Brane: A Framework for Programmable Orchestration of Multi-Site Applications

Onno Valkering
Multiscale Networked Systems group
University of Amsterdam
Amsterdam, The Netherlands
o.a.b.valkering@uva.nl

Reginald Cushing
Multiscale Networked Systems group
University of Amsterdam
Amsterdam, The Netherlands
r.s.cushing@uva.nl

Adam Belloum
Multiscale Networked Systems group
University of Amsterdam
Amsterdam, The Netherlands
a.s.z.belloum@uva.nl

***Abstract*—Regardless of the context and rationale, running distributed applications on geographically dispersed IT resources often comes with various technical and organizational challenges. If not addressed appropriately, these challenges may impede development, and in turn, scientific and business innovation. We have developed the Brane framework to support implementers in addressing these challenges. Brane utilizes containerization to encapsulate functionalities as portable building blocks. Through programmability, application orchestration can be expressed using an intuitive domain-specific language. As a result, end-users with limited programming experience are empowered to compose applications by themselves, without having to deal with the underlying technical details. They can do this from user-friendly interactive notebooks. In this paper, we introduce Brane, describe its components and features, and validate the framework with an implementation of a real-world scientific use case.**

***Index Terms*—Containerization, Domain-Specific Languages**

I. Introduction

There are different reasons why scientific and business endeavors must rely on multi-site infrastructures, i.e., the combined use of IT resources across different and typically geographically dispersed infrastructure domains. It might be because a single infrastructure provider cannot entirely fulfill an application's computing and storage requirements [6]. Alternatively, it might be because privacy constraints restrict centralized data aggregation, demanding federated data processing schemes [8]. Regardless of the context and rationale, the use of multi-site often comes with a variety of challenges [7], on top of the inherent requirement for applications to be distributed. Critical technical challenges are, e.g., achieving interoperability between heterogeneous resources, guaranteeing seamless portability of applications, and establishing inter-domain data management. Organizational challenges may also arise due to distributed collaboration with divided responsibilities, i.e., separate work needs to be integrated. The associated complexities of such challenges impede multi-site applications' development compared to applications targeting a single domain. If not addressed appropriately, this may lead to delayed or even missed scientific and business innovation.

This work received funding from the Dutch Science Foundation (NWO) in the Commit2Data program, through the EPI project (grant no. 628.011.028).

This paper introduces Brane[1], a framework that aims to simplify and streamline the development and deployment of complex multi-site applications. To achieve this, Brane provides a barebone system consisting of generic services and components as a starting point for multi-site applications. On top of that, Brane offers a uniform development method based on containerization for extending the initial barebone system. By not focusing on a single application class but rather emphasize the generic aspects and extendability, Brane is usable for myriad multi-site applications. Integration options are available to conveniently incorporate existing source code and/or to utilize optimizations of specialized external systems.

A. Separation of concerns

The development of a multi-site application often touches all layers of the stack [7], simplified here as the application, infrastructure, and network layers. For the infill of each layer, the responsibility typically lies with separate roles. Therefore, we've applied the separation of concerns (SoC) design principle to accommodate various roles with tools that correspond to their usual abstraction level and development method. Currently, Brane accommodates three different roles: system engineer, software engineer, and (data/domain) scientist. Each of these roles can make contributions to multi-site application development. Brane ensures seamless integration between contributions. The engineering roles typically contribute functionality to the infrastructure and networking layers, e.g., data transfers, integrations with external systems, and compute tasks. Such functionality contributions can be combined, as illustrated in Figure 1, into hierarchical routines using an expressive domain-specific language (DSL). Scientists operate in the application layer and contribute to top-level orchestration, which combines engineers' contributions into complex and coherent applications, e.g., data processing pipelines. To express the orchestration logic, scientists use a concise and intuitive DSL. Section III further describes the tools, DSLs, and the technical details underlying this SoC. An implementation following the SoC is provided in section IV.

[1]https://github.com/onnovalkering/brane

978-1-6654-0361-0/21/$31.00 ©2021 IEEE
DOI 10.1109/eScience51609.2021.00056

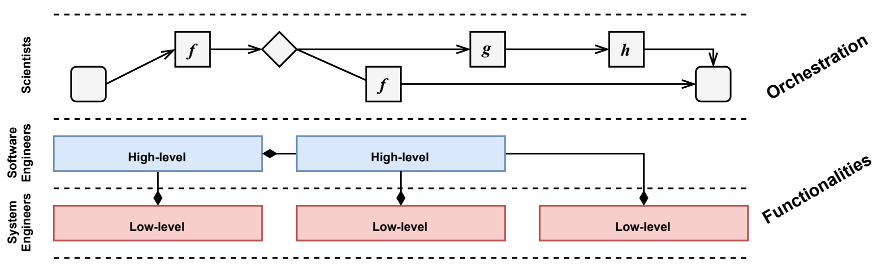

Figure 1. Brane's separation of concerns follows the roles and layers of the technical stack.

II. Related work

A. Function-as-a-Service

Brane's SoC approach is inspired by Cookery [4]. The Cookery framework also offers a DSL to compose applications from separately prepared building blocks, i.e., functionalities. However, Cookery focuses explicitly on cloud services and function-as-a-service (FaaS) [13]. It proposes a method to abstract away from the numerous vendor-specific APIs in order to develop cross-cloud applications. Fly [5] has a similar goal as Cookery, but takes a different approach. It employs source-to-source compilation to support different FaaS platforms and targets multi-cloud [20] deployments and cost-effectiveness.

B. Workflows

Scientific applications classifiable as data processing pipelines, i.e., workflows, are traditionally implemented using a workflow management system (WfMS) [17]. Typically, WfMSs do not allow users to fully control the underlying infrastructure's technical stack [21], making them not ideal for addressing multi-site challenges. Still, this does not mean that WfMSs are obsolete. Their reliability and performance are significant for distributed and on-site computation. The surrounding multi-site orchestration and operations can be performed by Brane, combining the strengths of both systems.

C. Microservices

Microservices [10], i.e. services with a single or minimal set of responsibilities, are used to create scalable and maintainable software architectures. Brane promotes the principles behind microservices to make functionality contributions (Section I-A) fine-grained and reusable. A popular industry practice is to use containers and container cluster managers [2], e.g. Kubernets[2], to deploy and manage microservice architectures. Brane is able to delegate running containers to such systems.

III. Architecture

Conceptually, the Brane framework architecture is composed of two loosely coupled parts: a programming model and a runtime system. The runtime system, by default, is a barebone starting point with only a minimal set of generic functionalities (Section I). The programming model is used to extend the runtime system, i.e., make contributions (Section I-A), in order to satisfy application-specific requirements.

[2]https://kubernetes.io

A. Programming model

The nucleus of Brane's programming model is the concept of packages. A package is the final product of an engineer's effort to encapsulate functionality as an extension for the runtime system. The encapsulation process consists of explicitly describing how Brane can execute the functionality's implementation, which can be arbitrary source code or a compiled binary. This description is in the form of one or more function definitions. Each function definition dictates a different way to execute the implementation. Brane provides a package builder tool[3] that combines the function definitions with the target implementation into a package. Brane also provides two additional package builders that can automatically derive function definitions from implementations that are based on a well-known specification. The first targets workflows written in the CWL specification [1]. This builder is Brane's initial step towards integration with WfMSs (Section II-B). The second builder targets Web APIs, e.g., cloud services and FaaS (Section II-A), and takes OpenAPI[4] specification as input. This builder also automatically generates the corresponding API client implementation to perform the specified HTTP requests.

All packages, regardless of the builder used, are in the OCI[5] image format. This containerization permits packages to be self-contained, i.e., to include all the necessary files and dependencies. Furthermore, all packages contain Brane's `branelet` binary. This binary is the image entrypoint and acts as a proxy between the runtime system and the encapsulated implementation, as illustrated in Figure 2. It uniformly exposes the packages' functions, verifies the input and output, and is responsible for runtime initialization and execution. The characteristics mentioned above qualify packages as standalone and composable building blocks. Uploading a package to the runtime system's registry makes it available to use in applications. Brane provides the tooling for uploading and managing packages. Several packages for generic tasks, e.g., basic mathematical operations and creating files, are built-in.

Brane features two DSLs, Bakery[6] and BraneScript[7]. Both are imperative programming languages and supports basic constructs: variables, objects, conditionals, loops, and functions.

[3]https://onnovalkering.gitbook.io/brane/package-builders/code
[4]https://github.com/OAI/OpenAPI-Specification
[5]https://github.com/opencontainers/image-spec/blob/master/spec.md
[6]https://onnovalkering.gitbook.io/brane/programming/bakery
[7]https://onnovalkering.gitbook.io/brane/programming/branescript

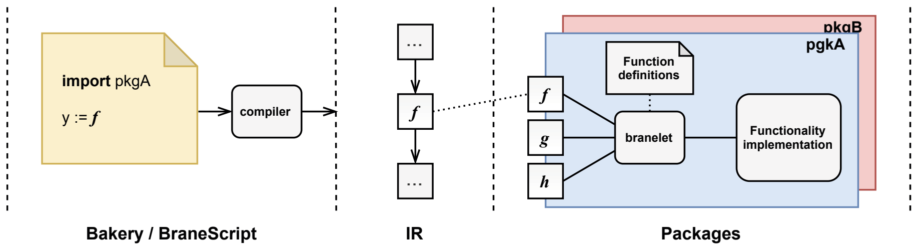

Figure 2. Source code is compiled into an intermediate representation (IR). The IR format preserves the reference to the package(s) that contain the function(s).

In the DSLs, after importing a package, as illustrated in Figure 2, its functions are callable as if they were functions from the DSL itself. The DSLs also facilitate hierarchical functionality. A DSL routine, wrapping one or more packages, can be uploaded to the registry and be treated as a new package.

In the Bakery language, statements follow an English sentence-like structure and typically take up only a single line. This makes Bakery source code easy to reason about and well-suited for scientists with limited programming experience (Section I-A). The Bakery call syntax of each function is part of its definition. It follows a pre-/in-/postfix pattern, e.g., a binary function has the template: `[prefix]` *`arg1`* `[infix]` *`arg2`* `[postfix]`. The rules for specifying the pattern differ based on the number of arguments. This mechanism enables the English sentence-like statements. Moreover, it allows for adaptation to domain jargon, which helps (fellow) scientists to modify and reason about Bakery source code.

BraneScript targets the engineering roles and scientists with sufficient programming experience. It has a C-like syntax and, compared to Bakery, has more advanced constructs, including ones specifically for multi-site applications (Section III-B). Unlike Bakery, the function call syntax is C-like as well, e.g., `fn(arg1, arg2)`. Moreover, DSL native functions, i.e. subroutines, and custom types can be declared in BraneScript.

Before any DSL source code can be executed, it first must be compiled into an intermediate representation (IR), as illustrated in Figure 2. This IR is what the runtime system understands and operates on. The IR functions as a decoupling between the programming model and the runtime system and opens the door for alternative DSL implementations. A type system is in place to ensure consistency between the DSLs, packages, and runtime system. All variables in Brane's programming model must have an associated type. Built-in value types are `boolean`, `integer`, `real`, and `string`. Arrays and custom types, i.e., objects, are supported as well.

The compiler connects to the registry to discover which packages are available in a Brane runtime system. This mechanism avoids the need to download potentially large packages locally when using the DSL to express orchestration logic.

Brane provides two programming interfaces. For the engineering roles, there is a read-eval-print-loop (REPL) [3]. Scientists may use an interactive Jupyter notebook, with support for widgets, which they are likely already familiar with [19].

```
on "location_1" {
    f();
}

on queryLocations("AMS") {
    g();
}
```

Listing 1: The `on` keyword annotates at which infrastructure site any package function (container) must be executed.

```
let results := parallel [
    {
        f();
    },
    on queryLocations("AMS") {
        return h();
    }
];
```

Listing 2: The `parallel` keyword indicates that the following array of closures (code blocks) can be executed in parallel.

B. Multi-site constructs

By default, the runtime system (Section III-C) will determine where it's most appropriate to run a package function, i.e., where to instantiate the corresponding container. To exert more control over this decision, BraneScript offers the `on` keyword. With the `on` keyword, code blocks can be annotated, as shown in Listing 1, to indicate where any scoped package function must be executed. The location can be identified using a constant literal, a variable, or an expression that will be evaluated at runtime, e.g., one or more function calls. Therefore, custom application-specific logic might be implemented to determine which location to use. This is especially useful if the Brane runtime system installation happens to be multi-tenant, i.e., used by multiple distinct applications and users.

The second multi-site specific construct is the `parallel` keyword. When put before an array of closures, i.e., code blocks with lexically captured variables, it indicates that the closures can be executed in parallel. Execution is unordered and the achieved degree of parallelism depends on the availability of resources at runtime. The closures' return values are made available as an aggregated array, as shown in Listing 2.

Combining the `on` and `parallel` keywords enables basic multi-site orchestration, an example is provided in Section IV.

C. Runtime system

The implementation of the runtime system is based on microservices (Section II-C). It comprises multiple fine-grained services that work together to interpret IR (Section III-A) and execute it as an application. Coordination between services is primarily event-driven [7], i.e., services can independently produce and react to events, and backed by Apache Kafka[8]. This design decouples the services and improves the ability to scale. Moreover, the resulting application event logs, stored in an Apache Cassandra database [16], are usable for reproducibility and auditing purposes [14]. The Xenon middleware [18] is employed to abstract away from remote access mechanisms, which might differ per resource, e.g., SSH for cloud virtual machines (VMs) and Slurm [24] for high-performance computing (HPC) clusters. The runtime system delegates the secure storage of credentials and other confidential information to a local HashiCorp Vault[9] instance. Figure 3 provides an overview of the runtime system's services and components.

The runtime system has two modes of operation. It can execute an application based on its complete IR. Alternatively, it can operate open-ended. In this mode, the runtime system sequentially processes chunks of IR for as long as they are received. In the Jupyter interface, a notebook cell corresponds to an IR chunk. This mode can be used for interactive and exploratory computing [9]. In both modes, when the runtime system encounters a function call IR instruction, it will schedule the corresponding package as a container on an appropriate remote resource. It will halt local IR processing until the output is received. Other instructions, e.g., loops and variable manipulations, are handled within the runtime system.

Next, a brief summary of each of the services is given.

1) API: Interaction with the runtime system is primarily done through the API, e.g., uploading and managing packages.

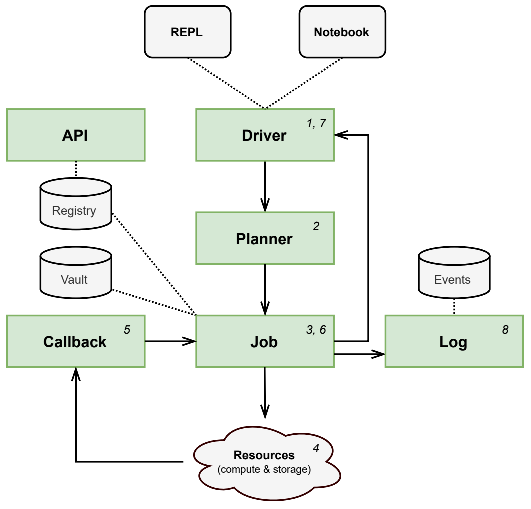

Figure 3. The services and components of the runtime system. The arrows indicate the typical execution loop when execution IR and package functions.

[8]https://kafka.apache.org

[9]https://www.vaultproject.io

2) Driver: The driver service interprets and runs IR and maintains the application state. When it encounters a package function call, it will emit a corresponding function call event.

3) Planner: If the driver emits a function call event without the location specified (section III-B), the planner service will determine the location and add it to the event. Due to the loose coupling, the planner service can easily be replaced by a custom implementation that conforms to the same interface[10].

4) Job: The job service translates complete function call events, i.e., those with a target location specified, to an execution of the corresponding container on a remote resource.

5) Callback: Updates and results from containers running on remote resources can enter the runtime system through the callback service. For every callback an event is generated.

6) Log: The log service constructs application event logs, and exposes them through a real-time) GraphQL [12] API.

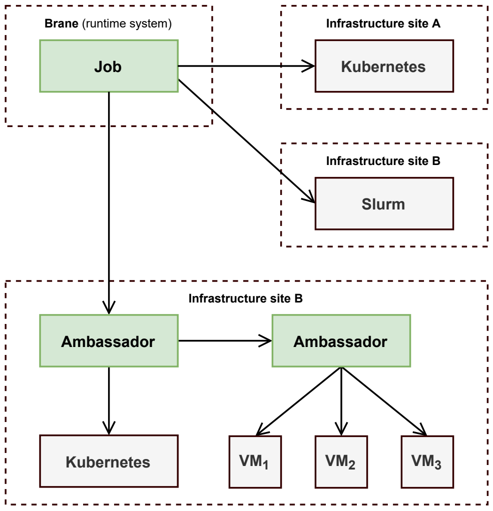

Figure 4. Layer(s) of indirection can be added between Brane and resources.

D. Remote resources

Apart from a compatible access mechanism, the only requirement for remote resources is the ability to run containers. Brane supports Docker and Singularity [15], as well as Kubernetes clusters. The runtime system can automatically convert packages (OCI images) to the proper container image format. Package authors do not have to prepare the conversions in advance. Because of the ability to convert and adapt, choosing a target from a pool of heterogeneous remote resources can be deferred and be programmatically determined during runtime.

By default, Brane requires direct access to remote resources. If this is not desired, a layer of indirection can be added using the ambassador service, as illustrated in Figure 4. With the ambassador service, organizations remain in full control over their infrastructures. It's possible to create custom implementations, as long as it adheres to the well-specified interface. The ambassador service can also be used to segment infrastructure sites, e.g., based on resource costs or desired scheduling rules.

[10]https://github.com/onnovalkering/brane/tree/master/specifications

```
// Brings the relevant functions into scope
import filesystem;
import lofar_lta;
import prefactor;

// ID of the LOFAR observation is the input
observation := 246403;
directory := new_directory;

// Files are staged from tape drives to a cache (remote)
staging := stage observation files;
wait until staging status = "success";

archives := download observation files to directory
measuresets := extract archives to directory;

skymap := calibrate measuresets;
return skymap; // The sky map is the output
```

Listing 3: A LOFAR calibration pipeline in the Bakery DSL.

IV. Application

As a real-world validation, a calibration pipeline from the LOFAR project [11] has been implemented using Brane. It generates sky maps from LOFAR observations, which astronomers use for exploratory research. The pipeline demands specific skills from outside the astronomy domain [22]. Including experience with HPC clusters to meet the pipeline's high hardware requirements. Since software from different packages has to be combined, knowledge of scripting is also required. With Brane, these technical tasks are performed and prepared by engineers with the relevant expertise. They only have to do this once and do not have to worry about compatibility with other engineers' work. Astronomers, i.e., scientists, are then free to compose, modify, and run the pipeline by themselves from the convenience of a JupyterLab notebook (Section III-A). Listing 3 contains the Bakery source code that corresponds to the calibration pipeline. The `observation` variable is the primary input variable. The statements that follow are self-explanatory. It downloads the observation files to a newly created directory, calibrates them, and marks the resulting sky map as output. Credentials are side-loaded from the vault (Section III-C), and the location of the new directory is determined during runtime. These details are hidden, out of convenience, from the astronomer. A detailed walkthrough and demonstration of the calibration pipeline is available online[11].

Listing 4 shows a BraneScript application, also available online[12], that exemplifies the training of a deep learning model using PyTorch's distributed data-parallel (DDP) training[13]. It starts by deploying a master service on `node1` and waits until the service is up and running. Then, in parallel (Section III-B), two worker services are started on `node2` and `node3`. The worker services receive the address of the master node as input. Once the world size is complete, i.e., a total of three nodes are started, PyTorch automatically starts coordinating the training. The application then waits until the master and worker services finish, which indicates training completion.

[11]https://onnovalkering.gitbook.io/brane/demonstrations/lofar
[12]https://github.com/onnovalkering/brane/tree/master/examples/pytorch
[13]https://pytorch.org/tutorials/beginner/dist_overview.html

```
// Brings the relevant functions into scope
import distributed_dl;

let world_size := 3;

on "node1" {
    // The master service is running on node 1
    let master := startMaster(world_size);
    master.waitUntilStarted();

    // After the master service is ready, workers
    // are started, in parallel, on nodes 2 and 3
    parallel [
        on "node2" {
            let w1 := startWorker(
                world_size, 1, master.address
            );

            w1.waitUntilDone();
        },
        on "node3" {
            let w2 := startWorker(
                world_size, 2, master.address
            );

            w2.waitUntilDone();
        }
    ];

    // Let the application run until completion
    master.waitUntilDone();
}
```

Listing 4: A deep learning application in the BraneScript DSL.

V. Future work

There are various opportunities for improving and extending the Brane framework. Additional package builders for well-known specifications, e.g., AsyncAPI[14] and gRPC[15], are possible. Furthermore, advanced multi-site constructs are needed to express more elaborate orchestration scenarios, such as synchronization between computing tasks and error handling. Allowing functions to emit custom status updates and render these as JupyterLab notebook widgets will improve exploratory computing. Further integration with WfMSs might include WfMSs, e.g., Toil [23], as compute targets and the conversion between Brane's DSLs and CWL. The networking layer currently remains underexposed. A distributed file system and multi-domain networking are crucial extensions.

VI. Conclusions

In this paper, we introduced the components and features of the Brane framework. The framework not only provides technical solutions to multi-site challenges but, through the SoC principle, also takes organizational aspects into account. The significant feature of Brane is the ability to convert and adapt locally developed functionalities to run on a variety of heterogeneous resources while guaranteeing seamless integration with other functionalities. Furthermore, scientists are empowered to compose and run applications by themselves and do exploratory research from the convenience of JupyterLab notebooks. Combined, this improves the overall productivity and ultimately speeds up scientific and business innovation.

[14]https://www.asyncapi.com
[15]https://grpc.io

References

[1] Peter Amstutz et al. "Common Workflow Language, v1.0". In: (2016).

[2] David Bernstein. "Containers and cloud: From lxc to docker to kubernetes". In: *IEEE Cloud Computing* 1.3 (2014), pp. 81–84.

[3] L Thomas van Binsbergen et al. "A principled approach to REPL interpreters". In: *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 2020, pp. 84–100.

[4] Mikolaj Branowski and Adam Belloum. "Cookery: A framework for creating data processing pipeline using online services". In: *2018 IEEE 14th International Conference on e-Science (e-Science)*. IEEE. 2018, pp. 368–369.

[5] Gennaro Cordasco et al. "Toward a domain-specific language for scientific workflow-based applications on multicloud system". In: *Concurrency and Computation: Practice and Experience* (2020), e5802.

[6] Reginald Cushing et al. "PROCESS Data Infrastructure and Data Services". In: *Computing and Informatics* 39.4 (2020), pp. 724–756.

[7] Reginald Cushing et al. "Towards a New Paradigm for Programming Scientific Workflows". In: *2019 15th International Conference on eScience (eScience)*. IEEE. 2019, pp. 604–608.

[8] Wenrui Dai et al. "Privacy preserving federated big data analysis". In: *Guide to big data applications*. Springer, 2018, pp. 49–82.

[9] Nicoletta Di Blas et al. "Exploratory computing: a comprehensive approach to data sensemaking". In: *International Journal of Data Science and Analytics* 3.1 (2017), pp. 61–77.

[10] Nicola Dragoni et al. "Microservices: yesterday, today, and tomorrow". In: *Present and ulterior software engineering* (2017), pp. 195–216.

[11] Michael P van Haarlem et al. "LOFAR: The low-frequency array". In: *Astronomy & astrophysics* 556 (2013), A2.

[12] Olaf Hartig and Jorge Pérez. "Semantics and complexity of GraphQL". In: *Proceedings of the 2018 World Wide Web Conference*. 2018, pp. 1155–1164.

[13] Scott Hendrickson et al. "Serverless Computation with OpenLambda". In: *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. 2016.

[14] Mieke Julie Jans, Michael Alles, and Miklos A Vasarhelyi. "Process mining of event logs in auditing: Opportunities and challenges". In: *Available at SSRN 1578912* (2010).

[15] Gregory M Kurtzer, Vanessa Sochat, and Michael W Bauer. "Singularity: Scientific containers for mobility of compute". In: *PloS one* 12.5 (2017), e0177459.

[16] Avinash Lakshman and Prashant Malik. "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[17] Ji Liu et al. "A survey of data-intensive scientific workflow management". In: *Journal of Grid Computing* 13.4 (2015), pp. 457–493.

[18] Jason Maassen et al. *Xenon 1.1.0*. Dec. 2015. DOI: 10.5281/zenodo.35415.

[19] Jeffrey M Perkel. "Why Jupyter is data scientists' computational notebook of choice". In: *Nature* 563.7732 (2018), pp. 145–147.

[20] Dana Petcu. "Multi-cloud: expectations and current approaches". In: *Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds*. 2013, pp. 1–6.

[21] Rafael Ferreira da Silva et al. "A characterization of workflow management systems for extreme-scale applications". In: *Future Generation Computer Systems* 75 (2017), pp. 228–238.

[22] Hanno Spreeuw et al. "Unlocking the LOFAR LTA". In: *2019 15th International Conference on eScience (eScience)*. IEEE. 2019, pp. 467–470.

[23] John Vivian et al. "Toil enables reproducible, open source, big biomedical data analyses". In: *Nature biotechnology* 35.4 (2017), pp. 314–316.

[24] Andy B Yoo, Morris A Jette, and Mark Grondona. "Slurm: Simple Linux Utility for Resource Management". In: *Workshop on job scheduling strategies for parallel processing*. Springer. 2003, pp. 44–60.