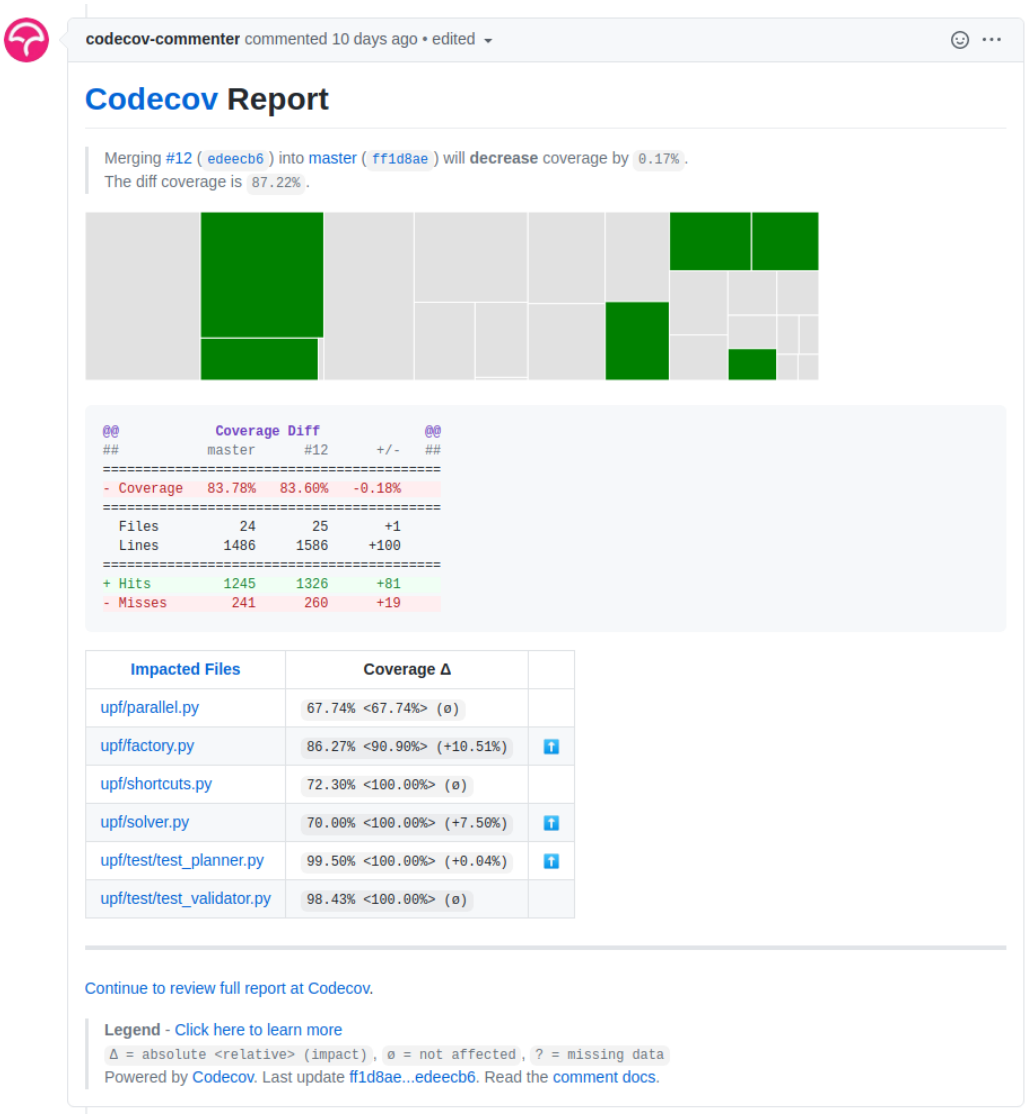

Figure 9 Codecov.io automatically comments on pull-requests showing the changes in coverage

3.3 Static type-checking

The UPF library is written in the Python programming language and several other software components of AIPlan4EU will be written in the same language. Python is a dynamic language that does not force the user to declare the types of variables and function parameters. In many situations this is a useful feature and it simplifies the coding, but for a library that is openly developed, having no type declaration on public functions and no type-checking could be a problem and generate confusion among developers and users. Thankfully, Python introduced an optional syntax for declaring types and the mypy[11] tool can be used to automatically perform a static type-checking of python code.

```
30  class ActionParameter:
31      """Represents an action parameter."""
32      def __init__(self, name: str, typename: upf.typing.Type):
33          self._name = name
34          self._typename = typename
35
36      def name(self) -> str:
37          """Returns the parameter name."""
38          return self._name
39
40      def type(self) -> upf.typing.Type:
41          """Returns the parameter type."""
42          return self._typename
```

Figure 10 Typed code in the UPF

As a coding rule within the project, we decided to enforce the types declaration on public functions and classes, while we allow for silent typing (i.e. not declaring types) for private code and in the body of functions. Our continuous integration automatically runs mypy at every commit and checks the type consistency. If an error is found the continuous integration fails and an error is reported to the developer and/or the pull-request reviewer.

[11] http://mypy-lang.org

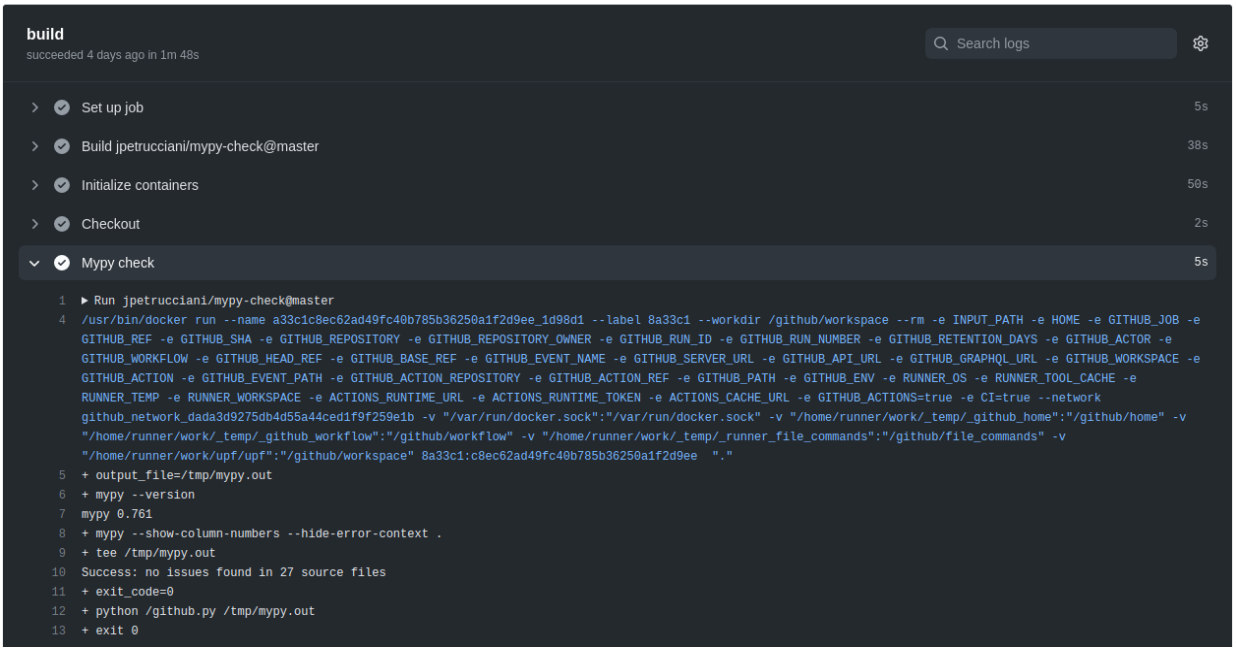

Figure 11 Mypy is automatically executed by the continuous integration of the UPF

4 Programming Languages and Interfaces

Each software component of AIPlan4EU is free to use the most appropriate programming language for the purpose at hand, but one central choice that was done within task T3.1 is the programming language and interfacing technologies for the UPF library. This is a central choice for the project and has ramifications on the choices for TSBs technologies as well as for planning interfaces. In order to operate this choice, we focused on two alternative implementation languages for the UPF, namely C++ and Python and we designed two alternative architectures. Then, we developed two proof-of-concepts of the two alternatives and empirically evaluated the performances as well as the software maintainability and limitations. In this section, we report the details of this analysis and the final decision of using the Python programming language that emerged after a thorough discussion with the project partners involved in WP3. All the proof-of-concept code was developed and is freely available in a separate repository: https://github.com/aiplan4eu/upf-poc.

4.1 General Requirements

While developing and analyzing the proof-of-concepts for the UPF library, we considered the following requirements that emerged from the discussion with the consortium within WP2 and WP3.

1. The UPF shall be available as a library, as well as a (containerized) service. The former is needed for several use-case where the UPF needs to run on the user premises, while the latter is needed for the integration within the AI on-demand platform and for the AIPlan4EU training program.
2. The UPF shall be able to use planning engines written in C/C++, Java/Scala and Python. These languages are used for the development of the planning engines that will be integrated within WP4.
3. The UPF shall be usable in Python, Java and C++ for developing TSBs. The discussion within WP2 highlighted that the programming languages used among the use-cases are heterogeneous.
4. It shall be possible to integrate the UPF in the AI on-demand platform Experiments. The platform uses ACUMOS for service orchestration and the UPF shall expose an ACUMOS-compliant gRPC[12] interface.

[12] https://grpc.io

5. The UPF library shall be usable as a jupyter notebook for training purposes.
6. It shall be possible to execute client code (i.e., code written in a TSB) from a planning engine. This is needed to implement custom behaviors such as custom, domain-dependent heuristics or to provide access to simulators for resources or planning actions.

These non-functional requirements have been discussed and agreed with all affected partners and the proof-of-concept activities focused on highlighting the trade-off of the two analyzed solutions in the scenarios highlighted by such requirements.

4.2 C++ Architecture

The first architect we developed and tested is based on the C++ programming language. We chose C++ as a representative of low-level languages that can be compiled into native machine code. The architecture we designed is depicted in the figure below.

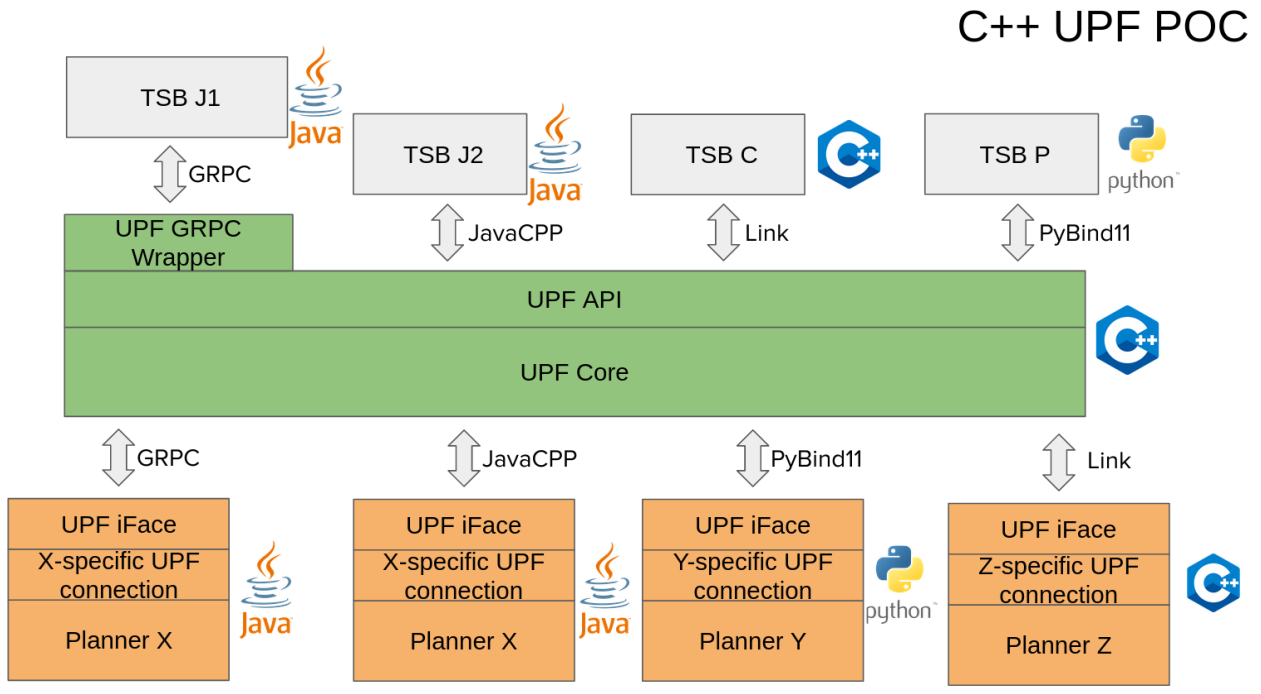

Figure 12 Architectural schema for the C++ UPF

In this solution, we implemented the UPF in C++ and we found a way to link planners written in C++, Python and Java using dynamic dlopen calls or, alternatively, through gRPC. The main technical issue of this solution is how to interface planners written in high-level programming languages such as Python or Java with the UPF as it is often less common to call these languages from C++ than calling C++ from such languages. This problem is exacerbated by the lack of a stable ABI for C++[13]: in fact, it is not guaranteed by the C++ standard that a C++ library compiled with a certain compiler (or compiler version) will correctly link with a client program compiled with a different compiler. The only sensible workaround solutions consists in either wrapping all the C++ APIs into pure C and using the "dlopen" call to load the libraries dynamically, or to use a language-independent Remote Procedure Call solution such as Google gRPC.
The former solution can be implemented in three different ways depending on the language of the planning engine.

[13] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4028.pdf

- for the C++ engines we can simply wrap the interface in C[14];
- for the Python engines we used PyBind11[15] to create a C interface that can call the Python code[16];
- for the Java engines we used JavaCPP[17] to create the C interface to call the Java code[18].

The gRPC solution is always possible but poses some limitations and performance issues. In fact, gRPC creates a socket server that exposes an interface that can be called by the UPF[19], but there is the overhead due to marshalling and unmarshalling objects (that is an object on one end of the communication needs to be serialized in a language-independent memory format and on the receiving end we need to read the message and convert it back to a usable representation in the target language); moreover there is also some overhead due to communication (even if this happens on a local machine, the TCP/IP stack of the OS is invoked and stimulated). In the empirical evaluation we quantified this overhead in different scenarios.

For the TSB side, we found solutions to call the UPF from different TSBs written in different languages:

- for TSBs in C++ we simply link the UPF[20];
- for TSBs in Python we use PyBind11 to create a Python wrapper around the UPF[21];
- for TSBs in Java we use JavaCPP to create a Java wrapper around the UPF[22].

The UPF can also expose its interface through a gRPC server and then it can be called by the AI4EU platform and by any other TSB[23].

All in all, this solution is feasible and very easy to use from the TSB perspective, but it is quite complicated for planning engines to be integrated and leaves on the ground the possibility of using object-oriented code in the integration of planning engines. Moreover, the use of a programming language that is compiled in native code, opens several issues for the interoperability of different platforms (e.g. between Linux, Windows and OSX) as the C++ code and the wrapping solutions shall be compilable and workable across these diverse systems.

4.3 Python Architecture

The second proof-of-concept is based on the python programming language that is nowadays very popular among AI frameworks (e.g. PyTorch or Tensorflow). The python-based architecture is depicted below.

[14] https://github.com/aiplan4eu/upf-poc/tree/main/planners/cpp/cpp_upf_wrapper
[15] https://github.com/pybind/pybind11
[16] https://github.com/aiplan4eu/upf-poc/tree/main/planners/python/cpp_upf_wrapper
[17] https://github.com/bytedeco/javacpp
[18] https://github.com/aiplan4eu/upf-poc/tree/main/planners/java/cpp_upf_wrapper
[19] https://github.com/aiplan4eu/upf-poc/tree/main/planners/java/grpc_cpp_client
[20] https://github.com/aiplan4eu/upf-poc/tree/main/upf/cpp/clients/cpp
[21] https://github.com/aiplan4eu/upf-poc/tree/main/upf/cpp/clients/python
[22] https://github.com/aiplan4eu/upf-poc/tree/main/upf/cpp/clients/java/JavacppClient
[23] https://github.com/aiplan4eu/upf-poc/tree/main/upf/cpp/grpc

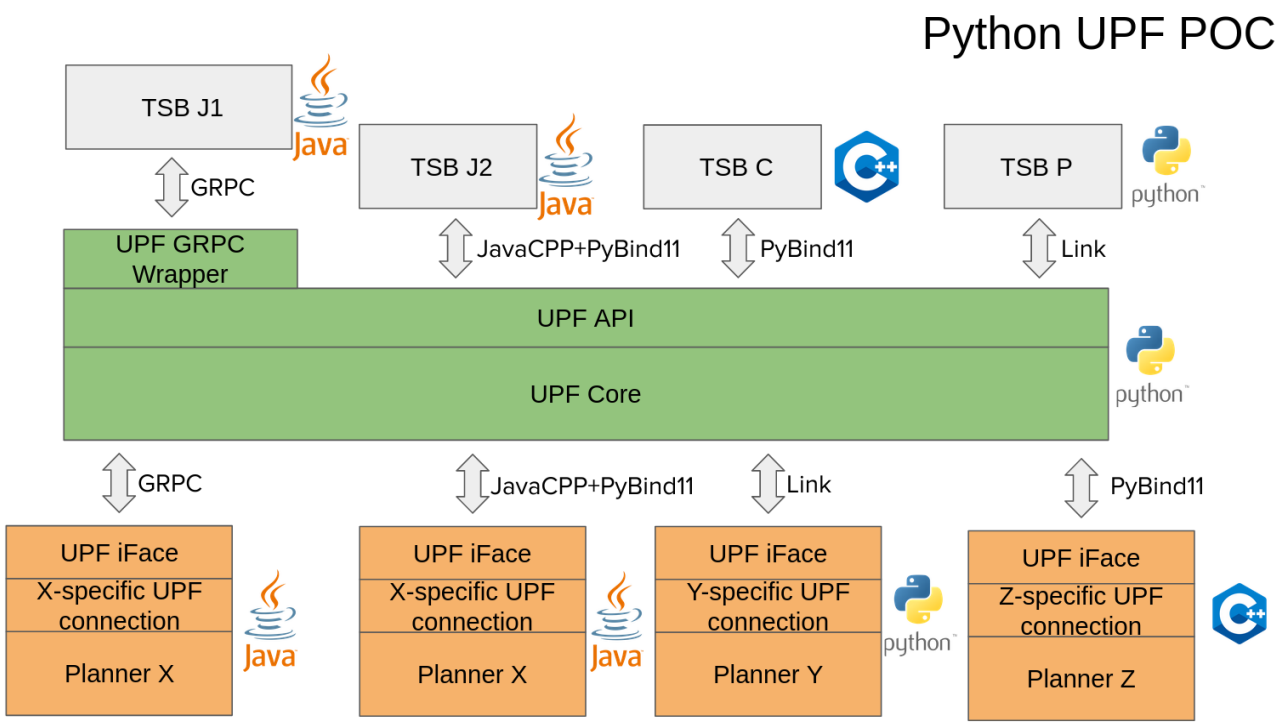

Figure 13 Architectural diagram for the Python UPF

In this second solution we implemented the UPF in Python and in this case we found different solutions to link planners and TSBs written in different languages.
The solutions through the gRPC framework works in the same way as the first solution[24] [25] because the gRPC library is language agnostic.

To dynamically import the planning engines in the python UPF, we used the "importlib" package that allows us to adopt these solutions:

- for the Python planning engines the link is automatically supported by the python language via a simple "import" statement[26];
- for the C++ engines we created a Python module using PyBind11[27];
- for the Java engines we created a Python module using both JavaCPP, that creates a C++ interface, and PyBind11, that creates the Python module using the C++ interface[28].

This solution is very simple for both Python and C++ and it allows the use of the full extent of the python language to provide convenient APIs to be implemented. Integrating Java or Scala planning engines requires a bit of labor, but is conceptually simpler than the integration of pure-C interfaces.

The link with the different TSBs works in this way:

[24] https://github.com/aiplan4eu/upf-poc/tree/main/planners/java/grpc_pyclient
[25] https://github.com/aiplan4eu/upf-poc/tree/main/upf/python/grpc
[26] https://github.com/aiplan4eu/upf-poc/tree/main/planners/python/py_upf_wrapper
[27] https://github.com/aiplan4eu/upf-poc/tree/main/planners/cpp/py_upf_wrapper
[28] https://github.com/aiplan4eu/upf-poc/tree/main/planners/java/py_upf_wrapper_javacpp

- for the Python TSBs is automatically supported by the python language via a simple "import" statement[29];
- for the C++ TSBs the call to the UPF can be done easily using PyBind11[30];
- for the JavaCPP TSBs the call can be done using both JavaCPP and PyBind11[31].

This solution turned out to be simpler to develop, as we encountered no linking errors and is much more portable than the C++ solution.

[29] https://github.com/aiplan4eu/upf-poc/tree/main/upf/python/clients/python
[30] https://github.com/aiplan4eu/upf-poc/tree/main/upf/python/clients/cpp
[31] https://github.com/aiplan4eu/upf-poc/tree/main/upf/python/clients/java/JavacppClient

4.4 Empirical Analysis

UPF	Client	Planner	Heuristic	Size=14	Size=15
Python	Python	cppplanner	No	1346.4	3321.6
Python	Python	cppplanner	Yes	903.4	2191.7
Python	Python	pyplanner	No	1397.8	3667.1
Python	Python	pyplanner	Yes	753.3	1690.2
Python	Python	jplanner	No	2349.6	8202.5
Python	Python	jplanner	Yes	3962.9	12175.4
Python	Python	jplanner_grpc	No	2810.1	8508.9
Python	Python	jplanner_grpc	Yes	15377.6	35287.0
Python	Java GRPC	cppplanner	No	1456.0	3577.0
Python	Java GRPC	cppplanner	Yes	21609.0	40139.0
Python	Java GRPC	pyplanner	No	1588.0	3872.0
Python	Java GRPC	pyplanner	Yes	15547.0	29073.0
Python	Java GRPC	jplanner	No	2776.0	8453.0
Python	Java GRPC	jplanner	Yes	29473.0	68194.0
Python	Java GRPC	jplanner_grpc	No	2409.0	9217.0
Python	Java GRPC	jplanner_grpc	Yes	29926.0	71745.0
Python	Java	cppplanner	No	1432.0	3436.0
Python	Java	cppplanner	Yes	2173.0	5114.0
Python	Java	pyplanner	No	1527.0	3991.0
Python	Java	pyplanner	Yes	1883.0	4798.0
Python	Java	jplanner	No	2455.0	8366.0
Python	Java	jplanner	Yes	8375.0	21358.0
Python	Java	jplanner_grpc	No	2614.0	8944.0
Python	Java	jplanner_grpc	Yes	43971.0	98459.0
Python	C++	cppplanner	No	1348.0	3348.0
Python	C++	cppplanner	Yes	992.0	2350.0
Python	C++	pyplanner	No	1493.0	3965.0
Python	C++	pyplanner	Yes	918.0	2077.0
Python	C++	jplanner	No	2552.0	8070.0
Python	C++	jplanner	Yes	4201.0	12876.0
Python	C++	jplanner_grpc	No	2413.0	9396.0
Python	C++	jplanner_grpc	Yes	16409.0	37517.0
C++	Python	cppplanner	No	1334.4	3275.9
C++	Python	cppplanner	Yes	1006.4	2378.3
C++	Python	pyplanner	No	1389.5	3708.7
C++	Python	pyplanner	Yes	1077.1	2421.5
C++	Python	jplanner	No	2461.2	8273.9
C++	Python	jplanner	Yes	4069.7	13137.7
C++	Python	jplanner_grpc	No	2361.1	9196.3
C++	Python	jplanner_grpc	Yes	11174.8	29679.1
C++	Java GRPC	cppplanner	No	1667.0	3659.0
C++	Java GRPC	cppplanner	Yes	10416.0	21404.0
C++	Java GRPC	pyplanner	No	1594.0	4059.0
C++	Java GRPC	pyplanner	Yes	9118.0	17454.0
C++	Java GRPC	jplanner	No	2657.0	8773.0
C++	Java GRPC	jplanner	Yes	13296.0	30345.0
C++	Java GRPC	jplanner_grpc	No	2599.0	8754.0
C++	Java GRPC	jplanner_grpc	Yes	20336.0	71649.0
C++	Java	cppplanner	No	1467.0	3779.0
C++	Java	cppplanner	Yes	1064.0	2999.0
C++	Java	pyplanner	No	1521.0	4370.0
C++	Java	pyplanner	Yes	1258.0	2740.0
C++	Java	jplanner	No	2497.0	8663.0
C++	Java	jplanner	Yes	4140.0	12888.0
C++	Java	jplanner_grpc	No	2517.0	9056.0
C++	Java	jplanner_grpc	Yes	12051.0	29425.0
C++	C++	cppplanner	No	1386.0	3337.0
C++	C++	cppplanner	Yes	847.0	2002.0
C++	C++	pyplanner	No	1551.0	3957.0
C++	C++	pyplanner	Yes	1030.0	2209.0
C++	C++	jplanner	No	2476.0	8418.0
C++	C++	jplanner	Yes	3844.0	12190.0
C++	C++	jplanner_grpc	No	2339.0	9319.0
C++	C++	jplanner_grpc	Yes	10057.0	27418.0

Figure 14 Empirical experimentation results. The numbers reported in the last two columns are in milliseconds of execution time.

The two solutions have been empirically tested with either a simple call (the client asks to find a plan for a given problem) and with a bidirectional communication (the client asks to find a plan for a given problem using a TSB-provided heuristic that needs to be evaluated often by the planner).

The experimental results show that the first solution is only marginally more efficient than the second one. Moreover, the overhead of gRPC communication is negligible for simple calls while it becomes extremely evident for frequent communication between the TSBs and the engines across the spectrum

Given the above results and the linking and portability issues exhibited by the C++ solution, the consensus within WP3 has been to proceed with the development of the UPF in the Python programming language.

5 Conclusion

This deliverable reported the activities carried on within task T3.1 that formalized and implemented the development practices to be used within the AIPlan4EU project. Moreover, we described and motivated the technological choices behind the UPF development that is currently undergoing. The practices and methodologies identified within will be transferred to the development of both Planner Interfaces and TSBs.
The UPF development will serve as validation for the choices and technologies that we presented; Deliverable D3.3. UPF Design will report the results of the evaluation as well as any changes or adaptations that will be deemed necessary for the development.