University of Padova

Department of Mathematics

Computer Science

Asynchronous AMPL REST interface written in Python 3.6

AMPLRESTAPI

Operations Research project report

Academic Year 2018-19

Alberto Schiabel | 1144672

Repository: https://github.com/jkomyno/amplrestapi

Contents

List of Figures

Listings

List of Tables

1 Abstract

The **AMPLRESTAPI** project intends to demonstrate how to create and use a custom asynchronous REST API layer written in modern Python to solve optimization problems in AMPL. This prototype augments the features provided by AMPL and allows users to solve an optimization problem from any Internet-capable device. Since every computation of a problem's solution is performed on the machine where the REST server runs, the users don't need to install an AMPL suite on their devices. Another possibility offered by the **AMPLRESTAPI** REST API is to dynamically adapt the problem constraints based on the input set cardinality.

The main aim of this report is to explain how the server application is structured and what's the optimization problem chosen as an example. The abstraction layer offered by a REST API can be further exploited to ease the way the user insert the problem data into the system, for example through a web interface or a mobile app. While a similar approach is possible and encouraged, it's currently out of the scope of this project.

The AMPL solver used by this project is **CPLEX**.
As of August 2019, the example problem in this project can be solved with a demo version of AMPL. The reader doesn't need to have any version of AMPL downloaded on his computer, as explained in greater detail in section 6 on page 10.

2 Technology Stack

In this section, the author presents a brief description of the technology stack that forms the core of the AMPLRESTAPI project.

2.1 REST API

A REST API is an *Application Program Interface* (API) that uses HTTP requests to exchange data between a client and a server. Every data exchange operation uses an HTTP verb (such as *GET* and *POST*) and points to a server URL.

A server that exposes a REST API is often referred to as a *RESTful service.* The RESTful server implemented in **AMPLRESTAPI** is stateless since it doesn't retain any data inserted by the user, it simply performs a computation a returns the result.

In this project, the data exchange between the client and the server uses the JSON format, which has been chosen because:

- It's human-readable;
- It's a reasonably light text format;
- It's extremely common in REST APIs.

Note: the author has already had significant experience with RESTful services.

2.2 Python 3.6

Python is a general-purpose programming language which is interpreted, dynamic and high-level. It combines the object-oriented paradigm with some functional programming features, such as list comprehension. **AMPLRESTAPI** has been developed using this programming language because:

- It's free and open-source;
- It has one of the most complete standard libraries among the most famous programming languages;
- Its high-level built-in data structures and dynamic semantics make it a good choice for rapid application development;
- It's the main programming language of reference for the scientific community, in particular for mathematicians, data analysts, and statisticians;
- There is an official AMPL client written in Python, *amplpy*;
- It's generally faster than domain specific languages like Matlab, Julia, and R.

Note: the author has had limited experience with Python 3.x, and had never used it for server development.

2.3 AIOHTTP and Asyncio

AIOHTTP is an asynchronous Server for Python and Asyncio.

Asyncio is the most famous Python library for writing single-threaded concurrent code using coroutines, multiplexing I/O access over sockets and other resources and running network clients and servers. Asyncio is what allows AIOHTTP (and, by extension, the API exposed by **AMPLRESTAPI**) to be asynchronous, reactive and non-blocking.

Note: the author has had never used AIOHTTP or Asyncio before this project.

2.4 Docker

Docker is a tool that makes it easy to create, deploy and run applications by using sandbox containers. Containers allow a developer to package up an application with all of the parts it needs to execute, such as a Linux-based operative system, the Python runtime, the third-party dependencies, and even initialization scripts, shipping it all out in one tiny package. Every Docker container is similar to a virtual machine, but it's much lighter and offers better performance. Each Docker container definition is stored in a special file called *Dockerfile*, from which a so-called Docker image can be created. Docker images are the easiest way to deploy a system to the cloud, for example making the server run on *AWS ECS*.

In this project, Docker has been used to relieve the interested reader from the struggle of installing the required dependencies and to make it possible to boot the server up in a matter of seconds. Optionally, it's possible to run this project without Docker, as shown in the subsection 6.2, but it's discouraged.

Note: the author has already had significant experience with Docker.

3 Example Problem

This project has been created to offer a facade that simplifies the interaction with AMPL solvers as well as dynamically creating constraints based on input data.
Thus, the author has decided to choose an optimization problem of medium complexity, so that the attention of the reader can stay focused on the main purpose of **AMPLRESTAPI**.

3.1 JIT - Just In Time Computation

The following paragraph presents a variation of the "Just In Time Computation" problem where each static constraint has been removed in favor of dynamic constraints, i.e. semantically equivalent constraints applied at runtime and derived from the input data.
From now on, this "Just In Time Computation" variation problem will be referenced as the *JIT* problem.

3.1.1 Problem Definition

A computational server must plan the execution of n batches on a single-processor machine, where $n > 0$. Each batch may take a different amount m of minutes, where $m > 0$. For simplicity, each item in m must be an integer value. The execution sequence $1, 2, ..., n$ is given and can't be changed. There can't be any temporal overlapping between the batches. For each batch, the desired delivery time is known. The delivery of the computed batches must be as on time as possible: a fixed fee must be paid for each minute early or late. Schedule the execution times in order to minimize the total fee.

3.1.2 Mathematical Model

Decisional Variables

- $start\_time_j$: minute when the server starts elaborating the batch j, $\forall j \in BATCH$;
- $delta\_time_j$: number of minutes wrong with respect to expected finish time of the job j, $\forall j \in BATCH$;

Parameters

- $duration_j$: number of minutes that each program j takes to complete, $\forall j \in BATCH$;
- $expected\_finish_j$: minute when the server is expected to stop processing the batch j, $\forall j \in BATCH$;
- $wrong\_time\_fee = \ldots \$$: fixed amount of dollars to pay for each program result that isn't computed in exactly the estimated time.

Objective Function and Constraints
Note: in order to compute $delta\_time_j\ \forall j \in BATCH$ it's necessary to both the jobs that finished early and the jobs that finish late. This can be represented using absolute values: $delta\_time_j = |start\_time_j + duration_j - expected\_finish_j|$. However, in order to fit this kind of constraint into a linear programming model, it must be linearized.

- **min**: $wrong\_time\_fee * \sum_{b \in BATCH} delta\_time_b$;
- **Absolute values linearization**:
  - ⋆ $delta\_time_j >= +start\_time_j + duration_j - expected\_finish_j$, $\forall j \in BATCH$;
  - ⋆ $delta\_time_j >= -start\_time_j - duration_j + expected\_finish_j$, $\forall j \in BATCH$;
- **Precedence and not overlapping of batches**: $start\_time_j + 1 >= start\_time_j + duration_j$, $\forall j \in BATCH$.

Domains

- $duration_j \in \mathbb{Z}$, $\forall j \in BATCH$;
- $delta\_time_j \in \mathbb{R}$, $\forall j \in BATCH$.

3.1.3 AMPL Code

Listing 1: AMPL code for the JIT problem

```
# set declarations
set BATCH; # set where each item maps to a single program

# param declarations
param duration{BATCH};         # duration of each program
param expected_finish{BATCH}; # time each program should take to complete
param wrong_time_fee;         # dollars to pay for each program result that
                              # isn't computed in exactly the estimated time

# variable declarations
var start_time{BATCH}; # starting time of each program
var delta_time{BATCH}; # quantity of time wrong with respect to expected arrival

# objective function
# minimize the fixed fee to pay for each computation started either early or late
minimize total_fee:
         wrong_time_fee * sum{b in BATCH} delta_time[b];

# wrong delta time is at least the exact wrong time.
# This is a linearization of a modulo equation
s.t. delta_time_abs_1{b in BATCH}:
     delta_time[b] >= - expected_finish[b] + (start_time[b] + duration[b]);
s.t. delta_time_abs_2{b in BATCH}:
     delta_time[b] >= + expected_finish[b] - (start_time[b] + duration[b]);
```

4 Program Description

This section describes the most relevant topics about the software implementation of the **AMPLRESTAPI** project.

4.1 Third-Party Dependencies

The software makes use of a small number of third-party Python packages, which are resumed in the following list:

- aiohttp@3.5.4: asynchronous Server for Python and Asyncio;
- amplpy@0.6.7: official AMPL client for Python;
- asyncio@3.4.3: library that provides asynchonicity support to Python methods;
- confuse@1.0.0: opinionated configuration management library;
- jsonschema@3.0.2: library used to validate a JSON input against a JSON file that follows the JSON Schema Specification.

Each of these dependencies (as well as their specific versions) is listed in the requirements.txt file.

4.2 Application Configuration

The application configuration is stored in the config/config-default.yaml file and is parsed using the *confuse* library. The project uses a bare minimum configuration, as shown in Listing 2.

Listing 2: Default YAML configuration

```
# Project version
version: 0.1.0

app:
  # IP of the server
  host: 0.0.0.0

  # Port exposed by the REST server
  port: 9001
```

4.3 Input Validation

Not all input data may be interpreted correctly by an AMPL model, and some constraints may only be applicable in a subset of possible inputs in a certain numeric domain. For this reason, the REST server must verify that the input data complies with a set of validity rules. Even if a single validity rule isn't respected, the input is rejected and an error message is returned to the user. The are three types of error that might occur while interpreting user input:

1. Parse errors;
2. Validation errors;
3. Semantic errors.

4.3.1 Parse errors

Parse errors are raised when the input is expressed in a structure and format that isn't parseable by the system in use. **AMPLRESTAPI** reads user input from the JSON body of HTTP POST requests.
If a request isn't expressed in an `application/json`-compliant format, the user will be returned an HTTP Bad Request error message.
These errors are caught by the custom `handle_bad_request_error` AIOHTTP middleware defined in the amplrestapi/middlewares.py file and returned to the user.

4.3.2 Validation errors

Validation Errors are raised when the input doesn't respect the structure and the numeric boundaries defined in the validation schema. In these cases, an error message is returned to the user in the form of an HTTP Unprocessable Entity Error. The middleware responsible of returning the appropriate error code is called `handle_unprocessable_entity_error` and is defined in the amplrestapi/middlewares.py file.

4.3.3 Semantic errors

Semantic errors occur when the input is defined in a properly parseable format and has the exact expected structure as the one required by the validation schema, but the input values don't respect the semantic rules of the problem, making it unsolvable. For example, in the JIT problem (see 3.1), the timestamps of the expected computation delivery of each job must be sorted in ascendant order. Just like validation errors, semantic errors are returned to the user in the form of an Unprocessable Entity Error message.

4.4 Dynamic AMPL Model Evaluation Process

The following list describes the steps required to dynamically evaluate and solve an AMPL model in **AMPLRESTAPI**. The example AMPL model refers to the JIT problem (see 3.1).

- Import model file in memory (only once for the entire server lifespan);
- Read model input data from the body of an HTTP request;
- Validate the input and throw an error in case of invalid values and/or input structure;
- Reset the AMPL wrapper instance;
- Evaluate the model;
- Generate dynamic constraints according to the cardinality of the input data (*n_batches*) and the specific problem to solve;
- Generate tabular parametric amplpy *DataFrames* over a set ($BATCHES$);
- Set the scalar value of the fixed cost ($wrong\_time\_fee$) to AMPL;
- Evaluate the model and measure the time required to find a solution and the number of iteration performed by *CPLEX*;
- Extract the most relevant results from the solved model;
- Return them to the user.

5 Code Structure

The project consists of several small Python packages, each of which provides a specific set of features. These packages are composable and decoupled to make it easy for other developers to expand the project and add new features.
In the following sections, each package will be presented briefly. Further explanations are available in the comments present in the source code.

5.1 amplwrapper

amplwrapper contains the AMPLWrapper class (defined in amplwrapper/ampl_wrapper.py), which is a Python class responsible for initializing an *amplpy.AMPL* instance, setting the default options and solver (*CPLEX*), augmenting the functionalities of amplpy and exposing the methods for interacting with the actual AMPL instance.

It also defines the OutputHandler and ErrorHandler implementations for AMPL, which are respectively defined in the AMPLOutputHandler class (amplwrapper/ampl_output_handler.py) and in the AMPLErrorHandler class (amplwrapper/ampl_error_handler.py). While *AMPLErrorHandler* just logs the errors and warning emitted by AMPL when evaluating the model, *AMPLOutputHandler* uses Regex to retrieve the total number of dual iterations necessary to solve the problem.

5.2 ampljit

ampljit contains the model definition of the 3.1 problem. It's completely decoupled from classes that handle HTTP requests or input validation. This package can be easily embedded in any other AMPL user interface, such as a terminal application, so its usage is not limited to a REST interface.

The *ampljit* package only contains the AMPL model definition of the problem (stripped of the dynamic constraints, which are appended at runtime by a Python method), the **solve** method needed to insert the input data in AMPL and retrieve the solution results (defined in ampljit/model.py), and a set of utilities needed to convert the input data in the most appropriate data structures. Most of the utilities are one-liners that make use of Python's powerful list comprehension.

5.3 amplrestapi

amplrestapi is the core of the REST interface. It uses *AIOHTTP* and *asyncio* to set up a REST server and HTTP router, as well as decorating said server with runtime error management middlewares. The only route currently exposed by the server is */problems/jit*, which executes the code defined in the *jit* subfolder, which wires the contents exported from the package that knows how to solve the JIT problem (ampljit) with the AIOHTTP classes needed to create a REST interface around them.

The *JITRouteHandler* class (defined in amplrestapi/jit/route_handler.py) receives an existing *amplwrapper.AMPLWrapper* instance and an appropriate solver method via **Dependency Injection** and initializes an *asyncio.Lock*. The lock is needed to appropriately handle concurrency. If no lock was present and two JIT-related requests reached the server at about the same time, both of them would receive the solution of the last request, even if the input data of the first request was different. Obviously, the response returned by the first request in this case would be wrong. This means that the injected *AMPLWrapper* instance is a shared resource that requires a blocking access policy.

Listing 3: AMPL server critical section detail

```
async def run(self, request: web_request.Request):
    # input parsing and verification is omitted

    # solve the problem in a critical section.
    # self.ampl is a shared resource whose access is regulated by an asyncio.Lock.
    async with self._lock:
        json_response = self._solver(ampl=self.ampl, n_batches=n_batches,
                                     wrong_time_fee=wrong_time_fee,
                                     duration_lst=duration_lst,
                                     expected_finish_datetime_str_lst=expected_finish_lst)

    # This block is outside of the previous critical section.
    # The computation results has been gathered and can be returned to the user.
    return web.json_response(json_response)
```

6 How to run the HTTP server

This section describes how to run the project prototype to solve Linear Programming optimization problems using the HTTP protocol.

6.1 Using Docker

This approach is the easiest and most encouraged. The requirements are the following:

- Stable internet connection (required only the first time the project is run);
- At least 8 GB of RAM;
- Docker version 19.03.1 or superior *previous versions should work too, but they have not been tested by the author*;

Note for Windows users: Docker can only be installed in Windows 10 Professional and Windows 10 Education.

To run the application:

- Open a terminal window that points to the root of the project;
- Type `docker-compose up --build` ;

The output should be something similar to the following text:

Listing 4: Example Docker output

```
Building amplrestapi
Step 1/7 : FROM leotac/ampl-notebook
 ---> 3227f6bc13ef
Step 2/7 : WORKDIR /app
 ---> f5ada6f83838
Step 3/7 : COPY requirements.txt /app/requirements.txt
 ---> ce7b83a3a37d
Step 4/7 : RUN pip install -r requirements.txt
 ---> 710aaf53fed2
Step 5/7 : COPY . /app
 ---> 6663062b9ace
Step 6/7 : EXPOSE 9001
 ---> Running in 193fdb631870
Removing intermediate container 193fdb631870
 ---> 333d9f9e28e0
Step 7/7 : CMD python -m amplrestapi
 ---> Running in 714a43b24a92
Removing intermediate container 714a43b24a92
 ---> 44b14101dfec

Successfully built 44b14101dfec
Successfully tagged operative-search-project_amplrestapi:latest
Attaching to amplrestapi
amplrestapi    | DEBUG:root:Running app on 0.0.0.0:9001...
```

6.2 Directly on an operative system (discouraged)

This approach is generally discouraged because:

- The system must have exactly the same version of Python as the one required;
- The system must have AMPL installed in a specific directory, otherwise the application isn't guaranteed to run correctly;
- The third party dependencies needed for this project may clutter the system.

The reader that doesn't wish to use Docker must:

- Install Python 3.6.9;
- Install AMPL and make it available to the PATH environment variable;
- Install Python dependencies: `pip install -r requirements.txt` ;
- Run the *amplrestapi* Python package: `python -m amplrestapi` .

7 Server Interaction

Once the **AMPLRESTAPI** server is up and running, HTTP requests can be sent to it via any HTTP client. In this tutorial, the HTTP clients considered are Postman and Curl. We will consider the JIT problem (3.1), which, if the server is running in the local machine, is accessible via **POST** requests at localhost:9001/problems/jit. Since the POST request payload is in *JSON* format, each request should add the *Content-Type: application/json* header.

7.1 API Documentation

A complete API documentation written in the *OpenAPI3* format is available at docs/openapi.yml. Each input and output field is documented in detail and contains an example. An HTML preview of the same documentation is also available at https://jkomyno.github.io/amplrestapi/index.htm

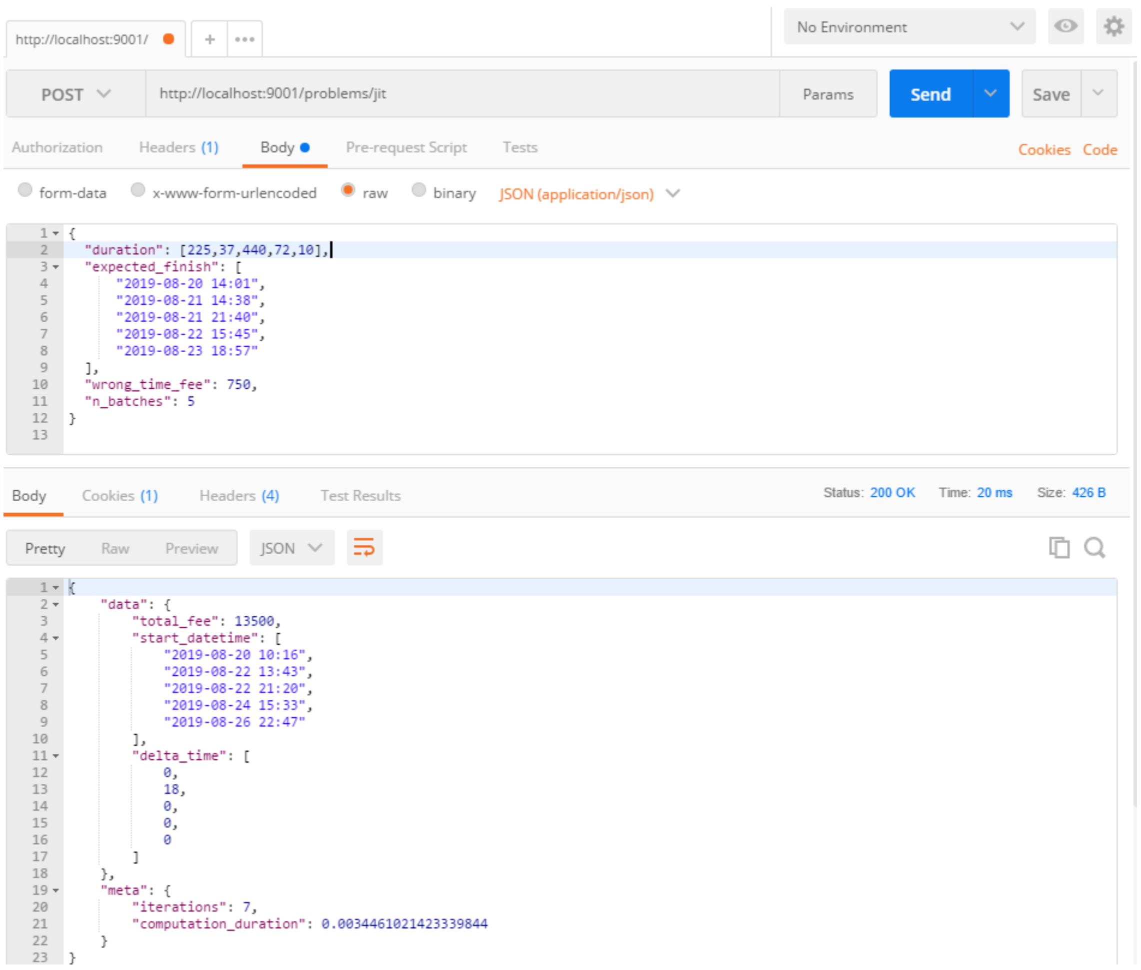
7.2 Solve a problem with Postman

In order to use **Postman**, it must first of all be downloaded from the official website.

Write "http://localhost:9001/problems/jit" to the request URL input field. Since each REST request should be a POST request, select the POST HTTP verb from Postman. In the "Body" tab, write the JSON payload and select the "raw" and "JSON (application/json)" options. When you're ready, you should click the "Send" button. The result should be similar to the one shown in Figure 1.

7.3 Solve a problem with Curl

Suppose that a BASH terminal window is open and that a file named *example-input-1.json* exists in the current directory. That file's content will be the payload of the request to be sent to the server. For this example, the author prefers to use a file instead of a JSON string due to string escaping issues that may arise and reduce readibility.



```
{
  "duration": [225,37,440,72,10],
  "expected_finish": [
      "2019-08-20 14:01",
      "2019-08-21 14:38",
      "2019-08-21 21:40",
      "2019-08-22 15:45",
      "2019-08-23 18:57"
  ],
  "wrong_time_fee": 750,
  "n_batches": 5
}
```

Status: 200 OK Time: 20 ms Size: 426 B

```
{
    "data": {
        "total_fee": 13500,
        "start_datetime": [
            "2019-08-20 10:16",
            "2019-08-22 13:43",
            "2019-08-22 21:20",
            "2019-08-24 15:33",
            "2019-08-26 22:47"
        ],
        "delta_time": [
            0,
            18,
            0,
            0,
            0
        ]
    },
    "meta": {
        "iterations": 7,
        "computation_duration": 0.0034461021423339844
    }
}
```

Figure 1: Example Postman usage to interact with **AMPLRESTAPI**.

Listing 5: Server request with Curl

```
# The "| json_pp" part can be omitted, as it's only used to beautify
# the JSON output of the server.

curl -H "Content-Type: application/json" \
     --data-binary @example-input-1.json \
     http://localhost:9001/problems/jit | json_pp
```

The result should be similar to the output displayed in Listing 6.

Listing 6: Server response with Curl and json_pp

```
{
   "meta" : {
      "iterations" : 8,
      "computation_duration" : 0.00409722328186035
   },
   "data" : {
      "delta_time" : [1, 0, 0, 0, 5],
      "total_fee" : 4500,
      "start_datetime" : [
         "2019-08-22 14:26",
         "2019-08-22 14:31",
         "2019-08-22 14:38",
         "2019-08-22 14:45",
         "2019-08-22 14:52"
      ]
   }
}
```

8 Conclusions

The author believes to have demonstrated how it's possible to create a REST layer over an AMPL-based resolutor, and how to apply dynamic problem constraints extracted from data inserted by the user.

8.1 Major obstacles found

The following paragraphs describe some of the major issues encountered by the author while preparing the **AMPLRESTAPI** project.

8.1.1 Amplpy documentation

Understanding how to use *amplpy* has been the most difficult aspect of developing this project. As of August 2019, the documentation of the official AMPL client for Python is quite lacking, and the given examples aren't as illustrative as a developer would hope. Several terms and class descriptions were based on some implicit assumptions that didn't seem to be explained anywhere (or, if they did, they weren't easily discoverable).

Luckily, amplpy is an open-source library, and some doubts have been solved carefully inspecting the source code. Trial and error in a Python interactive session has also been really helpful to understand what amplpy's *DataFrame* class was capable of, and what its constraints were.
The lack of up-to-date real-life examples of amplpy usage has also radically slowed down the development of **AMPLRESTAPI**.

8.1.2 Amplpy limitations

In some occasions, amplpy didn't offer all the features needed for this project, so they had to be implemented from scratch. For example, the author was interested in programmatically reporting the number of dual iterations required to solve a linear programming problem, but the API exposed by AMPL didn't offer this possibility. However, amplpy offers the possibility to manage all the log output emitted by AMPL via a Python class that implements the *amplpy.OutputHandler* Python class. This has been exploited to retrieve the number of dual iterations, combining the OutputHandler signals with Regex manipulation and a callback that would forward the number of iterations reported by AMPL. The code that accomplishes this feature can be found in the amplwrapper/ampl_output_handler.py file.

8.2 Future Work

Since the prototype features are currently quite limited (there's simply a REST facade and a single optimization problem described by an AMPL model), the community might benefit from a number of improvements, such as:

- Add new problem models, each of which should be exposed in a REST URL;
- Provide a web interface that allows users to interact with the model without needing to perform HTTP requests manually;
- Create a command line application that allows users to solve particular problems locally via a Python interface. Note that this is already possible because the **AMPLRESTAPI** packages that implement the dynamic version of AMPL problems are decoupled from the code executed by the server;
- Add a fixed threaded pool of AMPL instances for each problem, such that multiple. This would require changing the `asyncio.Lock` that regulates the access to the AMPL problem-solving critical region into a `asyncio.Semaphore` combined with a thread executor;
- Create a distributed microservice architecture where each AMPL problem is computed on a different node. This would perhaps be the most effective improvement, that, along with multiple server replicas, would guarantee a scalable service accessible to millions of people.

A Execution Examples

This chapter shows some example executions resulting from running the JIT problem (3.1) with different input data. The examples can be simulated with any tool capable of performing HTTP requests; in particular, we consider **Postman** and **Curl**, which are the *de facto* instruments to achieve this purpose.
Each example assumes that a **AMPLRESTAPI** server is running on localhost:9001. Please refer to section 6 (*"How to Run"*) in case of doubts.

Note: for the next examples, the *wrong_time_fee* penalty cost is set to 750.

A.1 5 programs, same hour of the same day

Table 1: Input data for 5 programs scheduled at the same hour of the same day

BATCH	duration	expected_finish
1	5	2019-08-22 14:32
2	7	2019-08-22 14:38
3	4	2019-08-22 14:42
4	7	2019-08-22 14:52
5	10	2019-08-22 14:57

Table 2: Output data for 5 programs scheduled at the same hour of the same day

BATCH	start_datetime	delta_time	total_fee	iterations
1	2019-08-22 14:26	1	4500	8
2	2019-08-22 14:31	0		
3	2019-08-22 14:38	0		
4	2019-08-22 14:45	0		
5	2019-08-22 14:52	5		

A.2 5 programs, different hours of the same day

Table 3: Input data for 5 programs scheduled at different hours of the same day

BATCH	duration	expected_finish
1	25	2019-08-22 14:01
2	37	2019-08-22 14:38
3	44	2019-08-22 15:42
4	17	2019-08-22 15:45
5	20	2019-08-22 18:57

Table 4: Output data for 5 programs scheduled at different hours of the same day

BATCH	start_datetime	delta_time	total_fee	iterations
1	2019-08-22 13:36	0		
2	2019-08-22 14:01	0		
3	2019-08-22 15:44	14	10500	7
4	2019-08-22 16:28	0		
5	2019-08-22 22:37	0		

A.3 5 programs, different hours of different days

Table 5: Input data for 5 programs scheduled at different hours of different days

BATCH	duration	expected_finish
1	225	2019-08-20 14:01
2	37	2019-08-21 14:38
3	440	2019-08-21 21:40
4	72	2019-08-22 15:45
5	10	2019-08-23 18:57

Table 6: Output data for 5 programs scheduled at different hours of the same day

BATCH	start_datetime	delta_time	total_fee	iterations
1	2019-08-20 10:16	0		
2	2019-08-22 13:43	18		
3	2019-08-22 21:20	0	13500	7
4	2019-08-24 15:33	0		
5	2019-08-26 22:47	0		

A.4 10 programs, same hour of the same day

Table 7: Input data for 10 programs scheduled at the same hour of the same day

BATCH	duration	expected_finish
1	15	2019-08-22 14:01
2	7	2019-08-22 14:15
3	5	2019-08-22 14:20
4	4	2019-08-22 14:22
5	10	2019-08-22 14:30
6	2	2019-08-22 14:39
7	5	2019-08-22 14:40

Continued on next page

Table 7 – Continued from the previous page

BATCH	duration	expected_finish
8	3	2019-08-22 14:46
9	7	2019-08-22 14:52
10	10	2019-08-22 14:59

Table 8: Output data for 10 programs scheduled at the same hour of the same day

BATCH	start_datetime	delta_time	total_fee	iterations
1	019-08-22 13:46	0	10500	22
2	019-08-22 14:06	3		
3	019-08-22 14:13	2		
4	019-08-22 14:18	2		
5	019-08-22 14:22	0		
6	019-08-22 14:33	2		
7	019-08-22 14:35	4		
8	019-08-22 14:42	0		
9	019-08-22 14:45	1		
10	019-08-22 14:52	0		

A.5 10 programs, different hours of the same day

Table 9: Input data for 10 programs scheduled at different hours of the same day

BATCH	duration	expected_finish
1	120	2019-08-22 07:01
2	50	2019-08-22 08:55
3	45	2019-08-22 09:35
4	90	2019-08-22 11:00
5	25	2019-08-22 11:25
6	28	2019-08-22 12:00
7	42	2019-08-22 12:40
8	72	2019-08-22 13:45
9	150	2019-08-22 16:15
10	39	2019-08-22 16:55

Table 10: Output data for 10 programs scheduled at different hours of the same day

BATCH	start_datetime	delta_time	total_fee	iterations
1	2019-08-22 05:01	0		

Continued on next page

24000 24

Table 10 – Continued from the previous page

BATCH	start_datetime	delta_time	total_fee	iterations
2	2019-08-22 08:55	10		
3	2019-08-22 10:45	5		
4	2019-08-22 13:30	0		
5	2019-08-22 15:00	0		
6	2019-08-22 16:25	7		
7	2019-08-22 16:53	5		
8	2019-08-22 18:35	2		
9	2019-08-22 22:47	2		
10	2019-08-23 01:17	1		

A.6 10 programs, different hours of different days

Table 11: Input data for 10 programs scheduled at different hours of different days

BATCH	duration	expected_finish
1	180	2019-08-22 17:01
2	360	2019-08-22 22:55
3	400	2019-08-23 09:35
4	90	2019-08-23 18:00
5	25	2019-08-23 19:25
6	228	2019-08-24 00:30
7	12	2019-08-24 05:40
8	472	2019-08-24 14:15
9	150	2019-08-24 16:15
10	39	2019-08-25 06:55

Table 12: Output data for 10 programs scheduled at different hours of different days

BATCH	start_datetime	delta_time	total_fee	iterations
1	2019-08-22 13:55	6	27000	18
2	2019-08-22 21:55	0		
3	2019-08-23 18:55	0		
4	2019-08-24 17:30	0		
5	2019-08-24 21:00	0		
6	2019-08-25 03:42	0		
7	2019-08-25 17:28	0		
8	2019-08-26 02:53	30		
9	2019-08-26 12:45	0		
10	2019-08-27 19:16	0		