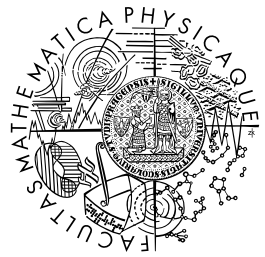

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Roman Vašut

Distributed job execution in IVIS Framework

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis:	prof. RNDr. Tomáš Bureš, Ph.D.
Study programme:	Computer Science
Study branch:	Programming and software development

Prague 2023

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank my supervisor, prof. RNDr. Tomáš Bureš, Ph.D., for his guidance and time. I would also like to thank my family for the support they have given me throughout my studies and beyond. Finally, I thank my friends and colleagues for their emotional and intellectual support.

Title: Distributed job execution in IVIS Framework

Author: Roman Vašut

Department: Department of Distributed and Dependable Systems

Supervisor: prof. RNDr. Tomáš Bureš, Ph.D., Department of Distributed and Dependable Systems

Abstract: This thesis tackles computation distribution in the IVIS data processing and visualization framework. In the existing versions, so-called Jobs are being executed only on the IVIS host machine, raising scalability concerns. The thesis attempts to allow the distributed execution on manually-provisioned machines, commercial cloud platforms, and an HPC cluster. It does so by introducing the ”executor” entity, ensuring adherence to the present Job architecture and, because the communication is done over the Internet, security. We introduce two auxiliary applications which manage the remote control of a machine and the management of a set of machines (a pool). We achieve parallelization of running Jobs. We also see the possibility of further extension to enable the usage of specialized hardware or more dynamic machine allocation.

Keywords: javascript, data processing, distributed computing, cloud

Contents

Chapter 1

Introduction

IVIS is a data processing and visualization framework and a client-server web application written in JavaScript. The framework allows its users to store and manage user-defined data collections and create visualizations of such data. Examples of visualization may include charts accessible via the application client. The framework also allows the execution of custom scripts that operate on the data. The entire lifecycle of these scripts, from saving the code, building the environment in which the scripts run to scheduling and executing the scripts, is managed by the IVIS server. In its current implementation, all scripts (otherwise known as jobs) are executed on the same machine as the IVIS server. This Job execution model may pose performance and scalability issues for larger or more resource-intensive framework applications.

This thesis extends the framework with the option to perform this computation on Internet-connected machines with Docker support. More abstractly, this thesis implements a way to execute the jobs on so-called "job executors". Concrete "job executor" types implemented in this thesis include:

- any machine with Docker support and Internet connection
- Slurm-based HPC cluster
- a fixed-sized pool of automatically managed virtual machine instances via the Oracle Cloud Infrastructure

The implementation defines an interface based on HTTP requests which serve as the primary communication protocol between a "job executor" and the IVIS server. On the IVIS server side, this interface exposes sensitive systems whose integrity is vital to the IVIS server's main functionality. That, among other security reasons, is why every client and server uses SSL certificates for authentication.

Two auxiliary Docker applications are implemented: the Remote Job Runner (RJR) and the Remote Pool Scheduler (RPS), both of which adhere to the same HTTP interface defined above. RJR encapsulates job environment building and running jobs. RPS serves as a proxy between multiple RJR instances and the IVIS server in a pool configuration. RPS's most important responsibilities are authentication (in the direction from an RJR to the IVIS server) and request delegation and scheduling (in the opposite direction).

Additionally, the Oracle Cloud Infrastructure pool implementation provides the means to control the pool resources via the existing IVIS UI.

The Slurm-based implementation utilizes the `slurm` scheduling program hosted on the HPC cluster's frontend node for job scheduling and failsafe output storage. Every operation is performed using a set of scripts that may schedule other scripts to be executed or interact with the IVIS server to adhere to the defined HTTP interface. Various other adaptations of existing tools and processes were made to ensure compatibility with the IVIS server's requirements.

The solution presented in this thesis achieves job execution distribution to the executors listed above. The distributed execution of jobs directly improves job throughput and improves hardware flexibility.

1.1 Thesis structure

The thesis is structured as follows. Chapter 2 provides a more detailed description of the current implementation and introduces key concepts that permeate the entirety of the IVIS application. In particular, it focuses on the existing architecture of job execution and the interactions expected by the IVIS client and server. It also introduces Docker, the containerization tool used in this thesis, the TLS protocol properties, Oracle Cloud Infrastructure, and SLURM.

Chapter 3 creates a precise image of the problem. It provides example use cases and the rationale behind the architectural decisions made. It also elaborates on the implemented job executor types.

Chapter 4 introduces the solution following the problem description of the third chapter. Chapter 4 elaborates on the lower-level IVIS server implementation and introduces key changes and additions to support the architecture described in chapter 3. We then move on to more detailed descriptions of the executor implementations.

In chapter 5, we compare the parallel execution of Runs on different machines with parallel execution on only the IVIS server to provide evidence that the extension provides a throughput boost to the IVIS server.

Appendix A provides more detailed information about using the IVIS server and accompanying software developed as part of the solution.

Appendix B shows use cases and screenshots of the IVIS Framework as visual material for the introductory chapter 2. It also contains screenshots of the newly integrated UI elements introduced by the solution.

Chapter 2

Background

IVIS framework allows data storage and processing with emphasis on the ability of the user to customize both the data formats and the computation. The framework is a client-server web application that offers data processing and visualizations. The user defines data processing in an administration interface. The administration interface also serves for the configuration and management of all IVIS-specific entities. Such entities are, for example, Panels that are used for data visualization purposes, e.g., the creation of charts of data managed by IVIS.

2.1 IVIS

The IVIS Client is built with React [1] and Sass [2]. Most of the critical UI elements display tables or create/edit forms of entities (e.g., Signal Sets, Signal Set Records, Tasks, Jobs, Job Runs). Most of the client-side implementation in this thesis concerns these UI elements and only makes light modifications or additions.

The IVIS server is a NodeJS [3] application that uses MySQL server [4] and Elasticsearch [5] for storage. The server provides all functionality required by the client, and during runtime, it can be broken down into the server and Task Handler processes.

2.1.1 Key concepts

IVIS uses many sorts of entities. The entities are manageable by authorized IVIS users and represent all of the critical IVIS concepts, ranging from data visualization

[1] `https://reactjs.org/`
[2] `https://sass-lang.com/`
[3] `https://nodejs.org`
[4] `https://www.mysql.com/`
[5] `https://www.elastic.co/elasticsearch/`

(so-called Panels) to data processing (Tasks, Jobs, Job Runs). In this section, we will describe IVIS entities and concepts, and for this thesis, we will specifically focus on a subset of all the entities. An example of the IVIS Client's visualization capabilities can be found in the Appendix B.2.

Singals and Signal Sets

IVIS introduces the concept of *Signal Sets* for data storage. A Signal Set is composed of *Signals*. Each *Signal Set Record*, an instance of a Signal Set, is stored using Elasticsearch, an indexing solution that can be queried to reveal the values of individual Signals within the Record. A Signal may be of various types, including an integer, boolean, text, JSON, timestamp, or blob. A simple example of a Signal Set is a Singal Set containing a timestamp Signal and a temperature Signal. Each Record would then be stored as a timestamp-float pair. Screenshots of the IVIS Client's Signal Sets UI can be found in the Appendix B.1.

Tasks and Jobs

For data processing, the IVIS framework defines a system of *Tasks* and *Jobs*. A Task is a template that defines the code to be executed on not-yet-specified data and parameters. Task data (in the form of Signal Sets) and parameters are specified later during the creation of a Job.

Tasks' code may be edited. Such action then affects Jobs derived from the edited Task. Each Task has a *type* and a *subtype*. A Task type defines the programming language of the Task code, and the subtype (dependent on the Task type) defines the environment the Task's code can rely on (for example, Python [6] packages installed).

A Job is always based on a Task. The Job specifies the data the Task's code is executed with by referencing a Signal Set. It also supplies other parameters the Task requires, including Job's *triggers*. If a Job is bound to a specific Signal Set, it may be set to be *triggered* by a Signal Set record creation. A Job may also have a periodic trigger, e.g., to run every 24 hours.

Job Run and Job State

A *Job Run* is an abstraction over the process executing the Job. Job Runs encapsulate the status, time of start and finish, and the output of a single execution of the Task code using Job-supplied parameters.

Running Job may utilize a *Job State*. Job State serves as a persistent storage of Job-related data between individual Job Runs. A Job's State may be a representa-

[6] `https://www.python.org/`

tion of a counter, the ID of the last Signal Set record used for computation or an intermediate result that may help with the execution of the next Job Run, such as the last value of a computation of an average.

The IVIS Client's Task and Job UI screenshots can be found in the Appendix B.3.

Task-Job-Job Run workflow

In the current implementation, the workflow for executing some code on some data is as follows:

The user defines a Task, mainly the Task's code. For now, only Python scripts are supported as Task code. Upon saving a Task, the Task is initialized and built by the IVIS server. The initialization and build steps produce an environment for the user's code and install dependencies (defined by the Task type and subtype).

A Job is created using an already existing Task. All Job parameters are supplied, including Signal sets and Job triggers.

Triggers are enforced, and a Job is requested to be run automatically whenever a trigger condition is satisfied. A Job may also be executed directly via the IVIS client from a browser after the Job is created.

When a Job is requested to be run, a Job Run is created and scheduled. If the Job can be run, the Task's code is executed with specified parameters, otherwise, the run is canceled. A single Job may not have two concurrently running or scheduled Job Runs.

Task Handler

Under the hood, the *Task Handler* process properly schedules all the steps mentioned above. It enforces the correct ordering of the workflow steps under some assumptions. One of such assumptions is, for example, that a Task's code is modified only when there are no Jobs of this Task running or scheduled. The Task Handler process is created as a child process of the IVIS server process, and it receives/sends messages from/to the IVIS server process.

Received messages indicate, for example, a request to start Task initialization, to start Task build, or to start or terminate a Job Run. Client-side requirements (such as live updates of Task build state) require the Task Handler to send messages to the IVIS server process.

Task Handler is also responsible for the actual execution of Jobs (execution of the Job's Task code). Task Handler wraps the process executing a Job by, for example, responding to the process' output and writing to the process' input. The Task Handler may also interact with the Job Run process in other ways to ensure that a Job may issue requests to the IVIS server described in the following section.

Job Runtime Support

The aspects of the Task/Job system require additional support from IVIS, such as a way to, for example, provide Elasticsearch (Signal Set storage) access and a way to communicate updates to the Job's State.

Note that the protocol described below (using and parsing input/output/other descriptors) is Task-type specific to *the only Task type* currently present, a Python type, and may be different for other Task types. The concepts and fundamental requirements, however, remain the same.

In the current implementation, each Job Run is therefore expected to read the first line of its standard input (describes the protocol, Task-type specific) containing IVIS instance-specific and Job-specific data necessary for the Job to fulfill its function (fundamental requirement). Items a Job may be interested in are, most importantly, the Elasticsearch connection description and Job's state. The Job input data also includes a description of associated Signal Sets. These Signal Sets are available via Elasticsearch.

For some operations, the running Job must have a way to indicate its need to interact with the IVIS server. This is done by sending and receiving messages in a Task-type-defined protocol.

The predefined message semantics are:

- Storing Job's state
- Creation of a derived Signal set (e.g., for the calculation of an average value)
- Renewal of an access token (used for uploading files)

For example, in the Python task type, messages are written to an additional file descriptor, monitored by the Task handler. A response to each message is written to the Run's standard input. The Job Run process may parse the response and make other decisions based on its contents.

Although sufficient, this support is very crude. This is why the Python Task type implementation provides a Python package that abstracts these interactions into a more usable interface. The package introduces the Ivis singleton that reads the Run's initial input, parses it, and initializes all necessary resources. The Ivis class also exposes an initialized Elasticsearch object and the higher-level functions for sending messages (wrapping the crude messaging mechanism).

2.1.2 IVIS project structure

This section describes the structure and technical details of the IVIS project to set the context for the solution implementation description. The project (without the extension) is available on GitHub [1].

On the client side, the main point of concern is the `client/src/root-trusted.js` file that defines the structure of the user interface rendered by the client. User interface contents are rendered using a `Panel` React element. General content, e.g., navbar links and action buttons, is specified in the structure object (returned by the `getStructure` function [7]) via strings and callback functions (called after entity data is fetched from the server).

The following subsections describe the directory structure of the server implementation.

Configuration directories

The `server/connfig` directory contains the main configuration file that also serves as the authorization system definition. It provides secrets, parameters, roles, and permissions for the entirety of the server implementation.

The `server/knex` directory contains sample data (`seeds`) and database migrations capturing incremental altering of the database structure with the option to roll changes back (in FIFO fashion).

Endpoints

The `server/routes` directory centralizes the definition of all IVIS server endpoints accessible via HTTP. This includes API for embedded applications, server-side events (SSE), and, most importantly, the REST API endpoints for all user-manageable entities. Most REST API definitions are tiny and immediately delegate work to appropriate *Data Model* functions.

Data Model Interface

Each entity managed by the IVIS server has its data manipulated via functions defined in an entity-specific file inside the `server/models` folder. This includes the CRUD operations required by the REST API and entity-specific operations and queries, e.g., Job Run Start/Stop, Task code saving, etc.

Task Handler

The Task Handler implementation lies inside the `server/services` folder. There, one may find both message processing and Job scheduling logic. Inside the `jobs` subfolder resides the run management code. This includes the general `run-manager.js` file and the `python-handler.js` file. In terms of the *Bridge design*

[7] `https://github.com/smartarch/ivis-core/blob/master/client/src/root-trusted.js#L90`

pattern, Tash Handler is in the *Abstraction* role, while the Python handler is a concrete *implementation*. Task Handler resolves messages to stop/start/cancel a run by selecting the run's Task type handler to realize the operations.

Because each Task type might require a different environment and due to the existence of Task subtypes, concrete handlers also implement the initialization and the build of their corresponding Task type (including all the subtypes).

The run manager provides common functions to handle all logic when running a Job. This includes run termination and run messages.

Shared definitions

The server's and client's shared functionality and definitions of constants are located in the `shared` directory. Most notably, the `tasks.js` file contains constants related to the keys of Task types and subtypes, significant filenames for the Task code, and Task state definitions. On the client side, shared constants, among other uses, allow the translation of IVIS domain-specific terminology.

In the following sections, we will focus on some technologies used in the implementation presented in this thesis. Namely, we will mention the TLS protocols and Docker.

2.2 Technologies used

In the solution, we use various technologies to ensure security requirements and ease of deployment. In some parts of the solution, we also interact with external services like a commercial cloud service provider and a local HPC cluster. The technologies and services are briefly described in the following sections to ease further reading.

2.2.1 TLS

TLS is a set of protocols that allow secure communication between a client and a server, primarily focusing on information interception and modification prevention [2].

Nowadays, TLS is an integral part of Internet communication, particularly the HTTPS protocol extension. Servers that support TLS authenticate themselves to the connecting client using certificates. Servers' certificates are backed by a chain of trust - a chain of certificates signed by *Certificate Authorities* with so-called Root Certificate Authorities (signing their own certificates themselves) being the first certificates in the chain. The client usually preconfigures a set of "trusted" CAs. To fulfill the purposes of the TLS protocols, "trusted" CAs should include only the widely accepted Root CAs.

For particular purposes, clients may allow other CAs to be "trusted". This can be done both globally for the entire client machine and (more importantly for this thesis) on (programmatically) individual client's basis (e.g., specifying a custom CA for a specific HTTPS client instance or using the `--ca-certificate` option of the standard utility `wget`).

For mutual authentication, TLS allows the client to send over their own certificate. Client authentication can be enforced on the server's side to restrict access for unknown clients. Clients may also be uniquely identified using the client's certificate properties, e.g., the certificate serial number.

2.2.2 Docker

Docker is an application suite for deployment of software in virtualized environments called containers. [3] Containers are individual isolated machine-like applications that run more or less independently (e.g., a database container may be completely independent, while a web server container may depend on the database container). Containers are defined and built using a Dockerfile - a file specifying steps (commands) to create a container from a base image. Examples of operations are running commands inside the container or copying files from the Docker host to the container. Base images may be based on lightweight (Alpine Linux) or popular (Ubuntu, CentOS) Linux distributions.

Networking

Containers are attached to a network, and multiple containers may share a network. By default, containers are isolated from the host machine's network(s). When running a container, ports or port ranges must be explicitly exposed on the host machine.

Docker networking also provides a name resolution service. If a container is assigned a *container name*, then this name may be used by other containers on the same network as an address instead of the IP address of the container when attempting communication.

Container Configuration

An application running inside a container may be configured or developed by mounting files and folders from the host machine to locations inside the container. This allows the application to be configurable without the need to rebuild the container image using the Dockerfile. Other means of configuration include, for example, setting environment variables when running a container.

Applications and Docker Compose

Multiple containers may form a *Docker Application*. Containers forming an Application exhibit a certain amount of dependency upon each other. The Docker Compose tool enables easier deployment and configuration of such Applications. Docker Compose parses and realizes the *docker-compose* file. This file, in a declarative fashion, defines Docker Applications by specifying individual containers (their Dockerfiles or images, if not built directly, and other container dependencies), network properties and container configuration.

2.2.3 Oracle Cloud Infrastructure

The Oracle Cloud Infrastructure (OCI) is a cloud service commercially provided by Oracle. [4] In the proposed solution, we utilize its networking and computing services.

OCI networking consists of a hierarchy of components. A Virtual Cloud Network (VCN) represents a virtual network and is the largest object in the OCI networking. The VCN is assigned an IP range and may be divided into subnets accordingly. Access to and from a subnet is managed by Security Lists, a virtual firewall attached to each subnet.

OCI Compute service provides the user with the means to create and manage virtual machines of their choosing. Performance tiers of VMs are called Shapes, and each compute VM must be attached to a VCN subnet.

2.2.4 SLURM

SLURM is an open-source software for cluster management and workload scheduling. [5] SLURM jobs are written in the form of standard Bash scripts enriched with some extensions allowing output capturing and customization of resource allocation (e.g., timeouts, memory, and processor nodes).

The SLURM tools like `srun` and `sbatch` allow the automation of individual steps of a job. The `srun` tool launches a command directly while `sbatch` is used to schedule scripts asynchronously.

Each Job running via any of the `slurm` tools can be uniquely identified. The job ID may be used to schedule jobs conditionally (after success, after termination).

The SLURM cluster management enables fine-grained control of access to the cluster resources using partitions. Each cluster user gains access to resources according to their membership in partitions. Access is limited in terms of priority, hardware nodes, and time limitations on the job runtime.

Chapter 3

Analysis

This thesis aims to allow Job execution outside of the IVIS server host. This chapter examines requirements, use cases, and the rough architecture of the solution.

3.1 Motivation

The possibility to separate the execution from the IVIS server host allows greater flexibility for the user and the IVIS instance administration. The user should be able to execute individual Jobs on specialized hardware, and the IVIS server should become more stable as it can support more concurrently running Jobs. More specific examples follow.

3.1.1 Increase in Job execution throughput

In the current implementation, the maximum amount of Jobs running in parallel depends on the IVIS server host hardware. IVIS server does not limit the number of running Jobs; therefore, the server may overload the host machine given enough resource-intensive Jobs.

The ability to run Jobs outside the IVIS server host would allow the offloading of resource-intensive Jobs to different machines, effectively increasing the number of Jobs that can be executed in parallel.

3.1.2 Utilizing specialized hardware

The delegation of Job execution, especially the cloud-based and SLURM cluster extensions, should allow the user to utilize specialized resources outside the IVIS server host. Examples may include standard scaling of computational resources

like an increase in processing power and memory capacity or allocating Task-specific hardware, such as powerful GPUs for CUDA [1] workloads, etc.

3.2 Solution architecture

From the perspective of a user or an administrator, we want to add the option to manage remote resources and allow the choice of computational resources when creating a Job. This implies creating at least one new IVIS entity (to represent the remote resource) and making various client-side additions to manage the new entities, and modifying the Job form (to specify where to execute a Job).

On the server's side, we must integrate as smoothly with the existing (and complicated) Job/Task functionality as possible. The solution roughly attempts this by intercepting remote Job Run's side effects and delegating them to the IVIS server. Relevant messages between the IVIS server and the remote Executor are sent over the Internet.

This section will examine how this thesis approaches the problem as a whole. We will introduce the concept of a Job Executor and analyze some technical challenges that arise from our attempt to distribute the Job execution in the IVIS framework.

3.2.1 Job Executor

The solution introduced in this thesis creates an abstract entity, a *Job Executor*. A Job Executor can be effectively thought of as a machine (or a set of machines) that can properly execute Jobs like they would be executed on the IVIS server. A Job Executor is expected to be able to receive simple Job-lifecycle commands (e.g., start a Job Run) and communicate the results of Job Runs back to the server via a unified interface.

In terms of IVIS entities, a Job Executor is a regular entity the user can manage. A Job Executor entity represents the machine available for execution, which is why the entity exposes the abstract machine's status (indicating readiness to execute Jobs) and logs (for troubleshooting - e.g. invalid cloud service credentials). The IVIS server itself is also classified as an immutable Job Executor. From now on, we will refer to Job Executors excluding the IVIS server as *remote* Job Executors.

3.2.2 Challenges

This section lists challenges that need to be addressed by the solution and roughly describes how the solution does so. In a nutshell, we need to ensure security and

[1] `https://developer.nvidia.com/cuda-toolkit`

proper Job Run behavior replication or emulation to integrate with existing IVIS functionality smoothly.

Security

Communication between the IVIS server and a *remote* Job Executor is generally conducted via the Internet. Because a running Job is provided with access to systems such as the Elasticsearch server and a running Job may issue requests with complex server-side side effects, the Job Run's actions and communication must be verified by the server.

Currently, the IVIS server assumes the (local) requests to be correct and trustworthy since Job Runs are launched and managed directly by the IVIS server. However, in the solution presented in this thesis (in an Internet-distributed scenario), this blind trust is not acceptable. This is because otherwise, malicious requests may be forged and sent both to the IVIS server and a remote Job Executor to exploit them.

Each access and request coming from a remote Job Executor thus needs to be authenticated and authorized before it is granted and performed, respectively. Similarly, the Job Executor exposes its computational capacity to the Internet and thus needs to authenticate the IVIS server as its master as well.

The contents of all communication between the IVIS server and a remote Executor also need to be protected from forgery and eavesdropping. First, message forgery would allow arbitrary remote code execution as the IVIS server would need, at some point, to provide the remote Executor with the Task's code it is supposed to execute (the attack vector is the unencrypted Task code sent over the Internet). Second, unauthorized data reading may pose a security risk if, for example, the IVIS server manages sensitive Signals and Signal Sets. This data exposure arises because the data would need to be transferred to the remote Executor during Job execution.

The solution approaches these problems using SSL/TLS certificates and certificate authorities (CAs). The IVIS server acts as a local CA. With each Job Executor creation and removal, the local CA creates and removes a client certificate key pair for the Job Executor. The server and Executors authenticate themselves mutually via server and client authentication defined by the TLS protocols. The SSL/TLS usage also provides communication encryption, preventing message forgery and eavesdropping.

Authorization is checked only on the server side to differentiate between individual Job Executors. This is done to ensure that no Executor may modify data related to Jobs whose Runs were assigned to a different Executor. Authorization is *not* done on the Job Executor side, i.e., Job Executor implementation is intended to communicate with only one "master" IVIS server instance.

Remote Task build, Task code updates

As the execution of a Job itself is done outside the IVIS server, the environment supporting the execution must also be replicated on a remote Executor. The entire Task building process is thus replicated on each remote Executor when needed by a Job running on that same Executor. The implementation assumes that when a Task can be built locally on the IVIS server, it can also be built remotely. In the case of a remote build failure, the implementation has no choice but to inform about the failure via the Job Run that could not be executed (as the remote build is triggered only by a request to run a Job). This is not identical to the IVIS server's handling of local build errors - an unbuildable Task cannot even be used to create Jobs.

Another issue arises when the underlying Task itself is modified. We recall that a Job is an instance of a Task - the Job specifies the data for execution and triggers when to run the Task code. Therefore when a user changes the Task code, type, or subtype, the environment must be rebuilt entirely. The Task modification is propagated to the remote Executors on the first affected Job run request. Each Job run request sends data used to build the Task, regardless of whether changes were or were not made. To recognize changes, the remote Executor examines this data (code, Task type, and subtype) and either schedules a build and a Run or just a Run. In the implementation, the original IVIS server assumption that the Task code is not modified when Jobs of that Task are running is reused in the remote Executor implementation.

Scheduling

The IVIS server's existing scheduling capabilities are reused in the remote execution scenarios. The only notable addition to the scheduling logic on the IVIS server side is checking the status of a remote Executor before delegating the Job run request to the Executor.

On the remote Executor side, a subset of the scheduling mechanisms of the IVIS server is used. In a pool configuration, the IVIS server sends a Job run request to a single node which is configured to schedule the Job Run among the pool peers. Finally, in the SLURM implementation, the cluster frontend's scheduler is used along with shell scripts taking care of proper scheduling of dependencies, such as the dependency on a build step terminating when the Task must be (re)built.

Remote Executor HTTP interface

The IVIS server needs to issue commands to remote Executors. Most remote Executors are therefore required to implement an interface for the operations required by the IVIS server. Most of the Executor implementations provide an HTTP

interface, the only exception being the SLURM-based Executor implemented via SSH commands and shell scripts executed on the SLURM cluster.

The interface endpoints/operations can be described as follows:

- initiate a Job Run (and build the entire Task if needed)
- get a Job Run's status (or report Run was not found)
- remove a Job Run from the remote Executor's data storage (a sort of a cleanup)
- stop a Job Run (or report Run was not found)
- remove a Task (cleanup purposes)

This interface gives the IVIS server all the control it needs over a Job Run's lifecycle. However, this interface covers only a part of the IVIS server's servicing of a Job Run.

IVIS server remote push interaface

The solution implemented in this thesis mimics the behavior of a Job Run as if it was running locally. Job scheduling, start, stop, and status requests are directly translated to the appropriate remote Executor's interface (most commonly the HTTP one). What remains is all the status reporting and state changes reported by the IVIS server's run manager. In other words, the actual Job outputs and success and failure reports up to this point were not addressed either by the remote Executor or by the IVIS server.

This thesis implements a push-based messaging of this information from the Job Executor to the IVIS server via the IVIS server's remote push interface. A pull model would be too resource-intensive for both the IVIS server and the remote Executors. This interface needs to address three types of messages which are received by the IVIS server when a Job is running locally:

- Status report when a Run terminates
- Event emission for client-side components (live run output)
- Run requests (request to store a Job's state, request to create a Signal Set)

This interface is implemented as an HTTP interface made of 3 endpoints, each addressing exactly one of the message types mentioned above.

In addition to these endpoints, the IVIS framework also exposes the Elasticsearch instance to the Internet. Up to this point, Elasticsearch has been exposed

only to localhost. However, remote Job Runs need to query Elasticsearch and thus require Elasticsearch to be accessible via the Internet. To avoid requiring Executor-specific modifications to the Task code in order to access the Elasticsearch instance from a remote Executor, Elasticsearch is exposed on TLS-authenticated ports. This restricts access to only remotely executing Jobs due to the usage of the local CA. TLS authentication is also used for all the HTTP endpoints with an additional layer of security in ensuring Executors modify data related to the runs only they are responsible for.

In the following sections, a rough overview of the individual remote Job Executor implementations is given.

3.2.3 Remote Job Runner

Remote Job Runner (RJR for brevity) is a Docker application consisting of an NGINX [2] web server acting as a proxy and a NodeJS container running the application logic. The RJR is the simplest form of a remote Job Executor, effectively acting as described in previous sections. The RJR, at its core, is a remotely-controllable subset of the Job running functionality of the IVIS server.

Architecture

The NGINX proxy container proxies all incoming traffic to the NodeJS application. The proxy ensures the security requirements by utilizing client authentication. IVIS server's CA certificate is allowed to access the RJR's HTTP interface serving to receive commands from the IVIS server to execute, stop, remove a Job Run, or query Job Run's status.

For the persistence of Job Run data, the RJR uses an SQLite database.

Each request that the RJR cannot fulfill is delegated to the IVIS server via the corresponding endpoint (Job state manipulation, event emission, etc.), and the response is returned to the Job Run. The request proxying is done by the NodeJS application that monitors the Job Run process itself. Figure 3.1 captures the interactions between the RJR and the IVIS server.

A Job Run accesses the Elasticsearch instance via Elasticsearch libraries. This implies that the runtime support must ensure a connection is made to the proper Elasticsearch instance and provide a certificate to gain access to it. Hence the RJR includes a modified version of the IVIS Python package. The modification guarantees correct certificates, and the Elasticsearch connection is configured upon Run's start.

[2] `https://nginx.org/en/`

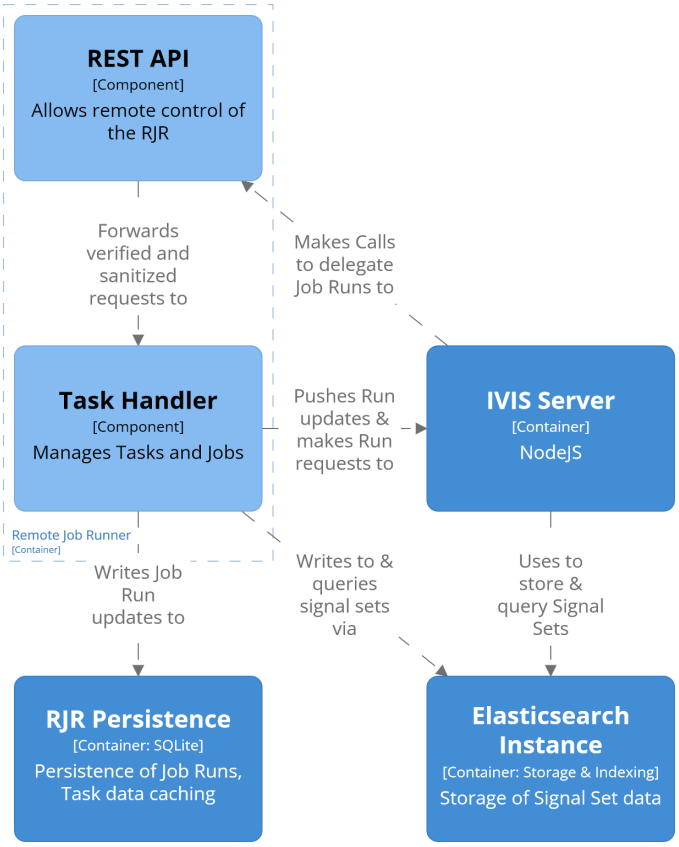

Figure 3.1 Diagram of the Remote Job Runner intended architecture

Deployment

To use the RJR, user intervention is needed in various steps of the Executor creation. First, the user needs to build the RJR application on their chosen machine and ensure it is reachable via the Internet. Second, the user must configure the Job Executor with necessary data, such as the address of the IVIS server. Third, the user must create the Job Executor of the RJR type in the IVIS client, manually copy the Executor's certificates, and provide the certificate of the IVIS certificate authority. After these steps, the RJR Docker application may run and IVIS server may utilize the machine as a remote Executor.

3.2.4 Remote pool

The remote pool is a general extension of the Remote Job Runner implementation. A remote pool organizes a set of RJRs under a single master node which hosts the Remote Pool Scheduler Docker application (RPS for brevity). The RPS host schedules and proxies all requests to the correct RJR instance. The interface of and communication with the RPS is designed to be nearly identical to the RJR interface.

Architecture

The RPS application uses an Apache container acting as a proxy in both directions and a NodeJS application for scheduling, routing, and verification. The RPS application is provided with addresses of the pool peers expected to carry out Job execution and implement the remote Executor HTTP interface. In the solution presented in this thesis, the pool peers are running the Remote Job Runner application described in previous sections.

The RPS acts as a scheduler and router for requests from the IVIS server toward the individual Executors (running RJR). Thus, when the pool is instructed to execute a Job, the RPS must keep a mapping of a Job Run to the address of the Executor responsible for its execution. This data can be later used to correctly route other requests, such as the request to retrieve the status of a Job.

The RPS application also performs request proxying in the reversed direction. When a Job Run needs to make a request to the IVIS server host (e.g., an Elasticsearch connection attempt), a proper client certificate must be used to sign the request. To simplify this process, the RPS application provides proxying endpoints to the pool peers and is configured to use its own certificate to forward requests to those endpoints. Figure 3.2 may clarify the idea behind the architecture.

The RPS requires SSL client authentication when receiving commands via the Remote Executor HTTP interface to maintain the security requirements. It also

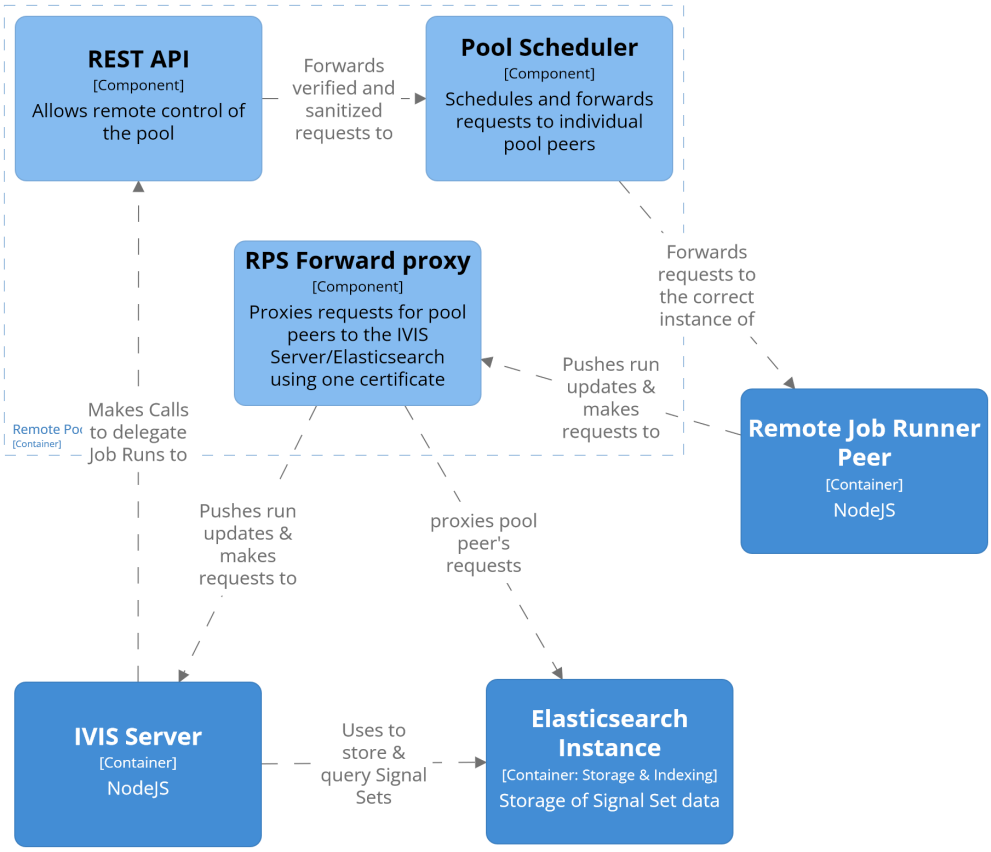

Figure 3.2 Architecture diagram of the RPS

requires that the pool peer proxying endpoints are accessible only by the pool peers and that their communication is not exposed to the Internet.

Deployment

It is possible to set up a remote pool using the RPS application manually. In this thesis, the remote pool concept is used in the implementation of the cloud-based Executor pool, which is deployed automatically.

In the case of manual deployment, individual peers must be configured in such a way that they are accessible only by the RPS host. Peers themselves (since unreachable from the Internet) do not use certificates and point all their requests toward appropriate RPS proxying endpoints.

The RPS host is configured similarly to the HTTPS-secured Remote Job Runner deployment described in the previous section. I.e., the RPS is set up on the host machine, the IVIS Remote Executor entity is created, and its certificates are manually injected along with the proper configuration of the IVIS server's endpoints. Additionally, the RPS must be provided with a list of addresses of the pool peers.

Some endpoints of the RPS application are Peer-only, and some are IVIS-server-only. Peer-only endpoints include the Elasticsearch proxying and IVIS server proxying endpoints. These endpoints can be mapped to a specific port which allows limiting of the traffic on the RPS host's firewall level.

3.2.5 Oracle Cloud Infrastructure pool

Support for a commercial cloud service provider is a desirable extension of the remote Executor concept. This extension adds a permanent user-manageable Oracle Cloud Infrastructure (OCI) homogenous virtual machine pool, given that the user has provided their OCI credentials. Usage of the OCI is arbitrary, and other cloud services can be integrated in the future.

Architecture

The OCI pool implementation automates the deployment of the remote pool consisting of an RPS application and RJR applications running on different virtual machines. The OCI pool implementation sets up the OCI networking and virtual machines to satisfy the remote pool constraints (mainly the endpoint isolation) and securely injects all required configurations so that the pool is ready to receive commands.

As the OCI is an Internet service, failure must be indicated to the user, and sufficient administrative tools and information must be available for usage and

display. These include failure logging, "forced" removal of an Executor (a dangerous operation supposed to be executed only when every other automated method of removal fails), and *Global Executor Type* state.

The Global Executor Type state contains information related to all remote Executors of the same type (e.g. the OCI Pool type). Some cloud services introduce rather tight limits to their services, e.g., the OCI limits the number of Virtual Cloud Networks. The Global Executor Type state ensures such limited resources can be shared among multiple instances of remote Executors of the same type. For example, the OCI Virtual Network is subdivided into isolated subnets, each allocated for one and only one OCI pool Executor. The allocation of the subnets is kept in the Global Executor Type state to allow more straightforward subnet allocation when creating a new Executor. Additionally, the state allows the removal of the entire Virtual Cloud Network on demand by storing the network's identifier.

Deployment

There are two criteria for the usage of this extension. First, the user must inject a valid OCI configuration file bound to an OCI API key pair and the key pair. These files are generated and downloaded from the OCI interface and are supplied on IVIS startup as part of the IVIS configuration. Second, the user should find and select the OCI Compute VM shape the pool peers should use. [3]

3.2.6 SLURM cluster

Finally, we introduce a fundamentally different remote Executor Type implementation. This Executor utilizes a local SLURM cluster by connecting to and issuing commands from a cluster's frontend node.

Architecture

The cluster's frontend node shares its `/home` directory with the cluster. This means that both the user of the frontend node and the cluster job may modify and access those files. Executor data is stored in this shared filesystem in predefined locations. Some data is persisted due to implementation details, to store the Task code or to make retrieval of run data possible in case the IVIS shuts down before a Job Run finishes.

As the frontend node is a rather limited machine, the frontend node serves only for scheduling Jobs and retrieval of outputs. Computation, the environment

[3] `https://docs.oracle.com/en-us/iaas/Content/Compute/References/computeshapes.htm`

setup, etc., must be done by scheduling cluster jobs carrying those tasks out. This means that neither the frontend node can serve as a host for the RJR application nor can the cluster itself, as the RJR must be available continuously and cluster jobs are time-limited (idle RJR instance would be a waste of the cluster's resources even if jobs were not time-limited).

IVIS server thus connects to the frontend node via SSH and executes commands on behalf of a remote Executor when needed. Job scheduling is delegated to the SLURM cluster management. All administrative tasks ranging from output collection and Job status retrieval to Job environment setup (building a Job), are implemented as Bash scripts utilizing the SLURM tools and their scheduling options.

Run push requests, normally performed by the RJR, are performed from within the Run itself. This is done with yet another modified version of the IVIS Python package and a Job Run wrapper script handling Run's output and status reporting.

Deployment

The IVIS server uses a single SSH key pair to connect to the SLURM frontend node. The user must provide the hostname of the frontend node, username, password, and the name of the partition to use by the Executor. On Executor creation, the IVIS server injects all the scripts and sets up the filesystem structure on the frontend node.

Chapter 4

Implementation

This chapter describes the implementation of the solution. Most changes and additions are made to the server side of the framework. The implementation attempts to integrate with much of the existing IVIS framework code, and even external applications (such as the RJR and RPS) follow the structure and architecture of the IVIS server. One section is dedicated to the few client-side changes and additions made behind the scenes. The last sections are focused on the individual executor types implemented.

4.1 Containerized deployment of the framework

To simplify deployment, a Docker application for the IVIS Framework was created. Initially, the IVIS repository contained setup scripts that severely modify the host machine. Reinstallation of the framework thus becomes quite cumbersome.

The IVIS Containerized repository contains the Docker application and a Dockerfile that builds the IVIS server and client. The application creates a proxy container for HTTPS communication towards the IVIS server, an Elasticsearch container, a MYSQL database container, and the IVIS server container.

As the implementation utilizes SSL/TLS protocol and requires that the IVIS acts as a Certificate Authority, certificates are created as a part of the setup process. Additionally, due to the usage of SSH (by the Slurm remote executor implementation), SSH key creation is also part of the setup process.

The only prerequisite for deployment is a valid registered domain name for all of the IVIS endpoints (as is required for the original script-based installation).

4.2 Newly introduced entities

The implementation introduces key entities representing the Job executor and the Global executor Type State. In the IVIS framework, each entity is associated with a relational database table of the entity type (e.g., `jobs`, `tasks`, etc.). This section focuses on the schemata of the tables associated with newly added entities.

Each IVIS-managed entity must have a `namespace` field for access-control purposes. The implementation fully integrates with both the namespace system and the entity-sharing system that the IVIS framework provides.

4.2.1 Job executor

A Job executor entity abstracts a machine or service capable of executing IVIS Jobs. Its `name` and `description` string fields exist for UX purposes; the `type` field specifies the type of the executor as distinguished in the previous chapter. Further, Job executor data (usually specific to the executor type) is stored in the `parameters` field in the form of stringified JSON, much like the parameters of a Task.

We recall that each executor is associated with its certificate. The certificate's serial number is used to authorize some Job requests from a remote executor. Thus, each executor has a `cert_serial` filled to mitigate filesystem accesses, as the number of requests that need to be verified may be significant. This should also reduce request latency.

The `status` and `log` fields serve troubleshooting purposes. The `status` field indicates the deployment or creation status (either ready, failure, or provisioning), and the `log` field contains additional information in the case of executor creation failure.

Finally, the `state` string field is a space for executor-type-specific data which must be persisted between server restarts. The usual format is, again, a stringified JSON.

To represent the localhost executor (the IVIS server host), the `job_executors` table is prepopulated with an entity of a special type `local` and a reserved identifier. The local executor may be neither modified nor deleted and is always in a "ready" state. An identical approach can be observed in the `root` namespace creation.

4.2.2 Global Executor Type State

This relatively simple entity keeps data common to all executors of a specific type. Thus, for each executor type, exactly one pre-created Global State entity

exists. Each State Entity has a type-specific data field `state`, a `log` field, and a `lock` field.

The `lock` field is utilized to ensure (via the ACID properties of the underlying relational database) that the global state is modified by one and only one Job executor (e.g., during an allocation of resources from a shared pool, such as subnets of an IP address range).

4.3 IVIS server Job lifecycle

This section describes the lifecycle of locally-run Jobs. This introduction is necessary to understand the modifications done by the solution presented in this thesis.

The IVIS server launches a separate process (the Task handler process) to take care of Task building and Job Run scheduling. The process receives messages from the IVIS server and, after some processing, places them in a work queue for further processing. The message types defined in the `shared/jobs.js` file represent purposes of the messages. Some types reference events of the the Task lifecycle (`INIT`, `BUILD` and `DELETE_TASK`), some are related to Job/Job Run Lifecycle (`RUN`, `SIGNAL_TRIGGER`, `STOP`, `DELETE_JOB`) and the `ACCESS_TOKEN` type is related to Job Run runtime support.

Because Task or Job Run lifecycle is specific to the Task's type, some messages' handling is dispatched based on the Task's type. The messages are dispatched to so-called handlers. An example of a handler may be found in the `server/services/jobs/python-handler.js` file. The handler is supposed to implement and export functions for handling certain steps (message types) of the Task or Job Run lifecycle, namely the `init`, `build`, `run`, `stop` and `remove` functions.

The *Task* lifecycle message types of concern are the initialization and build types. The initialization step includes the creation of the Task's directory, the creation of a virtual environment, and the installation of Task's-type-specific libraries. The build step should handle the propagation of any modifications of the Task's files made by the user.

The implementation makes several assumptions about the Task build system. First, Jobs may run concurrently. Second, Task builds, and initializations may run concurrently. Third, rebuilding/reinitializing a Task while the Job of that Task is running is not allowed, i.e., not accounted for.

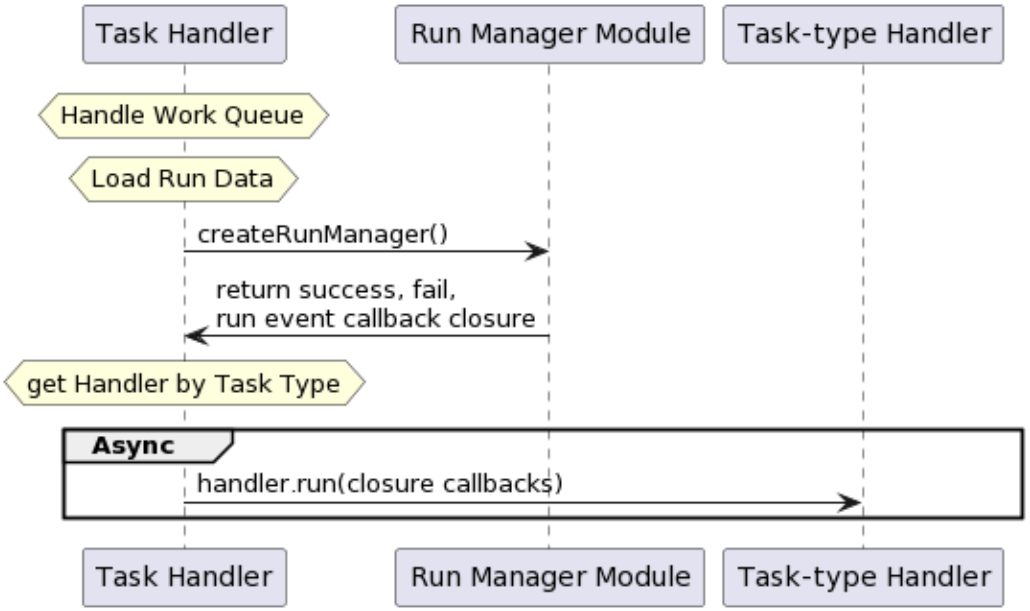

Figure 4.1 Sequence diagram of the first phase of Run start

4.3.1 Running a Job

We will describe the Python handler's (the only handler present so far) way of running a Job, and we will introduce the general procedures leading to Job execution on the IVIS server. On a technical level, this is replicated in the RJR implementation.

The scheduling checks whether the run conditions are met upon processing a `RUN` message type. These include the minimal defined delay between the same Job Runs or whether a Job is not already running. After checks, a Run configuration is created. The Run configuration contains all parameters necessary for a Job Run to be performed and includes, for example, identifiers of the Task, Job, Job Run, and most importantly, the Job Run input (connection details for Elasticsearch, assigned signal sets, and signals, and Job state). The configuration is then passed to the `createRunManager` function. The Run manager is a JavaScript closure encapsulating handlers for Job lifecycle events. More exactly, it is an enclosed object with `onRunFail`, `onRunSuccess` and `onRunEvent` handlers. The Run manager closure also periodically refreshes the Job Run's access token for the IVIS API and sandbox endpoints. Figure 4.1 roughly visualizes the initialization of a Job Run.

The Run manager handlers are general and are shared among all Task types as they are passed to the Task-type-specific handler (the Python handler). The manager-returned handlers are used to modify the overall Job Run's status (in the

case of `onRunFail` and `onRunSuccess`) or are used to communicate Job Run's requests at Job Run runtime (`onRunEvent`). The Task-type-specific handlers are thus expected to call the supplied Run management handlers when specific events occur (Run success, failure, or a Run request).

Run requests & Job runtime support

The Run Manager's `onRunEvent` function handles events occurring during a Job Run execution. These events indicate that the Job has emitted *output* or a *request*.

Job Run's output is processed and conditionally persisted by the Task Handler and may be viewed by the user in the Job Run's Output in the IVIS Client.

Job Run requests represent operations that need to happen under the supervision and assistance of the IVIS server. Currently, there are two Job request types (defined in the `shared/jobs.js` file): `STORE_STATE`, which requests storage of the supplied JSON as the Job's new state, and `CREATE_SIGNALS`, which specifies a new computed Signal to be created.

The Run Manager provides a general interface, and thus the Task-type-specific handlers must ensure that a Job Run can communicate its needs properly, i.e., all of the underlying inter-process communication (between the Task Handler and the Job Run process) is left to the implementation of the Task type handler. For the purposes of this thesis, the Python handler's communication will be described as most of the remote Job execution functionality is derived from it.

The Python handler (`server/services/jobs/python-handler.js`) implements Job execution by launching a a separate process, invoking a Python interpreter on the root Job file. In addition to standard I/O streams, it adds another file descriptor to the process. We will call this descriptor the file descriptor 3. The handler expects the Task code to utilize the IVIS Python package [1].

The handler registers callbacks on various events on the process' standard output and error and reacts to writes to the file descriptor 3 to implement runtime event handling. The Job Run process writes its requests to the file descriptor, the handler parses and propagates the request to be processed by IVIS, and writes the result to the standard input of the Job Run process. On the Job Run's side, the protocol is implemented in the IVIS Python package to separate the levels of abstraction (see the `_send_request_message` and `_send_request_message` methods). In addition, Run Manager-supplied callbacks signaling Run success and failure are registered for the exit event of the Job Run process.

The Python handler provides the Job Run with input data (configuration, connection detail, signals, parameters, etc.) in the form of a JSON string on the first line of the standard input. Parsing, utilization, and encapsulation of this

[1] main functionality implemented in `server/lib/tasks/python/ivis/ivis/helpers.py`

information are also done by the IVIS Python package in the constructor of the `Ivis` class. Therefore, all Tasks must construct this object as soon as possible.

In the Task code, the user may use any of the `create_signal_set`, `create_signal`, `store_state` methods to interact with the IVIS server. The `elasticsearch` property provides an initialized Elasticsearch library object capable of interacting with the Elasticsearch server. Figure 4.2 attempts to explain the runtime support visually. Arrows ending with circles represent the mandatory interface imposed by the Run manager. *The way these events are triggered* from the Task code and how the results are communicated back to the calling code is up to the Task-type implementation.

4.4 Integration of Executors

This section focuses on the modifications done concerning the Job entity, the lifecycle of Job Runs and the support of their remote execution.

The relationship of an executor to a Job is represented in a new field in the *Jobs* table. The `executor_id` field refers to the executor assigned to execute the Job.

4.4.1 Remote push HTTP interface

We need to allow remotely executing Jobs to have the same side effects on the IVIS server as a locally executing Job. These side effects include Run's status updates, event emissions for the Clientside updates, and the fulfillment of Run Requests (as they require the IVIS server to be completed). We create an HTTP interface suited for precisely those purposes.

The HTTP interface implementation assumes it is accessible only using a valid client certificate (signed by the IVIS local CA) and that the certificate's serial number is passed in the request headers. Recall that a Job executor's certificate serial number is saved in the `cert_serial` field of each Job executor. The certificate serial number, along with Job and Run identifiers sent as part of each request's body, is used to check whether the supposed An executor is authorized to manipulate the data it is requesting to manipulate.

The `/rest/remote/status` endpoint informs the IVIS server of a remotely-executed Run's progress. This includes the state (scheduled, running, success, failed) and, optionally, the output of the Run along with a Run's termination timestamp. As requests over the Internet may arrive in a different order, prioritization of state updates is implemented, with the terminating states (success, failed) having the highest priority. If a lower-priority state write is attempted, it is not persisted.

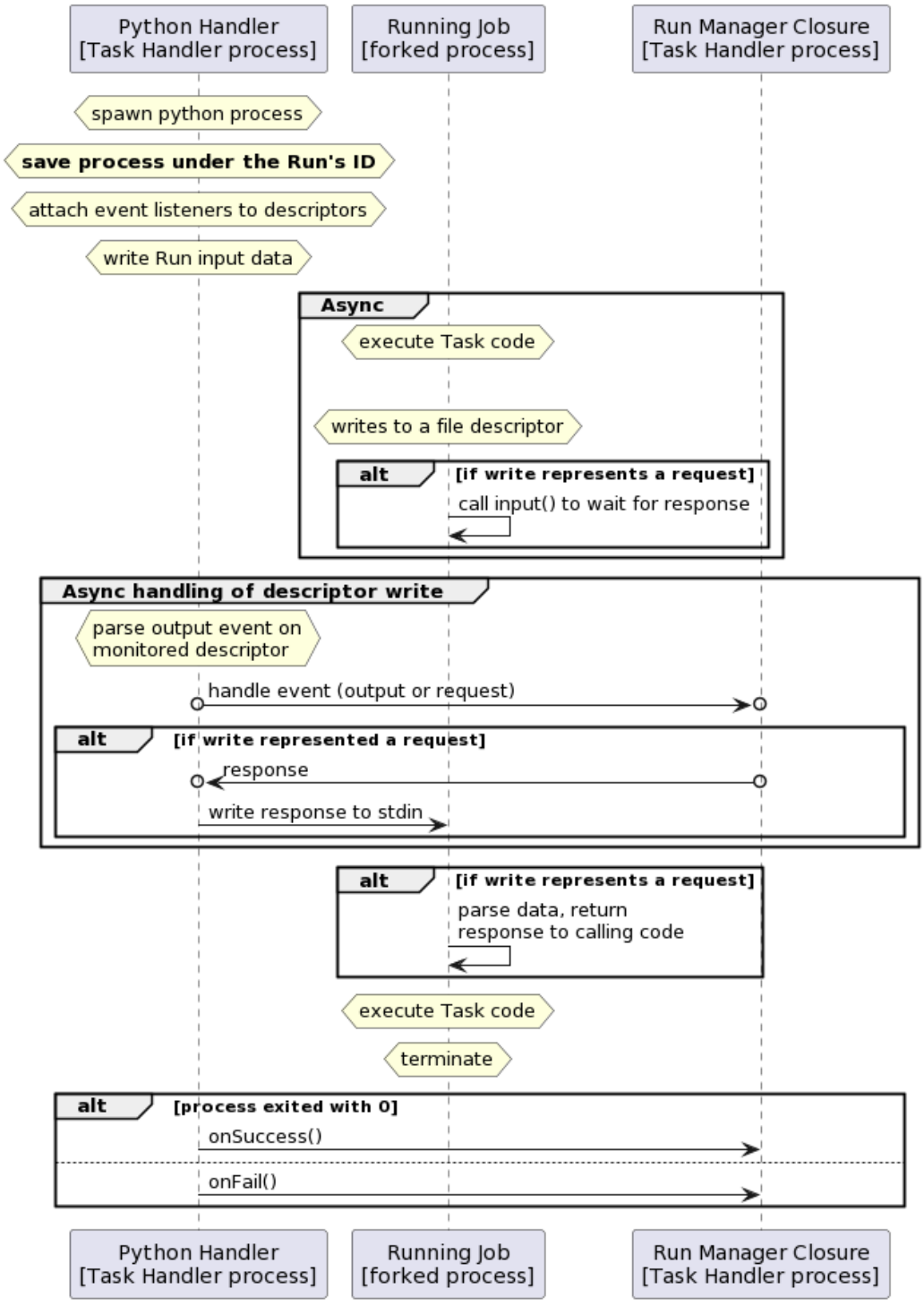

Figure 4.2 Sequence diagram of the execution of *a Python* Job

The `/rest/remote/emit` endpoint propagates Run's updates to the IVIS Client's components, such as the live-updated Run Output console. (available in the Run log or on the Task development page). The IVIS server implements serverside events endpoints, and the Client's Run Console component interacts with the endpoint to deliver live Run output updates. After remote-Run-related checks, the `emit` endpoints merely emulate the event as if it were emitted locally.

Finally, the `/rest/remote/runRequest` endpoint performs both Run request types (store Job state and create Signal set). After remote-Run-related checks, it performs the request exactly like a Run Manager. Contrary to the already described endpoints, the `runRequest` endpoint returns data in its response.

4.4.2 Job lifecycle modifications

We will describe critical changes to the Job lifecycle steps described above needed to integrate the Job executor concept. As mentioned, the remote execution scenarios expect that when a Task is buildable locally, it is probably also buildable remotely. Thus the current Task creation, initialization, and build are left unchanged to detect build failures.

Preparing the Task's code for transfer to a potential remote executor in the future is the only vital addition to this part of the lifecycle. This step is implemented on the Task Handler level by reacting to Task initialization success and Task build success. After these success events, the Task code is archived into a `tar` archive via the newly-added `task-archiver` module [2]. The module is also used to retrieve the archives when supplying them to the remote executors later.

For the implementation of the run and stop steps, we have to differentiate between local and remote Job execution on the Task Handler level because some of the steps made by the Task Handler are by the solution's architecture performed on the remote executor's side only. This includes the removal of build-related event listeners or the creation of the Run Manager closure. In the implementation, if we recognize that a run or a stop request is related to a remote executor, we delegate the request to the newly-added dispatcher function and immediately return. Figure 4.3 shows the Run stop is handled when running locally.

The dispatcher functions [3] perform run or stop requests based on the executor's type. They dispatch the requests to the lowest-level interface for communication with remote executors (HTTP for RJR and RPS-based, SSH for SLUMR types).

[2] `server/lib/task-archiver.js`

[3] defined in `server/services/jobs/remote-machine-handler.js`

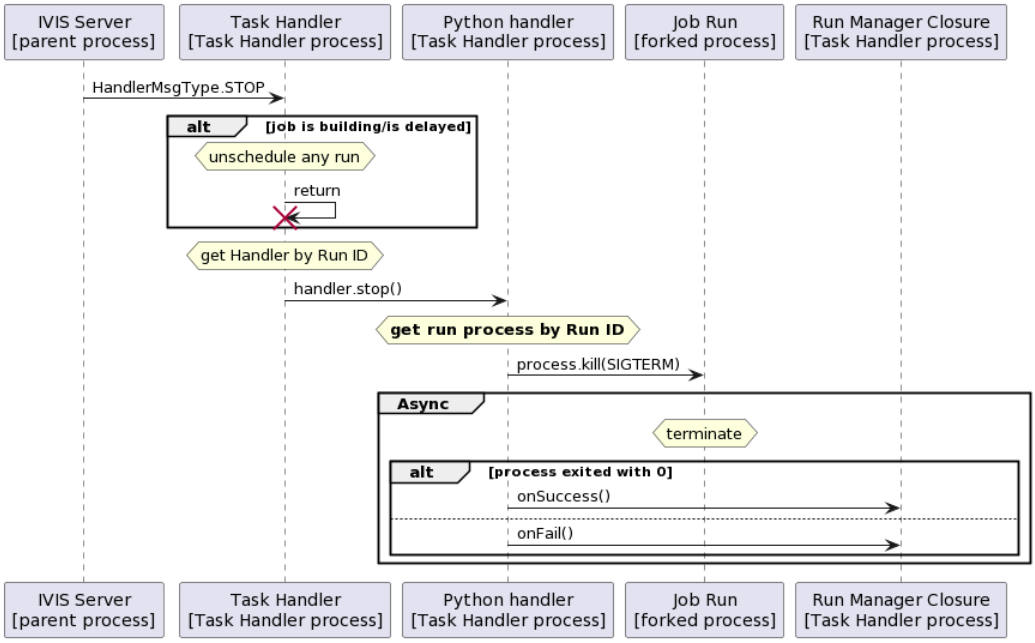

Figure 4.3 Sequence diagram of the local Run stop

4.4.3 OCI pool and SLURM specifics

As the OCI Pool and SLURM implementations are deployed automatically, some construction and destruction processes must occur. When creating a Job executor, the executor is created in the background, and the user is informed via the executor's status.

For OCI Pool, this means the allocation of an unused subnet where the RPS-backed pool will be deployed, as well as the VMs, which will host the RPS and RJR agents. After deployment, a necessary configuration is done by issuing commands via SSH.

For SLURM, SSH configuration is done, and executor's filesystem structure is created. Configuration mainly consists of the injection of various scripts to be executed via the `srun` and `sbatch` commands.

Destruction usually undoes all the mentioned steps made by initialization. To enable the user to handle initialization and removal failures manually, the Job executor entity may be *Forcefully Removed* . Forceful removal frees up resources on the IVIS server's side and removes the entity ignoring all errors that the removal process may emit. This means that remote resources such as subnet or VM allocation may leak. Forceful removal is therefore supposed to be used only when the automatic removal fails, and the user frees up all allocated resources (communicated via the IVIS client). Failure reporting and resource allocation

information is available in the executor log when a failure occurs.

4.5 Server-side additions and modifications

This section will list some of the changes made to the IVIS server. We will focus on the mentioned issues or mechanisms introduced in this thesis. All paths mentioned in this section are relative to the `ivis-core/server` directory.

4.5.1 Minor additions

For certificate support initialization, the `certs/remote` directory now contains a setup script that creates the IVIS server CA for certificate signing and a script for IVIS server's certificate creation. Lastly, there is the `remote_executor_cert_gen.sh` script used to create a client/server certificate for a remote executor. The `lib/remote-certificates.js` file exports helper functions for manipulation of the executor certificates and works in tandem with the scripts.

To simplify SSH usage elsewhere in the code, the `SSHConnection` class was implemented in the `lib/instance-ssh.js` file. This class wraps the `ssh2` [4] client to allow asynchronous command execution and connection reuse. The `lib/instance-ssh.js` file also exports some helper functions concerning file uploads and connectivity checks.

The `lib/task-archiver.js` file exposes helper functions for unified archiving of Task code.

4.5.2 Remote executor communication

The general entry point to executor communication is the `lib/remote-executor-comms.js` file. It declares the handlers for various operations required by the implementation based on the type of executor. It also implements essential communication for RJR and RPS, as they are just HTTP requests via a preconfigured HTTPS client. As the OCI Pool is implemented in terms of an RPS-backed pool, only the SLURM implementation must be implemented differently (SSH).

The SLURM pool communication is implemented in the `lib/pools/slurm.js` file. The implementation only delegates the requests (to run, stop, get the status of a Run, etc.) to the appropriate script injected when the executor was created. All communication with the cluster itself is done via SSH.

[4] `https://www.npmjs.com/package/ssh2`

4.5.3 Remote executor pool setup

As mentioned above, for automatically-deployed executors, the creation step differs by performing additional work in the background as opposed to plain storage of data from the user in the case of RJR and RPS pool executor types.

The OCI Pool setup works in 2 layers: the OCI layer and the VM (host machine) layer. On the OCI layer, we set up all OCI infrastructure to host the VMs. We use the official OCI SDK [5] and SSH. This is implemented in the `lib/poools/oci/basic/oci-basic.js` file.

On the host machine layer, we want to inject all data RJR and RPS need to function on the host. We do this as part of the instance setup process, and the majority of the configuration templates can be found in the `lib/poools/oci/basic/rjr-setup.js` file.

4.5.4 Data model modifications

The major addition of this thesis is the Job executor entity. Each entity in the IVIS Framework has represented in its own file inside the `models` folder. The `models/job-execs.js` file thus represents operations over our Job executor.

In addition to all the required functions enabling an entity to be integrated with the IVIS entity CRUD forms and basic manipulation of the Job executor data fields, we significantly modify the creation and deletion processes by adding background initializers and destructors. Intializers and destructors are defined in the `executorInitializer` and `executorDestructor` objects, respectively and are the primary mechanism for background setup/destruction of the OCI and SLURM implementations.

Other additions include the ability to view executor certificate data for certificate injection in case of manual executor setup (RJR and RPS Pool) and the implementation of forced removal.

4.5.5 REST API endpoints

IVIS server exposes its entities and operations on them via a REST API. We add standard CRUD operations for Job executors in the `/server/routes/rest/job-execs.js` file along with the job-executor-specific forced removal endpoint.

In the `/server/routes/rest/remote-run-push.js` file, we introduce the `/remote/emit`, `/remote/runRequest` and the `/remote/status` endpoints and verification functions. These endpoints should be restricted to remote executors only and are expected to be used by remote executors as described in the Remote Push HTTP interface section 4.4.1. All request handlers in this file expect a

[5] `https://github.com/oracle/oci-typescript-sdk`

certificate serial number supplied in the request headers. In the implementation presented in this thesis, this certificate number injection is configured on the level of the reverse proxy that proxies all requests to the IVIS server. Recall that the certificate serial number is used to restrict requests to make it more difficult for an executor to attempt to modify data on behalf of a different executor.

4.5.6 Task handler modifications

The Task Handler process dispatches the Task and Job Run lifecycle messages and initializes Runs in the IVIS's internal representation. With the addition of remote executors, some mechanisms need to be either bypassed or diverted.

In some places, we detect a remotely-executed Run and follow a new execution path. This includes the handling of a stop message and the handling of a start of a Job Run. In the following 2 paragraphs, we show the reasons for the change in execution flow.

When handling a *local stop* message, the handler, apart from stopping the Run, also issues an event in case the Run is being observed in the IVIS client. This is undesirable as the remote executor itself will issue these emissions when the Job Run actually stops.

When handling the start of a Job Run, similar logic applies. This combined with the basic need to dispatch remote run/stop messages according to the executor type implies the usage of the dispatch functions defined in the `server/services/jobs/remote-machine-handler.js` file.

4.6 Client-side additions

As the existing IVIS UI is quite expressive, no significant additions were needed. Apart from bugfixes [6] and adjustments [7], we only introduce the Job executor entity pages, including listing all executors and the CRUD form for the user. These additions can be located in the `client/src/settings/job-executors` folder.

One notable addition to the form system is the support for a password parameter type. The parameter system is a way to customize the forms and get special input from the user. This system is utilized in some Tasks. For example, when creating a Job, the moving average Task template makes the user enter a number for the mean window size. For the SLURM implementation, we need to store the user's credentials, and thus we add the `password` parameter type.

[6] server-side events URL detection when nonstandard ports are used

[7] `ParamTypes` field label and description translation

4.7 Remote Job Runner application

We recall that RJR enables IVIS to utilize a remote machine to Run its Jobs. The RJR is a Docker application consisting of an Nginx container acting as an HTTPS reverse proxy and a NodeJS application container that implements the RJR's functional requirements. The Nginx proxy is expected to accept only certificates of the supplied IVIS CA.

The following subsections will focus on how the NodeJS application works. When writing RJR, we attempted to mimic the file structure of the IVIS server. Thus, some path patterns seem familiar. All mentioned paths are relative to the RJR project root.

4.7.1 Remote control REST API

The Remote Control REST API serves the IVIS server to command the RJR. It is an HTTP interface, roughly described in the `swagger.yml` file and implemented in the `/src/app-build.js`, `/src/routes/run.js`, and `/src/routes/task.js` files. The implemented endpoints are:

- POST `/run/(run_id)` - starts a Run
- POST `/run/(run_id)/stop` - stops a Run
- GET `/run/(run_id)` - queries a Run's status
- DELETE `/run/(run_id)` - performs cleanup of a Run from the RJR's persistence
- DELETE `/task/(task_id)` - performs cleanup of a Task from the RJR's filesystem

4.7.2 Persistence and data model

The RJR uses a simple SQLite database to store two main data collections.

First, it keeps evidence of all running or finished Runs for the master IVIS instance. The `job_runs` table contains the Run output, error message, and a status field. Second, the data in `task_build_cache` table is used to detect whether a Task (re)build is necessary.

4.7.3 Job Run lifecycle

The RJR implements a subset of the original IVIS server's Job-running functionality. This was done to simplify the implementation process and ensure that a Run is performed as close to the original implementation as possible. This way of implementing the Job runtime management may also come in handy if more Task types and subtypes were to be supported in the future.

Run Manager, Job handlers, and the Task Handler process are defined in the `src/jobs` directory. Most of the implementation is identical to the IVIS server's implementation with minor tweaks regarding reducing work queue message types and adapting the handlers to support remote execution, e.g., reporting events to the IVIS server.

One major addition to the Run lifecycle is the implementation of a *build cache*. In our model, the IVIS server does not signal Task code changes to all remote executors. Instead, the Task code is bundled in the Job run request, and the Task's type, subtype, and code are compared with locally stored data on each executor. The executor decides whether to rebuild the Task (i.e., whether any of the type, subtype, or code has changed) for each Run request.

4.7.4 Python runtime package

For Python support, the IVIS server uses a special package that abstracts the communication between the Run and the IVIS instance. As RJR mimics most of the IVIS server's Job running functionality, most communication of the Run is proxied by the RJR to the IVIS server in the same way the IVIS server reacts to local Run messages.

The only difference is the Elasticsearch instance access. The IVIS server's Python package creates an Elasticsearch connection instance. Using the proxying approach is thus impractical. Since SSL secures the RJR's communication and Elasticsearch supports SSL configuration, we modify the initialization of the Python package. When creating the Elasticsearch connection, we supply the configured SSL certificates. As the client authentication is done by the IVIS HTTPS proxy, we gain RJR verification for free. For the data isolation, it is expected that Job Run inputs are always correct and that Jobs will not attempt to tamper with other Jobs's Elasticsearch data.

4.8 Remote Pool Scheduler application

The Remote Pool Scheduler (RPS) is a simple proxying and routing application. Its purpose is to represent multiple RJR applications in a pool configuration and to act as a communication point for the IVIS server.

It serves as a proxy for the RJRs to communicate with the IVIS server using only one SSL certificate and as a scheduler for the IVIS server. The RPS selects the pool peer to execute the Run, forwards the request to the chosen executor, and keeps a mapping of the Run ID to the correct executor should any other requests regarding that Run come from the IVIS server. Also note that the RPS pool architecture requires only one allocation of a public IP address after all pool peers are set.

4.8.1 Networking and container configuration

In the pool configuration, the security requirements guaranteed by SSL certificates are imposed only on the communication channel between the IVIS server and the RPS. The RPS expects that the pool RJR instances (pool peers) are inaccessible from the Internet and that the RPS has exclusive access to the subnet where the pool peers listen. The RPS's endpoints replace the pool peers' addresses to the IVIS server and Elasticsearch. This centralization simplifies the certificate management: only one executor certificate is required - the certificate the RPS is using to proxy pool peer requests.

RPS exposes four ports. Only one of those ports shall be exposed to the Internet. The rest should be exposed only to the pool peer network. The public port accepts IVIS server's requests and implements the same HTTP interface for Run management as the RJR. The three "peer-only" ports are forwarding ports where each RJR sends its requests intended for the IVIS server's trusted, sandbox and Elasticsearch endpoints.

The proxying architecture uses ports and does not utilize, for example, Docker networking (note that the RPS is attached to the host's network). This is due to configuration issues arising when configuring Docker and a firewall. The required configuration (as expected by the RPS) is currently achievable by modifying the firewall on the RPS and RJR pool peers.

4.9 OCI pool implementation

The OCI Pool Implementation combines the RPS-backed pool and the Oracle Cloud Infrastructure for automated deployment of remote resources. It utilizes the OCI SDK to instrument all OCI-specific components from the networking to the virtual machine running the RJR and RPS applications. The machines and the RJR and RPS applications are configured via SSH. This implementation demonstrates the automation of a homogenous virtual machine pool creation and removal.

To allow the implementation to support careful resource sharing across the

OCI pools, we use the remote executor Type State 4.2.1. We use the state to keep track mainly of the heavily-limited network resources.

Once the OCI API key is configured, this executor type implementation is automatic. The only required user input is the shape specification of the underlying pool machines and the pool size, P for brevity. The implementation creates P virtual machines of the same shape attached to the same (executor-only) subnet of a single VCN (see OCI 2.2.3). $P - 1$ pool machines run only the RJR application, and one selected machine (master peer) runs both the RPS and RJR applications. Figure 4.4 shows the deployment diagram of an OCI Pool and explicitly states the OCI networking elements used and their relation to the OCI pool implementation.

The majority of the implementation of this executor type manages the OCI resources belonging to an executor (a pool). The creation and configuration of the networking resources and the pool peers are implemented in the `server/lib/pools/oci/basic/oci-basic.js` file. This file exports only a minimal interface, interconnecting all the OCI SDK functionality and configuration templates from the `server/lib/pools/oci/basic/rjr-setup.js` file. During the configuration of the target machines, IVIS server instantiates these templates to get configuration commands for execution.

Other specific aspects implemented include the forced removal (4.4.3) and the general type locking (4.2.2) to prevent multiple OCI pool creation requests from racing each other for resources.

4.10 SLURM pool implementation

This section describes the most conceptually divergent implementation of a remote executor. The SLURM cluster is an HPC cluster managed by the SLURM workload management software and can be accessed only via its (in our case, low-powered) frontend node. The nature of mentioned remote executor architectures was that of a permanently-listening server software reacting to IVIS server's requests. This design is no longer possible due to the limitations of the cluster's frontend node, namely the performance, and networking. The SLURM executor is implemented in the `server/lib/pools/slurm/slurm.js` file.

The SLURM executor accesses the cluster frontend node via SSH. It executes a set of scripts injected during executor creation. These scripts perform desired actions, like running and stopping a Run, by invoking the job scheduling tools `srun` and `sbatch` to delegate most of the work to the cluster itself. We heavily rely on the fact that a SLURM job (our running script) may schedule other SLURM jobs.

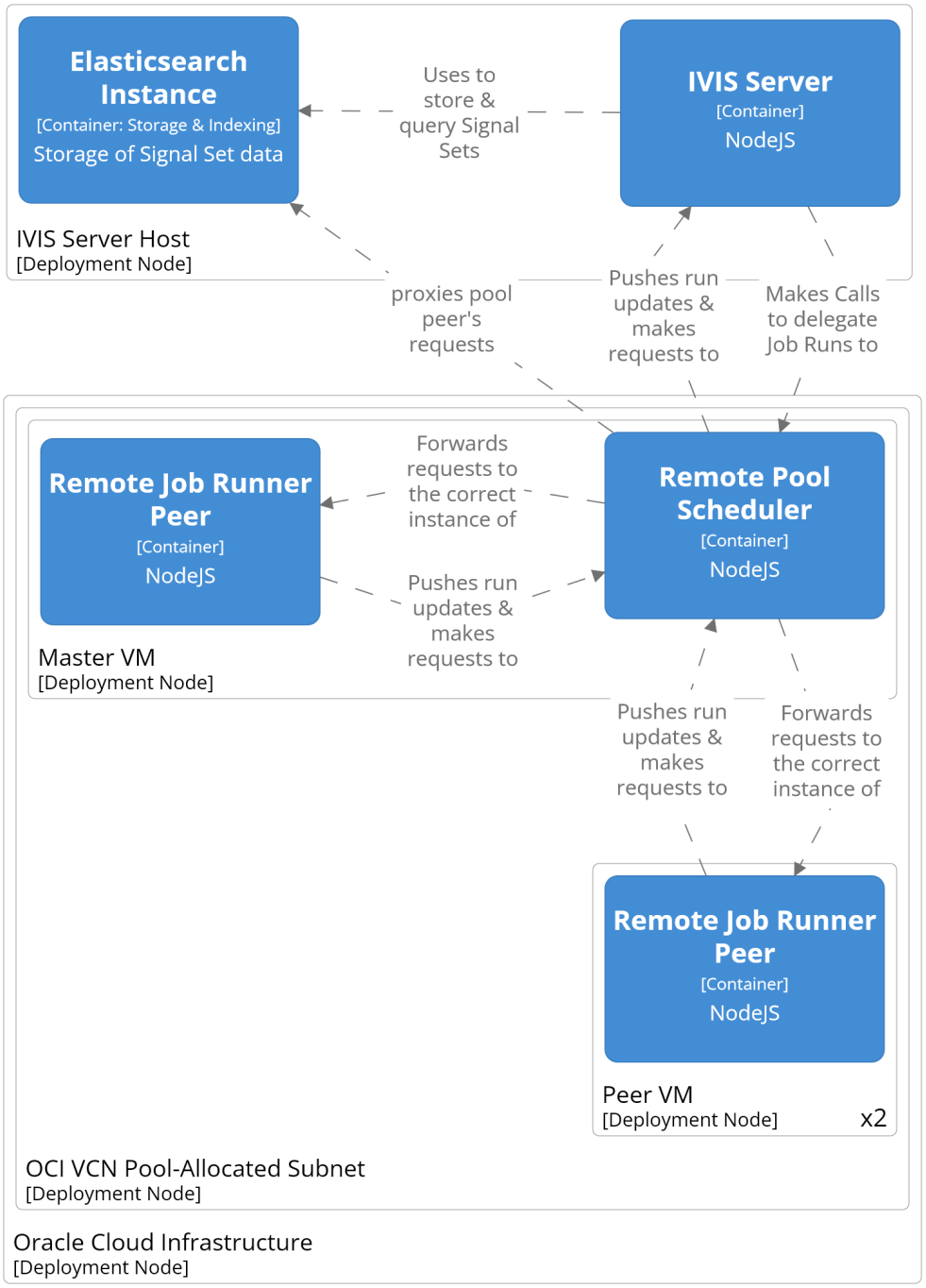

Figure 4.4 Deployment diagram of the OCI Pool (size = 3)

4.10.1 Executor setup on SLURM frontend node

We persist the executor data in the filesystem shared between the frontend node and the cluster. The creation of the SLURM executor on the IVIS server initializes executor-specific folder with subfolder structure for the following purposes:

- the `task` folder for Tasks - their extraction and build step
- the `cert`, certificate, folder
- the `inputs` folder for temporary Run input storage before the Run itself executes
- the `outputs` folder for Run output storage in case the IVIS server does not pick them up immediately
- the `cache` folder for detection of the need to rebuild a Task
- the `utils` folder for scheduling scripts, the IVIS runtime support Python package, and special wrappers

The exact path for each folder and executable is defined in the `server/lib/pools/slurm/paths.js` file. The classes `ExecutorPaths`, `TaskPaths` and `RunPaths` represent executor, Task, and Run-related paths (each parametrized by the respective entity ID). The classes also centralize the knowledge needed to create shell variable expansion strings for scheduling scripts' paths.

Using the path objects, the IVIS server also creates and injects scheduling scripts tailored for the executor's paths. The script structure is ready to be expanded with more Task types and subtypes and can be found in the `server/lib/pools/slurm/scripts.js` file. The file also exports functions for creating the script invocation commands to hide the command complexity.

4.10.2 Task and Run management scripts

Each Task type implements its own initialization and run script. Both the initialization and run steps are almost identical to the IVIS server's initialization and build and run actions. In addition to the expected (IVIS-server-like) workflow, the scripts indicate the success and failure to integrate with other higher-level scripts. Additional scripts perform the removal of all redundant outputs and manage build caching.

In the case of a step failure, the Run fail and build fail scripts inform the IVIS server of the events via an HTTP request. The high-level run and build,

stop, remove, and status scripts implement respective executor actions akin to the Remote Control Interface of the RJR 4.7.1. The server-side's implementation executes those high-level scripts via SSH.

Utilities

Finally, we introduce the "isolated utilities" repository. It is a collection of Task type-specialized Job runtime support utilities. We only need to support the Python Task type. Thus the repository contains the Python IVIS package modified so that the Run requests, usually delegated to the parent process of the Run, are handled by the package's IVIS class since all certificates and data are parsed and available in the IVIS class.

Additionally, there is the "wrapper" script that effectively launches and reports a Run. It does so by launching a subprocess, providing Run input data to the subprocess, and monitoring its outputs. It buffers the outputs and sends them to the IVIS server along with proper event emissions. The wrapper also properly terminates a Run by communicating Run-related events to the IVIS server.

Chapter 5

Results and discussion

From the architecture itself, we should see a significant increase in Job execution throughput. In this section, we will verify this claim and analyze the meaning of the results.

5.1 Evaluation

5.1.1 Methods

We created a simple benchmarking Task to stress the CPU. The Task simulates a high load on the IVIS server (request handling or running too many Jobs).

We intentionally limited the CPU performance available to the IVIS server Docker container by specifying `cpus:0.5` in the compose file. The limitation ensures a more evident CPU bottleneck and creates headroom should the host machine experience additional load. For host CPU usage monitoring, we used Zabbix [1] on another machine and installed the Zabbix Agent software on the IVIS server Host.

The benchmarking Task calculates as many SHA1 hashes in 60 seconds as possible and reports the count, start, and end time to the standard output. This is done ten times.

5.1.2 Data

We first gathered reference data for the IVIS server running only one benchmarking Job. We will state the statistical properties of the sample and, because the score will vary from machine to machine, we shall focus on the *relative ratios* of parallel execution in different scenarios. For two parallel Jobs, we expect roughly

[1] `https://www.zabbix.com/`

Listing 1 Benchamrking script.

```
import hashlib
from datetime import datetime

def bench(inp, seconds):
    startTime = datetime.now()
    print("start: ", startTime)

    counter = 0
    while (datetime.now() - startTime).total_seconds() < seconds:
        sha_1 = hashlib.sha1()
        for i in range(100000):
            sha_1.update(inp)
        sha_1.hexdigest()
        counter += 1
    print("end: ", datetime.now())

    return counter

inp="inputString".encode('utf-8')
batchLengthSecs=60

for i in range(10):
    print(bench(inp, batchLengthSecs))
```

Test Type	Average	Median	Average Deviation	Min	Max
Single Job	737.7	746.5	28.36	699	784
IVIS Parallel 1	376.6	379.5	9.28	358	398
IVIS Parallel 2	368	369.5	6.2	351	376
IVIS & SLURM	786.9	791	14.3	748	810
IVIS & OCI	776.5	774	16.2	753	815
IVIS & RJR	702.3	721.5	46.52	581	757
IVIS & RJR (clean)	731.375	728.5	13.625	712	757

Table 5.1 Benchmark results for Jobs executed on the IVIS server in different scenarios

Test Type	Average	Median	Average Deviation	Min	Max
OCI Single Job	2482.9	2488.5	14.74	2435	2514
OCI Parallel 1	2574.7	2577	6.28	2553	2585
OCI Parallel 2	2546.1	2548	7.26	2512	2558

Table 5.2 Benchmark results for Jobs executed on the OCI pool of size two

a 50% decrease in the benchmark value when running only on the IVIS server. For benchmarks running in parallel, but on *different machines*, we expect *little change* compared to the reference data.

During data gathering, the time gap between parallel jobs did not surpass 2 seconds (which is insignificant given the differences we are looking for) and the overall host machine CPU utilization was always below 25% (leaving enough room for all the processes).

5.1.3 Results

The *IVIS & RJR (clean)* result is the previous result cleared of outlier values *581* and *591*.

We can see that running Jobs can (obviously) overload the IVIS server. Even when comparing the worst-case measurements (that is, the ratio between minimum and maximum) of the Single and Parallel server-only benchmark, we see more than 40% performance drop.

We can also see that the delegation of Runs to other executors, as expected, helps the server maintain the performance level for locally-executed Runs. Due to the the sheer size of the observed ratios, we can conclude that the solution *has substantially increased Job execution throughput.*

The table dedicated to the OCI implementation shows that Job Runs are correctly distributed by the RPS. This fact may also be observed in the logs of the

RPS application.

5.2 Discussion

The execution throughput uplift we showed directly implies better scalability of the framework. Potential applications may include the automation of specialized HPC workloads based on real-time data, allowing GPU workloads in the framework or general expansion of commercial cloud support.

Chapter 6

Conclusion

We successfully extended the IVIS server with a way to execute Jobs both on manually set-up machines and automatically managed resources. We ensured the security and integration with the existing Job-running functionality by introducing the Remote Executor entity and locally-issued certificates.

We also implemented four executor types demonstrating the possibilities of the extension along with two auxiliary applications, the Remote Job Runner and the Remote Pool Scheduler, enabling job execution and machine aggregation, respectively.

The solution allows users to offload Jobs to different machines, bringing more hardware flexibility and performance (by scalability). The solution may be further improved by implementing more complex scheduling and pool management. For example, adding parametrized dynamic machine allocation in the cloud-based executors for cost optimization.

6.1 Future work

6.1.1 IVIS messaging

To further simplify the implementation of remote Executors, the IVIS server and Executors could use a centralized messaging system for both Run status updates and event emissions. This modification would also require careful examination of the existing IVIS codebase to assess the feasibility and possible extension into other parts of the IVIS framework.

6.1.2 More Executor types

We demonstrated support of technologies, ranging from hand-configured machines and automatically-deployed virtual machines to an HPC cluster with

limited frontend capabilities. The solution differentiates between Task types and subtypes. These facts combined should allow Jobs to use specialized hardware. For example, we might implement a Python-CUDA subtype and use an Executor that supports CUDA.

Bibliography

[1] SmartArch & contributors. *ivis-core*. Oct. 4, 2021. URL: https://github.com/smartarch/ivis-core.

[2] *RFC:8446 The Transport Layer Security TLS Protocol Version 1.3*. 2018. URL: https://www.rfc-editor.org/rfc/rfc8446.

[3] *Docker*. 2023. URL: https://www.docker.com/.

[4] *Oracle Cloud Infrastructure*. 2023. URL: https://www.oracle.com/cloud/.

[5] *Slurm workload manager (Overview)*. 2023. URL: https://slurm.schedmd.com/overview.html.

Appendix A

Usage Guide

A.1 Containerized IVIS setup

Read this section and then follow the `ivis-containerized/README.MD` file.

Three DNS A/CNAME records and a certificate for all those domains are required. If we own a domain `example.com`, we can use `ivis.example.com`, `sbox.example.com`, `api.example.com`.

For the Elasticsearch address, we recommend using the same name as for the trusted endpoint (in our example `ivis.example.com`).

In the git cloning steps, moving the entire `ivis-core` folder to `ivis-containerized/ivis-core` is also fine.

A.2 Using OCI pool executors

For OCI credential configuration, see the `ivis-containerized/README.MD` file.

The free `VM.Standard.E2.1.Micro` shape can be used. Be patient. The installation of required packages and prerequisites, including Docker, and the build of the RJR/RPS image may take a long time (especially on the Micro instances). Individual peer setup is done in parallel. SSH command execution may be monitored in the silly log (not enabled by default, to enable, modify `log.level` in the `ivis-containerized/config/ivis/default.yaml` file).

In the case of forced removal, terminate the pool peers in the OCI console and remove the associated subnet from the IVIS VCN.

A.3 Example tasks

Users may define their Tasks' code or use a "wizard" when creating a Task. Unfortunately, only the Moving Average wizard is correct at this point. Even the moving average Task was initially incorrect. However, because the Task uses both Signal Set manipulation and the Job State, we partially corrected it to showcase the functionality. Other Task wizards are incompatible both in terms of package dependencies and even in terms of the IVIS overall Job API.

To use the Moving Average task, please refrain from using the Timestamp Singal set. Create a Generic Signal Set and add an individual timestamp Signal (for example, an integer) and use this Signal as the timestamp Task parameter, as the task code is not adapted for those Signal sets.

For simpler testing, adjustment of the Moving Average Task to print the computed average on the standard output can be suitable - this way, the values will be shown in the Job's log.

A.4 Manual Remote Job Runner and Scheduler deployment

Follow the `ivis-remote-job-runner/README.MD` file. For the port parameter, choose the one from the Docker compose file. Remember to inject the IVIS-supplied certificates. These are available either from the Executor Settings (top right corner) or by clicking the star-shaped icon for the corresponding Executor in the Executors Table.

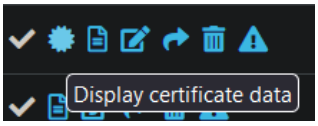

Figure A.1 Executor actions - display Certificates

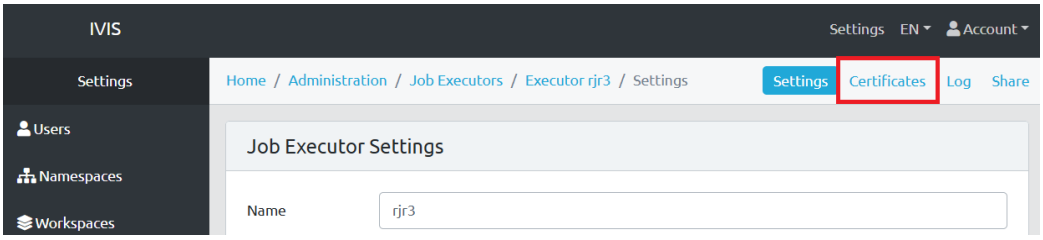

Figure A.2 Executor settings - display Certificates

As a hint, for RJR, usually only the `server_name` directive of the proxy configuration needs to be modified and the `ivisCore`, `useLocalCA`, and `es` sections of the YAML configuration file have to be adjusted.

The remote pool scheduler is supported on the IVIS server. The deployment, however, is much more elaborate than for the RJR. Preconfiguration of the RPS in a similar way to the RJR will be needed. However, it must also be ensured that the accessibility of the RPS ports as stated in the `remote-pool-scheduler/README.MD` file, section `Configuration` to guarantee security.

A.5 SLURM executor job output

For unknown reasons, very short SLURM Executor Jobs do not produce complete output. We suspect this is due to buffering reasons.

A.6 Online repositories

All the code described in this thesis is available on GitHub:

- IVIS-core: `https://github.com/BohdanQQ/ivis-core`
- IVIS-containerized: `https://github.com/BohdanQQ/ivis-containerized`
- RJR: `https://github.com/BohdanQQ/ivis-remote-job-runner`
- RPS: `https://github.com/BohdanQQ/ivis-remote-pool-scheduler`
- Remote Utilities: `https://github.com/BohdanQQ/ivis-isolated-runner-utils`

Appendix B

IVIS UI Screenshots

B.1 Signal Sets

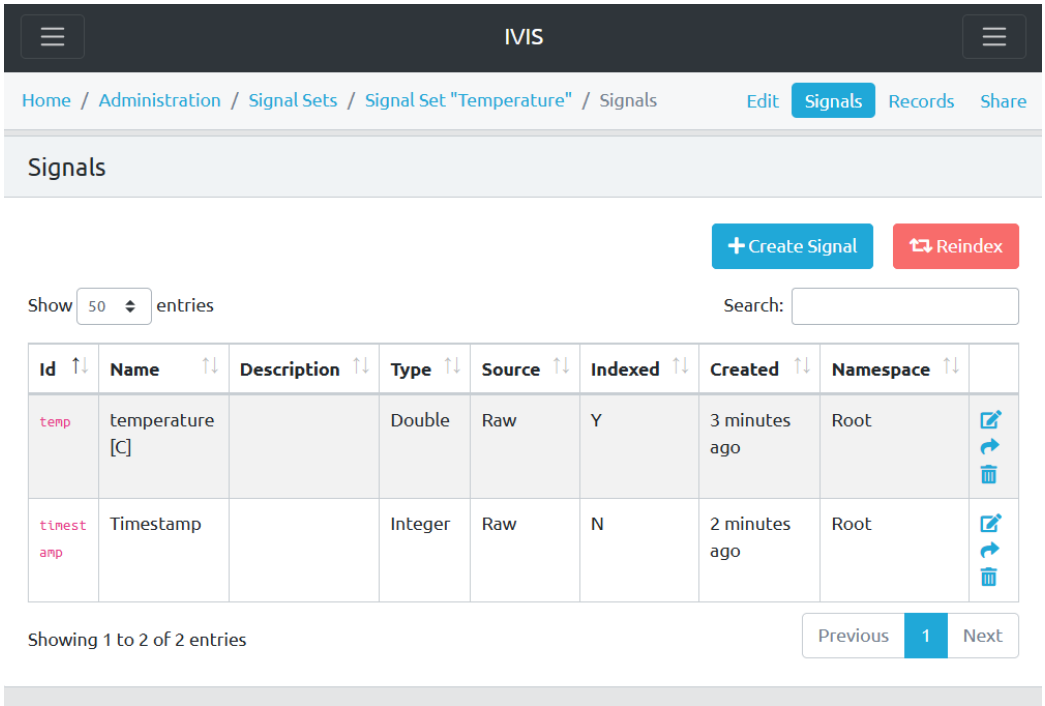

Figure B.1 The Signal Set Signal Table

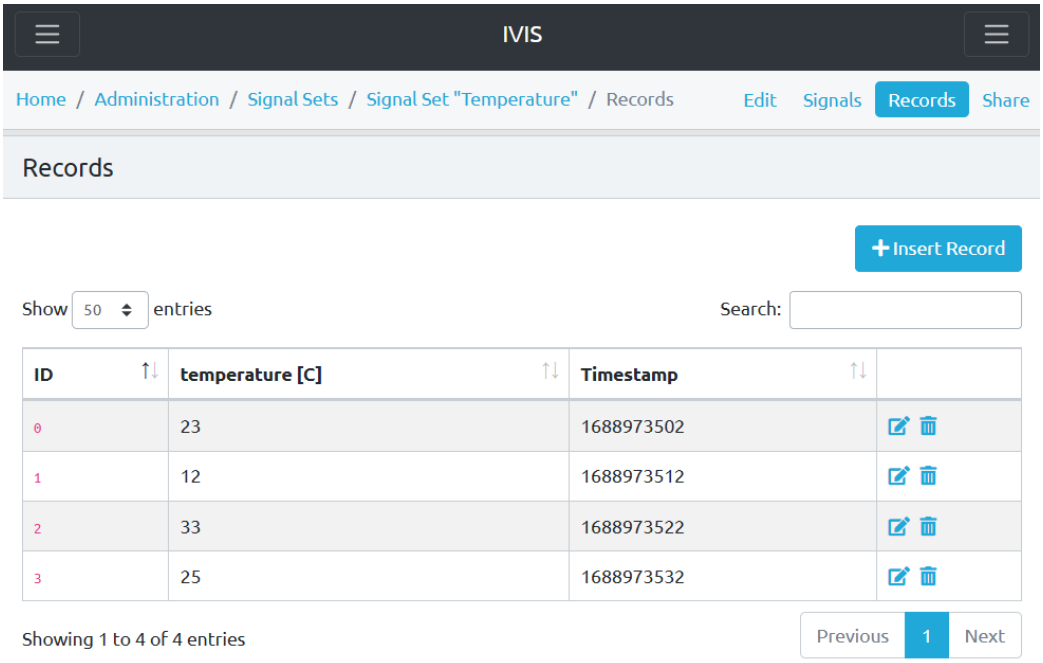

Figure B.2 The Signal Set Records Table filled with example data

B.2 Visualization

For the template definition, we used the example LineChart located in `ivis-core/examples/templates/linechart.js`.

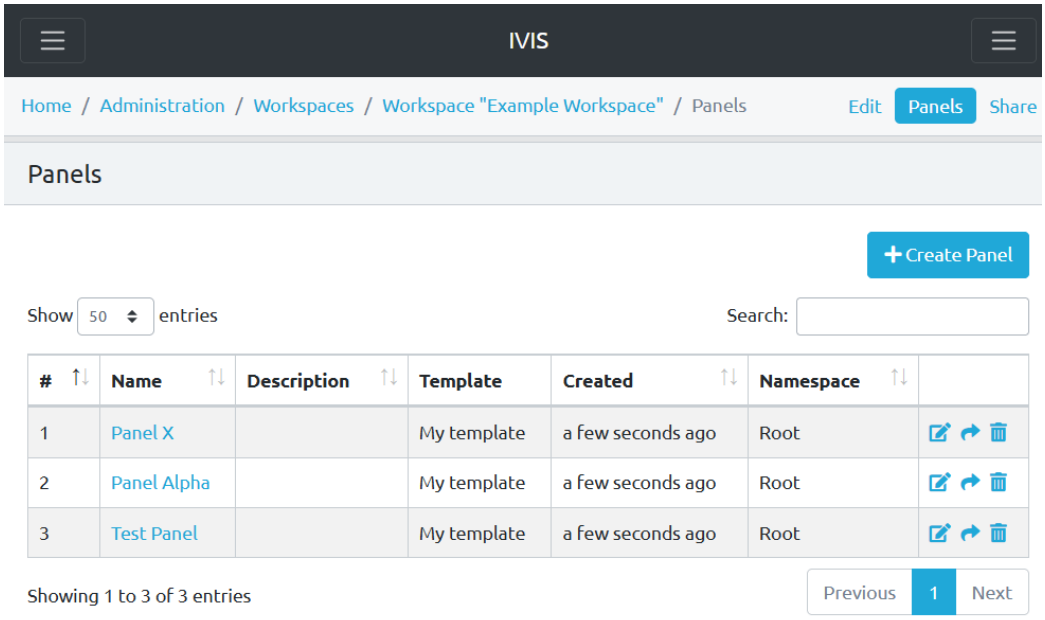

Figure B.3 The Panels Table

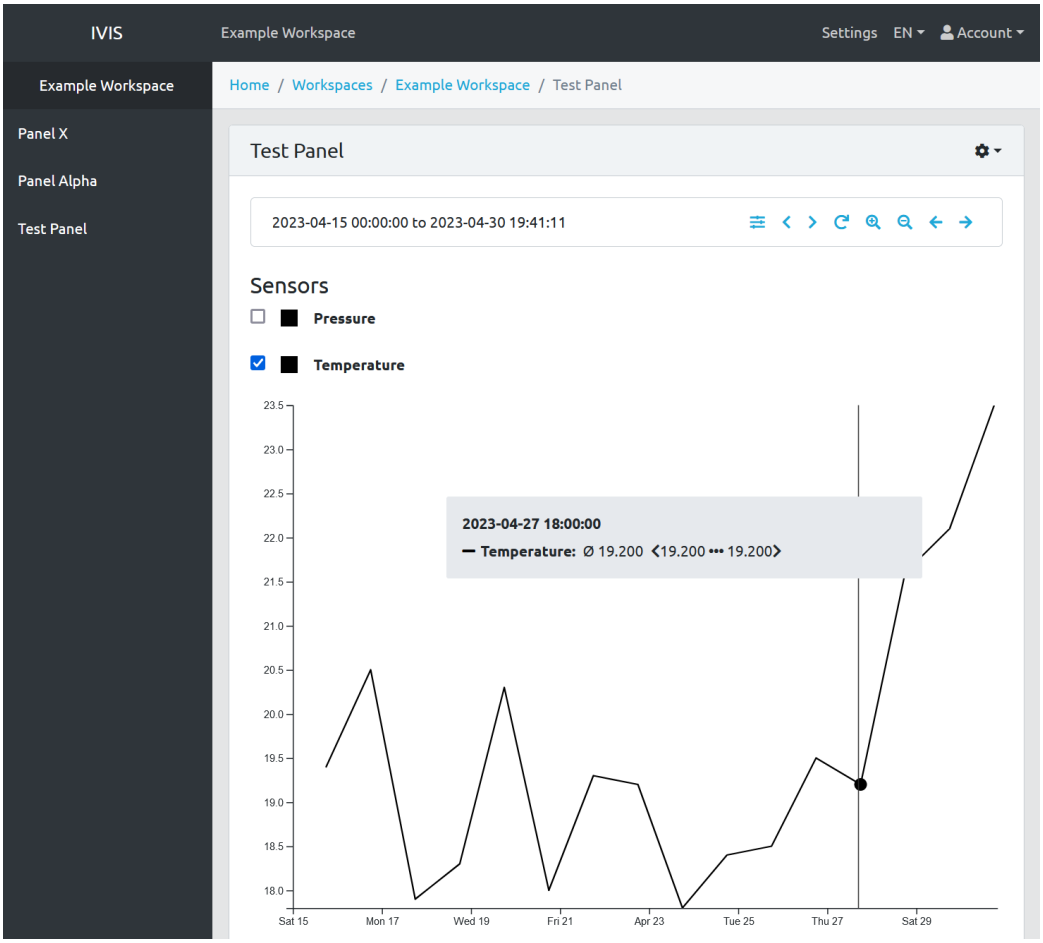

Figure B.4 Workspace Panel displaying a Signal Set visualization

B.3 Tasks and Jobs

In this section, we show screenshots of Task and Job lifecycles. We first create the Moving Average Task using the Moving Average wizard and make modifications to the Task's Python code so that it prints the moving average to its standard output. Then we create a Job with the created Task and supply parameters and run the Job once.

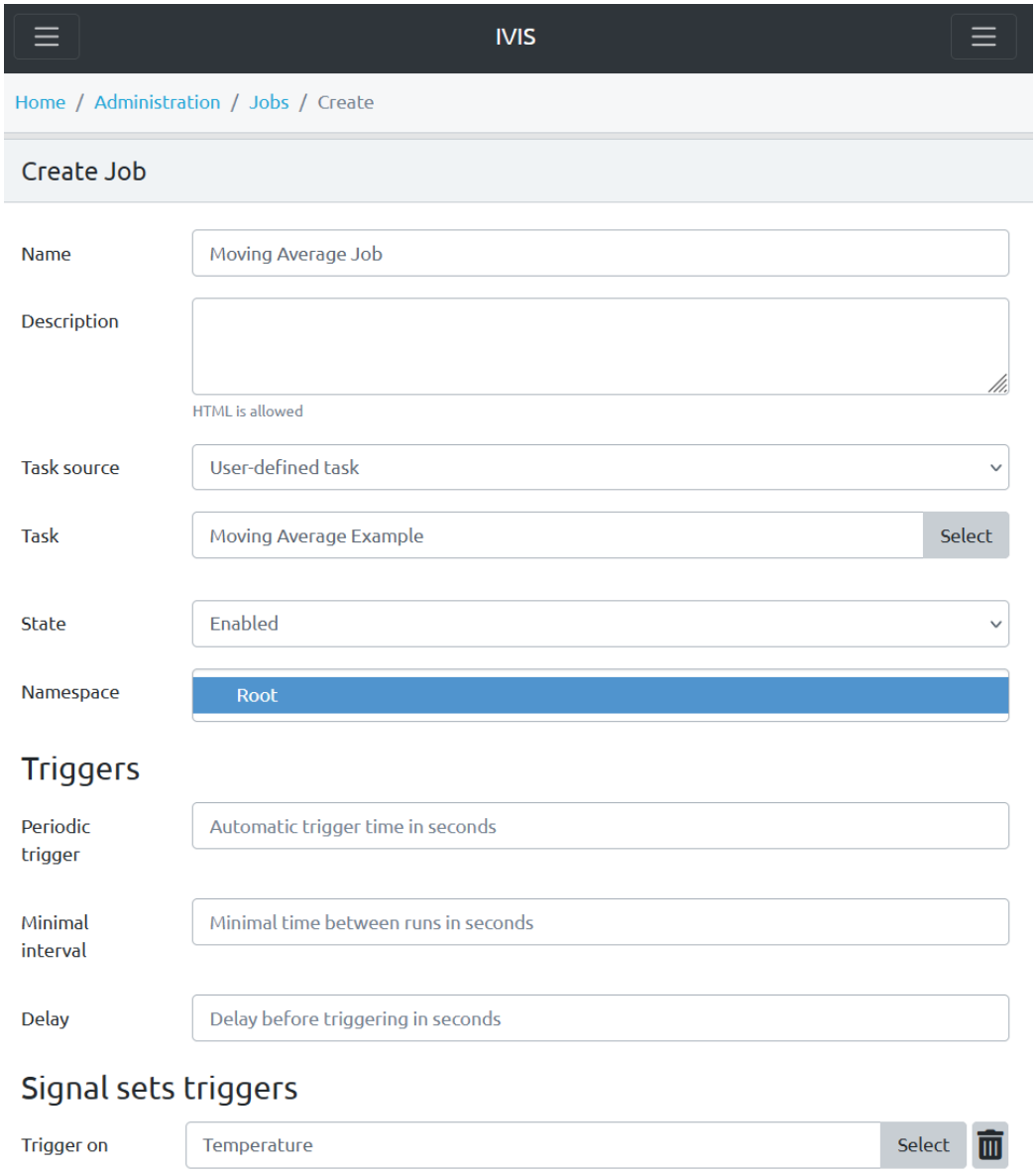

Figure B.5 The Job configuration form - basic parameters, triggers

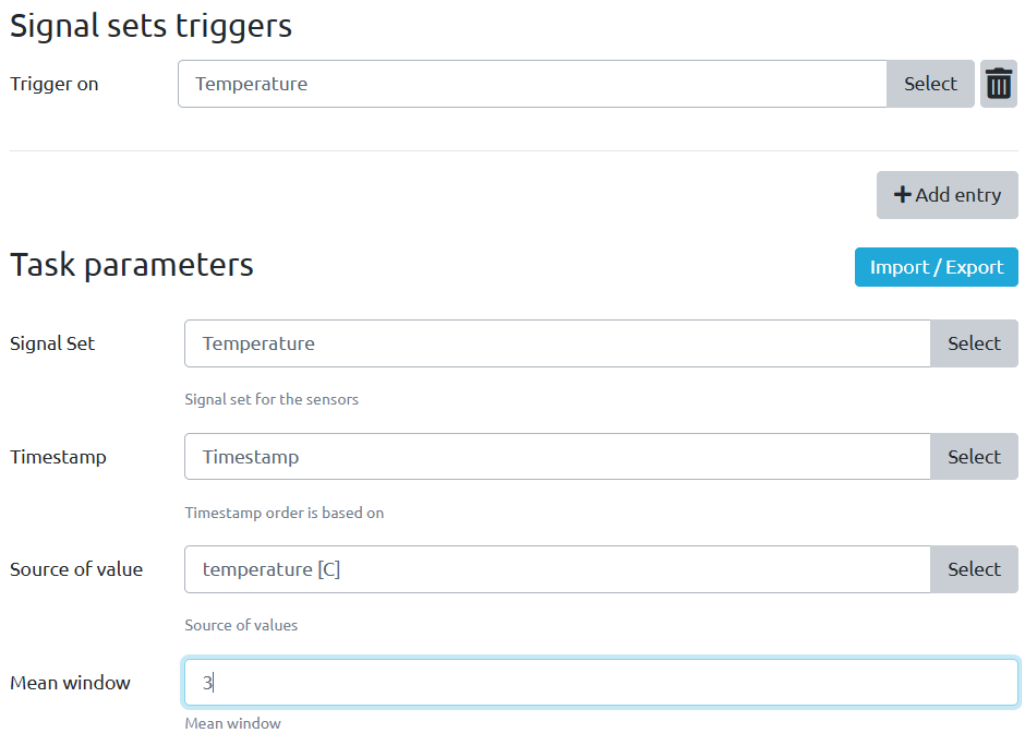

Figure B.6 The Job configuration form - Task parameters

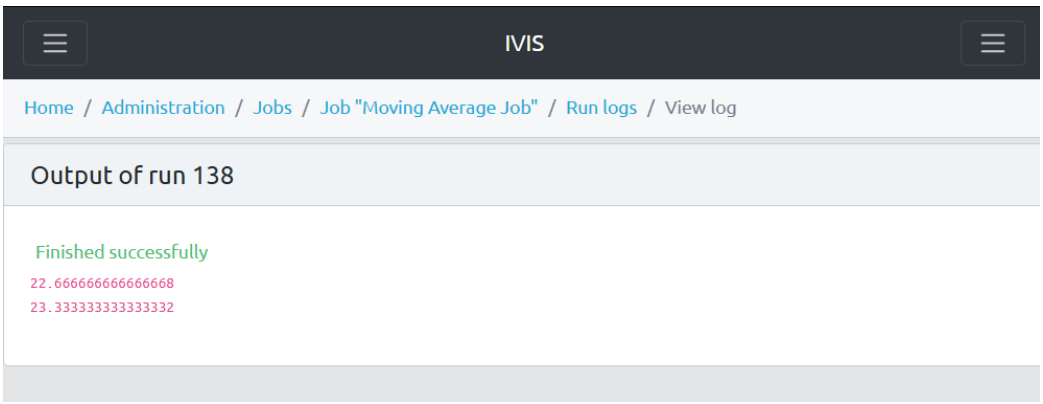

Figure B.7 The Run output of the finished moving average job with values printed out

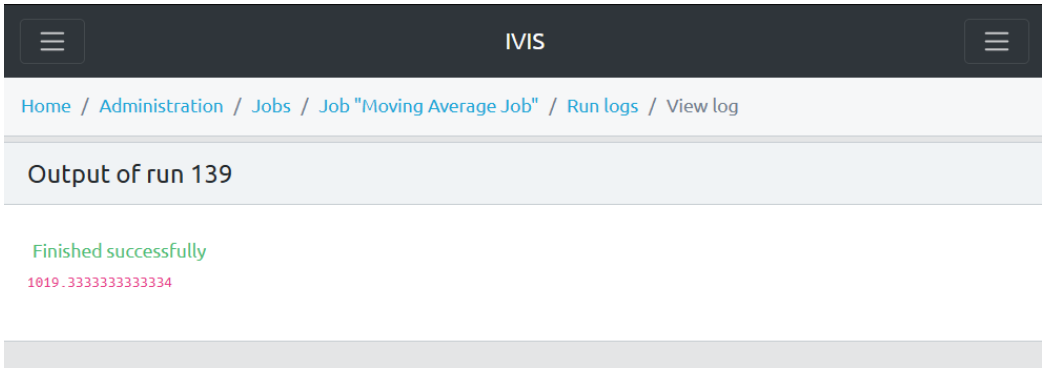

Figure B.8 Output of a triggered Run when the value 3000 was added in a Signal Set Record

B.4 Implementation screenshots

In this section, we showcase the UI of parts of the solution presented in this thesis.

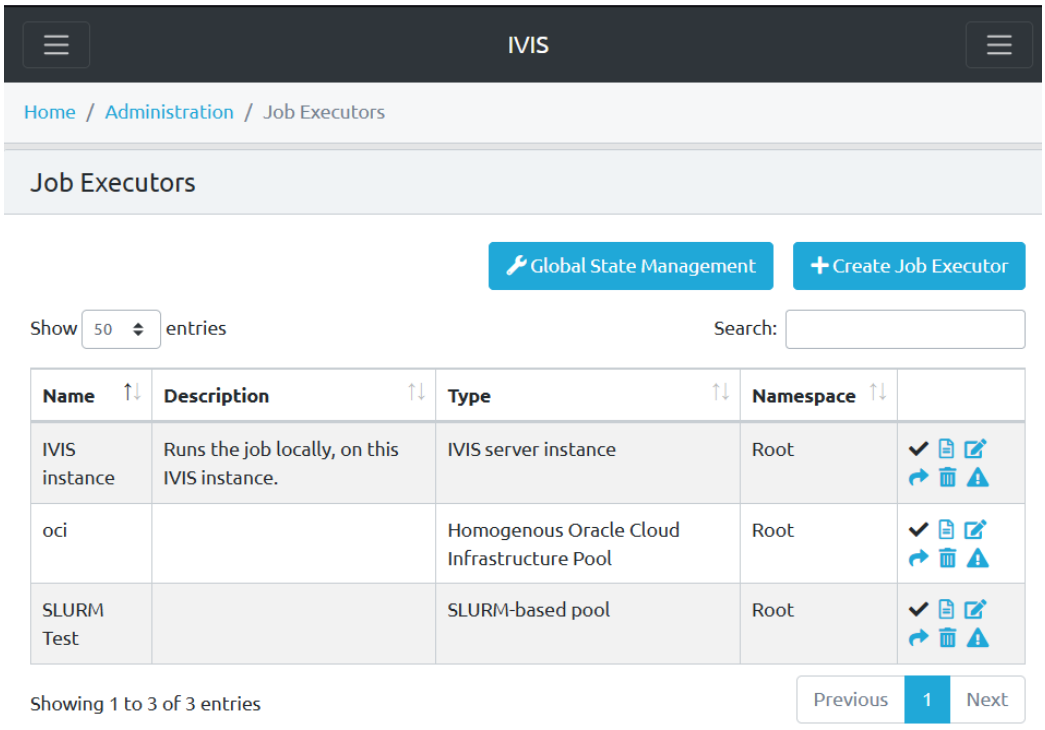

Figure B.9 The Executor table

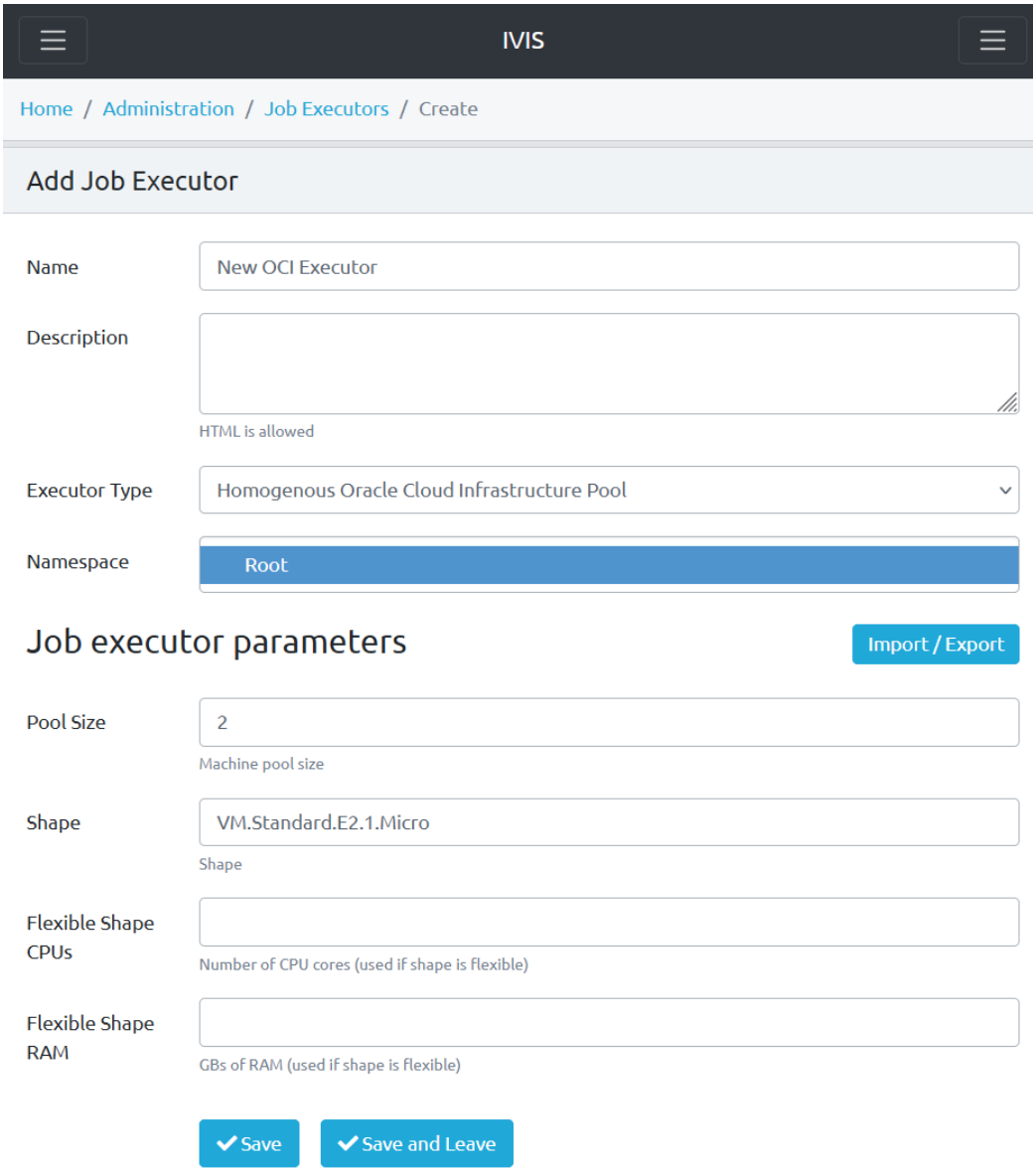

Figure B.10 The OCI Executor creation form

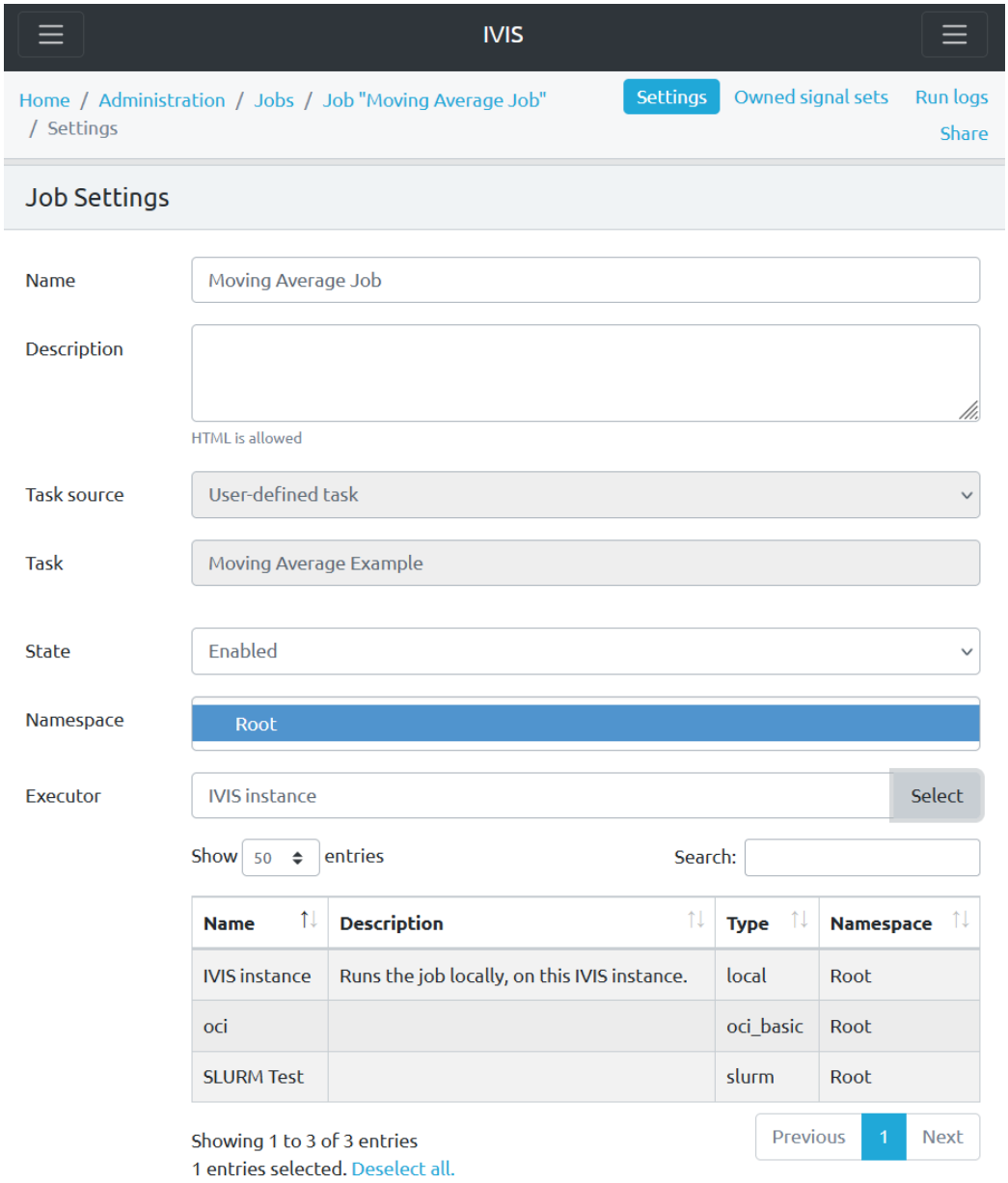

Figure B.11 The Executor selection list in the Job creation form