INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Scheduling System for Remote Control of Instruments used for Atmospheric Observation

Martin Schumann

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Scheduling System for Remote Control of Instruments used for Atmospheric Observation

Martin Schumann

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Stephan Hachinger (Leibniz-Rechenzentrum) Johannes Munke (Leibniz-Rechenzentrum) Jan Schmidt
Abgabetermin:	21. April 2022

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 19. April 2022

.....
(Unterschrift des Kandidaten)

Abstract

The Alpine Environmental Data Analysis Center (AlpEnDAC) "focuses on monitoring, understanding and forecasting of environmental changes in the Alpine region" [Alp]. For this purpose, AlpEnDAC collects instrument data streams from different sources and provides processing capabilities. To make more efficient use of the instruments, users should be able to control them remotely and, in addition, AlpEnDAC should be able to trigger automatic measurements for predefined events. Due to a variety of stakeholders (academia, other researchers, public agencies, …) with competing interests, who intend to use some of these instruments, proper scheduling of available measurement cycles is a key requirement. The main reasons are the high operating costs of the instruments and the discrepancy between the limited number of hours the instrument can be run and the larger number of people who want to use the instrument. There are many requirements that make efficient scheduling difficult, including: (1) priority (e.g., the instrument's owner has higher priority than other users), (2) an instrument's measurement and startup times, (3) maintenance, and (4) dependencies between instruments. To solve the scheduling tasks with their dependencies, numerous approaches exist, although the scheduling problem itself is NP-hard when a perfect schedule must be found. Fortunately, scheduling algorithms exist which work efficiently for shorter schedules with few constraints, and which can find semi-optimal solutions by using heuristics and approximations. A variety of software exists which aims to solve these problems, e.g., software written by NASA [NAS]. Their software is used, among other uses, for scheduling spacecraft instruments, including its requirements for operation. This thesis will (1) compare and contrast relevant research on scheduling, (2) perform a comprehensive requirements analysis, and (3) provide a case study by implementing a suitable scheduling solution for AlpEnDAC. The final implementation will be accessible via a REST-API and makes use of Apache Kafka, a distributed document streaming platform, to interact with other components of AlpEnDAC (e.g., Computing on Demand). The software architecture implemented for this thesis consists of three components: (1) a REST-API "frontend" that takes job requests to be scheduled from authorized users, (2) the scheduling software, which takes the jobs from the frontend and tries to find the optimum schedule which respects the various requirements and dependencies, and (3) a REST-API "backend" that provides a scheduled list of pending jobs for the instruments to consume and execute.

Contents

1 Introduction

The AlpEnDAC is a facility for Research Data Management and on-demand simulation, which "focuses on monitoring, understanding and forecasting of environmental changes in the Alpine region" [Alp]. It is run by the DLR, the University of Augsburg, the LRZ, the Environment Research Station Schneefernerhaus (UFS) and the bifa Umweltinstitut in the scope of a project collaboration. It serves scientists of the VAO collaboration. One task of the centre is to provide programmable interfaces to various instruments of partnered organizations, so that researchers and other authorized users can control the instruments, get data from them, or send requests for research jobs (such as taking a picture at a specified time) to be executed. These "OOD" interfaces (in AlpEnDAC terminology) and the centre make it possible for organizations to have access to each other's instruments and data, and enable collaborative research.

One of the instruments that is part of AlpEnDAC and a candidate for OOD is the Fast Airglow Imager (FAIM) [DLR] camera in Oberpfaffenhoffen. This device takes images of the mesopause for use in various projects. An illustration of the FAIM camera is shown in Figure 1.1.

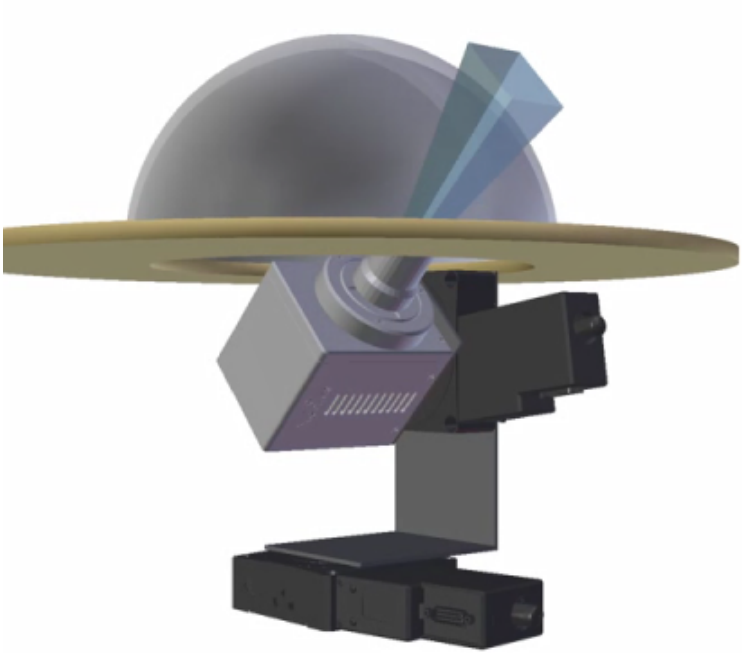

Figure 1.1: An illustration of the FAIM [Han21b].

The mesopause is a region in the atmosphere "where the temperature of the atmosphere reaches its lowest point" [MW]. It is approximately $80km$ above the ground [Par07], which can be seen in Figure 1.2.

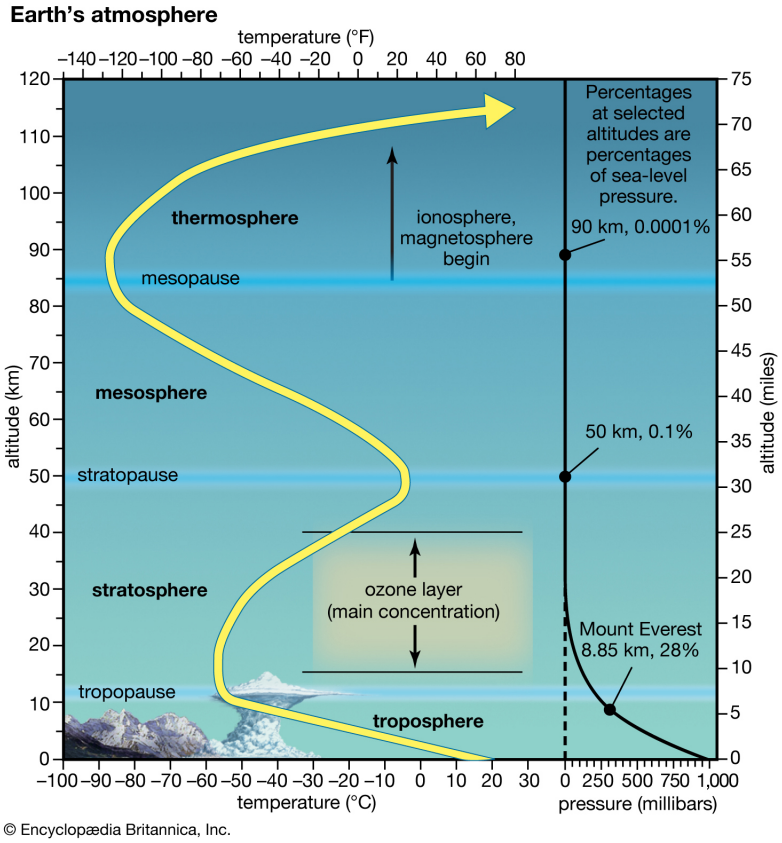

Figure 1.2: "Layers of Earth's atmosphere" [Enc]. The mesopause can be found at an altitude of 85km.

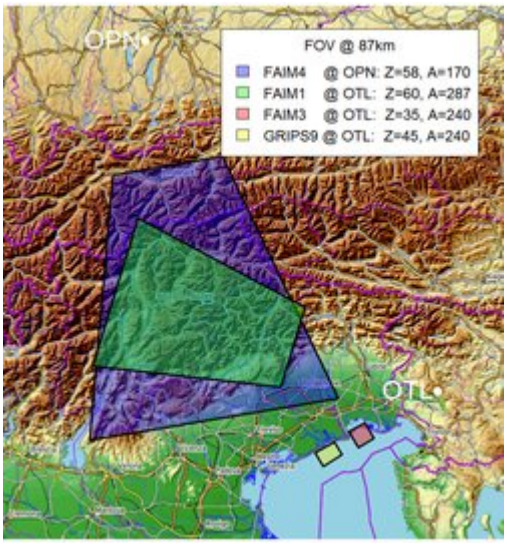

Figure 1.3: Location of various FAIM cameras in the Alps [DLR]. OPN stands for Oberpfaffenhoffen in Germany, OTL for Otlica in Slovenia.

When the instrument in Oberpfaffenhofen is used together with another FAIM instrument in Otlica, Slovenia, three-dimensional (stereoscopic) images can be taken. Similar to a 3D camera used for movies, the two images are offset in a way as to cover the same part of the sky from different vantage points, and by combining them, create a 3D image. 3D imaging makes analyzing changes in the mesopause easier [DLR]. The geographic location of the FAIM instruments can be seen in Figure 1.3.

The previous way of viewing, submitting and removing data and Computing-on-Demand jobs in AlpEnDAC was based on a complex PHP application, and the corresponding APIs made it hard to onboard new users and get them working with the endpoints quickly. OOD, i.e., the scheduling and instrument operation capabilities which are implemented in this work, have not been part of AlpEnDAC and its APIs until now.

The goal of this work is to provide a modern framework and API for submitting jobs and data to AlpEnDAC. This API can be viewed in OpenAPI [Ope] and can also schedule OOD events the way the instrument owners require. The implementation of this framework has the goal of making other future interfaces easier to create and program. Furthermore, a unified location and structure of the API using OpenAPI makes onboarding new users more streamlined. A scheduler is used here as a system which receives a list of possible jobs that should be executed and finds the optimal subset of jobs and their temporal ordering that can be actually be executed during a given time period. The reason a schedule is needed is that the camera can only take a picture of a certain portion of the sky at a time, and multiple users might want to point it to different locations at the same time. Therefore, a system that allows instruments to be used to their full potential is necessary to ensure some level of fairness and consistency for the jobs that are actually executed (an "optimal" schedule). What an optimal schedule actually is, depends on various factors and goals. A scheduler also allows for schedules to be more dynamic and efficient, e.g., an instrument might have service requirements, which can be included and changed in the schedule, and the rest of the schedule can adapt on the fly.

This work will provide an in-depth requirements analysis performed for a modern API with a configurable, reusable and generic scheduler. It will also describe how scheduling works, including the theoretical underpinnings. Finally, it will provide a case study, by describing the process of creating and implementing the system for FAIM in Oberpfaffenhoffen.

This process of creating the REST-API [Fie00], the framework and the scheduler, and putting it into production is done in multiple steps. In Chapter 2 explores the theoretical background regarding scheduling and OOD specific aspects. This includes researching the current state of scheduling, including important concepts and definitions related to scheduling and comparing different scheduling algorithms and strategies. Chapter 3 describes the requirements from the operators of FAIM (the DLR) and AlpEnDAC. These requirements include more general software related requirements, and specific requirements related to scheduling. Implementing the scheduling requirements in code for the scheduling library (in this case OR-Tools [PF]), so that the scheduler will create consistent and fair schedules for the instrument is described in Chapter 4. This chapter also discusses why OR-Tools was chosen compared to other libraries and describes its design. Chapter 5 describes the next steps, which are designing the software architecture that will provide the REST-API, and then programming, deploying and testing the API and integrating the scheduler. This chapter also includes the design of the new API to communicate with the instrument and allowing users to control adding and removing of jobs. It also includes a brief overview how the API is documented using OpenAPI and how the rest of the code is documented for

future users and developers. In Chapter 6, the specified requirements from Chapter 3 are compared to the implementation, which includes getting feedback from users and making changes if required. In Chapter 7, the work is concluded by a summary and an overview of future possible development.

2 Background

2.1 General Overview

There are multiple theoretical aspects that need to be considered in this work. One is of course scheduling itself, which is described in Section 2.2. Section 2.3 provides an overview of the API architecture. Another aspect, described in Section 2.4, is measuring the performance of a scheduler, both in terms of speed and "quality" of a schedule. Further theoretical aspects specific to OOD or FAIM, including some related to scheduling, are described in Section 2.5.

2.2 Scheduling

Scheduling is the process of creating a plan to achieve specified goals. These schedules can be done for a wide variety of objects, instruments, vehicles and even people. The goals can be varied, and include controlling a spacecraft [JMM+00] or allocating people on a factory floor. To achieve these goals, there is usually some metric used to define what a "good" or correct plan is, such as that the plan actually can be achieved by the scheduled entity, or e.g., that the amount of fuel used is minimized.

There are many tools and approaches that are able to solve scheduling problems [RPB13, JMM+00, BDP96, DCT19]. In this work, we concentrate on scheduling problems that are solved by using constraint programming, also known as CAIP. This type of scheduling is done by including various constraints that need to be observed when creating a plan to achieve the goal.

An example for a scheduling problem that includes people is the "job shop problem" as defined by [BDP96], which is used to test a wide variety of scheduling software [DCT19]. The problem consists in allocating people and tools to manufacture an unspecified item, and in creating a schedule that minimizes the time to completion [BDP96]. The time to completion is also known as a "makespan", i.e., "the time interval between the start of the first operation and the end of the last" [DCT19]. This example problem can include a variety of constraints and tuning parameters, such as number of workers, machines or time for each step.

However, constraints for scheduling can be very varied depending on the problem domain. Some very common constraints include that one action has to occur directly after another, actions cannot overlap, or that time between actions is needed, e.g., for cool down or movement of parts of an instrument. Constraints can also include dependencies, meaning that e.g., action A can only occur after action B has concluded and the amount of resources, such as fuel, are enough to carry out A. Another example of a constraint is that instruments should be used efficiently, e.g., a telescope should not turn in a lot of different directions one after the other, if it can be avoided.

The way CAIP solves scheduling problems is with constraint programming [Ros06], which revolve around algorithms to solve constraint satisfaction problems (CSP) or constraint

optimization problems (COP) [Apt03]. Constraint satisfaction problems are mathematical problems where the goal is to find one or more solutions to the problem (or prove that there are none) [Apt03][1]. In constraint optimization problems the goal is to find a solution and, if possible, finding the proven "optimal" solution [Apt03]. The difference to constraint satisfaction problems is that instead of just finding solutions to the problem under the given constraints, there is an "objective function" that defines what optimality means. Many problems are defined so that the goals are best met when a value is maximized or minimized, depending on the use-case. An "objective function" can then be calculated from various factors, and a constraint programming solver has the goal to minimize or maximize this function, while still fulfilling the constraints. An "optimal" solution is a solution that satisfies all constraints and the solver has proven it has found the best solution, as defined by the result of the "objective function".

For constraint optimization problems there is also a difference between hard and soft constraints [Apt03]. Hard constraints are constraints that must be met for a solution to even exist. Examples include constraints like the maximum possible of fuel which can be used, or the maximum possible time available. These constraints are often the most important, especially for use-cases such as rovers or production processes that have very specific time requirements.

Soft constraints meanwhile, do not need necessarily need to fully met for a solution to exist ("partially satisfied") [Apt03]. However, if a soft constraint is not met, a solution might be suboptimal [Zha02]. A lot of soft constraints can be modeled in constraint programming to have an associated "cost" for the constraint being violated, which feeds into the "objective function" [CCJK06]. This means that a solver will try to find better solutions to minimize these costs as much as possible [Zha02]. An example in job scheduling would be allowing overtime for workers to meet a production deadline. The soft constraint would be no overtime, which can be violated with the "cost" being modelled with the extra wages needed.

Soft constraints allow for solutions to be calculated more efficiently and quickly, because finding an "optimal" solution using only hard constraints might take a very long time to calculate (and can be NP-Hard in the worst case) [CCJK06, Zha02]. This is especially important for something such as scheduling, where a "good-enough" solution is better than a solution that has not finished calculating by the time it is needed. It is also highly beneficial for other, very dynamic optimization problems, or optimization problems that need to be run on less powerful hardware.

In general, constraint optimization problems are a good fit for scheduling problems, where often the goal is to minimize or maximize a value, for example the amount of time needed to run should be minimized or the amount of manufactured items in a production plant should be maximized.

Scheduling tools like OR-Tools [PF] or NASA's EUROPA [BBD+12] scheduling and planning tool use constraint programming to solve scheduling problems [RPB13, PF]. Both tools have built in types for a different kinds of constraints and scheduling primitives. These include intervals, which describe an event that has a start and end time, and can sometimes include additional attributes [RPB13, OR-b]. A goal can be set to maximize the cumulative duration of all events. The solver will then try to optimize for this goal while taking into account all other constraints, such as the horizon. The "horizon" is a concept which

[1] An example for a constraint satisfaction problem would be: find the values for a and b that satisfy $a+b < 5$.

some scheduling tools operate on [RPB13, PF]. It defines a range with predefined start and end integer values, and is often defined as the range from 0 (when tasks can start) to a maximum value, such as a fixed deadline for all tasks. Schedulers using this concept do not describe time in an "everyday format", but by a number (increasing with time, e.g., number of seconds) relative to a defined start time. The mapping between scheduler internal (relative) and "real-life" timestamps (such times in UTC) is done outside the actual solvers. For example, in this work, each night starts at a horizon value of 0 (dusk) and ends $n = \text{round}(t_{\text{dawn}}/\text{s} - t_{\text{dusk}}/\text{s})$ seconds later, defined as the end of the horizon. Each event's scheduler internal start and end timestamps are then calculated so that their "real-life" start and "real-life" end timestamps are relative to the dusk timestamp. An example of this mapping is: an event that starts at $20:00$, with dusk being at $19:30$ would have a relative start time in OR-Tools of $1,800$.[2]

OR-Tools also has "optional" intervals, which are intervals that do not need to be included. This means that if an interval cannot be scheduled given the constraints, it is excluded. A typical scheduling objective would be to maximize the cumulative duration of all *included* ("allowed") events. This is especially useful for creating dynamic schedules, where changes to the schedule mean that some events would need to be removed. Another advantage is that an "optimal" schedule can always be found, if considering only optional intervals, because if no events meet the constraints, then an empty result with a duration of 0 is still a valid solution. With normal intervals, all intervals need to be included[3], which means an "optimal" schedule, or even any solution might not be found.

The flexibility of constraint programming means that a multitude of constraints can be added and enforced to create a schedule, which makes automated scheduling for a large amount of use cases possible [BLPN01, Ros06].

2.3 APIs

To enable the scheduling software to be used by people and software alike, the solution was to create an HTTP REST-API, which is an API for communication between different systems using the HTTP protocol, in an architectural style defined by R. Fielding [Fie00]. HTTP REST-APIs are widely used [MPD10], which enables users, developers and integrators an easy way to work with the OOD system in many programming languages and with an easy learning curve.

This REST-API for OOD was further specified by following the JSON:API [JSO] standard, which defines a standard structure for JSON [Ecm15] documents which are sent and received over a REST-API. JSON was also chosen because it is a well known and widely used document syntax, which can also be used by many programming languages [BRSV17].

JSON:API allows for each document to have a similar structure, and it allows future documents for other OOD implementations to have a specified standard to adhere to. It also includes various required structural elements, e.g., for "data" documents it requires each JSON object to have an "id" and "type" [JSO]. This means that each object can be uniquely identified and has a known type, which makes working with large amounts of different data much easier. Each object having an id of type UUID4 [ITU] makes this uniqueness possible. This is because UUID4 (UUID version 4) values are random numbers with 2^{122} possible

[2] $20:00 - 19:30 = 30\,\text{min} = 30 \times 60\,\text{s} = 1,800\,\text{s}$

[3] At least in the way OR-Tools works.

values, which means the chances of a collision are very low [ITU]. JSON Schema [WAHD20] is also used to validate that the JSON documents are following the JSON:API specification, and it can also validate properties of the data themselves, e.g., that a JSON value is a date-time with the correct format.

2.4 Performance Metrics for Schedulers

There are many performance metrics for schedulers [SFS+14, DCT19]. One is of course speed of execution, i.e., how long it takes the scheduler to create a schedule. Another key metric is how well the scheduler then fulfill its stated goal, e.g., filling the time horizon (leaving the smallest amount of gaps) as much as possible, maximizing the number of events scheduled or the number of users who get their events scheduled [DCT19]. Other important performance metrics include if the schedule can update the schedule dynamically, how much time the update takes as well as the stability of the schedule.

Stability for a scheduler describes how much changing one event impacts the rest of the schedule, e.g., replacing one event with a slightly longer event might cause 30 events to be scheduled differently [FF10]. This is mostly not a problem if events are fixed, i.e., they have a well-defined start and end time and have no dependency constraints [FF10]. However, if the events are more dynamic, either with dependency constraints or events which can run at any time during a time horizon, small changes can have a big impact. Events scheduled anytime in a horizon means that the events have a specified duration, but not a specified start and end time. The start time is calculated by the scheduler to fit in the schedule where appropriate, and end time is just start time plus duration. Dependency constraints describe sequences of events, e.g., event A must occur before event B, otherwise the instrument can't function. Then, dynamic scheduling can cause the whole schedule to be reshuffled to try and fulfill the scheduling goals when events are inserted or removed. Therefore, measuring how well a scheduler can keep the schedule stable is often very important [FF10].

2.5 Domain Specific Concepts

This section describes OOD and FAIM specific theoretical concepts, such as how to handle timezones, changes to the time because of DST, and the concepts of dusk and dawn as used by the DLR.

One often repeating aspect of scheduling is the correct treatment of timezones and DST, where "missing" or "duplicated" hours can create problems. Governmental decisions to change timezones or the DST switching point (or to eliminate DST entirely) also add to complications in the long term. For FAIM, these issues are sidestepped because FAIM handles times strictly in UTC. Thus, our backend works only with UTC for the FAIM implementation and will only accept UTC dates in the JSON API. These restrictions eliminate issues related to timezones and DST very easily.

The FAIM camera only takes images at night, which would colloquially be described as the time between sunset and sunrise. However, the DLR uses the concepts of dusk and dawn, which occur at slightly different times than sunset and sunrise. All four of these words have official "civil" descriptions, i.e., at what angles of the sun these time periods start and end. The time period used by the DLR, which starts with dusk and ends with dawn, is also known as "civil twilight" [Ast, Glo].

Other theoretical concepts are possible angles for the FAIM camera, including the concepts of the azimuth and the zenith angle, which also need to be considered. The camera can be rotated in two directional axes in order to take pictures. A graphical explanation of these 2 axes and their names can be seen in Figure 2.1. In our case, the "observer" in the figure is the FAIM camera, and the "star" is the location where the camera is pointed.

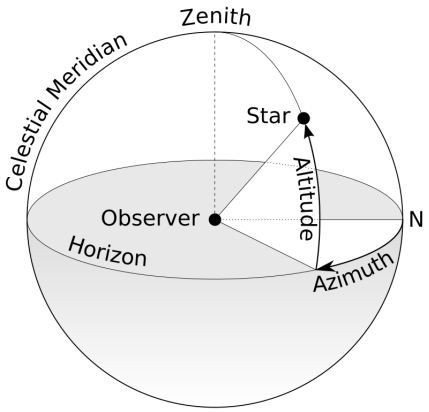

Figure 2.1: Graphical representation of the azimuth and the zenith angle [Wik21]. The zenith angle is obtained by subtracting the altitude from 90°.

The first axis the camera can move is up and down, which is specified in FAIM by the zenith angle (0° to 70°). This can be seen in Figure 2.1, where which angle the camera is pointed ("star") is calculated as the difference from the zenith point. This means an angle of 0° is the equivalent to the camera looking up as much as possible, whereas 70° means the camera angle is much closer to the horizon. 90° would be equivalent to the camera being flat with the horizon, which is however not possible because of mechanical limitations. The second axis the camera can move is left and right, which is specified by the azimuth (0° to 360°). Figure 2.1 illustrates this well: the angle is measured between the geographical/astronomical north direction (point "N") and the orientation of the "observer" on the horizontal plane.

Another theoretical concept is the difference between "scan" and "static" events. When the FAIM camera takes pictures, there are 2 different modes in which it can take pictures. In "scan" mode, the camera follows a predefined movement pattern to take a composite picture of the full sky (Example: Figure 2.2). This takes 123 seconds, and creates a total of 35 individual pictures, which can then be merged into one. One use for this is taking pictures during the night and then creating a timelapse sequence to track and analyse changes of the sky. This is also the reason that "scan" events are used to fill empty spaces in the schedule, because if there is not a certain portion of the sky that needs to be imaged, then getting a full image can provide more value. The "cost" of a "scan", however, is that each portion of the sky only gets imaged once every 123 seconds.

The other mode is the "static" mode. As the name implies, the FAIM camera is pointed to a certain static position. This position is defined by the azimuth and zenith angle values given by the user. The camera then takes a specified number of pictures, also defined by the user. Each image takes 0.5 seconds, and multiple images can be taken one after another. This mode is used for taking pictures of a specific portion of the sky, but can also be used to track something in the sky, such as a satellite, by defining when the camera should point where. The faster imaging of "static" events "enables the study of transient features in the

Figure 2.2: An full sky image taken from the FAIM camera [Han21b].

airglow" [HSWB16], i.e., observing events that can occur or change very quickly, which is otherwise not possible without a higher "temporal resolution" [HSWB16].

In this work, there is an overlap between the conceptual "scan" and "static" events, as described above, and `scan` and `static` events, which represent the actual implementation, including their JSON Schema representation. Although these are the same in their functionality and their theoretical underpinnings, they have separate styles to distinguish them, as a draft implementation of the system used the same concept of "scan" and "static" events, but had a different JSON Schema representation for both, known as `simple` events (see Subsection 6.1.2). In this work, `scan` and `static` will represent scan and static events as both the concept and their current implementation, whereas "scan" and "static" will only represent the concept.

3 Requirements Analysis

In this chapter, the requirements analysis for the system built in this work is presented. The analysis distinguishes Functional requirements and Non-functional requirements relating to different thematic aspects. In particular, we distinguish requirements for different components, including the REST-API and backend, the instruments and the scheduler. The first section deals with the stakeholders, which are the people and organizations involved in the project, and whose requirements are included in this analysis.

3.1 Stakeholders

There are multiple stakeholders for this project, including the LRZ, the DLR and other organizations which are part of the AlpEnDAC project. The LRZ and DLR are the primary stakeholders, as they are directly involved in the project, and its primary users and developers. The other organizations are the secondary stakeholders, as they might be users and could use the code for future development and their systems. The stakeholders can be divided into different groups: the users, the developers, and the operators of the instruments. The operators and developers are responsible for working and developing with the system, and are primarily responsible for its design. The users can be divided into different smaller groups, mostly dealing with their access privilege and priority in the schedule (with range 1 to 5, 5 being the highest). Although the operators are separate stakeholder group, they have some overlap with users, because they also consume output from the REST-API and use the instruments. Because they own the instruments, they therefore have the highest access privilege, meaning they have access to raw data, are able to operate the instrument as they see fit, and have the highest priority (5) in schedules. The second group of users is those who use the instrument and are part for the organization that owns the instrument. They have a lower priority, e.g., 3 or 4. The third group are users of the instrument from other institutions that are part of AlpEnDAC or are otherwise partnered with the organization of the owners. They have a priority of 2 or 3. The fourth group would be possible users of the instrument that have no direct affiliation with the operator's organization, such as scientists from other organizations, which would request access to work with the instruments. They have priority 1 or 2.

3.2 Requirements with Respect to Scheduling

This section focuses on goals and requirements specific for the OOD use case that was implemented in this work. The goals define which metric the scheduler is trying to optimize to create the "optimal" schedule for the OOD instruments. The requirements were created and refined in part with P. Hannawald at the DLR, including the FAIM specific requirements and checks (Appendix Section 1).

3.2.1 Functional Requirements Scheduling

We start out with the functional requirements related to scheduling:

1. The scheduler shall fill out the night with events as much as possible, meaning the time when the instrument is working is maximized.
2. Gaps in the schedule shall be filled with "scan" events, as defined in Section 2.5.
3. Events for a specified date A shall only be scheduled between day A (dusk) and on day $A + 1$ (dawn) UTC, in the location of the instrument[1].
4. Dates and times for all events shall be in UTC. Because UTC is never adjusted for DST, code dealing with possible bugs relating to clock adjustment shall not be included.
5. Stretch goal: Scheduling of the events should also take into account the priority of the requesting user. The higher the priority, the higher the chance an event will be scheduled, even if the goal specified in Item 1 is not fulfilled as optimally. Optimally here means the idle time is the minimum that can be achieved.
6. Scheduler should be easy to use and extend by future programmers.
7. Scheduler shall be easy to embed into or work with Python [VRD09], because existing code in AlpEnDAC is written in Python. This is to simplify development and working with the rest of the system. Using one programming language also makes maintenance easier, because language specific knowledge about syntax, tooling, integration and deployment is only need for one language. This allows maintainers to get started working with and extending the code much faster with less lead time.

3.2.2 Non-Functional Requirements Scheduling

Turning to the non-functional requirements related to scheduling, we have identified the following aspects:

1. Creating new schedules shall take at most 2 seconds. Creating new schedules entails both scheduling a new event, and if the old schedule changes, to create new filler events.
2. Sending malformed data, and data that cannot be scheduled for other reasons (such as not being inside the horizon or having overlap) shall always be rejected by the scheduler.
3. Schedules must always be executable by the instrument, meaning that overlapping events and events outside the horizon shall never be scheduled or sent to the instrument.
4. The scheduler shall always be able to create a schedule which fulfills all the requirements. This shall be done without crashing, running indefinitely or deadlocking, or not being able to create any kind of schedule.
5. The scheduler shall try to create an optimal schedule. An optimal schedule is defined here as one, where the cumulative duration of the scheduled events is the maximum possible sum, while respecting all requirements.

[1] The location of the FAIM camera in Oberpfaffenhofen, Germany has the coordinates latitude=48.087°, longitude=11.280°.

3.3 User Oriented Requirements with Respect to Interfaces/APIs

The interfaces and APIs shall fulfill certain usability requirements from a user perspective, like having a clear structure, being easy to use and understand (helped by OpenAPI) and being accessible.

3.3.1 Functional Requirements Interfaces/APIs

Here, we focus on the following functional requirements from a user oriented point of view:

1. The APIs shall be easily discoverable, meaning that new consumers can easily see how to use the API. The system shall use OpenAPI to show all the URL endpoints on `api.alpendac.eu`. OpenAPI must be used, because existing endpoints and infrastructure already exist in this format at the institutes participating in AlpEnDAC.
2. The APIs shall be accessible from `api.test.alpendac.eu`, and possibly `api.alpendac.eu`.
3. Changes to the APIs after deployment shall be minimal (such as fixing bugs) or on a different URL. An example would be `/v1/` for a scheduler which does not take into account priority of the users and `/v2/` for a scheduler which does, as described in Subsection 3.2.1.
4. The APIs shall be secured against unauthorized users. Here an existing authentication system using Kong already exists and shall be used.

3.3.2 Non-Functional Requirements Interfaces/APIs

Four non-functional requirements related to this portion of the system are included as well:

1. API shall return exact error messages, especially when sending a new event. This means when the data are validated with JSON Schema and the other checks (see Section 3.5), it shall return exactly what the problem with the data is.
2. Sending incorrect data shall not crash the API or cause problems without the user being aware.
3. Data returned from endpoints shall always be correct, both in syntax and logic.
4. Structure of JSON data should be easily understandable, and it should be easy to create the correct data for the endpoints. This means creating documentation and examples and validating data sent to the API.

3.4 Sustainability and Reusability Requirements on System

To make the system sustainable and reusable, it shall be able to work with minimal maintenance and shall be easily adaptable to multiple organizations to schedule their instruments. Because reusability of the system relies heavily on the quality and reusability of the code and quality of the documentation, Subsection 3.4.1 describes how this shall be enforced, including any software to enforce these requirements. Subsection 3.4.2 describes how the reusability and sustainability goals can be judged.

3.4.1 Functional Requirements Reusability

Functional requirements here mostly relate to "developer" focused concepts, e.g., code style, sourced code management and software packaging:

1. Packages, images and software shall be up-to-date.
2. All functions, classes and modules shall have comments, which are formatted correctly, enforced by pydocstyle [Pyd].
3. All functions, classes and modules shall be formatted according to pep8 [VRCW] and checked for simple errors, enforced by flake8 [Pytb].
4. Tests shall be exhaustive, and test a large amount of the functionality. They shall be run by pytest [Pyta].
5. Running the tests and linters shall be enforced and run in Gitlab CI [Gitc] and work without errors before the code is merged and deployed.
6. Generic code that can be used for multiple instruments and instrument specific code shall interoperate easily, by limiting the amount of coupling between the different parts.
7. Changing constraints in the scheduler shall be easy, while allowing substantial code reuse.
8. All code and related documents shall be stored in Git on the LRZ Gitlab [Gitb] server.

3.4.2 Non-Functional Requirements Reusability

Non-functional requirements for reusability focus mostly on documentation, as other reusability aspects are more difficult to quantify:

1. Documentation shall be exhaustive and shall include all aspects that enable a new developer to work with the code without needing to do a deep dive.
2. Documentation shall include how to extend the system.
3. Documentation shall include information on how to deploy the system, including new versions.
4. Code shall be clean. This includes well organized and structured files and folders. Also, functions, classes, etc. shall be organized logically and their names should be well-chosen and meaningful.

3.5 System/API Requirements

While Section 3.3 refers to the requirements of the API from an end-user perspective, this section focuses on the requirements to API and system from a designers' perspective, including ease of implementation and changes, speed of the code and maintainability. It also includes FAIM specific requirements for data validation and modeling.

3.5.1 Functional Requirements System/API

These functional requirements relate mostly to best practices in modern software design and management:

1. Code shall use Gitlab CI for building, tests and linting.
2. Code shall be deployed on the AlpEnDAC Kubernetes cluster.
3. Code shall run by default in Docker images.
4. System shall be secure with access controls and as "closed" as possible (i.e., the smallest amount of open ports, IP addresses, etc.)
5. Code shall be stable, there shall be no code crashes. If unforeseen crashes occur, they should cause no lasting effects. Easy recovery mechanisms shall exist, such as a redeployment or restart of the images in Kubernetes.
6. "Safety" of the data shall be ensured, i.e., no data gets lost during crashes etc. The data storage locations, such as a database, shall be on a NAS or similar, where scheduled and automated backups are run and verified.
7. Everything not stored in the database, git or in secret stores[2] shall be "ephemeral" [Docc], meaning that it could be deleted and recreated without any loss of data or functionality. For Docker images und kubernetes deployments this means that stopping, deleting and recreating has no lasting effect on other data, and recreating means no manual steps after creation [Docc].
8. Both sent and received data shall be validated with JSON Schema and other FAIM-specific checks (See Appendix Section 1).
9. FAIM specific requirements, described in Appendix Section 1, shall be implemented.

3.5.2 Non-Functional Requirements System/API

The non-functional requirements in this section overlap a bit with those in Subsection 3.4.2, but they have a broader scope instead of being focused on sustainability/reusability:

1. Tests for the code shall exist, and all tests and linters shall be run and pass before code is merged.
2. Changes shall be fast to implement and deploy.
3. Complete documentation, meaning that each function, method, class, package, etc. shall have a comment. The documentation shall also include how to run and deploy the code.
4. Good code quality. This includes linters (such as flake8 for Python) which check code quality, tests using pytest and manual reviews before code is merged in Gitlab.

[2] Passwords, certificates etc…are securely stored by the LRZ.

5. Code shall be reasonably performant, efficient and shall perform operations in near-real time.

6. Deployment shall be easily reproducible and upgradeable (Kubernetes and Docker). This means the number of manual steps the programmer needs to deploy a new version of the code shall be as low as possible. Fewer manual steps while upgrading and recreating the deployment means less work, but also means less possibility for errors.

7. Setting up a development environment shall be easy. This means installing the required tools and getting the code to run for development shall take at most 2 hours, not including time to download software.

8. Easy debuggability, with good logs. This means that the important functions shall created detailed logs. These logs shall include a timestamp and detailed information about the current state and the operation is being performed.

9. Implementation of change requests from stakeholder shall occur in a timely manner, depending on the nature of the changes requested.

10. Sending incorrect data shall not crash the application or cause silent problems.

3.6 Prioritization of Requirements

This section discusses how the various requirements should be prioritized with respect to the wishes of the stakeholders and other constraints, such as time. There are some requirements that can be relaxed or have lower priority, such as:

1. Speed of the code: as long as the speed is sufficient for usability, meaning the user does not have to wait more than 3 seconds (worst case), optimization of the code takes lower priority, including items such as close-to-real time (Subsection 3.5.2).

2. Performance of the code: Since the number of users currently is low, the kubernetes deployment does not need to include multiple replicas of the images or large amounts of resources.

3. As described in Subsection 3.2.1, adding a priority system to scheduling is a stretch goal, and so has a lower priority.

4. Also lower priority is “strictly clean” code, as described in Subsection 3.4.2. The code shall include good documentation and be understandable, but code organization and naming has a lower priority if the functionality is clear.

5. Another lower priority requirement concerns extra (“nice to have”) features, which might enable easier or faster use of certain aspects, but are not part of core functionality.

The requirements that are the most important are the stability and correctness of the code. This means that the deployed code does not crash from user input and shall not crash from bugs. If some bugs does take down the system, Kubernetes restarts the containers, but bugs should not cause constant crashing, especially if the crash is right after startup. Correctness

of the code is a central aspect. Especially requirements in scheduling (Section 3.2) are crucial, because the scheduler always has to function correctly. Also, user input and data sent to FAIM and other instruments is always checked for completeness, syntactical validity of the JSON, and logic requirements described in Appendix Section 1. If these aspects are not prioritized, then the usefulness for stakeholders and users is low, because they cannot use the code effectively. Also, bugs and incorrect functionality mean frustration for users and less incentive to use the new system. The goal of the requirements are higher productivity for the users, lower costs, flexibility, easy of use, number of features, and faster development of new features.

4 Scheduling Software

4.1 Selection of a Scheduling System Matching our Requirements

There are a few different types of scheduling software, such as OR-Tools, NASA's EUROPA or IBM's "ILOG CPLEX Optimization Studio", among others, each with different trade-offs [BBD+12, DCT19, SFS+, SFS+14, RPB13]. This section discusses how the various requirements can be met by different software and which was chosen for this implementation.

4.1.1 Two Scheduling Systems Evaluated in Depth

For the implementation of our system, two scheduling software libraries were evaluated in depth: OR-Tools and EUROPA, which were chosen because they are both opensource and have seen a lot of real world application [BBD+12, NAS, DCT19, RPB13]. Table 4.1 compares these libraries by their key characteristics. The information is sourced from the OR-Tools website [PF] and two papers describing EUROPA's implementation [BBD+12, RPB13].

Table 4.1: Comparison of Scheduling Software

#	OR-Tools	EUROPA
Type	General Optimization software, can be used for scheduling	Planning software, can be used for planning and scheduling
Solver	Multiple open source and commercial solvers can be used; includes a built-in SAT Solver	Custom solver algorithms and engines for rules and constraints
Creator	Made by Google	Made by NASA
License	Opensource (Apache License 2.0)	Opensource (NASA Open Source Agreement)
Language Bindings[a]	C++, Python, C#, and Java	Java and C++
Constraint language[b]	Python	NDDL

[a] Language bindings describe which programming languages can natively interact with the library.
[b] Constraint language describes the language in which the constraints are written, which might be different from the binding language.

4.1.2 Requirements for the Scheduler

Because the different kinds of scheduling software have different capabilities, we have determined multiple requirements, as described in Section 3.2 and Appendix Section 1 (specific to FAIM), to ensure that the scheduler works well with our use case. There are also certain requirements that any scheduler under consideration needs to fulfill, which are not directly relevant to the performance of the actual scheduling.

One important general requirement is that the scheduler should be easy to embed/work with in Python. Because OR-Tools has a Python version as a package, this means calling OR-Tools is a simple as calling any other library in Python. However, EUROPA is written in C++ and has Java bindings, which means communication with the Python code directly is more difficult, without writing bindings or having some other interoperability method.

Another general requirement is, that the scheduler should be fast to create and update schedules. Both OR-Tools and EUROPA are written in C++, which allows for a lot of low level optimizations and their execution speed is quite optimized as measured by multiple benchmarks [SFS$^+$, BBD$^+$12, DCT19, RPB13].

The third general requirement is that the scheduler should be easy to use and extend. For OR-Tools, there exists good documentation and community forums where one can ask questions. Also, the number of people using OR-Tools means that there is a larger user-base of people and solutions to problems. For EUROPA, the tool includes many more features, because it does "planning, scheduling and constraint reasoning" [BBD$^+$12]. Planning is used to describe how and in what order instruments, vehicles and other "actors" should do actions, such as moving somewhere or manipulating objects. This planning is then subject to constraints like the amount of fuel or size of a location. Constraints can be used for both planning and scheduling. While this is very useful for NASA's applications, such as controlling a spacecraft or a rover [BBD$^+$12, Chapter 6], there is a number of features that are not needed in AlpEnDAC and make working with EUROPA more daunting. EUROPA also uses a custom language called NDDL to describe the models used for scheduling. Although this language is an extension to a widely used planning language known as "PDDL" [GHK$^+$98], it differs in multiple aspects, and so some concepts between the languages are different [BBD$^+$12, Section 2.1]. These language differences make the ease-of-use for future implementers more difficult, as there is smaller amount of available documentation, examples and experienced programmers for NDDL. Another difference that makes EUROPA more difficult to pick up and use, is that the language also only has a plugin for Eclipse, whereas the code for scheduling with OR-Tools is written directly in Python, meaning a lot more tooling and IDEs are available.

4.1.2.1 Decision for using OR-Tools

Although both OR-Tools and EUROPA can fulfill the requirements mentioned above, the easy embed-ability of OR-Tools and the fact that it has better usability for our use case, means that OR-Tools was selected as the scheduler for our implementation.

4.2 Performance Metrics

There are many performance metrics for schedulers, as described in Section 2.4. In terms of execution speed, both OR-Tools and EUROPA are both quite performant [RPB13, SFS$^+$].

OR-Tools has even won gold medals in the MiniZinc Challenge [SFS+], which benchmarks the quality and speed of solving various constraint programming problems [SFS+14, DCT19].

However, due to the number of events per night being quite small and static, execution speed is of lower priority. Also, since adding/removing events is currently done in a linear queue and non-scheduled events get deleted, this means any impact a new event has is quite minimal, because it might cause some removals and possibly the recreation of the filler events. A queue also means that the hardware running the scheduler does not suffer performance problems when a lot of users try to modify events at the same time (the so called "thundering herd"). Another performance benefit is because the frontend is decoupled from the backend, the frontend can multiplex future backends for other instruments, which means their schedulers can work concurrently. Although future backends could include more dynamic scheduling (Section 2.4) which would require a moderate amount more computing power, the efficiency of both schedulers and the powerful AlpEnDAC Kubernetes servers do not make this a concern.

The other performance aspect is the "quality" of a schedule the scheduler can generate. Quality encompasses both correctness, i.e., how well the constraints are upheld, and how well the scheduler can reach the specified objective(s), e.g., minimizing the execution time for a job. Performance in terms of quality can be easy to measure in some cases, because if an "optimal" solution exists, i.e., a solution which is provably the best solution for the specified objective(s), then as long as all constraints are specified and enforced correctly, this solution has the highest "quality".

For OR-Tools and EUROPA the correctness aspect is not a problem, as both being constraint programming based schedulers means that if a "feasible" scheduling solution is found, it is guaranteed to satisfy all specified constraints [RPB13, OR-a].

However, measuring how well the scheduler can solve constraint optimization problems to meet the specified objective(s) is much harder. For constraint optimization problems multiple solutions can be found, some of which are not optimal (known as "feasible") [SAT, RPB13]. Proving which solution is the optimal solution can be harder and more computationally expensive [Ros06]. Furthermore, the solver might also not find a solution or, if the problem requires a lot of resources to solve, the computation might be cancelled before completing, which means it is not known if a solution exists or not [SAT, Ros06].

In our use-case, the provably (calculated by OR-Tools) optimal solution, i.e., the solution which satisfies the goal of maximum cumulative duration, always exists. A feasible solution, i.e., where at least one solution which satisfies all constraints, always exists because all events being optional means an empty schedule satisfies all constraints. The proof for the optimality is that, as long as events conforming to the constraints are available, the maximization goal forces the scheduler to add these events so that the cumulative duration is maximized. OR-Tools is also able to prove that a solution is optimal relatively quickly and easily, because the complexity of the model is low enough that the solver can evaluate all possible solutions in a reasonable time.

4.3 Implementation of the Scheduler

This section describes how OR-Tools was used in the AlpEnDAC scheduler, including technical explanations, design decisions and trade-offs.

One advantage of OR-Tools is that it has a built-in `Interval` object, which can encode an

event with start, end and duration times. In this scheduler implementation, when events do not fit in the schedule or create a suboptimal scheduler, they are dropped by the scheduler and are not included in the schedule. OR-Tools has a built-in data type that makes this behavior easy to implement. Known as an `OptionalInterval` object, it is the same as an `Interval` object, but also includes an extra `is_present` Boolean variable. OR-Tools SAT solver can then include/exclude these `Interval`s automatically by changing the variable, depending on if they fulfill the goal(s) while conforming to the specified constraints [Opt]. When the solver has finished calculating, all `OptionalInterval` objects where the boolean variable is `True` are returned as the schedule.

There are multiple requirements and goals for the scheduler. Because of time and complexity constraints, two versions of the scheduler were created in this work. The first version of the scheduler (known as "v1"), which is currently deployed, does not include priority as part of its constraints. This is because adding priority in the scheduler code and the system adds complexity to implementation and usage, and priority-based scheduling is less important when the number of users is very small. The second version (known as "v2"), which implements the stretch goal, as described in Item 5 (Subsection 3.2.1), does include priority calculations when creating the schedule. This version exists and is fully functional, but has not been deployed because of time constraints. It is part of the scheduler repository in Gitlab and has tests that confirm the full functionality.

For all versions, there shall be no overlap of the events. Events cannot overlap, because the FAIM camera can only take one picture at a time, so parallel operation is not possible. If an event that could be added does overlap with another, then a decision made on which event should be kept and which to discard.

For "v1", the goal for the scheduler is maximum cumulative "user-event" duration, i.e., the sum of the lengths of all user events that are included should be maximized. There are also similar possible implementations e.g., to minimize the spaces between events or to maximize the number of events. This might mean that the total duration of user requested events might be lower depending on the distribution of the events, but the number of events that are scheduled might be higher. Although there are tradeoffs between each implementation, maximum duration was chosen because it allows for the most amount of time that the instrument is taking "real" pictures (pictures requested by a user). As described in Item 1 (Subsection 3.2.1), gaps in the schedule are filled with `scan` events so that the camera spends most of its time taking pictures. However, while these `scan` events are useful, time spent taking pictures for users have priority. These filler `scan` events are also *not* included in the cumulative duration to be maximized by the scheduler.

For "v2", maximum "user-event" duration is the secondary goal. The primary goal is that events with the highest priority are included first, and lower priority events are only included if they do not inhibit higher priority events. In this version, each user of FAIM is assigned a priority level of 1-5, with 5 being the highest. This allows, e.g., users from the DLR to have priority in taking pictures over users from other research facilities. The downside of this approach is that if events overlap, the higher priority event always gets selected, even if this means that the duration suffers. This means that there might be less time spent taking pictures for users overall, but for higher priority users they are more likely to get the images they have requested.

There are multiple steps the scheduler runs through to create a new schedule, and then a high level explanation of the steps is as follows. This code is inspired by the job shop example code [OR-b] from OR-Tools, which was heavily modified for our implementation.

The first step is to create the OR-Tools variables, that are needed for the solver, from the Event Types (`scan` and `static`) and add them to the model. The second step is to call `model.AddNoOverlap`, which adds each event to the model and automatically guarantees no overlap in the resulting schedule. The third step is to add a constraint that the start and end of each event is inside the horizon[1], meaning that events will only be accepted if they are between dusk and dawn for the same day as the event. The next step is adding the goal(s) for scheduler version "v1" or "v2" (as described above). This is done by adding more constraints and a "goal function", such as `model.Maximize(sum (duration_of_included_tasks))`, which tries to maximize the cumulative sum of all included tasks. The next step is to instantiate a solver, e.g., OR-Tools' "CP-SAT Solver", which will solve the model. The solver will return a status type, which describes if the model could be solved, and if the solution is optimal. For our implementation, the solver will always return `OPTIMAL`, if the model is syntactically valid (i.e., `MODEL_INVALID` is not returned). This is because all events are set as optional, so an optimal schedule could also be an empty schedule (see Section 2.2). The last step is to get the new schedule from the solver, and convert it back to the internal data types used for the JSON, and to update the database.

[1] See definition in Section 2.5.

4.4 An Example Schedule

Figure 4.1 shows a series of 50 randomly generated `static` events. These `static` events were generated to occur between the 14.6.2021 at 11pm and the 15.6.2021 at 3am, each having a random duration of between 1 and 500 seconds. Random `scan` events were not generated, as the same principles apply as to `static` events for illustration purposes.

For each figure, the x axis shows the date and time in 30 minute increments. The y axis contains the first 8 characters[2] of the event's UUID4. The length of each event is represented by a bar, which represents its duration.

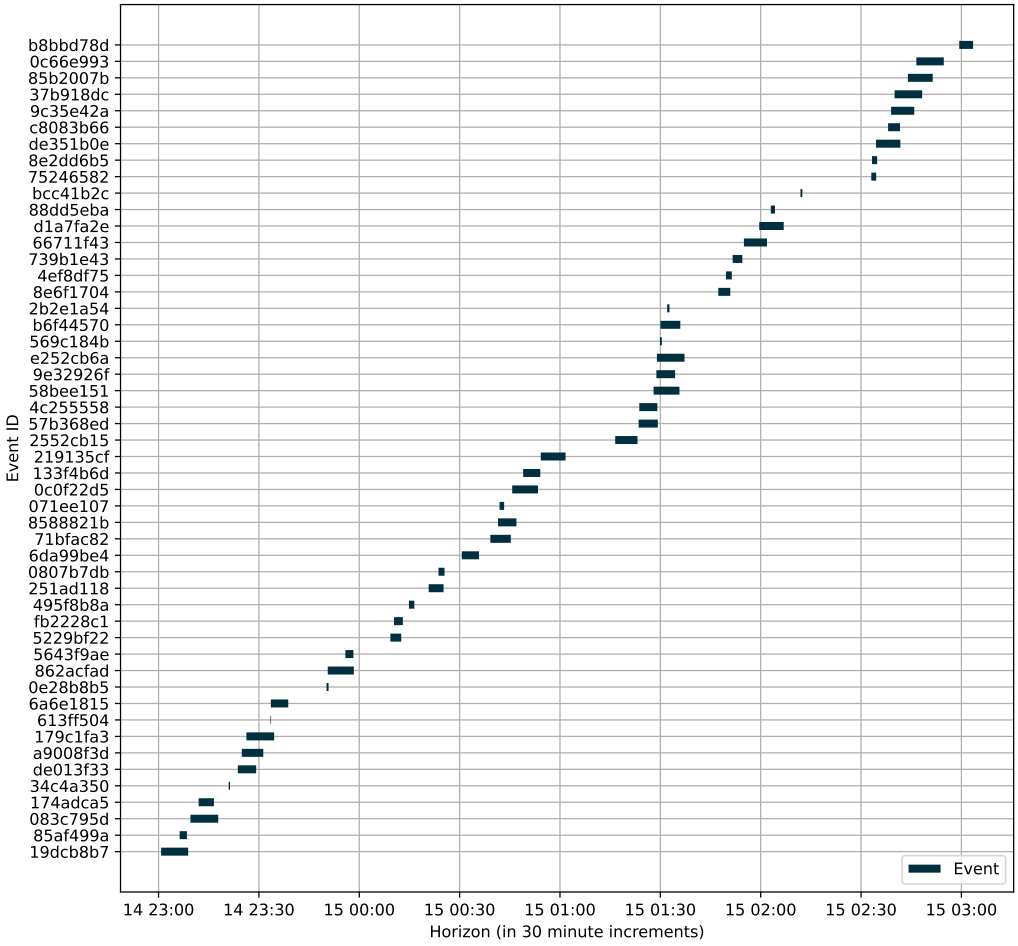

Figure 4.1: 50 randomly generated `static` events, sorted by their starting time.

Figure 4.2 shows the 24 `static` events that remain after the 50 events from Figure 4.1 have been scheduled. The cumulative duration of this subset is $7,323$ seconds, i.e., 2 hours, 2 minutes and 3 seconds out of a total of 4 possible hours.

Filling the schedule with `scan` events increases the cumulative duration of all events to $13,227$ seconds, i.e., 3 hours, 40 minutes and 27 seconds, which means the camera spends

[2] 8 was chosen, because it provides enough difference so there is little chance of overlap, but it is not as long as to make the image labels too long.

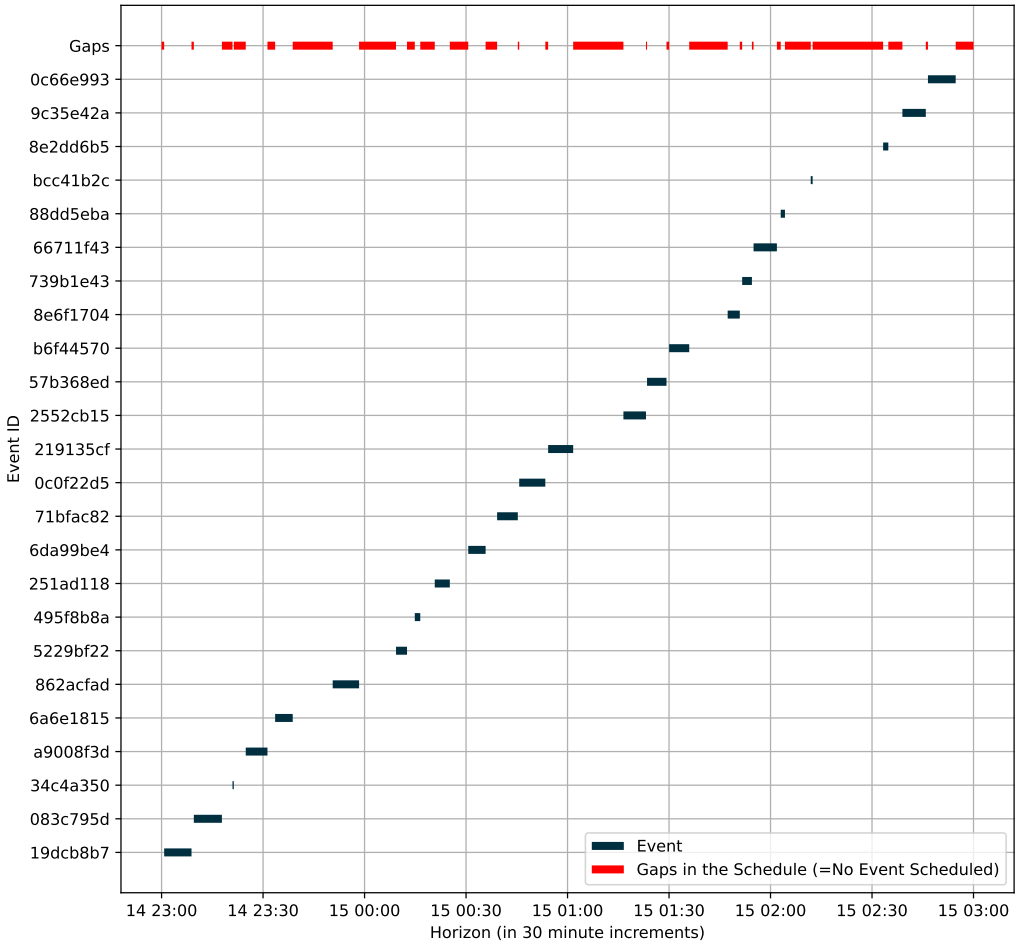

Figure 4.2: The subset of events from Figure 4.1 after being scheduled. The red lines with the label "Gaps" represent all instances in the schedule where no event exists, i.e., where the instrument is idle.

8.14% (19 minutes and 33 seconds) idle in this example. This is a reduction of idle time of 5,904 seconds, which means the instrument is idle for 1 hour, 28 minutes and 34 less than before. The reason this value is not smaller is that each `scan` event is 123 seconds, so any gaps in the schedule smaller than this can't be filled. The events were generated at random, which means the possible schedules which can be calculated and how well any gaps can be filled is defined by the "luck of the draw".

Figure 4.3 illustrates how much of the schedule is filled with `scan` events, which can be identified by the label "Filler Scan" on the x axis. All `scan` events were merged into one line in the figure to make visualization easier, as a total of 48 events would make the image larger and less straightforward. The figure also visualizes how much filling possible gaps in the schedule dramatically decreases the amount of idle time when compared to the "Gaps" in Figure 4.1.

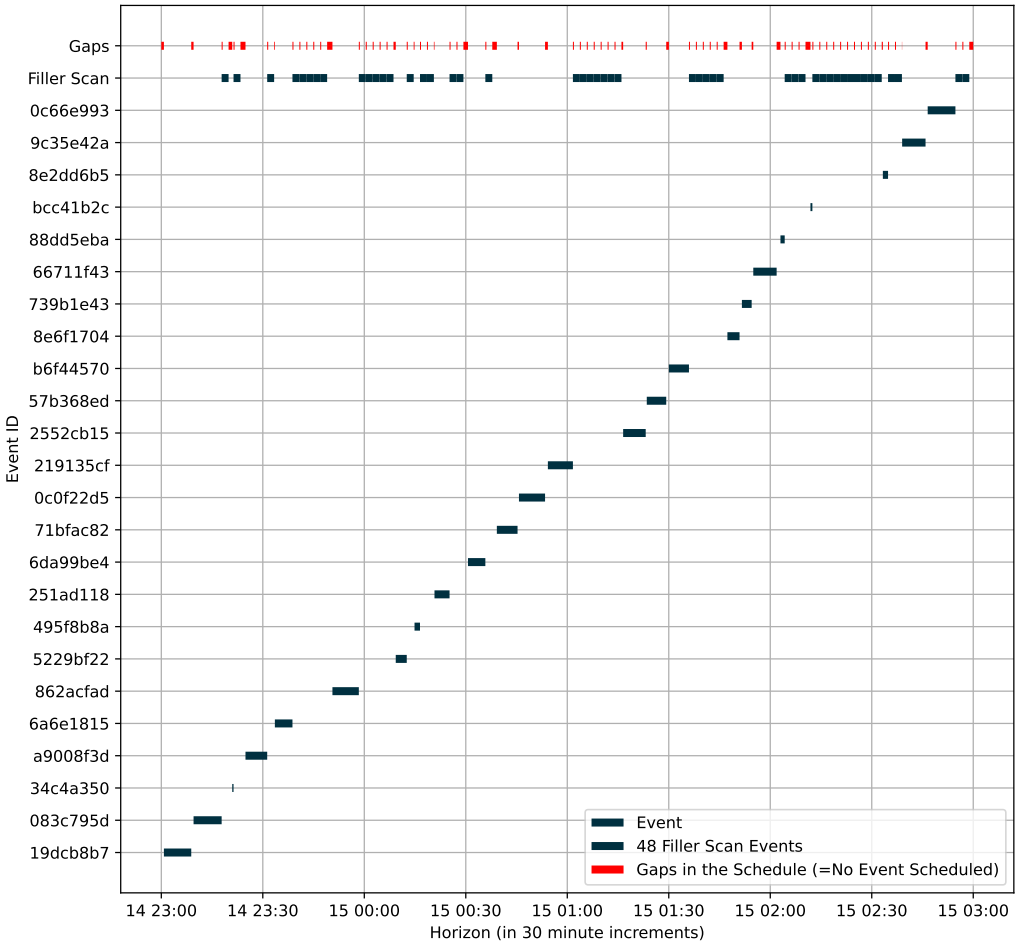

Figure 4.3: The schedule from Figure 4.2, where any possible gaps have been filled by `scan` events. The red lines with the label "Gaps" represent all the gaps in the schedule, where the amount of space is to low to fit a `scan` event.

5 REST-API and Implementation

In this chapter, we describe the implementation of the OOD system, for which the requirements were specified in Chapter 3. The description includes both the REST-API as our external interface, and the scheduler. The entire source code is provided separately and part of this work.

5.1 Structure

The structure of the OOD REST-API and its interfaces is split into 3 parts: frontend, scheduling backend and "sidecar" backend. The frontend is access controlled with an API key, which is enforced by Kong (Figure 5.1). All the code is run in Docker images on LRZ's Kubernetes cluster, and all communication between the various parts is run and controlled inside the cluster. Everything inside the Kubernetes cluster is shown in Figure 5.1 inside the light blue and green rectangles[1]. The use of PostgreSQL [Theb] to store the events makes working with and updating schedules much easier than using files directly. It is also a well known database that is used in a lot of applications, including in other AlpEnDAC projects.

The first part of the structures is the frontend, where users can see a schedule, a list of events and which instruments exist. Users with the correct permissions are also able to submit events to be scheduled and delete events that exist in the database. The frontend can be seen in Figure 5.1 inside the green rectangle. It includes the Docker image `OOD-API`, which receives requests for various endpoints from the `AlpEnDAC UI`, which is what the users interact with directly. The frontend is also responsible for communication with the instruments.

The second and third parts are the scheduling backend and the "sidecar" backend. The scheduling backend, known as the "main" scheduler component, can be seen in Figure 5.1 labeled as `OOD-Scheduler Main`. This part is responsible for actual scheduling operations, by receiving add or delete requests for events from the frontend (which are sent over Apache Kafka [Kafa]), and then adjusting the schedule accordingly. This includes adding or deleting events from the PostgreSQL database, and updating the schedule with new filler scan events, as described in Subsection 3.2.1, Item 2. Delete requests are always fulfilled, and no message is sent back over Kafka. Add requests are processed by the scheduler, and a message is sent back over Kafka whether the event is accepted into the schedule or rejected. `OOD-API` then relays this information back to the user. Kafka was chosen for this, because it enables more durable (i.e., less chance of data getting lost in transit) and scalable communication between the frontend and backend.

The "sidecar" backend/component is responsible for communicating with the `OOD-API` and the PostgreSQL database. It can be seen in Figure 5.1 labeled as `OOD-Scheduler Sidecar`. It provides "internal" URLs that the `OOD-API` calls, for example `/schedule` calls an internal `/schedule` URL. The sidecar then reads the events for the current date from the database,

[1] The dark blue logo at the bottom left of the figure is the Kubernetes logo

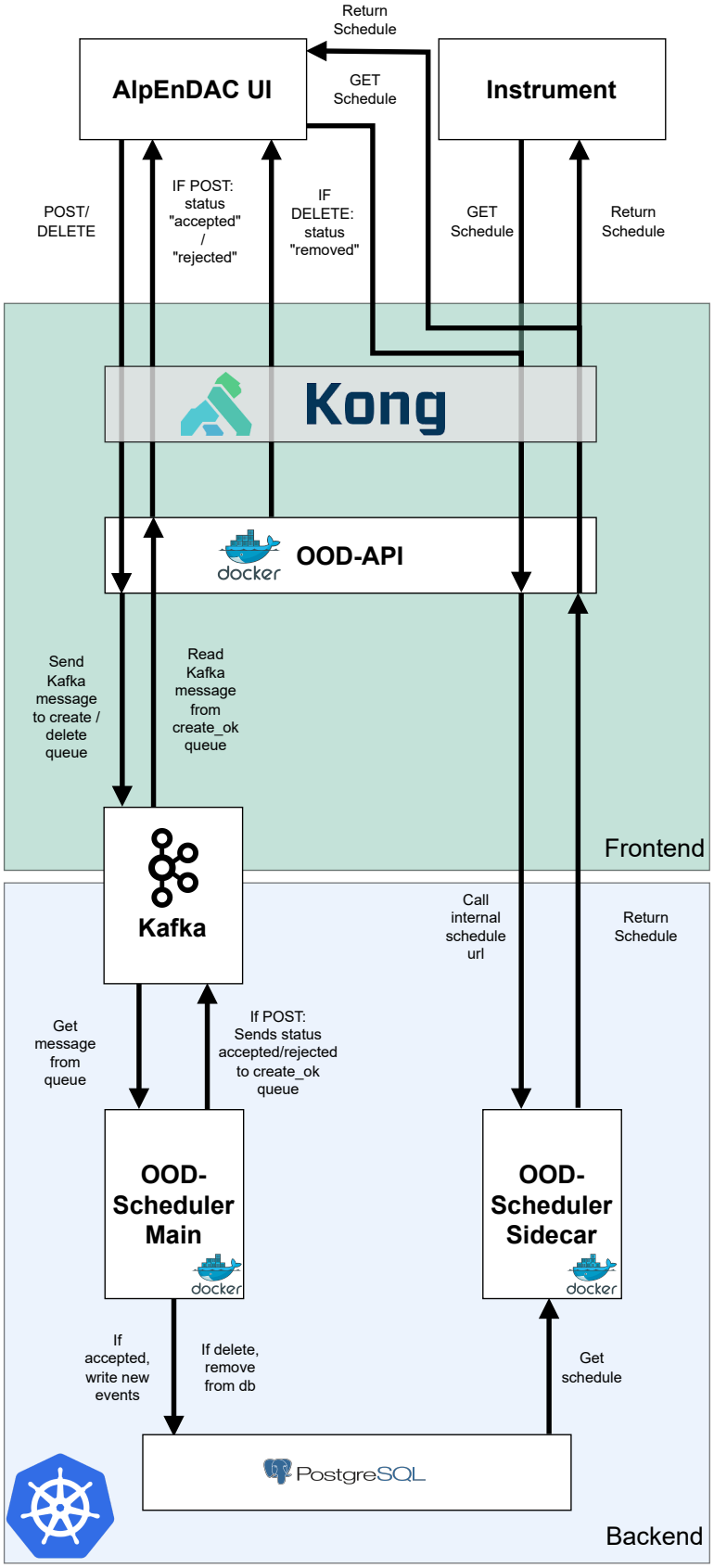

Figure 5.1: A more detailed view of the implemented architecture of the AlpEnDAC operating on demand system, modified from [Gö21]. The light blue rectangle contains the backends and the green rectangle contains the frontend.

makes sure that the filler scan events are up-to-date, and then sends the full JSON schedule to `OOD-API`.

The splitting up of the code and endpoints into 2 Docker images allows for a better separation of concerns. While the `OOD-API` is directly connected to the outside world through Kong, the sidecar only communicates inside Kubernetes, which allows for better security. Another benefit is that changes to the scheduler internals (database, tweaks to OR-Tools usage, etc.), that do not affect the user API, or the communication between the frontend and the backend, can be made without a full redeployment. An illustrative example of this, is that the `OOD-API` Docker image had an uptime of 77 days, before being updated to a version with new functionality, while during this timeframe the `OOD-Scheduler` images where changed multiple times without a loss of functionality. It also allows for better future scaling, because the number of images that communicate directly to the database or are changing the schedule can be changed independently of the number of front-end (user-facing) Docker images.

5.2 REST-API endpoints

Table 5.1: REST-API endpoints for the OOD API.

URL[a]	HTTP Method	Request Data[b]	Response Data[b]	Comments
`/v1/instrument`	GET	-	Ids, types and attributes of all instruments	Currently contains only the FAIM instrument. The additional attributes are attributes relevant to a specific instrument
`/v1/instrument/<id>/schedule`	GET	-	Schedule for current day	Includes filler `scan` events, see Item 2 (Subsection 3.2.1)
`/v1/instrument/<id>/event`	GET	-	List of all scheduled events in the database	Includes scheduled events from the past
`/v1/instrument/<id>/event`	POST	Object, with a single event	`status` of `Added` or `Rejected`	The user sends an event to the scheduler, which tries to schedule this event, and the API returns if the event was added to the schedule
`/v1/instrument/<id>/event`	DELETE	Object with id	`status` of `Removed`	The user sends which event they want deleted, by providing the unique id which identifies the event

[a] `<id>` refers to a UUID4 encoded id value, which uniquely identifies each instrument.
[b] Data is always a JSON:API style document. Events are always of `scan` and `static` event type (FAIM specific, see Section 2.5).

5.3 Frontend

The REST-API is used to request the use of resources, see all events which are active, get a daily schedule for each instrument and receive information about the instruments themselves. A description of the REST-API endpoints is shown in Table 5.1. OpenAPI is used as a user-facing interface to see and work with the API. All OOD URLs have `/v1/` as part of the

URL, to differentiate between the endpoints if the interfaces change, e.g., if version 2 of the scheduler[2] were implemented, `/v2/` would be used.

Data exchange is done with multiple different JSON documents following the JSON:API format. This means that every `data` JSON:API document contains a UUID4 `id`, a `type` describing what the document contains (e.g., `"static"` for a `static` event) and `attributes`, which contain any additional data. There is another JSON:API "type" which might be returned, the `errors` response type. This gets returned if errors occur calling any URL, e.g., validation of the input failed or Kafka was unavailable. An error return also sets an HTTP error code that is not `200`, depending on what the problem was (see Appendix Section 2). The documents for events are `scan` and `static` events, as defined in Section 2.5. They follow all the requirements and formats which are described in Appendix Section 1. An example can be seen in Appendix Section 3.

Although the output format is the same for the `schedule` and `event` endpoints, the returned data is different. The `schedule` URL returns the schedule for the current day, which is defined as all events that occur between tonight dusk and tomorrow dawn UTC. These times are calculated using the Astral [Ken] library. This endpoint is currently also specifically tailored toward FAIM.

The `event` URL, on the other hand, returns all events in the database that are or were scheduled. This means it includes events in the past, which were part of a past schedule.

When deletion of an event is requested, the scheduler in the backend (Section 5.4) recalculates the schedule after each delete, which might have cascading changes for the rest of the schedule.

5.4 Scheduling Backend

The backend takes the requests for use of the instrument from the frontend and tries to create an optimal schedule that maximizes the cumulative duration of the instrument usage while fulfilling the other requirements as described in Section 3.2.

When a request to add an event to the schedule comes from frontend, it is sent as a JSON document to the scheduling backend using Kafka. This can be seen in Figure 5.1, where the `scheduler-main` Docker image polls the Kafka queue for new events. When a new event is received in the queue, the scheduler pulls in all currently scheduled events from the PostgreSQL database, which occur during the same night (night A) as the new event. The scheduler then tries to add the event into the schedule for night A by trying to create an updated schedule using OR-Tools. A schedule is returned from OR-Tools, which might be updated and include the new event, or it is the same schedule as before[3]. If the new event is scheduled, it might mean removing other, less "optimal" (here: shorter) events out of the schedule. In version 2 of the scheduler, events with a lower priority might be removed instead.

If the request was accepted, i.e., the new event can be scheduled, then an "accepted" message is sent back to the frontend as JSON using Kafka. The frontend then returns this data back to the user, as shown in Figure 5.1. Concurrently, the new event is written to the database and other events that no longer fit in the schedule are deleted from the database.

[2] as described in Item 5 (Subsection 3.2.1)

[3] The schedule can be the same, because OR-Tools' solver enables stable solutions. See Section 4.2 and Section 2.4.

On the other hand, if the event can't be scheduled, then a "rejected" message is sent back to the frontend, which the frontend returns to the user with an HTTP code of `409` (see Appendix Section 2) to indicate failure. The database stays the same as before.

5.5 Instrument Backend

This backend provides a REST-API which is read by the instruments themselves, to see what they have to do. The backend is run by the `OOD-Scheduler Sidecar` Docker image (Figure 5.1), which communicates between the PostgreSQL database, which contains the schedule and the instrument through the `OOD-API` frontend.

The returned data from the backend doesn't contain exact instructions, i.e., the steps that need to be fulfilled for an instrument to take a measurement or move around, but only the desired actions, e.g., take a picture at 10:45 am for 10 seconds at angle 20 deg. If and how these desired actions are actually run is controlled and implemented by the owners of the instruments themselves. The reason not to include the actual execution steps for the schedule is because the instruments are large and expensive, which means stakeholders do not want to entrust software-driven instrument control to a third party. The separation of concerns as we have implemented them here makes it possible to program a largely instrument-agnostic system.

For FAIM, the software running the instrument downloads a nightly schedule from the `/schedule` endpoint once at 15:00 UTC and then executes the events in the schedule. After downloading, the computer controlling the FAIM instrument converts the JSON data to a different plain text format used to control the instrument, where the information for one event is one line of text. The computer then runs these commands line by line during the night. It checks the start timestamp and does the next instruction when the current time in UTC is the same as the start timestamp for the next event. The images taken by the FAIM camera are then saved and also sent back to the OOD system's storage system, which stores all the images, organized by event ids, for future use by others.

6 Discussion and Analysis

6.1 Performance of System with Regards to the Requirements

This section compares and contrasts how many of the requirements from stakeholders were met, and how close the actual system came to the requirements, as defined in Chapter 3. It also includes the reasoning behind some design and architecture decisions and why some requirements or their implementation where changed.

6.1.1 Performance with Respect to Scheduling

This section describes how well the system able to schedule, and what its capabilities are (priority, different goals of the scheduler, …).

The scheduler meets all requirements set in Subsection 3.2.1. It is able to fill out the night with as many events as possible, and for all gaps that are large enough, generated scan events are included. These scan events are written into a separate table in the database, which means there is a separation between "real" and generated events. The stretch goal was partially met: a version that includes scheduling with priority. A fully functional implementation is included in the repository, however it has not yet been integrated into production. Easy embed-ability with Python and the rest of the code was achieved, because OR-Tools was used, which has Python bindings and a Python package. The scheduler also makes sure that events for a specified date only take place between dawn and dusk by adding a constraint to the OR-Tools' model.

The requirements in Appendix Section 1 were met. All dates and times for events are in UTC. OR-Tools itself uses relative times for the horizon, as described in Section 2.2. All other aspects of the scheduler use and enforce UTC.

The scheduler almost completely meets the non-functional requirements as described in Subsection 3.2.2, with the only requirement which was not completely met being Item 1. It was not fully met because the complete process of creating new schedules, starting when the user sends the JSON request and ending when they receive a JSON response, takes approximately 1–2 seconds, but can sometimes spike up to 3 seconds. This end-to-end time can vary quite a bit, with values as low as 0.8 seconds being observed. However, the highest amount of time is lost to communication latency sending and receiving data between the frontend and backend using Kafka. The actual scheduling, where an old schedule and a possible new event are input to OR-Tools and a new schedule is returned to the scheduler system, is very fast and takes less than two seconds. The scheduler only accepts correct data and rejects malformed or otherwise unschedule-able data. It also always creates a schedule that fulfills all the requirements, and can always be executed by the instruments, owing to the use of the OR-Tools constraint system. All events are set as optional, which means the scheduler does not need to include any events. In fact, because even an empty schedule is an optimal solution if no events meet the correct criteria, this means that a schedule is always created and the scheduler can run without crashing or running indefinitely. In v1

of the scheduler, the scheduler always tries to create the most optimal schedule as defined in Item 5 (Subsection 3.2.1). In v2 of the scheduler, where the priority is also taken into consideration, the goal from Item 5 (Subsection 3.2.1) is the secondary goal. The primary goal is, that if events overlap, the event with the highest priority should be chosen, even if the sum of the duration is lower as a result.

6.1.2 Performance with Respect to Interface/API Requirements

All performance requirements as described in Section 3.3 were met. Because there was already existing infrastructure on `alpendac.eu`, OpenAPI and Kong were already deployed, which made it easier for all the functional requirements to be met.

In the beginning, before the first deployment to production, the APIs were changed a few times, during multiple discussions with P. Hannawald from the DLR, with respect to P. Hannawald's wishes, the goals of the LRZ and the author [Han21a]. This does not conflict with the requirement from Item 3 (Subsection 3.3.1), because the changes to the APIs occurred before the first deployment into production. After the deployment into production, there were only minimal functional changes, which were confined to which `HTTP` return codes were used for `DELETE` and `POST` operations. All other changes were bugfixes or fixes to enforce stricter validation of user input.

The main changes to the APIs before the first production deployment included switching what type of event values where returned from `/v1/instruments/<instrument_id>/schedule` and adding `/v1/instruments/<instrument_id>/event`. The first draft version included what was known as `simple` events, which were the only events returned from `/schedule`. They represented the "static" event concept, i.e., one orientation of the camera per event (one `azimuth` and one `zenith` value), and were therefore similar to the current `static` events. However, representing only "static" events meant that what is now one `scan` event had to be converted by being split into 35 `simple` events (which is the number of orientations for one scan) to then be returned by `/schedule`. This quickly became a big number of JSON events that need to be created and returned, especially when all possible gaps need to be filled with "scan" events. At 8 hours between dusk and dawn[1], that would mean, in the case with no events other than "scan" events, a total of 234 "scan" events per night.[2]. At 35 images (one per orientation) and therefore `simple` events per "scan" event, that equates to $234 * 35 = 8,190$ `simple` events per night. This was a huge number of JSON objects, which made both performance and readability suffer. Performance, because the large number of objects meant big files and a longer time to calculate and later ingest the values. Readability, because trying to look at 8,190 JSON objects, instead of the current equivalent of 234, became very unwieldy, especially when looking for problems, or later for trying to get a sense of how images should be grouped.

These problems meant that the current system returns `scan` and `static` events from both endpoints. Also changed was that instead of including the duration in the output of a `simple` event, the number of photos each `static` event should take was included instead. The duration value was removed, as number of photos was more useful for `static` events, while having similar functionality (0.5s = 1 photo). Duration was also useless for `scan` events, which always have the same duration. P. Hannawald changed his code that operates the camera to use these two data types, which improved readability, understandability and

[1] a reasonable value, but can vary wildly depending on the time of the year

[2] $8\,\mathrm{h} = 28,800\,\mathrm{s}$ with $123\,\mathrm{s}$ per scan event means $floor(28,800\,\mathrm{s}/123\,\mathrm{s}) = 234$ events

performance from both ends. Readability, because the amount of events sent are much smaller and can be skimmed more easily. Understandability, because fewer events means it is easier to get an overview compared to before, where trying to see which `simple` events would correspond to each "scan" or "static" event was quite unwieldy. Performance, because the large amounts of JSON that needed to be generated and sent took a large amount of resources, and might have required some architectural changes, such as writing parts of the code in another language for better performance.

Furthermore, the amount of commands sent to the camera are much fewer. This is because the commands sent to the camera are line based (see Section 5.5), which meant that, for the old API, each *individual* photo was its own line, which made for much worse readability and understandability. The new API means that individual `scan` photos are grouped as one line and as one event.

6.1.3 Performance with Respect to Sustainability and Reusability Requirements

Almost all requirements described in Section 3.4 have been met. The only requirement which was partially met was Item 1 in Subsection 3.4.1, all other requirements in both subsections were completely met. Although the software is mostly up-to-date, some third-party libraries have released new major versions with breaking changes and were not updated to these new major versions.

As required in Item 6 (Subsection 3.4.1), the scheduling implementation is easy to extend, because it itself has no validation for the events except the basics required of almost any schedule (events inside horizon, no overlap etc.). Validation of events specific to FAIM, as described in Appendix Section 1, is done by functions in `models.py`. This separation of concerns enables the scheduler to be reused more easily, because any instrument-specific logic is dealt with before it is sent to the OR-Tools scheduler. It also means adding new instruments consists mainly of adding new models with new validation, and the actual scheduler implementation only needs very little if any changes. The only changes needed are to adapt the scheduler are to extract the relevant data from the new models (`start` and `end` times, and a unique `id` for each event), assuming the goal, maximum coverage of the horizon, is the same. If the goal is different, the code is written so that adding, removing or changing constraints is relatively easy and the rest of the scheduling framework can be reused, which fulfills the requirement in Item 7 (Subsection 3.4.1).

In the Gitlab repositories, code is only merged and deployed when all steps are completed successfully in Gitlab CI. This means that all functions, classes and modules have correctly formatted comments, enforced by a CI job running pydocstyle. Furthermore, `flake8` is run by the CI which enforces code formatting and also checks for simple errors. All code and code documentation is stored on the LRZ Gitlab server.

All the non-functional requirements (Subsection 3.4.2) have been met. The documentation includes how to start working on new code easily, by providing instructions and code to set up and run all the required packages and Docker images needed. It also includes how to extend the system. The code is well organized into files and folders, depending on their functionality and where else the code might be needed. Well organized code also eliminates import cycles, e.g., 2 files import other libraries and also each other, what can cause code issues. The names for functions, classes, etc. are well-chosen and accurately describe their functionality.

6.1.4 Performance with Respect to System/API Requirements

All requirements as described in Subsection 3.5.1 have been met.

Kubernetes allows a lot of the requirements described in the section to be met without much configuration. This is because Kubernetes automatically restarts containers when they crash, and because deployment of new Docker images means the old images are deleted, the "ephemeral" aspect is guaranteed. Using PostgreSQL also means that crashes imply no data corruption or loss in the database, because of the ACID guarantees. While crashed images might lead to an event getting lost after being sent but not yet written to the database, all accepted events in the database are secure. However, the user is able to see if the event was successfully added if they get back an "accepted" message. All fatal problems before this message is sent cause the event to not be accepted, and the HTTP return code of `504` reflects this. AlpEnDAC's Kubernetes cluster is also able to connect to LRZ's Network Attached Storage (NAS), which allows secure storage of the database files, including regular and automatic backups. The checks listed in Appendix Section 1, which are necessary for the implementation of the system to be correct and usable by the DLR, have been completely implemented.

Almost all requirements described in Subsection 3.5.2 have been met or exceeded. Only Item 8 was not fully met, because some logging statements do not have direct context and are somewhat terse. The quality of the code is high, enforced by Gitlab's CI and merge request systems, where any new changes will only been merged if all stages (tests/linters/builds) finish without errors. Gitlab CI also automatically creates new Docker images from the code after it is approved and merged, which eliminates both work and the possibility of images being out of date or created incorrectly. Although bugs cannot be avoided, the system has been created so that bugs should not cause silent problems, crashes mean that the system will be restarted automatically, and incorrect data will not cause crashes. Docker and docker compose allow for a development environment to be set up easily, because all the relevant programs and code can be run with these two systems. Deployment is very simple using Kubernetes, where just a `kubectl apply -f k8s/*.yaml` call is needed. This means deployment is both easily reproducible and fast, and upgrading is easy, because Kubernetes takes care of the upgrade process, and automatically pulls new Docker images created by Gitlab's CI and starts them.

6.2 Usability and Stakeholder Feedback

In this section, we describe how well the system works and is usable according to the various stakeholders, such as the DLR and LRZ.
The feedback from Patrick Hannawald (DLR) is:

> The communication with you regarding the user requirements worked wonderfully. In several iterations you adapted the API in such a way that on the one hand it is very general purpose, but at the same time meets the requirements for the specific test instrument (FAIM). At the same time, you modified my existing R script so that it works perfectly with the changed API. This helped me a lot. In my view, the API can also be used very well and easily in the future.

[Han21c]

The feedback from the LRZ AlpEnDAC team is:

> The APIs and scheduling system developed match the AlpEnDAC architecture very well. They have enabled the AlpEnDAC team to reach Operating-on-Demand milestones in the project and comprehensively fulfill the requirements of our instrument owners. Importantly, the developed system with its flexible architecture is an excellent building block for future AlpEnDAC services.

[LRZ22]

This positive feedback reassures us in our belief that system is based on correct design decisions and in our assessment that it fulfills all relevant requirements well.

7 Conclusion and Outlook

This work aimed to create a modern framework and REST-API for the AlpEnDAC's OOD work package, which enables management of data and jobs for various instruments in the project. The goal of this work was reached for the FAIM instrument, where the REST-API sees real-world usage and is used for scheduling image capture events each day. Jobs can be scheduled for the FAIM instrument, and the images taken from the camera are now additionally stored on the AlpEnDAC servers. The REST-API also has an OpenAPI interface, which makes onboarding new users easier and allows integrations to be created without in-depth knowledge, just by using the documentation provided by the OpenAPI.

Our work is based on an in-depth requirements analysis with respect to scheduling, APIs/interfaces, and the system and support structures needed to create high quality functionality and code. The theoretical aspects of scheduling have been illuminated, which will provide guidance for future changes or maintenance of the scheduler. A widely used scheduling framework (OR-Tools) has been chosen as the central component, and the backend and frontend components have been implemented as required. The entire system and framework is based on state-of-the-art technology, including widely-used platforms such as Kafka and PostgreSQL. Modern CI/CD and deployment mechanisms (containerisation, Kubernetes) and appropriate Source Code Management/code documentation systems are used.

There are multiple future projects that could be based off of this work. Some would be more OOD or FAIM specific, i.e., adding more instruments, their constraints and attributes, or different ways of specifying when events should occur. Another future possibility would be using this system for other projects/groups which need scheduling for their instruments e.g., other institutions which are part of AlpEnDAC, but are not part of the OOD project. To this end, we have made the scheduler largely independent of any FAIM specific code. Therefore, the aim of using our framework for other instruments requires very few if any fundamental changes. For a new instrument, only very few changes would be needed. One change for a new instrument would be modifying the Flask [Pal] server code to return different data when URLs are called with new `instrument_id`s. The other change would be creating new JSON data models for the new instruments which include their attributes and any specific data constraints. The rest of the code can be easily reused and modified, especially the scheduler and any code interacting with PostgreSQL or Kafka. A relatively easy future improvement would be to get version two of the scheduler into a deployable state.

Further possible future work for the scheduler is to have people submit preferences in terms of if the first event slot does not work, then use the second etc. OR-Tools allows flexibility for such work to occur, and there might already be existing strategies for solving these problems, which just need to be added to the code. Other future improvements for the scheduler could include recurring events with custom repetition dates and preferences, or to extend the types of constraints considered, e.g., weather-related constraints.

We are confident that the system provided will thus be one of the key components for AlpEnDAC to provide new, innovative services, and to constantly adapt them to the needs of the community.

Appendix

1 FAIM Specific Checks

1. Timezone must be UTC.
2. Date and time must be in the ISO8601 or the RFC3339[1] format, with the `Z` suffix as the timezone offset for UTC, instead of `+00:00`, as allowed by RFC3339 [KN02].
3. End time must be after start time.
4. Specific checks for `static` events (see Section 5.3):
   a) The zenith angle (defined as `zenith` in the code) must be between 0 and 70 degrees and have the format of 1 to 3 digits before the decimal, and always 3 digits after the decimal.
   b) The azimuth must be between 0 and 360 degrees and have the format of 1 to 3 digits before the decimal, and always 3 digits after the decimal.
   c) The duration (end time - start time) times 2 must be equal to the number of photos. This is because 1 photo takes 0.5 seconds to be captured.
   d) Number of photos must be a whole number.
5. Specific checks for `scan` events (see Section 5.3):
   a) The duration (end time - start time) must equal 123 seconds, which is the length of time a scan needs to work.

2 HTTP Return Code Explanation

This section describes the possible HTTP return codes for all OOD endpoints. Descriptions are similar to those in OpenAPI, which can be found on `api.alpendac.eu`.

Table 1: HTTP Return Code Explanation

Code	Description
200	Specified action completed successfully.
400	`POST`: Validation failed from user input.
404	URL could not be found.
409	Event has not been added to the schedule. (`POST` only)
504	`GET`: Gateway timeout, when internal endpoints do not respond. `POST`: There was an error with the communication to kafka.

[1] which is a 'profile of the ISO 8601 standard' [KN02], meaning most of the standard overlaps.

3 Example JSON Output


```
{
  "data": [
    {
      "type": "static",
      "id": "022539c2-4717-49b7-834e-91b5cae9653c",
      "attributes": {
        "start_time": "2022-02-16T18:42:24.913Z",
        "end_time": "2022-02-16T18:42:25.913Z",
        "zenith": "60.300",
        "azimuth": "284.000",
        "number_of_photos": 2
      }
    },
    {
      "type": "scan",
      "id": "022539c2-4717-49b7-834e-91b5cae9653c",
      "attributes": {
        "start_time": "2022-02-16T18:42:24.913Z",
        "end_time": "2022-02-16T18:44:27.913Z"
      }
    }
  ],
  "links": {
    "self": "https://api.alpendac.eu/v1/instrument/022539c2-4717-49b7-834e-91
        ↪ b5cae9653c/schedule"
  }
}
```


This JSON document is output which could be returned from the `/event` endpoint. The only difference to this document, if it were returned from `/schedule`, is that there would be filler `scan` events that fill all possible gaps in the schedule.

Glossary

ACID Short for "atomicity, consistency, isolation and durability", which can make a database much more reliable [HR83]. p. 36

AlpEnDAC Alpine Environmental Data Analysis Center. pp. 1, 3, 11–13, 15, 20, 21, 27, 28, 36, 37, 39, 47

API Application Programming Interface. pp. 3, 7, 13, 14, 30

Astral Python library for calculating "[t]imes for various positions of the sun: dawn, […], dusk, […]" [Ken]. p. 31

azimuth Also known as the horizontal angle, it describes the angle between 2 objects that are on the same "horizontal plane" [Mah]. For FAIM, it describes how much the camera can turn left or right. pp. 9, 41, 47

CAIP Constraint-based Attribute and Interval Planning [FJ03]. p. 5

constraint optimization problem As defined in [Apt03]. pp. 5, 6, 21

constraint satisfaction problem As defined in [Apt03]. pp. 5, 6

dawn Uses the civil twilight definition of when the sun goes up [Ast]. "[T]he time of morning […] when the sun is 6° below the horizon" [Glo]. pp. 8, 12, 23, 31, 33, 34

DLR German Aerospace Center. pp. 1, 3, 8, 11, 22, 34, 36

Docker Software for running code in virtualized "containers", which isolates the code and resources from the rest of the system [Docb]. pp. 15, 16, 27, 29, 31, 32, 35, 36, 43

docker compose "Compose is a tool for defining and running multi-container Docker applications." [Doca] It uses a YAML file that includes all the configuration for the Docker containers. p. 36

DST Daylight Savings Time. pp. 8, 12

dusk Uses the civil twilight definition of when the sun goes down [Ast]. "[T]he time of …[the] evening when the sun is 6° below the horizon" [Glo]. pp. 7, 8, 12, 23, 31, 33, 34

EUROPA Short for "Extensible Universal Remote Operations Planning Architecture", it is a planning and scheduling "library and tool set" developed by NASA [BBD+12]. pp. 6, 19–21

FAIM Acronym for "Fast Airglow Imager", an infrared camera [DLR]. pp. 1–3, 5, 8–12, 14, 15, 17, 20, 22, 30–32, 35, 39, 43, 46, 47

flake8 "flake8 is a python tool [...to] check the style and quality of [...] python code" [Pytb]. pp. 14, 15

Flask "Flask is a [...] web application framework" [Pal]. p. 39

Functional requirements Requirements that describe how a system should work internally, e.g., given an input, what should the output of the system be. It includes items such as "[...] calculations, technical details, data manipulation and processing, and other specific functionality" [SG05, p. 110]. p. 11

Gitlab A platform for hosting and managing git [Gita] code repositories. pp. 14, 15, 22, 35, 36

Gitlab CI "GitLab CI/CD is a tool for software development using [...] Continuous Integration (CI)" [Gitc], integrated with the GitLab platform. CI systems can run tests and linters, build code etc. automatically, for example after each commit, which enforces that all the relevant steps are always run [Gitc]. pp. 14, 15, 35, 36

HTTP Short for Hypertext Transfer Protocol, it "is an [...] protocol designed to transfer information between networked devices" [Clo]. In this work, it is used synonymously with HTTPS, which is the encrypted version of HTTP, but is otherwise identical [MDN]. pp. 7, 31, 36, 41

JSON "JSON is a lightweight, text-based, language-independent syntax for defining data interchange formats" [Ecm15]. pp. 7, 8, 13, 17, 23, 29, 31–35, 39, 42

JSON Schema JSON Schema is used to give a specific structure and types to JSON documents. It is similar to types in static programming languages, and allows JSON documents to be well-formed and their validity checked. It also serves as documentation, which makes working with JSON files from other sources easier and less error-prone [WAHD20]. pp. 8, 10, 13, 15

JSON:API JSON:API is a specified format for JSON documents, and also a specification for how JSON documents are sent between the server and the client [JSO]. pp. 7, 8, 30, 31

Kafka "[...] Kafka is an open-source distributed event streaming platform [...]" [Kafa]. Event streaming describes a system where messages are sent, received and processed in "real-time" from different sources, and then stored, in this case by Kafka [Kafb]. pp. 27, 31, 33, 39

Kong Kong Ingress Controller (enables security and access controls to the Kubernetes cluster) [Kon]. pp. 13, 27, 29, 34

Kubernetes "[... A]n open-source system for automating deployment, scaling, and management of containerized applications." [Thea]. pp. 15, 16, 21, 27, 29, 36, 39

LRZ Leibniz Supercomputing Centre. pp. 1, 14, 15, 27, 36, 37

NDDL Language used for planning created at NASA, it is an extension to a widely used planning language known as "PDDL" [GHK+98], although there are multiple differences [BBD+12, Section 2.1]. pp. 19, 20

Non-functional requirements Requirements which do not define how a system should work in detail or how an implementation should be coded, but which rather "[…] include[s] how easy the software is to use, how quickly it executes, how reliable it is, and how well it behaves when unexpected conditions arise" [SG05, p. 113]. This includes requirements such as "[t]he maximum number of seconds it must take to perform a task" [SG05, p. 113], which does not describe how this requirement should be achieved, but rather how to measure if the requirement was achieved. p. 11

OOD Operating-on-Demand. pp. 1, 3, 5, 7, 8, 11, 27, 30, 32, 39, 41

OpenAPI "[A] standard, language-agnostic interface to RESTful APIs", which makes working with APIs simpler to use and to implement in multiple programming languages [Ope]. pp. 3, 13, 30, 34, 39, 41

OR-Tools "OR-Tools is an open source software suite for […] constraint programming" [PF], and can be used for scheduling [OR-a]. pp. 3, 6, 7, 19–23, 29, 31, 33, 35, 39

pep8 "Style Guide for Python Code" [VRCW]. p. 14

PHP A programming language [PHP]. p. 3

PostgreSQL "PostgreSQL is a powerful, open source object-relational database" [Theb]. pp. 27, 31, 32, 36, 39

pydocstyle "[…A] static analysis tool for checking compliance with Python docstring conventions" [Pyd]. pp. 14, 35

pytest A Python testing framework [Pyta]. pp. 14, 15

Python The Python programming language [VRD09]. pp. 12, 15, 19, 20, 33, 43

REST-API A REST-API is an API, which follows the REST (Representational state transfer) principles as outlined in [Fie00], which describes how to structure an API architecturally. pp. 3, 7, 11, 27, 30, 32, 39

UTC Coordinated Universal Time [KN02]. pp. 7, 8, 12, 31–33, 41

UUID4 Short for "Universally Unique Identifier" version 4 [ITU]. It is an 128 bit number that should uniquely identify an object. A version 4 UUID is defined in the standard as one which is "random-number-based", i.e., the numbers used to generate the UUID "shall be a randomly or pseudo-randomly generated 60-bit value" [ITU]. This is in comparison to other UUID versions, which uses names or times as input to generate the UUID value [ITU]. pp. 7, 24, 30, 31

VAO Virtual Alpine Observatory [Bav]. p. 1

zenith angle Describes the angle between an object in the sky (known as the zenith) and a vertical line, as seen from the observer [Hon]. For FAIM, it is used as one of the coordinates along which the camera can turn. pp. 9, 41, 47

List of Figures

Bibliography

[Alp] AlpEnDAC: *AlpEnDAC Homepage*. https://www.alpendac.eu/. – accessed on 5-12-2020

[Apt03] Apt, Krzysztof: *Principles of constraint programming*. Cambridge New York : Cambridge University Press, 2003. – ISBN 0521825830

[Ast] Astronomical Applications Department, U.S. Naval Observatory: *Rise, Set, and Twilight Definitions*. https://web.archive.org/web/20190927120626/http://aa.usno.navy.mil/faq/docs/RST_defs.php. – accessed on 18-10-2021

[Bav] Bavarian State Ministry of the Environment and Consumer Protection: *The Virtual Alpine Observatory (VAO)*. https://www.vao.bayern.de/. – accessed on 26-02-2022

[BBD+12] Barreiro, Javier ; Boyce, Matthew ; Do, Minh ; Frank, Jeremy ; Iatauro, Michael ; Kichkaylo, Tatiana ; Morris, Paul ; Ong, James ; Remolina, Emilio ; Smith, Tristan u. a.: EUROPA: A platform for AI planning, scheduling, constraint programming, and optimization. In: *4th International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS)* (2012)

[BDP96] Błażewicz, Jacek ; Domschke, Wolfgang ; Pesch, Erwin: The job shop scheduling problem: Conventional and new solution techniques. In: *European journal of operational research* 93 (1996), Nr. 1, S. 1–33

[BLPN01] Baptiste, Philippe ; Le Pape, Claude ; Nuijten, Wim: *Constraint-based scheduling: applying constraint programming to scheduling problems*. Bd. 39. Springer Science & Business Media, 2001

[BRSV17] Bourhis, Pierre ; Reutter, Juan L. ; Suárez, Fernando ; Vrgoč, Domagoj: JSON: data model, query languages and schema specification. In: *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI symposium on principles of database systems*, 2017, S. 123–135

[CCJK06] Cohen, David A. ; Cooper, Martin C. ; Jeavons, Peter G. ; Krokhin, Andrei A.: The complexity of soft constraint satisfaction. In: *Artificial Intelligence* 170 (2006), Nr. 11, S. 983–1016

[Clo] Cloudflare, Inc: *What is HTTP? | Cloudflare*. https://www.cloudflare.com/learning/ddos/glossary/hypertext-transfer-protocol-http/. – accessed on 09-03-2022

[DCT19] Da Col, Giacomo ; Teppan, Erich C.: Industrial size job shop scheduling tackled by present day CP solvers. In: *International Conference on Principles and Practice of Constraint Programming* Springer, 2019, S. 144–160

[DLR] Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR), Earth Observation Center (EOC): *DLR - Earth Observation Center - Die Mesopause in 3D*. https://www.dlr.de/eoc/desktopdefault.aspx/tabid-11932/20674_read-50219/. – accessed on 27-10-2021

[Doca] Docker: *Overview of Docker Compose | Docker Documentation*. https://docs.docker.com/compose/. – accessed on 12-01-2022

[Docb] Docker: *What is a Container? | App Containerization | Docker*. https://www.docker.com/resources/what-container. – accessed on 12-01-2022

[Docc] *Best practices for writing Dockerfiles | Docker Documentation*. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/#create-ephemeral-containers. – accessed on 09-04-2022

[Ecm15] Ecma, Standard: *ECMA-404 The JSON Data Interchange Format*. 2015

[Enc] Encyclopædia Britannica: *Layers of Earth's atmosphere*. https://www.britannica.com/science/mesopause/images-videos#/media/1/376810/99826 accessed on 06-02-2022

[FF10] Fattahi, Parviz ; Fallahi, Alireza: Dynamic scheduling in flexible job shop systems by considering simultaneously efficiency and stability. In: *CIRP Journal of Manufacturing Science and Technology* 2 (2010), Nr. 2, S. 114–123

[Fie00] Fielding, Roy T.: *Representational State Transfer (REST)*. University of California, Irvine, 2000

[FJ03] Frank, Jeremy ; Jónsson, Ari: Constraint-based attribute and interval planning. In: *Constraints* 8 (2003), Nr. 4, S. 339–364

[GHK+98] Ghallab, Malik ; Howe, Adele ; Knoblock, Craig ; McDermott, ISI D. ; Ram, Ashwin ; Veloso, Manuela ; Weld, Daniel ; SRI, David W. ; Barrett, Anthony ; Christianson, Dave u. a.: PDDL| The Planning Domain Definition Language. In: *Technical Report, Tech. Rep.* (1998)

[Gita] *Git*. https://git-scm.com/. – accessed on 16-04-2022

[Gitb] *Gitlab*. https://about.gitlab.com/. – accessed on 31-01-2022

[Gitc] *GitLab CI/CD*. https://docs.gitlab.com/ee/ci/. – accessed on 31-01-2022

[Glo] Global Monitoring Laboratory (GML) of the National Oceanic and Atmospheric Administration: *Solar Calculator Glossary*. https://gml.noaa.gov/grad/solcalc/glossary.html#civiltwilight. – accessed on 18-10-2021

[Gö21] Götz, A.: *AlpEnDAC OoD Architektur*. April 2021. – From Internal Dokumentation

[Han21a] Hannawald, Patrick: *OOD and FAIM Discussions*. 2020 –2021. – sent in an email

[Han21b] Hannawald, Patrick: *FAIM camera illustration*. 2021. – sent in an email

[Han21c] Hannawald, Patrick: *Feedback OOD*. 2021. – sent in an email

[Hon] Honsberg, C.B. and Bowden, S.G.; "Photovoltaics Education Website": *Elevation Angle | PVEducation*. `https://www.pveducation.org/pvcdrom/properties-of-sunlight/elevation-angle`. – accessed on 12-01-2022

[HR83] Haerder, Theo ; Reuter, Andreas: Principles of transaction-oriented database recovery. In: *ACM computing surveys (CSUR)* 15 (1983), Nr. 4, S. 287–317

[HSWB16] Hannawald, Patrick ; Schmidt, Carsten ; Wüst, Sabine ; Bittner, Michael: A fast SWIR imager for observations of transient features in OH airglow. In: *Atmospheric Measurement Techniques* 9 (2016), Nr. 4, S. 1461–1472

[ITU] ITU-T Study Group 17: *Information technology - Procedures for the operation of object identifier registration authorities: Generation of universally unique identifiers and their use in object identifiers*. `https://handle.itu.int/11.1002/1000/11746`. – accessed on 10-03-2022

[JMM+00] Jónsson, Ari K. ; Morris, Paul H. ; Muscettola, Nicola ; Rajan, Kanna ; Smith, Benjamin D.: Planning in Interplanetary Space: Theory and Practice. In: *AIPS*, 2000, S. 177–186

[JSO] *JSON:API — Latest Specification*. `https://jsonapi.org/format/`. – accessed on 18-10-2021

[Kafa] *Apache Kafka*. `https://kafka.apache.org/`. – accessed on 31-01-2022

[Kafb] *Apache Kafka Introduction*. `https://kafka.apache.org/intro`. – accessed on 11-02-2022

[Ken] Kennedy, Simon: *astral*. `https://pypi.org/project/astral/`. – accessed on 18-10-2021

[KN02] Klyne, Graham ; Newman, Chris: Date and Time on the Internet: Timestamps / RFC Editor. Version: July 2002. `https://www.rfc-editor.org/rfc/rfc3339`. RFC Editor, July 2002 (3339). – RFC. – ISSN 2070–1721. – accessed on 18-10-2021

[Kon] Kong: *Kong Kubernetes: Ingress Controller Solutions to Manage Clusters*. `https://konghq.com/solutions/kubernetes-ingress/`. – accessed on 11-12-2021

[LRZ22] LRZ AlpEnDAC Team: *Feedback OOD*. 2022. – sent in an email

[Mah] Mahun, Jerry: *Chapter G. Horizontal Angles*. https://jerrymahun.com/index.php/home/open-access/i-basic-principles/88-i-g-horizontal-angles?showall=1. – accessed on 12-01-2022

[MDN] MDN contributors: *HTTPS - MDN Web Docs Glossary: Definitions of Web-related terms | MDN*. https://developer.mozilla.org/en-US/docs/Glossary/https. – accessed on 09-03-2022

[MPD10] Maleshkova, Maria ; Pedrinaci, Carlos ; Domingue, John: Investigating web apis on the world wide web. In: *2010 eighth ieee european conference on web services* IEEE, 2010, S. 107–114

[MW] Merriam-Webster: *"Mesopause." Merriam-Webster.com Dictionary*. https://www.merriam-webster.com/dictionary/mesopause. – accessed on 17-01-2022

[NAS] NASA Ames Research Center: *Nasa Scheduling Software*. https://ti.arc.nasa.gov/tech/asr/groups/planning-and-scheduling/software/. – accessed on 5-12-2020

[Ope] OpenAPI Initiative: *OAI/OpenAPI-Specification: The OpenAPI Specification Repository*. https://github.com/OAI/OpenAPI-Specification. – accessed on 17-8-2021

[Opt] *Python Reference: CP-SAT NewOptionalIntervalVar | OR-Tools*. https://developers.google.com/optimization/reference/python/sat/python/cp_model?hl=en#newoptionalintervalvar. – accessed on 12-04-2022

[OR-a] *About OR-Tools | Google Developers*. https://developers.google.com/optimization/introduction/overview. – accessed on 29-01-2022

[OR-b] *The Job Shop Problem | Google Developers*. https://developers.google.com/optimization/scheduling/job_shop. – accessed on 24-02-2022

[Pal] Pallets: *Flask | The Pallets Projects*. https://palletsprojects.com/p/flask/. – accessed on 31-01-2022

[Par07] Park, Chris: *mesopause*. http://dx.doi.org/10.1093/acref/9780198609957.013.4927. Version: 2007

[PF] Perron, Laurent ; Furnon, Vincent: *OR-Tools*. https://developers.google.com/optimization/. – Version 8.1, 2019-7-19

[PHP] *PHP: Hypertext Preprocessor*. https://www.php.net/. – accessed on 18-11-2021

[Pyd] *pydocstyle's documentation*. https://www.pydocstyle.org/en/stable/. – accessed on 02-04-2022

[Pyta] Pytest developers: *pytest homepage*. https://docs.pytest.org/en/stable/. – accessed on 24-02-2022

[Pytb] Python Code Quality Authority: *GitHub - PyCQA/flake8*. https://github.com/pycqa/flake8. – accessed on 24-02-2022

[Ros06] Rossi, Francesca: *Handbook of constraint programming*. Amsterdam Boston : Elsevier, 2006. – ISBN 9780080463803

[RPB13] Rajan, Kanna ; Py, Frédéric ; Barreiro, Javier: Towards deliberative control in marine robotics. In: *Marine Robot Autonomy*. Springer, 2013, S. 91–175

[SAT] *CP-SAT Solver | OR-Tools*. https://developers.google.com/optimization/cp/cp_solver#cp-sat-return-values. – accessed on 15-03-2022

[SFS+] Stuckey, Peter J. ; Feydy, Thibaut ; Schutt, Andreas ; Tack, Guido ; Fischer, Julien: *MiniZinc Challenge 2021 Results*. https://www.minizinc.org/challenge2021/results2021.html. – accessed on 07-03-2022

[SFS+14] Stuckey, Peter J. ; Feydy, Thibaut ; Schutt, Andreas ; Tack, Guido ; Fischer, Julien: The minizinc challenge 2008–2013. In: *AI Magazine* 35 (2014), Nr. 2, S. 55–60

[SG05] Stellman, Andrew ; Greene, Jennifer: *Applied software project management*. "O'Reilly Media, Inc.", 2005

[Thea] The Kubernetes Authors: *Kubernetes*. https://kubernetes.io/. – accessed on 11-12-2021

[Theb] The PostgreSQL Global Development Group: *PostgreSQL*. https://www.postgresql.org/. – accessed on 31-01-2022

[VRCW] Van Rossum, Guido ; Coghlan, Nick ; Warsaw, Barry: *PEP 8 – Style Guide for Python Code*. https://www.python.org/dev/peps/pep-0008/. – accessed on 26-02-2022

[VRD09] Van Rossum, Guido ; Drake, Fred L.: *Python 3 Reference Manual*. Scotts Valley, CA : CreateSpace, 2009. – ISBN 1441412697

[WAHD20] Wright, Austin ; Andrews, Henry ; Hutton, Ben ; Dennis, Greg: JSON Schema: A Media Type for Describing JSON Documents / Internet Engineering Task Force. Version: Dezember 2020. https://datatracker.ietf.org/doc/html/draft-bhutton-json-schema-00. Internet Engineering Task Force, Dezember 2020 (draft-bhutton-json-schema-00). – Internet-Draft. – Work in Progress

[Wik21] Wikimedia Commons: *File:Azimuth-Altitude schematic.svg — Wikimedia Commons, the free media repository*. https://commons.wikimedia.org/w/index.php?title=File:Azimuth-Altitude_schematic.svg&oldid=599340224. Version: 2021. – accessed on 07-02-2022

[Zha02] Zhang, Weixiong: Modeling and solving a resource allocation problem with soft constraint techniques. (2002)