International
Virtual
Observatory
Alliance

IVOA Execution Planner

Version 1.0

IVOA Working Draft 2023-07-01

Working Group
GWS

This version
https://www.ivoa.net/documents/ExecutionPlanner/20230701

Latest version
https://www.ivoa.net/documents/ExecutionPlanner

Previous versions
This is the first public release

Author(s)
Dave Morris

Editor(s)
Dave Morris

Abstract

One of the long term goals of the IVOA has been to enable users to move the code to the data. This is becoming more and more important as the size and complexity of the datasets available in the virtual observatory increases.

The IVOA Execution Planner provides a step towards making this possible.

The IVOA Execution Planner is designed to address a specific question; given an executable thing, e.g. a Python program or Jupyter notebook. What facilities are available to run it?

To do this, the IVOA Execution Planner specification defines a data model and webservice API for describing executable things and the resources needed to execute them.

Together these components enable a user to ask a simple question *"Where (and when) can I execute my program?"*

This in turn enables users to move code between science platforms. Allowing them to develop their code on one platform and then apply it to a different dataset by sending it to execute on another platform.

Status of this document

This is an IVOA Working Draft for review by IVOA members and other interested parties. It is a draft document and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use IVOA Working Drafts as reference materials or to cite them as other than "work in progress".

A list of current IVOA Recommendations and other technical documents can be found at https://www.ivoa.net/documents/.

Contents

Acknowledgments

The authors would like to thank all the participants in the IVOA and ESCAPE projects who have contributed their ideas, critical reviews, and suggestions to this document.

Conformance-related definitions

The words "MUST", "SHALL", "SHOULD", "MAY", "RECOMMENDED", and "OPTIONAL" (in upper or lower case) used in this document are to be interpreted as described in IETF standard RFC2119 (Bradner, 1997).

The *Virtual Observatory (VO)* is a general term for a collection of federated resources that can be used to conduct astronomical research, education, and outreach. The International Virtual Observatory Alliance (IVOA) is a global collaboration of separately funded projects to develop standards and infrastructure that enable VO applications.

1 Introduction

The IVOA Execution Planner specification defines two webservice interfaces, the ExecutionPlanner and the ExecutionWorker, and a common data model for describing executable tasks.

Together these provide a common interface for service discovery, resource allocation and execution scheduling across a heterogeneous federation of different types of execution platform.

- ExecutionPlanner webservice – a discovery service to find execution platforms, allocate resources and schedule execution.
- ExecutionWorker webservice – an asynchronous service for executing tasks (based on the IVOA UWS pattern).
- ExecutionPlanner data model – a common data model for describing an executable thing and its resource requirements.

1.1 Role within the VO Architecture

The IVOA Architecture(Arviset and Gaudet et al., 2010) provides a high-level view of how IVOA standards work together to connect users and applications with providers of data and services. Fig. 1 shows the role the IVOA Execution Planner plays within this architecture.

In response to the increasing size and complexity of the next generation of science datasets a number of IVOA members are developing intergrated science platforms which bring together the datasets co-located with the compute resources needed to analyse them.[1][2]

The science platforms make extensive use of the IVOA data models and vocabularies to describe their datasets, and use the IVOA data access services to find and access data from other data providers. In addition, some of the

[1] https://data.lsst.cloud/
[2] https://rsp.lsst.io/index.html

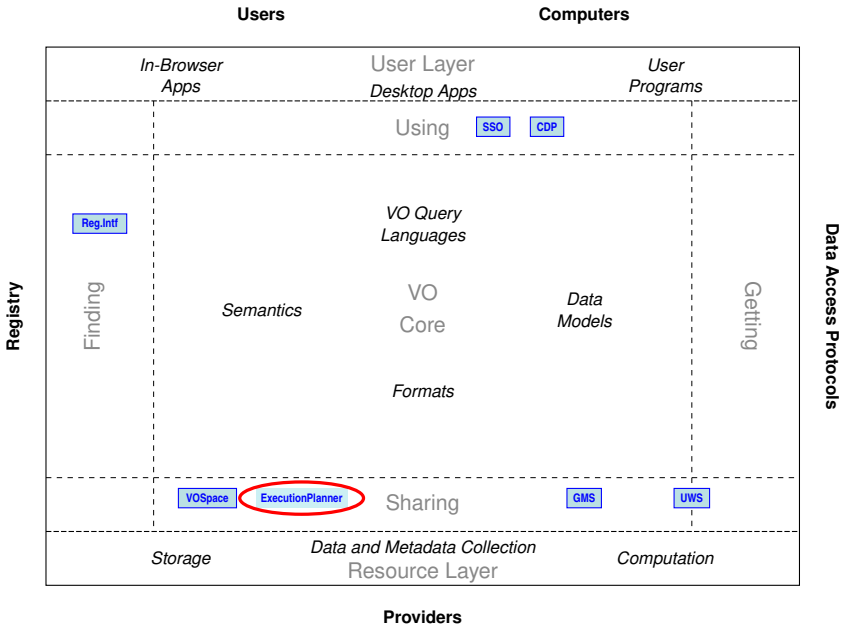

Figure 1: Architecture diagram showing the IVOA Execution Planner's role in the IVOA

science platforms use IVOA VOSpace services to manage data transfers to and from local storage co-located with the compute resources.

However, to date the IVOA does not provide any APIs or webservice interfaces that enable science platforms to exchange the software used to analyse the data. The IVOA Execution Planner provides a step towards making this possible.

This places the IVOA Execution Planner in the same region of the IVOA architecture as the IVOA VOSpace specification (Graham and Morris et al., 2009), providing an infrastructure level service that enables service discovery, resource allocation and execution scheduling across a heterogeneous federation of execution platforms.

The IVOA Execution Planner specification refers to the IVOA Single-Sign-On standard (Taffoni and Schaaf et al., 2017) for authentication (see section xx)and the IVOA Credential Delegation Protocol (Plante and Graham et al., 2010) for delegating credentials to other services.

The IVOA Execution Planner specification also describes how to register an ExecutionPlanner service in the IVOA Registry (Benson and Plante et al., 2009), making it findable within the wider context of the VO.

1.2 Executable things

To understand the problem that the IVOA Execution Planner is trying to solve it is useful to describe what an *executable* thing is in this context. In general terms, this document refers to something that can be executed, or run, as an *executable*.

To explain what this means we can start with a science domain function that we want to perform. For example, the mathematical concept of the square root of a number. We can calculate the square root of a positive number using the Newton–Raphson algorithm[3] which produces successively closer approximations to the result. However, in general case, this mathematical description of the algorithm would not be considered to be an *executable* thing.

We can write a Python program to use this algorithm to calculate the square root of a number. This is the first identifiable *executable* thing in our example. To be able to use this *executable* thing, you would need a computing resource with the appropriate hardware and software environment. In this case, a computing resource with the Python interpreter installed along with any additional Python modules required by the program. This environment is often referred to as the Python runtime.

In the context of science platforms and data science, a common pattern is to provide this environment using an OCI[4], or Docker[5] container, to package the Python program and runtime together as a single binary object. This package, or container, is itself an *executable* thing. One which requires a different execution environment than the original Python program. The aim of containerization is to package *executable* things together with the software environment they need as a binary object that interfaces with a standard execution environment, referred to as the *container runtime*. To be able to use this *executable* thing, you would need a computing resource with the appropriate hardware and software environment. In this case, a computing resource with the OCI container runtime installed.

We could also create a Jupyter notebook that demonstrates how to use our Python program. This is the third *executable* thing in our example. One which provides an interactive environment for the user to experiment with. As before, to be able to use this *executable* thing, we would need a computing resource with the appropriate hardware and software environment. In this case, a computer with the Jupyter notebook platform installed along with all the Python modules needed by our Python program. In the context of science platforms and data science, a common pattern is to provide this environment as a webservice that allows the user to connect to the Jupyter notebook via a web browser.

[3] https://en.wikipedia.org/wiki/Newton%27s_method
[4] https://opencontainers.org/
[5] https://docs.docker.com/get-started/what-is-a-container/

From one algorithm that implements a science domain function, we have created three different *executable* things. A Python program, an OCI container packaging the Python program, and an interactive Jupyter notebook that demonstrates how to use the Python program. Each of which requires a different computing environment to execute. A basic Python runtime, the OCI container runtime, and a Jupyter notebook service.

We may also want to consider the data that we are applying the algorithm to. If we are running some small experiments to learn how to use the algorithm, then a basic computing resource will probably be sufficient. However, if we have a dataset of ten million numbers that we want to process, then we may need to consider adding extra storage to handle the input data and the results. For a large dataset it may also be worth using a GPU to accelerate the calculation steps for such a large dataset.

The IVOA Execution Planner data model provides a way to describe what each of these *executable* things are and what resources are needed to execute them. This can include things like number of CPU cores and amount of memory it needs, whether it needs a GPU, the location of the input data, the storage space needed to perform the calculation and the storage space needed to save the results.

1.3 Type URIs and specifications

Several parts of the IVOA Execution Planner data model follow a common pattern, using a URI to identify a `type` followed a `spec` section to fill in the details. This pattern is adopted from a similar pattern used by Kubernetes to describe components deployed in the system.

However, the Execution Planner data model specifically recomends using a long lasting resolvable HTTP URL as the type identifier.

The reasoning behind this is that if a service provider notices several clients are requesting *'encabulation'* on their compute nodes, the service provider may not know what this means. However, if we use a resolvable URL to identify the type[6], a service provider can use it to find a human readable description and decide whether they want to provide it on their platform.

To the webservice both of these requests are just a simple string comparison against a dictionary of known terms:

```
Request  - Can this platform provide 'encabulation' ?
Response - YES|NO
```

```
Request  - Can this platform provide 'https://tinyurl.com/encabulation
    ' ?
Response - YES|NO
```

[6] https://tinyurl.com/encabulation

The second form provides a simple mechanism for linking the type identifier to a human readable description.

While it is possible to use IVOA registry URIs to identify the `type`, this specification recomends using a simple HTTP/S URL. This lowers the barrier to entry and makes it simpler for the end user to resolve the URL into a description.

2 Client-server conversation

The core idea behind the IVOA Execution Planner is based on a conversation between a client and one or more ExecutionPlanner services to discover where, how, and when, an *executable* thing can be executed.

The conversation starts with the client sending a description of the *executable* thing they want to run to the ExecutionPlanner services, which respond with a top level `YES|NO` answer, and if the answer is `YES`, a list of offers describing how it can be executed on their platform.

```
Request  - Can this platform execute <task> ?
Response - YES, list of <offer>[]
```

The client can then choose which of the offers it wants to use and reply with a message accepting the offer. In response the ExecutionPlanner will mark the resources in the offer as reserved, initiate a *job* in an ExecutionWorker service to execute the task and reply with details of how to access it.

```
Request  - I accept <offer>[n]
Response - <job details>
```

The client can then use the connection details in the *job* response to contact the ExecutionWorker service and track its status.

Note that the client does not need to cancel the offers made by the other ExecutionPlanner services. Offers are only valid for a limited period of time, and expire naturally when they reach the end of their lifetime.

2.1 The *executable*

At the simplest level the client just needs to check whether a platform is able to execute a particular type of *executable* task. For example, *"Is this platform able to run a Jupyter notebook?"*

In order to do this, the request needs to specify the task type, e.g. Jupyter notebook, along with details about it, e.g. where to fetch the notebook from.

The information in this part of the data model will be different for each type of *executable*. Rather than try to model every possible type of *executable* in one large data model, the data model for each type is described in an extension to the core data model.

To support this, the core data model defines two fields:

- `type` - a URI identifying the type of *executable*.
- `spec` - a place holder for type specific details.

```
# ExecutionPlanner client request.
# Details of the executable.
executable:

  # A URI identifying the type of executable.
  type: "https://www.purl.org/ivoa.net/executable-types/example"

  # The details, specific to the type of executable.
  spec: {}
```

2.1.1 Jupyter notebook

The data model for each type of *executable* defines the metadata needed to describe that particular type. For example, the data model for a Jupyter notebook needs to describe where to fetch the source code for the notebook from.

```
# ExecutionPlanner client request.
# Details of the executable.
executable:
  # A URI identifying the type of executable.
  type: "https://www.purl.org/ivoa.net/executable-types/jupyter-
    notebook"

  # The details, specific to a Jupyter notebook.
  spec:
    notebook: "https://.../example.jpnb"
```

It may also include a reference to a `requirements.txt` file that describes any additional Python libraries needed to run the notebook.

```
# ExecutionPlanner client request.
# Details of the executable.
executable:
  # A URI identifying the type of executable.
  type: "https://www.purl.org/ivoa.net/executable-types/jupyter-
    notebook"

  # The details, specific to a Jupyter notebook.
  spec:
    notebook: "https://.../example.jpnb"
    requirements: "https://.../requirements.txt"
```

2.1.2 OCI container

The data model for an OCI container needs to describe what operating system and system architecture the container is built for, and where to fetch the binary image from.

```
# ExecutionPlanner client request.
# Details of the executable.
executable:
  # A URI identifying the type of executable.
  type: "https://www.purl.org/ivoa.net/executable-types/oci-container"

  # The details, specific to an OCI container.
  spec:
    os:    "linux"
    arch:  "amd64"
    repo:  "ghcr.io"
    image: "ivoa/oligia-webtop"
    version: "ubuntu-2022.01.13"
```

This pattern of using a `type` URI to identify the type of thing, and then a `spec` block to add the type specific details is used in several places in the Execution Planner data model. This enables us to keep the core data model relatively small, defining the common aspects needed to describe an *executable* thing and the resources it needs while allowing the data model to be extended to describe a wide range of different types of things.

This pattern makes it easy for projects outside the core IVOA community to add new types of *executable* things and resources appropriate for their science domain.

Using a URI to identify the task type means that implementations do not need to understand all of the different possible types of *executable*. If a service doesn't recognize a particular type, it can simply reply `NO`.

```
Request  - Can this platform execute <unkown-type> ?
Response - NO
```

2.2 Resources

At the next level the client may need to check whether a platform has sufficient compute resources needed to execute a particular task. For example, *"Can this platform provide enough resources to run this Jupyter notebook?"*

In order to do this the request would not only need to describe the *executable* itself, but also the minimum level of compute resources needed in terms of CPU cores, memory, GPUs and disc space needed to execute it.

2.2.1 Compute resources

The data model for describing compute resources is, in most cases, common to all types of *executable*, so the data model for these requirements are defined as part of the core Execution Planner data model.

It is important to note that all of the resource requirements are optional. As in the example from the previous section, a request to execute a simple Jupyter notebook does not need to include any resource details.

```
# ExecutionPlanner client request.
# Details of the executable.
executable:
  # A URI identifying the type of executable.
  type: "https://www.purl.org/ivoa.net/executable-types/jupyter-
    notebook"

  # The details, specific to a Jupyter notebook.
  spec:
    notebook: "https://.../example.jpnb"
```

As long as this Jupyter notebook only needs a minimal set of resources to run, e.g. 2 CPU cores, 2G of memory and 20G of disc space, then this task probably doesn't need any additional resources.

However, if this Jupyter notebook needs a specific type of GPU to function properly, then it can be added to the request by specifying a compute resource with the specific type of GPU.

```
# ExecutionPlanner client request.
# Details of the executable.
executable:
  # A URI identifying the type of executable.
  type: "https://www.purl.org/ivoa.net/executable-types/jupyter-
    notebook"
  # The details, specific to a Jupyter notebook.
  spec:
    notebook: "https://.../example.jpnb"

# Details of the resources needed.
resources:
  compute:
  - name: "compute-001"
    type: "https://www.purl.org/ivoa.net/resource-types/generic-
    compute"
    spec:
      extras:
      - name: "nvidia-gpu"
        # A URI identifying the type of GPU.
        type: "https://tinyurl.com/nvidia-ad104"
```

With this detail added to the request, platforms that are not able to provide this kind of GPU would simply reply `NO`.

```
Request  - Can this platform provide a 'NVIDIA AD104 GPU' ?
Response - NO
```

Note that a platform does not need to know what a "NVIDIA AD104 GPU" is to be able to reply with a sensible aswer. If a platform receives a request for a resource that it doesn't understand, it MAY simply reply `NO`.

```
Request  - Can this platform provide <unknown extra> ?
Response - NO
```

The only platforms that will reply `YES` are ones that understand what a "NVIDIA AD104 GPU" is and are able to provide access to one.

The data model for the GPU resource follows the same extendable pattern as the data model for the *executable*. A `type` URI to identify the type of GPU, and a `spec` section for type specific details, e.g. the minimum amount of memory and number of shaders.

```
# ExecutionPlanner client request.
  ....
# Details of the resources needed.
resources:
  compute:
  - name: "compute-001"
    type: "https://www.purl.org/ivoa.net/resource-types/generic-
    compute"
    spec:
      extras:
      - name: "nvidia-gpu"
        # A URI identifying the type.
        type: "https://tinyurl.com/nvidia-ad104"
        # The details, specific to a 'NVIDIA AD104 GPU'.
        spec:
          minmemory: 20
          minshaders: 6144
```

This pattern make it easy for projects to add new types of compute resources to their platforms. All they need to do is to choose a URL to identify the `type` and they can add their own type specific details to the `spec` section.

```
# ExecutionPlanner client request.
  ....
# Details of the resources needed.
resources:
  compute:
  - name: "compute-001"
    type: "https://www.purl.org/ivoa.net/resource-types/generic-
    compute"
```

```
  spec:
    extras:
    - name: "Xilinx FPGA"
      # A URI identifying the type.
      type: "https://www.purl.org/ivoa.net/resource-types/xilinx-
vu19p"
      # The details, specific to a 'Xilinx FPGA'.
      spec:
        logicCells: 8938
        DSPslices:  3840
```

Note that in both these examples, the URL used to identify the `type` uses some level of indirection to make the URL more robust.

At the time of writing, the best resource we could find to describe NVIDIA's AD104 GPU is an entry in a database curated by the TECHPOWERUP website[7]. While this page provides a lot of useful techical detail about the component, the URL itself is vulnerable to changes in the design of the TECHPOWERUP website.

To make this more robust we can use a URL redirect service to create a more permanent URL that we have control over[8]. If the TECHPOWERUP website is redesigned, we can update our redirect to match.

There are several options to choose from:

- A URL shortening service.
  `[https://tinyurl.com/nvidia-ad104]`
- The PURL service[9] provided by the internet archive[10].
  `[https://www.purl.org/ivoa.net/resource-types/nvidia-ad104]`.
- A page in our GitHub repository that directs the reader to more resources[11].
  `[https://github.com/..../resource-types/nvidia-ad104.md]`.

Similarly, the best resource we could find to decribe the Xilinx FPGA provided by Amazon AWS F1[12] instances is the product details page on the Xilinx website[13]. Both of these URLs are informative, but are vulnerable to changes in their websites.

To make this more robust, our example uses a PURL redirect to a page in our GitHub repository that describes the type of FPGA that our project

[7] https://www.techpowerup.com/gpu-specs/nvidia-ad104.g1013
[8] https://tinyurl.com/nvidia-ad104
[9] https://purl.archive.org/
[10] http://www.archive.org/
[11] https://github.com/Zarquan/ExecutionPlanner/blob/main/types/resource-types/nvidia-ad104.md
[12] https://aws.amazon.com/ec2/instance-types/f1/
[13] https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus-vu19p.html

requires. The page in our GitHub repository can point to additional details about the FPGA and the tools needed to use it.

- A PURL redirect to provide a long lasting URL.
  [https://www.purl.org/ivoa.net/resource-types/xilinx-vu19p]
- A page in our GitHub repository describing the FPGA and the tools needed to use it[14].
  [https://github.com/..../resource-types/xilinx-vu19p.md]

2.2.2 Minimum and maximum

The data model for describing compute resources includes elements for specifying the numeric size and number of resources such as CPU cores, memory and storage.

If the Jupyter notebook in our example needs a minimum of 8 CPU cores and 16G of memory to be able to perform its calculations, then this can be specified in the compute resource.

```
# ExecutionPlanner client request.
  ....
# Details of the resources needed.
resources:
  compute:
  - name: "compute-001"
    type: "https://www.purl.org/ivoa.net/resource-types/generic-
    compute"
    spec:
      mincores:   8
      minmemory: 16
```

All of the data model elements for specifying the size or number of resources are defined as pairs of minimum and maximum values. This allows a conversation between the ExecutionPlanner client and services to discover the best platform to execute the task.

The client requests the minimun resources it needs, and each service responds with a set of offers which specify the maximum level of resources they can offer.

For example, if a platform is able to provide double the compute resources, 16 CPU cores and 32G of memory, then it can indicate this by specifying higher maximum values in its response.

```
# ExecutionPlanner service response.
  ....
# Details of the resources offered.
```

[14] https://github.com/Zarquan/ExecutionPlanner/blob/main/types/resource-types/xilinx-vu19p.md

```
resources:
  compute:
  - name: "compute-001"
    type: "https://www.purl.org/ivoa.net/resource-types/generic-
    compute"
    spec:
      mincores:   8
      maxcores:  16
      minmemory: 16
      maxmemory: 32
```

This response represents an offer to start with a minimum of 8 CPU cores and 16G of memory as requested, with the option to use a maximum of 16 CPU cores and 32G of memory if needed.

The client may receive different offers from different platforms and can pass this information on the the user to allow them to choose the offer that best fits their use case. The our example notebook may specify a minimum of 8 CPU cores and 16G of memory, but an offer of twice the resources allows the user more scope for experimenting with more data or more complex algorithms.

This scalable compute resource represents something like a Kubernetes platform where the execution can start with a minimum configuration and scale on demand up to a maximum limit.

This is slightly different to a platform like Openstack which allocates resources in specific blocks, defined by the set of *'flavors'* available on that particular platform. If the smallest flavor of virtual machine available on the platform has 16 CPU cores and 24G of memory, then the service can represent that by setting the minimum values in its offer to match available resources.

```
# ExecutionPlanner service response.
  ....
# Details of the resources offered.
resources:
  compute:
  - name: "compute-001"
    type: "https://www.purl.org/ivoa.net/resource-types/generic-
    compute"
    spec:
      mincores:  16
      maxcores:  16
      minmemory: 24
      maxmemory: 24
```

This response represents an offer to start with a fixed allocation of 16 CPU cores and 24G of memory.

An ExecutionPlanner MAY NOT make an offer with less than the minimum resources requested. For example, if an Openstack platform only has

a virtual machine flavor with 1 CPU core and 2G of memory, then it MAY NOT offer this resource as it is less than the requested minimum.

Note that the term *'compute resource'* specifically avoids stating whether the notebook will be run in an Openstack virtual machine or a Kubernetes cluster. As far as the user is concerned, it should not matter. As long as the compute resource provides sufficient CPU cores and memory for the notebook to execute, the technical details of how the platform implements should not be relevant at this level of abstraction.

The user just wants their notebook to run. The technical details of what platform is used to run it should be someone else's problem.

2.2.3 Storage resources

The resources section of the request can also be used to specify storage resources.

There are two main types of storage resources:

- Internal storage resources that are provided by the platform.
- External storage resources that are provided by an external source.

There are two types of internal storage resources:

- Ephemeral storage available for the duration of the *job*, created when the *job* starts and released when the *job* is completed.
- Persistent storage that exists beyond the lifetime of the *job*, created before the *job* starts and remaining after the *job* has completed.

There are two levels of persistent storage:

- Managed resources that are created and deleted by the platform.
- Unmanaged resources that are created and deleted by an external entity.

The simplest of these are ephemeral storage resources managed by the execution platform. For example, if the Jupyter notebook in our example requires 1024G of space to perform its calculations, then this can be specified in the request by defining an ephemeral storage resource.

```
# ExecutionPlanner client request.
  ....
# Details of the resources needed.
resources:
  ....
  storage:
  - name: "scratch-space"
```

```
    type: "https://www.purl.org/ivoa.net/storage-types/ephemeral-
    storage"
    spec:
      size: 1024
```

To enable the Jupyter notebook to access this storage, we need to add a corresponding `volume` element to the compute resource that describes where to mount the storage resource.

For example, the following request asks for 1024G of ephemeral storage that is mounted at `/temp` in the filesystem of the compute resource. The compute volume is linked to the storage resource by the name of the storage resource.

```
# ExecutionPlanner client request.
  ....
# Details of the resources needed.
resources:
  ....
  compute:
  - name: "compute-001"
    ....
    volumes:
    - name: "temp-volume"
      resource: "scratch-space"
      path: "/temp"
      mode: "rw"
  storage:
  - name: "scratch-space"
    type: "https://www.purl.org/ivoa.net/storage-types/ephemeral-
    storage"
    spec:
      size: 1024
```

This pattern of separating the details of how a storage resource is implemented from the details of how it is mounted inside a computing resource is based on a pattern used by Kubernetes to describe storage volumes and their mount points within containers[15].

This pattern can also be used to define a storage resource that imports data from an external source. For example, if the user wanted to use the Jupyter notebook to analyse data stored in an external S3 system, this can be done by defining an external storage resource that describes where the data is, and a corresponding volume mount inside the compute resource.

```
# ExecutionPlanner client request.
  ....
# Details of the resources needed.
resources:
```

[15] https://kubernetes.io/docs/concepts/storage/volumes/

```
  ....
  compute:
  - name: "compute-001"
    ....
    volumes:
    - name: "data-volume"
      resource: "source-data"
      path: "/data"
      mode: "r"
  storage:
  - name: "source-data"
    type: "https://www.purl.org/ivoa.net/storage-types/amazon-s3"
    spec:
      endpoint: "https://.../echo"
      bucket: "example-data"
```

Again, this pattern of separating how the data is stored outside the system and how it appears inside the compute resource borrows heavily from the pattern used by Kubernetes to describe persistent volumes[16].

The same pattern can be used to describe a storage resource that can be used to save the analysis results, by defining a persistent storage resource allocated by the system, and a corresponding volume mount inside the compute resource.

```
# ExecutionPlanner client request.
  ....
# Details of the resources needed.
resources:
  ....
  compute:
  - name: "compute-001"
    ....
    volumes:
    - name: "results-volume"
      resource: "results-storage"
      path: "/results"
      mode: "rw"
  storage:
  - name: "results-storage"
    type: "https://www.purl.org/ivoa.net/storage-types/persistent-
    storage"
    spec:
      lifecycle: "managed"
      minlifetime: "1D"
      minsize: 100
```

By setting the storage resource type to `https://.../persistent`, and setting the `lifecycle` to `managed`, the client is asking the ExecutionPlanner

[16] https://kubernetes.io/docs/concepts/storage/persistent-volumes/

service to take care of allocating the storage and managing its lifecycle. It is up to the ExecutionPlanner service to decide where to store the data and how make it accessible after the *job* has completed.

For example, a science platform may have a Rucio storage system co-located with the compute platform which it uses to store user generated data. In which case the ExecutionPlanner service would respond with an offer that stores the results in the Rucio system and provides details of how the user can access it after the *job* has completed.

```
# ExecutionPlanner service response.
  ....
# Details of the resources offered.
resources:
  ....
  compute:
  - name: "compute-001"
    ....
    volumes:
    - name: "results-volume"
      resource: "results-storage"
      path: "/results"
      mode: "rw"
  storage:
  - name: "results-storage"
    type: "https://www.purl.org/ivoa.net/storage-types/rucio-storage"
    spec:
      lifecycle: "managed"
      minlifetime: "1D"
      maxlifetime: "5D"
      minsize:  100
      maxsize:  200
      endpoint: "http://...."
      domain: "Project 51"
      objectid: "cdc78e7d-8032-497e-9a5b-01c720ea2223"
```

In this example, the client requested at least 100G of storage available for 1 day and in response the service is offering up to 220G available for 5 days stored in a Rucio system co-located close to the compute platform. It is up to the client to check if it can access the particular Rucio system described in the response before it accepts the offer.

Making an offer with the `lifecycle` set to `managed` and the `maxlifetime` set to `5D` means that the service will manage the lifecycle. The storage will be available for 5 days after the *job* completes and then it will be deleted automatically. The client doesn't need to worry about tidying up afterwards.

It is important to note that at this point in time the storage is proposed, but not yet allocated. The persistent storage is only allocated if the client accepts this particular offer. This allows an ExecutionPlanner service to

make multiple offers with different storage options, allowing the client to select and accept the one that best fits its use case.

The same data model can be used the other way around as well. If the client already knows where it wants the data to be stored, for example at a specific VOSpace location, then it can specify this in the request.

```
# ExecutionPlanner client request.
  ....
# Details of the resources needed.
resources:
  ....
  compute:
  - name: "compute-001"
    ....
    volumes:
    - name: "results"
      path: "/results"
      mode: "rw"
  storage:
  - name: "results"
    type: "https://www.purl.org/ivoa.net/storage-types/vospace-storage
    "
    spec:
      endpoint: "http://...."
      path: "/experiment-21/results"
      lifecycle: "unmanaged"
```

It is up to the ExecutionPlanner service to work out if it is able to access the VOSpace location and mount it as a volume in the compute resource, using either its own authentication, or a delegated form of the authentication provided by the client.

If it can access the VOSpace location, then the ExecutionPlanner MAY respond with an offer, otherwise if it can't access the VOSpace location for whatever reason the ExecutionPlanner MUST respond with `NO`.

Note that in this example, the client has specified the lifecycle as `unmanaged`, which means that the ExecutionPlanner is not involved in managing the creation or deletion of the data in VOSpace. It is also possible for the client to ask the ExecutionPlanner service to manage data in an external resource.

```
# ExecutionPlanner client request.
  ....
# Details of the resources needed.
resources:
  ....
  storage:
  - name: "results"
    type: "https://www.purl.org/ivoa.net/storage-types/vospace-storage
    "
    spec:
```

```
      endpoint: "http://...."
      type: "container"
      path: "/experiment-21/results"
      lifecycle: "managed"
      maxlifetime: "2D"
```

In this example, the client is specifying an external VOSpace location to store the data, but it is asking the ExecutionPlanner service to manage the lifecycle, creating the location in VOSpace at the start of the *job* and deleting it 2 days after the *job* completes.

This negotiation of who is responsible for creating and deleting storage resources enables a client to put together a workflow of interconnected steps, with the ExecutionPlanner services managing the lifecycle of the resources and releasing them automatically after they are no longer needed.

2.3 Authentication

The client may also want to check whether the user account making the request has sufficient access rights and resource quota needed to execute a task.

This is equivalent to asking: *"Do I have permission to use these resources to run this Jupyter notebook?"*

There are two ways for the ExecutionPlanner service to check the user's identity, using implicit and explicit authentication.

The implicit method is for the client to authenticate to the ExecutionPlanner service as normal and the service will use the identity from the request headers to check if the user has sufficient access rights and resource quota to execute the *job*.

If the webservice call to the ExecutionPlanner service is authenticated using OpenIDConnect (OIDC)[17], then the ExecutionPlanner MUST include a reference to the implicit authentication in its reply, including enough non-secret information to identify the authenticated identity.

```
# ExecutionPlanner server response.
....
authentication:
- name: "http-request"
  type: "https://.../oidc"
  mode: "implicit"
  spec:
    subject: "user@domain"
```

If the client uses an unknown authentication method in the webservice call and the ExecutionPlanner service recognises that an anthentication attempt has been made, then it MUST reject the webservice call with a 401

[17] https://auth0.com/docs/authenticate/protocols/openid-connect-protocol

"Unauthenticated" response. If the ExecutionPlanner service does not realise that an anthentication attempt has been made, then this MAY result in the request being treated as an anonymous unauthenticated call.

The data model also allows for an explicit statement of identity in the request. The data model for authentication follows the same pattern as previous sections, defining a `type` URI to identify the type of authentication method, and a `spec` section for the type specific details.

For example, to explicitly include a basic authentication with username and password in the request:

```
# ExecutionPlanner client request.
....
authentication:
- name: "basic-auth"
  type: "https://.../basic"
  spec:
    username: "..."
    password: "..."
```

This pattern allows the client to supply multiple authentication methods in the request. The ExecutionPlanner service SHOULD select the authentication methods it accepts from those in the request and include the name and type of the methods in its reply along with enough non-secret information to identify the authenticated identity.

For example, if the client supplies both basic and token authentication in the request:

```
# ExecutionPlanner client request.
....
authentication:
- name: "basic-auth"
  type: "https://.../basic"
  mode: "explicit"
  spec:
    username: "..."
    password: "..."
- name: "token-auth"
  type: "https://.../token"
  mode: "explicit"
  spec:
    subject: "user@domain"
    token: "..."
```

If the ExecutionPlanner service only accepts the token authentication, it MUST skip the basic authentication and only include the name and type of the token based authentication in its reply.

```
# ExecutionPlanner server response.
....
```

```
authentication:
- name: "token-auth"
  type: "https://.../rfc7519"
  mode: "explicit"
  spec:
    subject: "user@domain"
```

If the ExecutionPlanner service does not recognise or support any of the authentication methods included in the request, then the service MUST reject the request and reply with `NO`.

```
Request  - Can I authenticate with <unknown method, unknown method, or
     unknown method> ?
Response - NO
```

The result is that an offer made by an ExecutionPlanner service SHOULD include details of all the authenticated identities that are allowed to use the offer.

If the original question is equivalent to: *"Do **these identities** have sufficient access rights and quota to run this Jupyter notebook?"*

Then the response from the ExecutionPlanner service is equivalent to: *"This offer asserts that **these identities** have sufficient access rights and quota to run this Jupyter notebook?"*

The client MUST use one of the authentication methods listed in the offer when contacting the ExecutionPlanner service to accept the offer and start the *job*. The client MUST continue to use the same authentication method when making subsequent requests to the ExecutionWorker service to access the *job* status and results.

The Execution Planner specification includes an initial set of authentication methods corresponding to the methods defined in the IVOA Single-Sign-On standard(Taffoni and Schaaf et al., 2017)

- ....
- ....

However, the data model also allows an ExecutionPlanner service to accept authentication methods that are not covered by the IVOA SSO standard. A client is free to use any authentication method, including ones not covered by the IVOA SSO standard. It is up to the ExecutionPlanner service to decide how it handles the authentication infomation it receives.

This means that the Execution Planner services can be deployed in other domains outside the IVOA, without requiring them to adopt the full IVOA SSO standard.

2.4 Date and time

The `datetime` part of the data model enables the client and server to have a conversation about when a *job* can be executed.

The client can specify one or more time periods when it would like to start the *job*, and the minimum duration that it thinks would be needed to complete the *job*.

The ExecutionPlanner service may respond with one or more offers that specify when the *job* would start and the maximum duration that the *job* would be allowed to consume. It is then up to the client to select which of the offers best suits its use case.

The start time for a *job* is expressed as an array of time intervals, as defined by ISO 8601 (International Organization for Standardization, 2004). Specifically, type 1 or type 2 intervals (start/end and start/duration), excluding type 3 and 4 intervals (duration/end and duration only) and excluding repeats[18][19].

Note that the duration part of the interval applies to the start time, specifying a time range during which the *job* may start.

If no duration is specified, this means an absolute start time; e.g. the *job* SHOULD start at 11:30 on 14 August.

```
# ExecutionPlanner client request.
....
datetime:
- start: "2023-08-14T11:30Z"
```

If the start and end are specified, this means the *job* SHOULD start somewhere between the start and end values; e.g. the *job* SHOULD start between 11:30 and 12:00 on 14 August.

```
# ExecutionPlanner client request.
....
datetime:
- start: "2023-08-14T11:30Z/T12:00Z"
```

If a duration is specified, this means the *job* SHOULD start somewhere between start and the start plus duration; e.g. the *job* SHOULD start between 11:30 and 12:00 on 14 August.

```
# ExecutionPlanner client request.
....
datetime:
- start: "2023-08-14T11:30Z/PT30M"
```

The ExecutionPlanner service SHOULD respond with one or more offers that start within the ranges specified in the request. The start times in the

[18] https://www.iso.org/iso-8601-date-and-time-format.html
[19] https://en.wikipedia.org/wiki/ISO_8601

offers MAY be more precise than the start times in the request, but they MUST all occur within at least one of the ranges specified in the request.

The client MAY also specify the maximum and minimum execution duration, expressed as time periods as defined by ISO 8601.

For example, for the unattended batch mode execution of an OCI container, the user might not be concerned about when their *job* starts, but they may want to specify the minimum duration needed to complete the task.

In which case, the client may simply request a minimum duration of 1 hour.

```
# ExecutionPlanner client request.
....
datetime:
- minduration: "1H"
```

The ExecutionPlanner service MAY respond with offers that start at different times and set different values for the maximum duration.

It MAY offer a maximum duration of 1 hour in the morning, starting at 11:00.

```
# ExecutionPlanner server response.
....
datetime:
- start: "2023-08-14T11:30Z"
  minduration: "1H"
  maxduration: "1H"
```

It MAY also offer a longer allocation of 4 hours later in the evening, starting sometime between 22:00 and 23:00.

```
# ExecutionPlanner server response.
....
datetime:
- start: "2023-08-14T22:00Z/PT1H"
  minduration: "P1H"
  maxduration: "P4H"
```

Different values for start time and duration can be combined with different values for the compute resources to make a range of different offers to the client.

For example, if a client asks for 2 cores and 2G of memory for 1 hour sometime on 14 August:

```
# ExecutionPlanner client request.
  ...
resources:
  compute:
    mincores: 2
    minmemory: 2
datetime:
```

```
  - start: "2023-08-14/P1D"
    minduration: "P1H"
```

The ExecutionPlanner service may respond with 2 offers, the minimum 2 cores and 2G of memory for 1 hour starting at 11:30, and a larger offer of up to 8 cores and 8Gb of memory for up to 4 hours starting between 22:00 and 23:00.

```
# ExecutionPlanner server response.
offers:
- name: "offer-001"
  executable:
    ...
  resources:
    compute:
      mincores:  2
      maxcores:  2
      minmemory: 2
      maxmemory: 2
  datetime:
    - start: "2023-08-14T11:30Z"
      minduration: "P1H"
      maxduration: "P1H"

- name: "offer-002"
  executable:
    ...
  resources:
    compute:
      mincores:  2
      maxcores:  8
      minmemory: 2
      maxmemory: 8
  datetime:
    - start: "2023-08-14T22:00Z/T23:00Z"
      minduration: "P1H"
      maxduration: "P4H"
```

It is then up to the client to decide which offer better suits their use case. Accept the limited offer in the morning, or accept the more generous offer with more resources and more time later in the day.

If an ExecutionPlanner service is offering more than one option for the `datetime` section it MUST make separate offers for each different option. Even if all of the other parameters are the same, e.g. compute and storage resources, the ExecutionPlanner service MUST NOT include more than one time slot in the same offer. Technically the data modelallows an array of values for the `datetime` section, but this would impose unnecessary complexity on the client for no real gain in user experience.

2.5 Triggers and callouts

The final part of the data model enables the client to set up webservice API calls both to and from the ExecutionPlanner service that can be used to control the state of a *job*.

2.5.1 Actions and triggers

The `triggers` section of the data model allows the client to set up webservice endpoints that will trigger actions that control the state of a *job* in an ExecutionWorkerservice.

The simplest of these webservice calls is a `value-update` endpoint. This sets up a `POST` webservice endpoint that a remote system can call with a set of name-value pairs.

```
POST
name1: value1
name2: value2
name3: value3
```

The following example asks the ExecutionPlanner service to set up a webservice endpoint that will trigger an action depending on the value of the `status` element in the `POST` data it receives.

```
# ExecutionPlanner client request.
....
triggers:
- name: "trigger-001"
  type: "https://www.purl.org/ivoa.net/webservice-types/value-update"
  spec:
    method: "POST"
    content-type: "yaml"
    conditions:
    - name:  "status"
      value: "COMPLETED"
      action: "start"
    - name:  "status"
      value: "FAILED,CANCELLED"
      action: "cancel"
```

Which action taken is defined in the `conditions` section of the `trigger`. In this case the action depends on the value of the `status` field in the `POST` data.

- If the value of `status` is `COMPLETED`, then start this *job*.
- If the value of `status` is `FAILED` or `CANCELLED`, then cancel this *job*.

Setting up a trigger to start a *job* if the value of a `status` field is `COMPLETED` may sound counter-intuitive, but the typical use case for triggers like this is to control this *job* based on the result of a *job* in an upstream service that is performing the step before this in a workflow. In which case, we want this *job* to wait until the previous step has completed before starting. Hence the trigger that waits until the `status` of the **previous** step is `COMPLETED` before starting this step.

Defining a `start` action in the `triggers` section will modify the effect of the start time declared in the `datetime` section. The ExecutionPlanner service will not automatically start the *job* when the specified start time is reached. Instead the ExecutionPlanner service will wait until the start action has been triggered before it starts the *job*.

- If the trigger is called before the specified start time is reached, the event is recorded, but no action is taken. The ExecutionPlanner service will wait until the start time is reached before starting the *job*.
- If the start time is reached before the trigger has been called, the event is recorded, but no action is taken. The ExecutionPlanner service will wait until the trigger is called before starting the *job*.
- If the trigger is called within the start time range, the ExecutionPlanner service will start *job*.
- If the start time range expires before the trigger has been called, the ExecutionPlanner service will cancel the *job*.
- If the trigger is called after the start time range, it has no effect. The *job* should already have been cancelled.

In effect, the start time range acts as a constraint on the action of the trigger. Preventing it from starting the *job* too early or too late. As far as resource allocation is concerned, the ExecutionPlanner service allocates the required compute and storage resources from the beginning of the start time range. This means the *job* is ready to start as soon as the trigger is called. If the start time range expires before the trigger has been called, the resources are released as normal when the *job* is cancelled.

2.5.2 Remote callouts

The `callouts` part of the data model forms the other half of a callout-trigger pair, enabling a client to set up webservice API calls to remote services that the ExecutionPlanner service will call at specific points in the lifecycle of a *job*.

The following definition asks the ExecutionPlanner service to `POST` the name-value list defined in the `content` template to a remote webservice endpoint when the status of this *job* changes to be one of `RUNNING`, `COMPLETED`, `FAILED` or `CANCELLED`.

```
....
callouts:
- name: "callout-001"
  type: "https://www.purl.org/ivoa.net/webservice-types/value-update"
  spec:
    triggers:
    - state: "RUNNING,COMPLETED,FAILED,CANCELLED"
    method: "POST"
    endpoint: "http://ws.example.org/update"
    content-type: "yaml
    content: |
      name:  {{job.name}}
      date:  {{system.date}}
      ident: {{job.ident}}
      status: {{job.status}}
```

When the status of this *job* changes to be one of the specified values, the ExecutionPlanner service will `POST` the document described in the `content` template to the HTTP `endpoint`, filling in the `...` markers in the template with values from the data model representing the *job* at that point in time.

In this example, the `...` markers in the `content` template will be filled in with the current values of the *job* `name`, `ident` and `status`, along with the current local system time on the ExecutionPlanner service sending the callout.

In a normal execution sequence, this callout would be called twice. Once when the *job* `status` is set to `RUNNING`, and again when the *job* `status` is set to `COMPLETED`.

2.5.3 Linked worflow

The following example describes how the callout and trigger functions can be used to set up a 2 step workflow containing step-a and step-b, with step-b waiting until step-a has been completed.

The first part of setting up the workflow is to configure the second step, step-b, with a trigger that will wait until it receives a `value-update` webservice call with the value of `status` set to `COMPLETED` before starting the *job*.

```
name: "step-b"
executable:
  ...
resources:
  ...
```

```
triggers:
- name: "trigger-001"
  type: "https://www.purl.org/ivoa.net/webservice-types/value-update"
  spec:
    method: "POST"
    content-type: "yaml"
    conditions:
    - name:  "status"
      value: "COMPLETED"
      action: start
    - name:  "status"
      value: "FAILED,CANCELLED"
      action: cancel
```

Note that the client doesn't set the endpoint location for the trigger, that needs to come from the ExecutionPlanner service. The webservice endpoint location for each trigger may be different for each of the offers in the ExecutionPlanner service response.

```
....
offers:
- name: "offer-001"
  ....
  triggers:
  - name: "trigger-001"
    type: "https://www.purl.org/ivoa.net/webservice-types/value-update
    "
    spec:
      method: "POST"
      content-type: "yaml"
      endpoint: "https://..../offer-001/trigger-001"
      ....
```

The second part of setting up the workflow is to configure step-a with a `value-update` webservice callout to the endpoint address from step-b when the status of step-a changes.

```
name: "step-a"
executable:
  ...
resources:
  ...
callouts:
- name: "callout-001"
  type: "https://www.purl.org/ivoa.net/webservice-types/value-update"
  spec:
    triggers:
    - status: "RUNNING,COMPLETED,FAILED,CANCELLED"
    method: "POST"
    endpoint: "https://..../offer-001/trigger-001"
    content-type: "yaml
```

```
    content: |
      name:  {{job.name}}
      date:  {{system.date}}
      ident: {{job.ident}}
      status: {{job.status}}
```

The start times on both step-a and step-b can be set to a range that starts today and lasts for a day. This will ensure that even if the triggers don't get called, and neither *job* is executed, both *job*s will be cancelled and the resources released when the start time range expires at the end of the day.

```
name: "step-a"
executable:
  ...
resources:
  ...
datetime:
  - start: "2023-08-14/P1D"
```

The client can also set up a `managed` storage resource in VOSpace to transfer data between the steps which is automatically created and deleted by the ExecutionPlanner service.

Adding a `managed` storage resource to step-a which points a location in a VOSpace service with a `maxlifetime` and `minlifetime` set to 1 day means that the VOSpace location will be automatically created when step-a starts, and automatically deleted 1 day after step-a completes.

This gives step-b enough time to collect the results but also ensures that the storage resource is eventually released after a day.

```
name: "step-a"
executable:
  ...
resources:
  ....
  storage:
  - name: "result-storage"
    type: "https://www.purl.org/ivoa.net/storage-types/vospace-storage
    "
    spec:
      endpoint: "http://...."
      path: "/step-a/results"
      lifecycle: "managed"
      minlifetime: "1D"
      maxlifetime: "1D"
```

The data in this location can be made available to step-b using an `unmanaged` resource that points to the same location in VOSpace.

```
name: "step-b"
```

```
executable:
  ...
resources:
  ....
  storage:
  - name: "input-storage"
    type: "https://www.purl.org/ivoa.net/storage-types/vospace-storage
    "
    spec:
      endpoint: "http://...."
      path: "/step-a/results"
      lifecycle: "unmanaged"
```

If all goes to plan, when the start time for step-a is reached, the first ExecutionPlanner service will allocate the managed space in VOSpace, and then start the execution of step-a. As soon as step-a starts, the *job* status changes to `RUNNING`, which triggers a callout from this ExecutionPlanner service to the second ExecutionPlanner service running step-b. This first callout will have no effect, as the trigger for step-b is configured to only perform an action on a `status` value of `COMPLETED`, `FAILED` or `CANCELLED`.

The executable in step-a can access the storage location using an internal volume mount that refers to the external storage location.

```
name: "step-a"
executable:
  ...
resources:
  ....
  compute:
  - name: "compute-001"
    ....
    volumes:
    - name: "results-volume"
      resource: "results-storage"
      path: "/results"
      mode: "rw"
  ....
  storage:
  - name: "results-storage"
    type: "https://www.purl.org/ivoa.net/storage-types/vospace-storage
    "
    spec:
      endpoint: "http://...."
      path: "/step-a/results"
      lifecycle: "managed"
      minlifetime: "1D"
      maxlifetime: "1D"
```

As far as the code inside the executable is concerned, it writes its results to a filesystem directory at `/results`. The code inside the executable does

not need to know anything about the details of where the data is stored. The ExecutionPlanner service is responsible for ensuring that anything written to `/results` in the local filesystem is transferred to `/step-a/results` in the remote VOSpace service.

When step-a completes, its status is updated to `COMPLETED`, which triggers another callout to the ExecutionPlanner service running step-b. This time, the `status` value of `COMPLETED` triggers an action to start step-b.

Although step-a has completed, the managed space in VOSpace is not deleted. The managed resource has a `minlifetime` of `1D`, which means that the space will not be deleted until a day after step-a has completed. This allows sufficient time for step-b to execute, reading its input data from the results of step-a. The second ExecutionPlanner service runs step-b to completion as normal, freeing up any resources it was allocated at the end.

As with step-a, step-b is configured to mount the same VOSpace location as a filesystem directory.

```
name: "step-b"
executable:
  ...
resources:
  ....
  compute:
  - name: "compute-001"
    ....
    volumes:
    - name: "input-volume"
      resource: "input-storage"
      path: "/inputs"
      mode: "r"
  ....
  storage:
  - name: "input-storage"
    type: "https://www.purl.org/ivoa.net/storage-types/vospace-storage
    "
    spec:
      endpoint: "http://...."
      path: "/step-a/results"
      lifecycle: "unmanaged"
```

As far as the code inside the executable is concerned, it reads its input data from a filesystem directory at `/inputs`. The code inside the executable does not need to know anything about the details of where the data is stored.

Finally, the `managed` VOSpace storage location is automatically deleted a day after the completion of step-a by the ExecutionPlanner service that was assigned to manage its lifecycle.

3 Separation of concerns

Separate who knows what .. The players:

- The researcher who creates the notebook
- The developer who creates the container
- The publisher who publishes the data
- The user who is running the analysis

4 Request and response

Details of the request and response messages. Timeline of request, offer and accept.

4.1 ExecutionPlanner

Simple `POST` REST webservice. Single parameter, the *job* description.

Response contains a simple `YES` or `NO` and a list of offers for how the *job* can be executed on this platform.

4.2 ExecutionWorker

Derived from IVOA UWS, using a similar data model.

Accepting an offer on an ExecutionPlanner creates a job on the associated ExecutionWorker, with status PENDING until the *job* is started automatically at the designated time.

The offer from the ExecutionPlanner will contain the URL *job* in the ExecutionWorker.

- ExecutionPlanner is responsible for creating and starting the *job*.
- Client can use a HEAD request to check network access at anytime.
- Client can use a GET request to check the status of the *job* at anytime.
- Client can cancel the *job* at anytime.

5 General requirements

- ....
- ....
- ....

6 Federated architecture

The ExecutionPlanner and ExecutionWorker services are designed to be used at multiple levels within an organization.

At the low level, an ExecutionPlanner and ExecutionWorker pair may be implemented as a single web-application linked to a simple Docker execution service. The configuration may be hard coded to only accept a fixed white list of container images and a fixed allocation of compute resources for each job. e.g. the platform will only run a fixed set of container images, and only provides 2 CPU cores, 2G of memory and a maximum 1HR execution time.

A project or organization may deploy several of these low level services, providing a range of different capabilities.

Given a new *job* to execute, a client can simply poll each of the services to see which one will make an offer to execute it.

The client does not need to have any prior knowledge about the services apart from their endpoint address. This information could come from the IVOA Registry, or it could simply be provided as a flat list in a configuration file for the client.

A more flexible architecture could add another ExecutionPlanner service a level above the low level task specific services. This service would be configured with a list of the local task specific services and act as an aggregator service sitting between the client and the low level services. This aggregator service would forward a copy of each request it receives to each of the low level services in its list and then aggregate the offers from the individual responses into a single response that is sent back to the client.

A simple aggregator service would just forward every request to all the low level services below it, regardless of what the request contained.

A more complex aggregator service may have some prior knowledge about what types of task or compute resources each of the lower level services were able to offer, enabling it to make more informed decisions about which low level serviecs to send the requests to.

The aggregator service may also have an understanding of the location of datasets within the organization and be able to route requests to different low level services depending on which datasets were required.

The aggregator service may expose the low level ExecutionWorker endpoints in its responses, or it may implement a proxy interface acting as an aggregator for the ExecutionWorker services as well.

This configuration could be used to provide ExecutionPlanner and ExecutionWorker service interfaces that bridged a firewall. Providing a public interface for external clients and forwarding the requests to the internal ExecutionPlanner and ExecutionWorker services that are not accessible from outside the firewall.

A large organization with multiple sites couls also deploy a single high level ExecutionPlanner and ExecutionWorker interface that handles task execution for the whole organization, forwarding the requests to mid-level ExecutionPlanner and ExecutionWorker services at each site which in turn forwards the requests to individual low-level ExecutionPlanner and ExecutionWorker services within the local site networks.

The implementation at each level may be different, providing different levels of routing and aggregation based on internal knowledge of the capabilities of the level below, but the ExecutionPlanner and ExecutionWorker interfaces would be the same at each level in the organization.

This could be expanded to include another level of ExecutionPlanner and ExecutionWorker services that cross organization bondaries, providing a gateway that allows users from one project to access services from other projects and organizations. This gateway service may also provide an authentication translation layer between external public accounts internal project specific identities and authentication methods.

Federated cloud diagram ....

- Project - SKA
  - Project gateway
  - Regional centres
  - Local data centres
  - Low level services
    - Slurm batch services
    - Docker container services
    - Jupter notebook services
- Project - LSST
  - Project gateway
  - Regional centres
  - Local data centres
  - Low level services

        * Slurm batch services
        * Docker container services
        * Jupter notebook services

- Project - CTA
    - Project gateway
    - Regional centres
    - Local data centres
    - Low level services
        * Slurm batch services
        * Docker container services
        * Jupter notebook services

- Project - CERN
    - Project gateway
    - Regional centres
    - Local data centres
    - Low level services
        * Slurm batch services
        * Docker container services
        * Jupter notebook services

7 Example use cases

This section contains a set of examples ranging from very simple to complex, with a complete listing of the data model used to describe each case.

Before we get into some of the more complex examples it is worth reiterating that all of the elements in the data model are optional, so it is perfectly legitimate for a simple implementation to only accept one of the simplest examples with no extras, and answer `NO` to everything else.

7.1 Simple notebook

Just a simple Jupyter notebook. ...

7.2 Notebook with dates

A Jupyter notebook with specific date ranges for start time. ...

7.3 Notebook with data

A Jupyter notebook with specific input data. ...

7.4 Notebook with compute

A Jupyter notebook with specific input data and compute resources. ...

7.5 Simple container

A simple OCI container. ...

7.6 Container with data

An OCI container with specific input data. ...

7.7 Container with compute

An OCI container with specific input and output data, and compute resources. ...

7.8 Container triggering notebook

An OCI container that notifies the user and launches a Jupyter notebook session when the container finishes executing. ...

7.9 Kubernetes Helm chart

A complex Helm chart with multiple elements launched as a single task. ...

7.10 Spark cluster

A complex Spark cluster analysis launched as a single task. ...

A Changes from Previous Versions

No previous versions yet.

A Resource type URLs

https://www.purl.org/ivoa.net/executable-types/example https://github.com/Zarquan/ExecutionPla
types/example-executable.md

https://www.purl.org/ivoa.net/executable-types/jupyter-notebook https://github.com/Zarquan/
types/jupyter-notebook.md

https://www.purl.org/ivoa.net/executable-types/oci-container https://github.com/Zarquan/Exec
types/oci-container.md

https://www.purl.org/ivoa.net/resource-types/generic-compute https://github.com/Zarquan/Exe
types/generic-compute.md

https://www.purl.org/ivoa.net/resource-types/nvidia-ad104 https://github.com/Zarquan/Execut
types/nvidia-ad104.md

https://www.purl.org/ivoa.net/resource-types/xilinx-vu19p https://github.com/Zarquan/Executi
types/xilinx-vu19p.md

https://www.purl.org/ivoa.net/storage-types/ephemeral-storage https://github.com/Zarquan/Ex
types/ephemeral-storage.md

https://www.purl.org/ivoa.net/storage-types/amazon-s3 https://github.com/Zarquan/Executionl
types/amazon-s3.md

https://www.purl.org/ivoa.net/storage-types/persistent-storage https://github.com/Zarquan/Exe
types/persistent-storage.md

https://www.purl.org/ivoa.net/storage-types/rucio-storage https://github.com/Zarquan/Executio
types/rucio-storage.md

https://www.purl.org/ivoa.net/storage-types/vospace-storage https://github.com/Zarquan/Execu
types/vospace-storage.md

https://www.purl.org/ivoa.net/authentication-types/basic-auth https://github.com/Zarquan/Exe
types/basic-auth.md

https://www.purl.org/ivoa.net/authentication-types/oidc-auth https://github.com/Zarquan/Exec
types/oidc-auth.md

https://www.purl.org/ivoa.net/authentication-types/rfc7519-token https://github.com/Zarquan/
types/rfc7519-token.md

https://www.purl.org/ivoa.net/webservice-types/value-update https://github.com/Zarquan/Exec
types/value-update.md

References

Arviset, C., Gaudet, S. and IVOA Technical Coordination Group (2010), 'IVOA Architecture Version 1.0', IVOA Note 23 November 2010. http://doi.org/10.5479/ADS/bib/2010ivoa.rept.1123A

Benson, K., Plante, R., Auden, E., Graham, M., Greene, G., Hill, M., Linde, T., Morris, D., O'Mullane, W., Rixon, G., Stébé, A. and Andrews, K. (2009), 'IVOA Registry Interfaces Version 1.0', IVOA Recommendation

04 November 2009, arXiv:1110.0513.
http://doi.org/10.5479/ADS/bib/2009ivoa.spec.1104B

Bradner, S. (1997), 'Key words for use in RFCs to indicate requirement levels', RFC 2119.
http://www.ietf.org/rfc/rfc2119.txt

Graham, M., Morris, D. and Rixon, G. (2009), 'VOSpace specification Version 1.15', IVOA Recommendation 07 October 2009.
http://doi.org/10.5479/ADS/bib/2009ivoa.specQ1007G

International Organization for Standardization (2004), 'Data elements and interchange formats – information interchange – representation of dates and times'.
http://www.iso.org/iso/catalogue_detail?csnumber=40874

Plante, R., Graham, M., Rixon, G. and Taffoni, G. (2010), 'IVOA Credential Delegation Protocol Version 1.0', IVOA Recommendation 18 February 2010, arXiv:1110.0509.
http://doi.org/10.5479/ADS/bib/2010ivoa.spec.0218P

Taffoni, G., Schaaf, A., Rixon, G. and Major, B. (2017), 'SSO - Single-Sign-On Profile: Authentication Mechanisms Version 2.0', IVOA Recommendation 24 May 2017, arXiv:1709.00171.
http://doi.org/10.5479/ADS/bib/2017ivoa.spec.0524T