On-demand virtual research environments using microservices

Marco Capuccini[1,2], Anders Larsson[3], Matteo Carone[2], Jon Ander Novella[3], Noureddin Sadawi[4], Jianliang Gao[4], Salman Toor[1] and Ola Spjuth[2]

[1] Department of Information Technology, Uppsala University, Uppsala, Sweden
[2] Department of Pharmaceutical Biosciences, Uppsala University, Uppsala, Sweden
[3] National Bioinformatics Infrastructure Sweden, Uppsala University, Uppsala, Sweden
[4] Department of Surgery and Cancer, Imperial College London, London, United Kingdom

ABSTRACT

The computational demands for scientific applications are continuously increasing. The emergence of cloud computing has enabled on-demand resource allocation. However, relying solely on infrastructure as a service does not achieve the degree of flexibility required by the scientific community. Here we present a microservice-oriented methodology, where scientific applications run in a distributed orchestration platform as software containers, referred to as on-demand, virtual research environments. The methodology is vendor agnostic and we provide an open source implementation that supports the major cloud providers, offering scalable management of scientific pipelines. We demonstrate applicability and scalability of our methodology in life science applications, but the methodology is general and can be applied to other scientific domains.

INTRODUCTION

Modern science is increasingly driven by compute and data-intensive processing. Datasets are increasing in size and are not seldom in the range of gigabytes, terabytes or even petabytes and at the same time large-scale computations may require thousands of cores (*Laure & Edlund, 2012*). Accessing adequate e-infrastructure therefore represents a major challenge in science. Further, the need for computing power can vary a lot during the course of a research project and large resources are generally needed only when large-scale computations are being executed (*Lampa et al., 2013*; *Dahlö et al., 2018*). To this extent, moving analyses to cloud resources represents an interesting opportunity from an investment perspective. Indeed, cloud resources come as a configurable virtual infrastructure that can be allocated and released as needed with a pay-per-use pricing model (*Armbrust et al., 2009*). However, this way of procuring resources introduces a layer of complexity that researchers may find hard to cope with; configuring virtual resources requires substantial technical skills (*Weerasiri et al., 2017*) and it is generally a tedious and repetitive task when it is done on demand. Therefore, when running scientific applications

Corresponding author
Marco Capuccini,
marco.capuccini@farmbio.uu.se

on cloud, there is a need for a methodology to aid this process. In addition, to promote sustainability, this methodology should be generally applicable over multiple research domains, hence allowing to compose working environments from established scientific software components.

The idea of allocating composable, on-demand working environments on a "global virtual infrastructure" was envisioned by *Candela, Castelli & Pagano (2013)*. These working environments, which comprehensively serve the needs of a community of practice, are commonly referred to as Virtual Research Environments (VREs). *Roth et al. (2011)* and *Assante et al. (2019)* identify cloud resources as the underlying "global virtual infrastructure" for these systems and provide two similar implementations that offer on-demand allocation of VREs. Both implementations enable to dynamically compose VREs from a collection of scientific applications, which are nevertheless installed directly on Virtual Machines (VMs). Following this approach, due to the remarkably heterogeneous landscape of scientific software packages, one will almost inevitably encounter conflicting dependencies (*Williams et al., 2016*). The technology that has been recently introduced under the umbrella of microservice-oriented architecture (see 'Microservice-oriented architecture and technology') has cleared the way for a remedy to this problem, providing an improved mechanism for isolating scientific software (*Williams et al., 2016*). The idea consists of introducing an engine that leverages on kernel namespaces to isolate applications at runtime. The resulting software environments are lightweight, easy and fast to instantiate, and they are commonly referred to as containers. Noticeable efforts in leveraging this technology to deliver improved VREs were made by the PhenoMeNal project (in medical metabolomics) (*Peters et al., 2018*), by the EXTraS project (in astrophysics) (*D'Agostino et al., 2017*) and by the Square Kilometer Array (SKA) project (in radio astronomy) (*Wu et al., 2017*). However, despite of microservice-oriented applications being consider the gold standard of cloud-native systems, EXTraS and SKA run their VREs on dedicated servers. Here we introduce a general methodology to allocate VREs on demand using cloud resources—which we have also implemented in PhenoMeNal.

The methodology that we introduce in this paper addresses a number of research questions that arise when designing on-demand VREs using microservices. Firstly, allocating virtual infrastructure and setting up the required middleware is hard for non-IT experts. Thus, we face the question of how to provide a seamless allocation procedure for scientists while still enabling a good level of configurability for a specific set up. Secondly, scientists should be able to run VREs on multiple clouds while operating with the same immutable infrastructure and tooling ecosystem. When using public cloud resources, this is challenging due to the heterogeneity of vendor-specific features and tools. Further, it is common in academic settings to leverage commodity clouds that run on premises. While it is important to support these systems as regulations may forbid certain datasets to be handled in public settings, commodity clouds offer a reduced set of features; we face the question of how to enable immutable VREs in commercial and commodity settings. Lastly, we face the question of how to provide VREs that scale reasonably well. To this extent, there are two main aspects that we cover in this paper: (1) scaling of scientific analyses and (2) scaling of VRE instantiation. In connection to this second point, it is important to consider

that our methodology is designed around the idea of on-demand, short-lived deployments; high availability is not crucial while instantiation speed is of great importance.

Based on our methodology, we implemented KubeNow: a comprehensive open-source platform for the instantiation of on-demand VREs. Please notice that we use the term platform as opposed to Platform as a Service (PaaS), because KubeNow comes with a Command-Line Interface (CLI) that operates from the user's workstation—rather than providing a publicly available Application Programming Interface (API). The platform is currently in production as part of the PhenoMeNal project and we employ such use case to demonstrate the applicability of the proposed methodology.

In summary, our key contributions are as follows.

- We introduce a general methodology for on-demand VREs with microservices ('On-Demand VREs with Microservices'). The methodology enables: (1) simplicity in VRE instantiation, (2) VRE allocation over commercial and commodity clouds and (3) scalable execution of scientific pipelines on cloud resources.
- We provide an open source implementation, named KubeNow, that enables instantiating on-demand VREs on the major cloud providers ('Implementation').
- We demonstrate the applicability and the scalability of our methodology by showing use cases and performance metrics from the PhenoMeNal project ('Evaluation'). In connection to our first research question, concerning simplicity, this also contributes in showing how researchers with little IT expertise were able to autonomously allocate multi-node VREs using KubeNow.
- We evaluate the scalability of KubeNow in terms of deployment speed and compare it with a broadly adopted microservice architecture installer ('Deployment automation scalability').

MICROSERVICE-ORIENTED ARCHITECTURE AND TECHNOLOGY

The microservice architecture is a design pattern where complex service-oriented applications are composed of a set of smaller, minimal and complete services (referred to as microservices) (*Thönes, 2015*). Microservices are independently deployable and compatible with one another through language-agnostic APIs, like building blocks. Hence, these blocks can be used in different combinations, according to the use case at hand. This software design promotes interoperability, isolation and separation of concerns, enabling an improved agile process where developers can autonomously develop, test and deliver services.

Software container engines and container orchestration platforms constitute the cutting-edge enabling technology for microservices. This technology enables the encapsulation of software components such that any compliant runtime can execute them with no additional dependencies on any underlying infrastructure (*Open Container Initiative, 2016*). Such software components are referred to as software containers, application containers, or simply containers. Among the open source projects, Docker emerged as the de-facto standard software container engine (*Shimel, 2016*). Along with Docker, Singularity

has also seen considerable adoption by the scientific community as it improves security on high-performance computing systems (*Kurtzer, Sochat & Bauer, 2017*). Even though container engines like Docker and Singularity serve similar purposes as hypervisors, they are substantially different in the way they function. When running a VM, an hypervisor holds both a full copy of an Operating System (OS) and a virtual copy of the required hardware, taking up a considerable amount of system resources (*Vaughan-Nichols, 2016*). In contrast, software container engines leverage on kernel namespaces to provide isolation, thus running containers directly on the host system. This makes containers considerably lighter and faster to instantiate, when compared to VMs. Nevertheless, containers have a stronger coupling with the OS, thus if they get compromised an attacker could get complete access to the host system (*Manu et al., 2016*). Hence, in real-world scenarios a combination of both VMs and containers is probably what most organizations should strive towards.

In current best practices, application containers are used to package and deliver microservices. These containers are then deployed on cloud-based clusters in a highly-available, resilient and possibly geographically disperse manner (*Khan, 2017*). This is where container orchestration frameworks are important as they provide cluster-wide scheduling, continuous deployment, high availability, fault tolerance, overlay networking, service discovery, monitoring and security assurance. Being based on over a decade of Google's experience on container workloads, Kubernetes is the orchestration platform that has collected the largest open source community (*Asay, 2016*). Other notable open source orchestration platforms include *Marathon (2018)*, which is built on top of the Mesos resource manager (*Hindman et al., 2011*), and Swarm which was introduced by Docker (*Naik, 2016*).

ON-DEMAND VRES WITH MICROSERVICES

In this section we introduce the methodology that enables on-demand VREs. The methodology is built around the microservice-oriented architecture, and its companion technology. Here we explain our solution on a high level, thus not in connection to any specific software product or vendor. Later in this paper ('Implementation') we also show an implementation of this methodology that builds on top of widely adopted open source tools and cloud providers.

Architecture

Figure 1 shows a general architecture for on-demand VREs. The architecture is organized in three layers: *Cloud Provider*, *Orchestrator* and *Microservices*. In describing each layer we follow a bottom-up approach.

Cloud Provider

At the lowest level, the *Cloud Provider* layer manages virtual resources at infrastructure level. In our methodology this layer enables to dynamically procure infrastructure when a VRE is instantiated. Physical resources can be outsourced (public cloud), in house (private cloud) or anywhere in between (hybrid cloud).
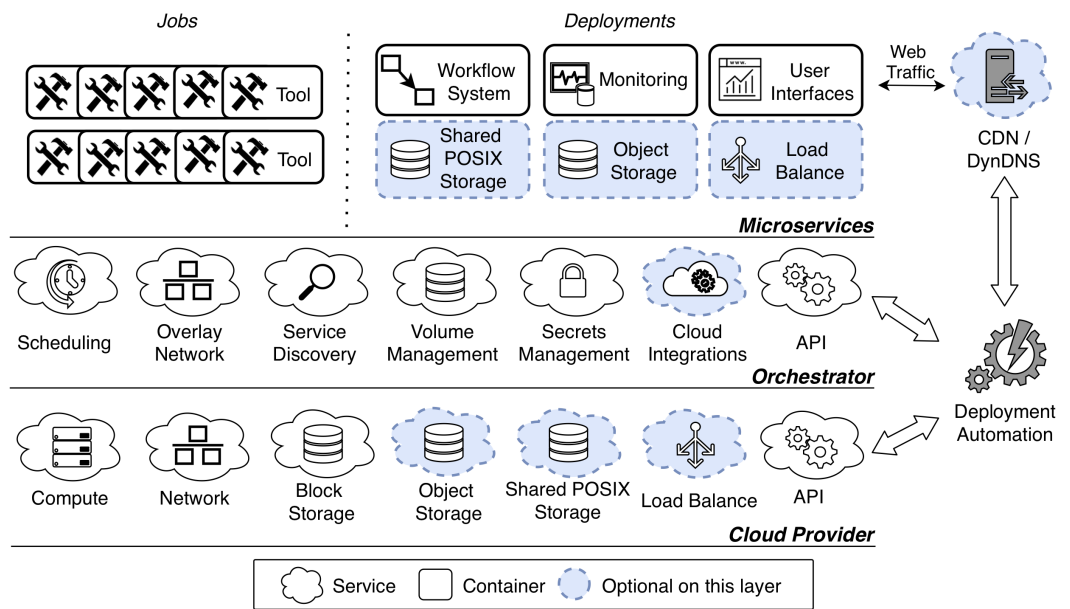
Figure 1 Microservice-oriented architecture for on-demand VREs. The architecture is organized in three layers: *Cloud Provider*, *Orchestrator* and *Microservices*. The two lowest layers offer necessary services to the above layer. In particular the *Cloud Provider* manages virtual resources at infrastructure level, and the *Orchestrator* manages microservices that run as application containers. The uppermost layer run a set of container-based microsrvices for a certain community of practice. The VRE is instantiated through a deployment automation, which may also configure a Content Delivery Network (*CDN*) and a Dynamic Domain Name System (*DynDNS*) to serve the *User Interfaces*.

There are a few necessary services that a cloud system should offer to serve the purpose of a VRE. First, a *Compute* service should enable for booting and managing the VMs that will provide computing power. Second, a *Network* service should provide management for VMs interconnection, routing, security rules and other networking-related concerns. Third, a *Block Storage* service should provide volumes management for VMs. Finally, an *API* should provide programmatic access to the all of the other services (to enable automation).

Apart from these basic requirements, VREs need a few other services that may not be offered by commodity providers (such as moderately sized university installations). Luckily, their implementation as microservices is relatively easy as we describe in 'Microservices'— and it is crucial in commodity settings. First, it is important to point out that the main purpose of VREs is to run computations through scientific tools. These tools can be run dispersively in the virtual cluster, thus needing a shared file space for synchronization and concurrent dataset handling. This cannot be provided via block storage, as usually it does not allow for concurrent access. Concurrent access may be achieved via *Object Storage*, a well-established storage service that is capable of providing shared file spaces (*Karakoyunlu et al., 2013*). As the name suggests the service manages files as objects, thus being substantially different from POSIX storage systems. This may represent a challenge in the context of VREs, as scientific tools can usually only operate on a locally-mounted POSIX space. However, this challenge can be tackled by third party products (such as

Cloudfuse (2019), that can abstract and mount the object storage as a POSIX file system. As an alternative to object storage, some cloud providers recently started to offer *Shared POSIX Storage*, which enables concurrent access on POSIX file spaces. Some examples include *Amazon Elastic File System (2019)*, *Google Cloud Filestore, (2019)*, *Azure NetApp Files (2019)* and *OpenStack Manila (2019)*. Nevertheless, in contrast to object storage, this solution did not yet reach a consensus in terms of implementation and functionalities across different providers.

Finally, a cloud provider may offer a *Load Balance* service. As the name suggests, this service can be used to load balance incoming traffic from a certain public IP to a configurable set of VMs or microservices. In the context of VREs, this can be useful to expose many services under a single public IP (as related quotas may be limited).

Orchestrator

As we mentioned in the introduction, our methodology makes use of application containers to improve the isolation of scientific software environments. A few cloud providers offer native management for container instances (*Amazon Elastic Container Service, 2019*; *Google Cloud Run, 2019*; *Azure Container Instances, 2019*; *OpenStack Zun, 2019*), nevertheless these solution are strongly coupled with vendor-specific tooling and they are seldom supported by commodity cloud systems. Hence, to promote portability of VREs, it is preferable to not rely on container-native cloud environments. However, when leveraging solely on VMs there is no straightforward way to manage disperse containers. This is where the *Orchestrator* is important, as it abstracts VM-based clusters so that containers can be seamlessly scheduled on the underlying resources. There are a few orchestration platforms available in the open source ecosystem (as we discussed in 'Microservice-oriented architecture and technology'), and our methodology is not tied to any of these in particular. However, there are a few services that an *Orchestrator* should offer to support on-demand VREs.

First, a *Scheduling* service should support cluster-wide resource management and scheduling for application containers. This service should also manage container replication across the cluster, and reschedule failed container (possibly to different nodes in case of VM failure). Since containers can be scheduled across many VMs, an *Overlay Network* should provide interconnection among them. In addition, a *Service Discovery* mechanism should provide the means to retrieve container addresses in the overlay network. This usually comes as a DNS service that should only be available inside the cluster.

In order to provide data persistency and synchronization between replicas, a *Volume Management* service should offer container volumes operations across the cluster. This means that containers should be able to access a shared volume, possibly concurrently, from any host. Since this represents a major challenge, on this layer volume management should only represent an abstraction of an underlying storage system, such as a *Block Storage* or a *Shared POSIX Storage*. Apart from file spaces, the *Orchestrator* should be able to manage and mount secrets, such as encryption keys and passwords, in the containers through a *Secret Management* service.

Cloud Integrations may be optionally offered by the orchestrator, and be beneficial in the context of VREs. This service enables to dynamically provision resources on the underlying layer. For instance, on-demand VREs with *Cloud Integrations* may dynamically procure load balancers and cloud volumes for the managed containers. Finally, the *Orchestrator* should provide an *API* to allow programmatic access to its services (enabling automation).

Microservices

The set of services for a certain community of practice run as container-based microservices, on top of the orchestration platform. While we envision the previous layers to be exchangeable between communities of practice, this layer may offer substantially different functionalities, according to the application domain. Luckily, microservices-oriented systems for different scientific domains (e.g., PhenoMeNal, EXTraS and SKA) are very similar in their design, allowing us to give a general overview of this layer.

First, we make a distinction between *jobs* and *deployments*. *Jobs* are mainly application containers that run scientific tools, to perform some analyses. The idea consists of instantiating each processing tool, execute a part of the analysis, and allowing it to exit as soon as the computation is done. In this way the analysis can be divided into smaller blocks and distributed over the cluster.

Deployments should include a *Workflow System*, a *Monitoring Platform* and *User Interfaces*. *Workflow Systems* (or similar analytics services) enable to define and orchestrate distributed pipelines of containerized tools. For the containerized tools scheduling to work, it is crucial that the selected workflow system is compatible with the underlying *Orchestrator*. *Monitoring Systems* collect cluster-wide performance metrics, logs and audit trails, possibly aggregating them in visual dashboards. *User Interfaces* provide graphical access to the workflow and monitoring systems, and possibly enable interactive analysis through the execution of live code. An important remark is that as interfaces are typically stateless, their implementation as functions (*Baldini et al., 2017*) should also be considered when the integration with the workflow systems and the monitoring platform is feasible.

Finally, on this layer *Shared POSIX Storage*, *Object Storage* and *Load Balance* may be implemented as container-based microservices, if not provided by the underlying commodity cloud service. Many available open source projects provide these services and support the major orchestration platforms, thus making the implementation relatively simple (see 'Implementation').

Content delivery network and dynamic domain name system

Content Delivery Networks (*CDNs*) are geographically disperse networks of proxy servers (*Pathan & Buyya, 2007*). The main goal of a CDN is to improve the quality of web services by caching contents close to the end user. Even though this is not particularly beneficial for short-lived systems, modern CDNs offer additional benefits that are relevant for on-demand VREs. In fact, when proxying web traffic, CDNs can provide seamless HTTPS encryption, along with some protection against common attacks (e.g., distributed denial of service). Since modern CDNs can be configured programmatically via APIs, this provides an easy way to setup encryption on-demand. When comparing with Let's Encrypt (*Manousis et al., 2016*), this system has the advantage of seamlessly issuing and storing a single certificate.

This is relevant for on-demand systems, as they may need to be instantiated multiple times in a relatively short period of time, thus making important to reuse existing certificates. In contrast, Let's Encrypt only enables to issue new certificates leaving their management up to the users.

Dynamic Domain Name System (*DynDNS*) is a method that enables automatic DNS records update (*Vixie et al., 1997*). Since on-demand VREs are instantiated dynamically, each instance can potentially expose endpoints on different IP addresses. *DynDNS* enables to automatically configure DNS servers, so that endpoints will always be served on a configurable domain name.

Even though we recommend adoption for user friendliness, *CDNs* and *DynDNS* are optional components. Secure Shell (SSH) tunnelling and virtual private network gateways are valid alternatives to securely access the endpoints. In addition, it is relatively simple to discover dynamically allocated IP addresses by using the cloud API.

Deployment automation

Setting up the presented architecture requires substantial knowledge of the technology, and it may represent a challenge even for a skilled user. Furthermore, for on-demand VREs this time-consuming task needs to be performed for each instantiation. Therefore, on-demand VREs should include a *Deployment Automation*. The automation should operate over multiple layers in the architecture, by setting up the infrastructure through the cloud *API* and by setting up the microservices through the orchestrator *API*. In addition, the automation should also configure the *CDN* and *DynDNS* when required. The deployment automation should be based on broadly adopted contextualization tools. These can be cloud-agnostic, thus supporting many cloud providers, or cloud specific. Cloud-agnostic tools are usually open source, while cloud-specific tools may be licensed. The former has the advantage of generalizing operations over many providers, while the latter might offer commercial support.

No matter which set of contextualization tools is chosen, the deployment automation should offer a common toolbox that operates across all of the supported cloud providers. To this extent, contextualizing the system automatically across multiple commercial and commodity clouds is going to be challenging. For the *Orchestrator* layer one could in principle rely on managed setup automations. However, this approach has the disadvantage of tailoring the orchestration layer to vendor-specific tooling. The same stands when relying on managed storage and load balance. Further, these services are seldom provided by commodity clouds. Therefore, our recommendation is to automate the setup of the orchestration layer without relying on managed services—which also has the advantage of making this layer immutable across providers. Along the same lines, we recommend to automate the setup of storage and load balancing as microservices. This not only gives the user the possibility of deploying this services when not offered by the commodity cloud of choice, but also enables for not tailoring the analyses to any vendor-specific storage system.

IMPLEMENTATION

We provide an open source implementation of our methodology, named KubeNow (*KubeNow GitHub organization, 2019*). KubeNow is generally applicable by design, as it does not explicitly define the uppermost layer in Fig. 1. Instead, KubeNow provides a general mechanism to define the microservices layer, so that communities of practice can build on-demand VREs according to their use cases.

KubeNow is cloud-agnostic, and it supports Amazon Web Services (AWS), Google Cloud Platform (GCP) and Microsoft Azure, which are the biggest public cloud providers in the market (*Bayramusta & Nasir, 2016*), as well as OpenStack (the dominating in-house solution (*Elia et al., 2017*)). This is of great importance in science as it allows to take advantage of pricing options and research grants from different providers, while operating with the same immutable infrastructure. Furthermore, supporting in-house providers enables to process sensitive data, that may not be allowed to leave research centers.

KubeNow implements *Object Storage*, *Shared POSIX Storage* and *Load Balance* in the microservices layer. This is a straightforward solution to maximize the portability of on-demand VREs. In fact, these services may not be available in certain private cloud installations, and their APIs tend to differ substantially across providers (requiring orchestrators and microservices to be aware of the current host cloud). On the other hand, leveraging on cloud-native services may be beneficial in some cases. As an example, using cloud-native storage enables to persist the data on the cloud, even when the on-demand VRE is not running. Thus, KubeNow gives the possibility to skip the provisioning of *Object Storage*, *Shared POSIX Storage* and *Load Balance*, leaving their handling to the communities of practice in such case.

Finally, KubeNow is built as a thin layer on top of broadly-adopted software products. Below follows a summarizing list.

- Docker (*Shimel, 2016*): the open source *de facto* standard container engine.
- Kubernetes (*Asay, 2016*): the orchestration platform that has collected the largest open source community.
- GlusterFS (*GlusterFS, 2019*): an open-source distributed file system that provides both shared POSIX file spaces and object storage.
- Traefik (*Traefik, 2019*): an open-source HTTP reverse proxy and load balancer.
- Cloudflare (*Cloudflare, 2019*): a service that provides CDN and DynDNS.
- Terraform (*Terraform, 2019*): an open-source IaC tool that enables provisioning at infrastructure level.
- Ansible (*Ansible, 2019*): an open-source automation tool that enables provisioning of VMs and Kubernetes.
- Packer (*Packer, 2019*): an open-source packaging tool that enables packaging of immutable VM images.

Configurability

Figure 2 shows a sample KubeNow configuration. In a KubeNow cluster there are four main node entities that can be configured: *master*, *service*, *storage* and *edge*. Apart from
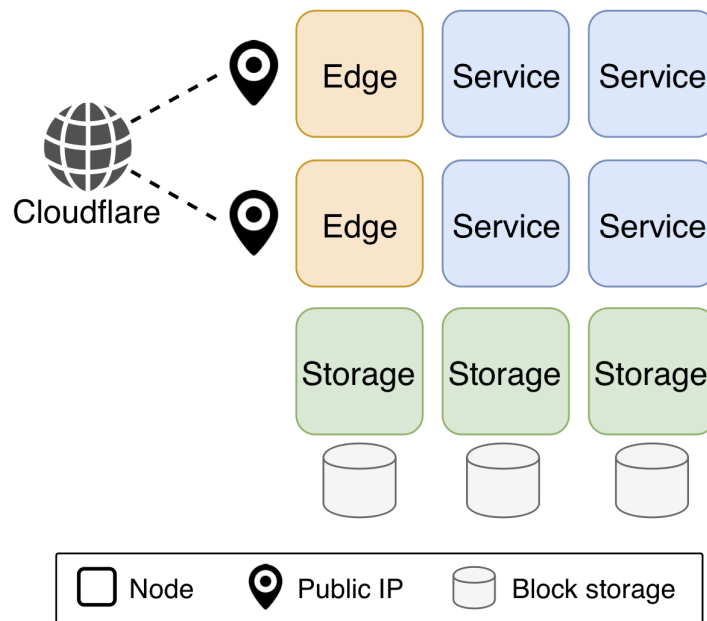
Figure 2 KubeNow sample configuration. There are four main node entities in a KubeNow cluster which are managed via Kubernetes. Apart from the master node, which runs the Kubernetes API, the user can chose how many instances of each node entities to deploy. Service nodes run the user application containers. Storage nodes run GlusterFS, and they attach a block storage volume to provide more capacity. Edge nodes run Traefik to load balance Internet traffic to the application containers, and each of them is associated to a public IP. Further, Cloudflare manages DNS records for the edge nodes IP, and optionally proxies Internet traffic to provide encryption.

the master node, the user can chose how many instances of each node entity to deploy. By default, each node shares the same private network that allows incoming traffic only on SSH, HTTP and HTTPS ports. The master node manages various aspects of the other nodes, retaining the cluster status and running the Kubernetes API. The current implementation of KubeNow does not support multiple master nodes. This is because the purpose of KubeNow is to enable on-demand processing on cloud resources. Under this assumption, deployments are supposed to be short lived, hence high availability is not crucial. Service nodes are general-purpose servers that typically run user containers. Storage nodes run GlusterFS, and they are attached to a block storage volume to provide additional capacity. Finally, edge nodes are service nodes with an associated public IP address, and they act as reverse proxies and load balancers, for the services that are exposed to the Internet. In order to resolve domain names for the exposed services, a wildcard record is configured in the Cloudflare dynamic DNS service (*Cloudflare, 2019*), such that a configurable base domain name will resolve to the edge nodes. In addition, the traffic can be proxied through the Cloudflare servers, using a fully encrypted connection. When operating in this mode Cloudflare provides HTTPS connections to the end user, and it protects against distributed denial of service, customer data breach and malicious bot abuse.

Apart from the typical setting that we show in Fig. 2, some other configurations can be used. Excluding the master node, each node entity is optional and it can be set to any

```
$ kn init <provider> <dir>          $ kn init --preset <preset> \
$ cd <dir>                          $    <provider> <dir>
$ kn apply                          $ cd <dir>
$ kn helm install <package>         $ kn apply
```

      (a) Manual configuration   (b) Preset system

Listing 1: **KubeNow CLI user interaction.** The `init` subcommand sets up a deployment directory for a certain cloud provider. When choosing to configure KubeNow manually the user does not specify any preset and moves to the deployment directory, where some configuration files need to be edited (see Listing 1b). Alternatively, one can chose to initialize the deployment with a preset made available by the community of practice (see Listing 1b). The `apply` subcommand then deploys KubeNow as specified in the configuration files. Lastly, the `helm` subcommand is used to install the application-specific research environment. When using the preset system this last step is not necessary, as the Helm packages that compose the VRE are installed automatically as specified in preset.

replication factor. For instance, when IP addresses are particularly scarce, it is possible to not deploy any edge node, and to use the master node as reverse proxy instead (this may often be the case for commodity cloud settings). The same stands for the storage nodes, that can be removed when an external filesystem is available. In addition, for single-server setups, it is possible to deploy the master node only, and to enable it for service scheduling. Finally, since for entry-level users it can be difficult to reserve a domain name and set it up with Cloudflare, it is possible to use NIP.IO (*NIP.IO, 2019*) instead. NIP.IO provides for an easy mechanism to resolve domain names without needing any configuration (e.g., *foo.10.0.0.1.nip.io* maps to *bar.10.0.0.2.nip.io* maps to *10.0.0.2*, etc.).

Command-line interface

The KubeNow deployment automation is available as a CLI, namely kn, that has the goal of making cloud operations transparent. Indeed, we envision researchers to autonomously set up cloud resources, without performing complex tasks outside their area of expertise. The kn CLI wraps around a Docker image that encapsulates Terraform, Ansible and a few other dependencies, hence Docker is the only client-side requirement. Listing 1a shows a typical user interaction. The user starts by initializing a deployment directory for a certain cloud provider with the `kn init` command. The deployment directory contains some template files that need to be filled in. These files can be used to configure how many of each of the available node entities to deploy (see 'Configurability'), as well as low level parameters such as node flavors, networking and credentials. This way of configuring the deployment hides complex Kubernetes operations that would otherwise be needed to specialize the nodes. Once the user is done with the configurations, the deployment is started by moving into the deployment directory and by running the `kn apply` command. This command sets up Kubernetes as well as the KubeNow infrastrucuture (Fig. 2). Finally, the application-specific research environment is installed on top of KubeNow, by running *Helm (2019)* (the Kubernetes package manager). Even if preparing Kubernetes packages

requires substantial expertise, ready-to-use applications can be made available through Helm repositories.

Listing 1b shows an easier way of deploying a VRE, which trades off configurability. Indeed, configuring the deployment can be hard for inexperienced users. Using the preset system, the user can specify a preset provided by the VRE's community of practice which populates the configuration files with a common setup for the cloud provider of choice. In this way the user only needs to fill in the cloud credentials and optionally make some configuration adjustment. Following this approach, the configuration files also include the Helm packages that need to be installed, thus the `kn apply` command can bring up the complete setup automatically.

Enabling scalable deployments

Enabling fast and scalable deployments is crucial when leveraging cloud infrastructure on-demand. In fact, if the deployment time grows considerably when increasing the number of nodes, the VRE instantiation time likely dominates over the analysis time, making less appealing to invest in large-scale resources.

In order to achieve fast and scalable deployments, there are two main ideas that we introduced in our automation. First, the instances are booted from a preprovisioned image (collaboratively developed via *Travis CI, 2019*). When the image is not present in the cloud user space, the deployment automation imports it, making all of the consecutive deployments considerably faster. Using this approach, all of the required dependencies are already installed in the instances at boot time, without paying for any time-consuming download. The second idea consists in pushing the virtual machines contextualization through *Cloud-init (2019)*, by including a custom script in the instances bootstrap. In this way, the machines configure themselves independently at boot time leading to a better deployment time scaling, when compared to systems where a single workstation coordinates the whole setup process (as we show in 'Evaluation'). This latter approach is even more inefficient when the deployment automation runs outside of the cloud network, which is a quite common scenario.

EVALUATION

We evaluate our methodology using KubeNow as implementation. Being based on Kubernetes, our system benefits from the resilience characteristics provided by the orchestration platform. Resilience in Kubernetes was previously discussed and studied (*Vayghan et al., 2018*; *Netto et al., 2017*; *Javed et al., 2018*) and it is trusted by several organizations (*Neal, 2018*); thus, we do not show a resiliency evaluation here. Instead, we show how the adoption of our methodology enable scientific analysis at scale ('Full analysis scaling'). In particular, we show that running POSIX and object storage as microservices, through KubeNow, offer a scalable synchronization space for parallel scientific pipelines while enabling VREs on commodity clouds—thus validating the design that we show in Fig. 1. Further, we show how KubeNow scales in terms of deployment speed on each cloud provider, also in comparison with a broadly adopted Kubernetes installer ('Deployment automation scalability'). Regarding this last point, it is not our intention to compare the

cloud providers in terms of speed or functionalities, but to show that the deployment scales well on each of them.

Execution of scientific analysis

KubeNow has been adopted by the PhenoMeNal project to enable the instantiation of on-demand VREs (*Peters et al., 2018*). The PhenoMeNal project aims at facilitating large-scale computing for metabolomics, a research field focusing on studying the chemical processes involving metabolites, which constitute the end products of processes that take place in biological cells. Setting up the required middleware manually, when running PhenoMeNal on demand, was originally a complex and repetitive task which made the whole process often infeasible. The adoption of KubeNow has helped the PhenoMeNal community to automate on-demand deployments that now boil down to running a few commands in the resercher's workstation.

On top of KubeNow, the PhenoMeNal VREs run a variety of containerized processing tools as well as three workflow systems, a monitoring platform and various user interfaces. More in detail, the VREs provide Luigi (*Luigi, 2018*), Galaxy (*Goecks et al., 2010)* and Pachyderm (*Pachyderm, 2019*) as workflow systems and the Elasticsearch Fluentd Kibana stack (*Cyvoct, 2018*) as monitoring platform, all of which come with their built-in user interfaces. In addition, PhenoMeNal VREs also provide Jupyter (*Jupyter, 2019*) to enable interactive analysis through a web-based interface.

PhenoMeNal VREs have seen applications in mass spectrometry, nuclear magnetic resonance analyses as well as in fluxomics (*Emami Khoonsari et al., 2018*). Even though these three use cases come from metabolomics studies, they are substantially different and require different tools and pipelining techniques. This suggests that our methodology is generally applicable and supports applications in other research fields.

Parallelization of individual tools

Gao et al. (2019) and *Novella et al. (2018)* used the PhenoMeNal VREs to parallelize three individual metabolomics tools: Batman (*Hao et al., 2012*), FeatureFinderMetabo (*FeatureFinderMetabo, 2018*) and CSI:FingerID (*Duhrkop et al., 2015*). In these two studies different choices were made in terms of infrastructure setup, utilized workflow system and cloud provider. However, in both cases the parallelization was performed by splitting the data into N partitions, where N was also the number of utilized vCPUs, and by assigning each partition to a containerized tool replica. Gao et al. ran their analysis on 2000 1dimensional spectra of blood serum from the MESA consortium (*Bild et al., 2002*; *Karaman et al., 2016*), while Novella et al. processed a large-scale dataset containing 138 mass spectrometry runs from 37 cerebrospinal fluid samples (*Herman et al., 2018*).

In both studies the performance is evaluated in terms of measured speedup when increasing the number of utilized vCPUs. The speedup was computed as T_N/T_1 where T_N is the running time of the parallel implementation on N cores and T_1 is the running time of the containerized tool on single core (measured on the same cloud provider). Gao et al. used the Luigi workflow system to parallelize Batman on Azure and on the EMBL-EBI OpenStack *EMBL-EBI Cloud (2019)* installation. When running on Azure they used 10
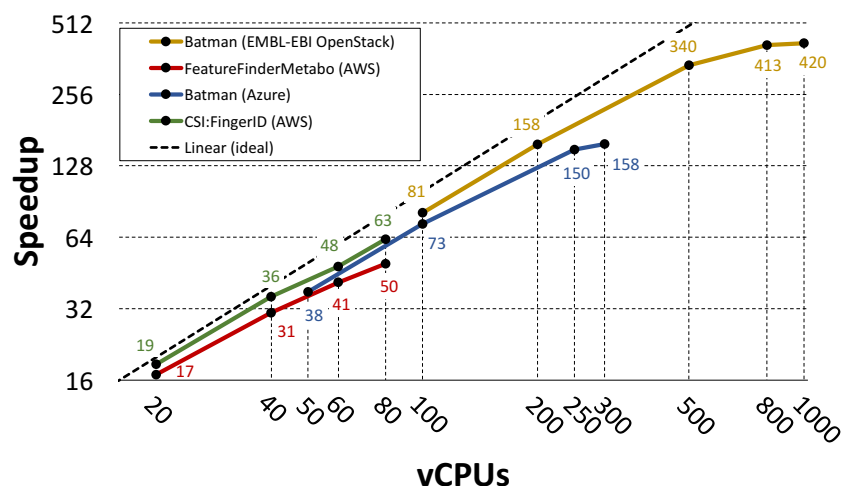
Figure 3 Speedup plot for three containerized tools. The plot shows speedups for three containerized tools that were parallelized using the PhenoMeNal on-demand VRE on different cloud providers. Please notice the logarithmic scale (in base 2) on both axes.

service nodes with 32 vCPUs and 128GB of RAM each, and 1 storage node with 8 vCPUs and 32GB of RAM. On the EMBL-EBI OpenStack they used 55 worker nodes with 22 vCPUs and 36GB of RAM each, and 5 storage nodes with 8 vCPUs and 16GB of RAM each. Under these settings they run on 50, 60, 100, 250 and 300 vCPUs on Azure, and on 100, 200, 500, 800 and 1000 vCPUs on the EMBL-EBI OpenStack.

Novella et al. used the Pachyderm workflow system to parallelize FeatureFinderMetabo and CSI:FingerID on AWS. They run their experiments on AWS, using the *t2.2xlarge* instance flavor (eight vCPUs and 32GB of RAM) for each node in their clusters. They used five service nodes and three storage nodes when running on 20 vCPUs, eight service nodes and four storage nodes when running on 40 vCPUs, 11 service nodes and six storage nodes when running on 60 vCPUs, and 14 service nodes and seven storage nodes when running on 80 vCPUs.

Figure 3 shows the measured speedup for each tool in the referenced studies. Even though these tools differ in terms of CPU and I/O demands, their speedup has a close to linear growth up to 500 vCPUs. For the Batman use case, the speedup starts to level out at 300 vCPUs when running on Azure and at 800 vCPUs when running on the EMBL-EBI OpenStack. However, we point out that Gao et al. used only one storage node when running on Azure, meaning that in such case more I/O contention occurred.

Full analysis scaling

Emami Khoonsari et al. (2018) used the PhenoMeNal VRE to scale the preprocessing pipeline of MTBLS233, one of the largest metabolomics studies available on the Metabolights repository (*Haug et al., 2012*). The dataset consists of 528 mass spectrometry samples from whole cell lysates of human renal proximal tubule cells. This use case is substantially different from the previous benchmarks, as the analysis was composed by several tools chained into a single pipeline, and because the scalability was evaluated over
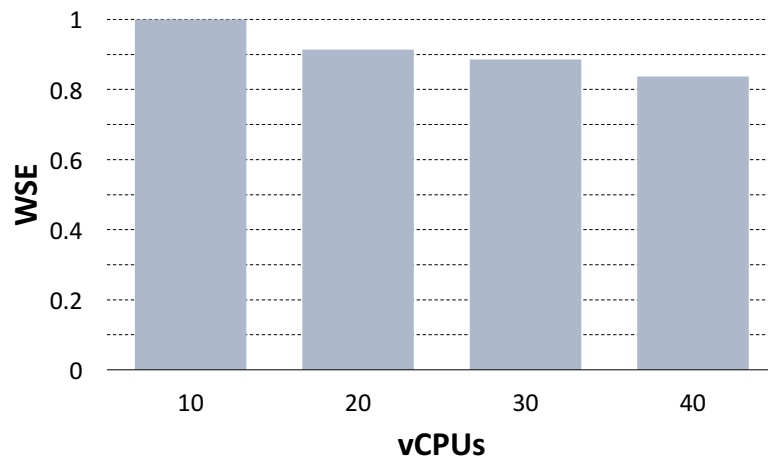
Figure 4 WSE plot for the MTBLS233 pipeline. The plot shows the Weak Scaling Efficiency (WSE) for the MTBLS233 pipeline, executed using the PhenoMeNal on-demand VRE on an OpenStack-based provider.

the full workflow. However, the parallelization was again implemented by assigning a roughly equal split of the data to each container replica. The scalability of the pipeline was evaluated by computing the Weak Scaling Efficiency (WSE) when increasing the number of utilized vCPUs.

The pipeline was implemented using the Luigi workflow system on the SNIC Science Cloud (SSC) (*Toor et al., 2017*), an OpenStack-based provider, using the same instance flavor with 8 vCPUs and 16GB of RAM for each node in the cluster. To compute the WSE, the analysis was repeatedly run on 1/4 of the dataset (10 vCPUs), 2/4 of the dataset (20 vCPUs), 3/4 of the dataset (30 vCPUs) and on the full dataset (40 vCPUs). Then, for $N = 10, 20, 30, 40$ the WSE was computed as T_{10}/T_N where T_{10} was the measured running time on 10 vCPUs and T_N was the measured running time on N vCPUs. Figure 4 shows the WSE measures. There was a slight loss in terms of WSE when increasing the vCPUs, however at full regimen the Khoonsari et al. measured a WSE of 0.83 indicating good scalability. The loss in WSE is due to growing network contention when increasing the dataset size. This problem can be mitigated by implementing locality-aware scheduling for containers (*Zhao, Mohamed & Ludwig, 2018*), and we leave this as future work.

Deployment automation scalability

In order to evaluate how KubeNow deployment automation scales over different cluster sizes, we measured and analyzed its deployment time for each of the supported cloud providers: AWS (Frankfurt region), Azure (Netherlands region), GCP (Belgium region) and OpenStack (provided by *EMBL-EBI Cloud (2019)* and located in the United Kingdom). Then, where applicable, we repeated the measurements using *Kubespray (2019)*, a broadly-adopted Kubernetes cloud installer, to make a comparison. The experiments were carried out from a local laptop, thus envisioning the common scenario where a researcher needs to set up a one-off cluster, in a remote cloud project. More specifically, the laptop was

an Apple MacBook Pro (model A1706 EMC 3071) running on the Uppsala University network (Sweden). We measured time for multiple instantiations on the supported cloud providers, doubling the size for each cluster instance. Apart from the size, each cluster had the same topology: one master node (configured to act as edge), and a 5-to-3 ratio between service nodes and storage nodes. This service-to-storage ratio was shown to provide good performance, in terms of distributed data processing, in our previous study (*Emami Khoonsari et al., 2018*). Hence, we started with a cluster setup that included one master node, five service nodes and three storage nodes (eight nodes in total, excluding master) and, by doubling size on each run, we scaled up to 1 master node, 40 service nodes and 24 storage nodes (64 nodes in total, excluding master). For each of these setups we repeated the measurement five times, to consider deployment time fluctuations for identical clusters. Finally, the flavors used for the nodes were: t2.medium on AWS, Standard_DS2_v2 on Microsoft Azure, n1-standard-2 on GCP, and s1.modest on EMBL-EBI OpenStack.

Comparison between KubeNow and Kubespray

To make the comparison as fair as possible, we used the Kubespray deployment automation that is based on Ansible and Terraform (the same tools that are used in KubeNow), which uses a bastion node to enable the provisioning with a single IP address. It is worth repeating that public address scarcity is a common issue when dealing with commodity cloud installations, hence we tried to minimize their usage in our experiments. For large deployments, the Kubespray documentation recommends to increase the default maximum parallelism in Ansible and Terraform. Since in our experiments we planned to provision up to 64 nodes, we set the maximum parallelism to this value for both KubeNow and Kubespray. To the best of our knowledge, Kubespray makes storage nodes available only for OpenStack deployments, hence the comparison was possible only on the EMBL-EBI OpenStack provider. Figure 5 shows the results for KubeNow and Kubespray in comparison.

Deployment time fluctuations for repeated runs, with the same cluster size, were not significant. However, there is a significant difference in terms of scalability between the two systems. In fact, we observe that Kubespray deployments scale poorly, as they increase in time by a large factor when the cluster size doubles. On the other hand, when doubling the number of nodes, KubeNow time increases by a considerably smaller factor, thus providing better scalability. The gap between the two systems becomes of bigger impact as the deployments increase in size. In fact, for the biggest deployment (64 nodes) KubeNow is ∼12 times faster than Kubespray.

To understand why such a big difference occurs, it is important to highlight how the deployment automation differs in the two systems. Kubespray initiates deployments from vanilla images, and it orchestrates the installation from a single Ansible script that runs in the user workstation (outside of the cloud network). Provisioning vanilla images is not only more time consuming, but it also causes more and more machines to pull packages from the same network as the deployments increase in size, impacting scalability. In the same way, the central Ansible provisioner that orchestrates Kubespray's deployments becomes slower and slower in pushing configurations as the number of nodes increases. As we mentioned
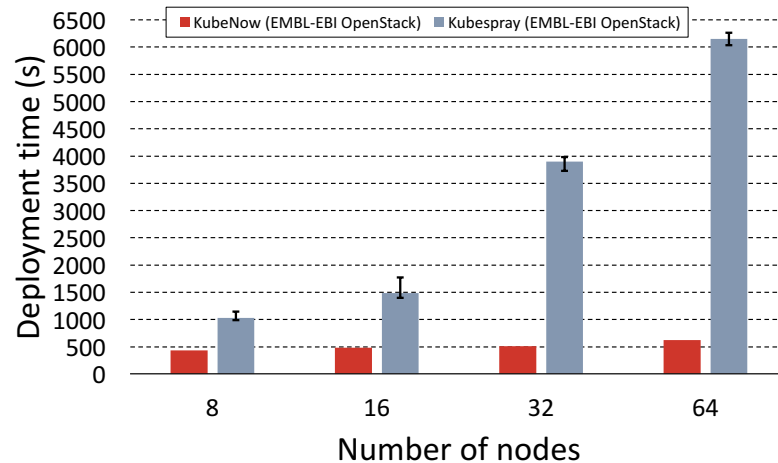
Figure 5 **KubeNow and Kubespray deployment time comparison.** The plot shows the deployment time, for different cluster sizes (number of nodes), when using KubeNow and when using Kubespray. The experiments were performed on the EMBL-EBI OpenStack. Error bars for KubeNow can be seen on Fig. 6.
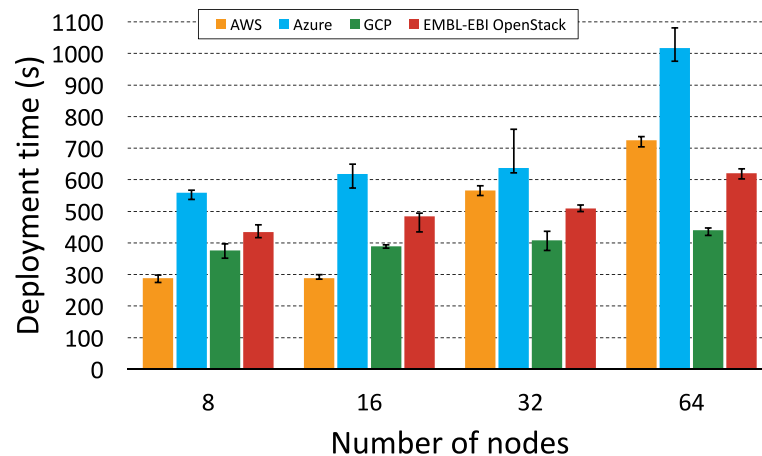
Figure 6 **KubeNow deployment time by cloud provider.** The plot shows the deployment time for different cluster sizes (number of nodes) on each of the supported cloud providers.

earlier, KubeNow solves these problems by starting deployments from a preprovisioned image, and by decentralizing the dynamic configuration through cloud-init.

Evaluation on multiple cloud providers

Figure 6 aims to highlight interesting differences in KubeNow's deployment scaling over different cloud providers. Again, deployment time fluctuations for repeated runs, with the same cluster size, were not significant. We got the best scaling on GCP and EMBL-EBI OpenStack, where every time we doubled the number of provisioned nodes we measured a considerably small increase in deployment time. When deploying on Azure, we always measured a slightly longer time than on the other providers, which increased by a relatively

small constant up to 32 nodes. However, when we increased the number of nodes to 64, the deployment time on Azure almost doubled. Finally, on AWS deployment time was better than on the other providers for small clusters (8 and 16 nodes). However, when provisioning 32 and 64 nodes, AWS time increased by a larger factor, and it almost doubled when we scaled from 16 to 32 nodes.

When provisioning on different cloud providers, KubeNow uses the same deployment strategy, which consists in creating the infrastructure with Terraform, and in waiting for the decentralized dynamic configuration to be completed on each node. The same Ansible contextualization is then applied to make small adjustments in the deployment, on every cloud provider. Since the deployment strategy is not cloud-specific, differences in deployment time among clouds are due to the infrastructure layer, which is managed independently by the providers. Finally, it is important to point out that cloud providers can make changes in the infrastructure layer, impacting the results that we show in this study.

DISCUSSION

The presented methodology differs from the state of the art, as it makes use of the microservice-oriented architecture to deliver on-demand VREs to scientists. This improves isolation of VREs components, and enables to assemble workflows of highly-compartmentalized software components through the adoption of application containers. Achieving scalability by using VMs as isolation mechanism would otherwise be unfeasible, due to the overhead introduced by the guest operating systems.

The implementation for our methodology, namely KubeNow, has been adopted by PhenoMeNal: a live European collaboration in medical metabolomics. Various partners in PhenoMeNal successfully deployed and leveraged KubeNow-based VREs on the major public cloud providers as well as on national-scale OpenStack installations, including those provided by EMBL-EBI (*EMBL-EBI Cloud, 2019*), de.NBI (*de.NBI cloud, 2019*), SNIC (*Toor et al., 2017*), CSC (*CSC cloud, 2019*) and CityCloud (*CityCloud, 2019*). By referring to use cases in PhenoMeNal, we have shown the ability of our methodology to scale scientific data processing, both in terms of individual tool parallelization ('Parallelization of individual tools') and complete analysis scaling ('Full analysis scaling'). It is important to point out that since the analyses are fully defined via workflow languages, the pipelines are intrinsically well documented and, by using KubeNow and PhenoMeNal-provided container images, any scientist can reproduce the results on any of the supported cloud providers.

When comparing KubeNow with other available platforms provided by the IT industry, such as Kubespray, it is important to point out that our methodology is conceived for analytics, rather than for highly-available service hosting. This design choice reflects a use case that we envision to become predominant in science. In fact, while the IT industry is embracing application containers to build resilient services at scale, scientists are making use of the technology to run reproducible and standardized analytics. When it comes to long-running service hosting, long deployment time and complex installation procedures

are a reasonable price to pay, as they occur only initially. In contrast, we focus on a use case where researchers need to allocate cloud resources as needed. Under these assumptions there is a need for adopting simple, fast and scalable deployment procedures. KubeNow meets these requirements by providing: (1) an uncomplicated user interaction (see 'Enabling scalable deployments') and (2) fast and scalable deployments (see 'Deployment automation scalability').

Microservices and application containers are increasingly gaining momentum in scientific applications (*Peters et al., 2018*; *D'Agostino et al., 2017*; *Wu et al., 2017*; *Williams et al., 2016*). When it comes to on-demand VREs the technology presents some important advantages over current systems. Our methodology is based on publicly available information by three research initiatives in substantially different scientific domains (PhenoMeNal, EXTraS and SKA). It is important to point out that EXTraS and SKA provide microservices-oriented VREs primarily as long running platforms, and they do not cover on-demand instantiation, while our methodology made this possible in PhenoMeNal. The requirements in terms of VRE infrastructure are similar across domains, which allowed us to design our methodology as generally applicable. Hence, we envision our work and the presented benchmarks as valuable guidelines for communities of practice that need to build on-demand VRE systems.

CONCLUSION

Here, we introduced a microservice-oriented methodology where scientific applications run in a distributed orchestration platform as light-weight software containers, referred to as on-demand VREs. Our methodology makes use of application containers to improve isolation of VRE components, and it uses cloud computing to dynamically procure infrastructure. The methodology builds on publicly available information by three research initiatives, and it is generally applicable over multiple research domains. The applicability of the methodology was tested through an open source implementation, showing good scaling for data analysis in metabolomics and in terms of deployment speed. We envision communities of practice to use our work as a guideline and blueprint to build on-demand VREs.

ETHICAL APPROVAL AND INFORMED CONSENT

Human-derived samples in the datasets are consented for analysis, publication and distribution, and they were processed according to the ELSI guidelines (*Sariyar et al., 2015*). Ethics and consents are extensively explained in the referenced publications (*Gao et al., 2019*; *Herman et al., 2018*; *Ranninger et al., 2016*).

ACKNOWLEDGEMENTS

We kindly acknowledge contributions to cloud resources by SNIC, EMBL-EBI, CityCloud, CSC, AWS and Azure.

Funding

This research was supported by The European Commission's Horizon 2020 programme under grant agreement number 654241 (PhenoMeNal) and the Nordic e-Infrastructure Collaboration (NeIC) via the Glenna2 and Tryggve2 projects. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Grant Disclosures

The following grant information was disclosed by the authors:
The European Commission's Horizon 2020 programme: 654241.
Nordic e-Infrastructure Collaboration (NeIC).

Competing Interests

The authors declare there are no competing interests.

Author Contributions

- Marco Capuccini conceived and designed the experiments, analyzed the data, contributed reagents/materials/analysis tools, prepared figures and/or tables, performed the computation work, authored or reviewed drafts of the paper, approved the final draft.
- Anders Larsson conceived and designed the experiments, contributed reagents/materials/analysis tools, authored or reviewed drafts of the paper, approved the final draft.
- Matteo Carone performed the experiments, analyzed the data, prepared figures and/or tables, performed the computation work, approved the final draft.
- Jon Ander Novella, Noureddin Sadawi and Jianliang Gao performed the experiments, analyzed the data, contributed reagents/materials/analysis tools, performed the computation work, approved the final draft.
- Salman Toor and Ola Spjuth conceived and designed the experiments, authored or reviewed drafts of the paper, approved the final draft.

Data Availability

The data in the study by Gao et al. is publicly available: Available at https://doi.org/10.6084/m9.figshare.c.4204022. Detailed instructions for reoroducing the analysis can be found at https://github.com/csmsoftware/phnmnl-scalability.

Novella et al. and Khoonsari et al. used public data from the Methabolights repository, and in particular datasets: MTBLS558 and MTBLS233. Detailed instructions for reoroducing the analyses can be found at https://github.com/pharmbio/LC-MS-Pachyderm and at https://github.com/phnmnl/MTBLS233-Jupyter, respectively.

KubeNow is publicly available as open-source software: https://github.com/kubenow/KubeNow Detailed instruction for deploying KubeNow and reproducing the experiments in 'Deployment automation scalability' can be found at https://kubenow.readthedocs.io.

PhenoMeNal is publicly available as open-source software: https://github.com/phnmnl/phenomenal-h2020/wiki.

REFERENCES

Amazon Elastic Container Service. 2019. *Available at https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ECS_instances.html* (accessed on 16 May 2019).

Amazon Elastic File System. 2019. *Available at https://aws.amazon.com/efs* (accessed on 16 May 2019).

Ansible. 2019. *Available at https://www.ansible.com* (accessed on 16 May 2019).

Armbrust M, Fox A, Griffith R, Joseph AD, Katz RH, Konwinski A, Lee G, Patterson DA, Rabkin A, Stoica I, Zaharia M. 2009. Above the clouds: a berkeley view of cloud computing. Technical report UCB/EECS-2009-28. EECS Department, University of California, Berkeley.

Asay M. 2016. Why Kubernetes is winning the container war. *Available at http://www.infoworld.com/article/3118345/cloud-computing/why-kubernetes-is-winning-the-container-war.html* (accessed on 16 May 2019).

Assante M, Candela L, Castelli D, Cirillo R, Coro G, Frosini L, Lelii L, Mangiacrapa F, Marioli V, Pagano P, Panichi G, Perciante C, Sinibaldi F. 2019. The gCube system: delivering virtual research environments as-a-service. *Future Generation Computer Systems* **95**:445–453 DOI 10.1016/j.future.2018.10.035.

Azure Container Instances. 2019. *Available at https://azure.microsoft.com/en-us/services/container-instances* (accessed on 16 May 2019).

Azure NetApp Files. 2019. *Available at https://azure.microsoft.com/en-us/services/storage/netapp* (accessed on 16 May 2019).

Baldini I, Castro P, Chang K, Cheng P, Fink S, Ishakian V, Mitchell N, Muthusamy V, Rabbah R, Slominski A, Suter P. 2017. Serverless computing: current trends and open problems. In: *Research advances in cloud computing.* Springer, 1–20 DOI 10.1007/978-981-10-5026-8_1.

Bayramusta M, Nasir VA. 2016. A fad or future of IT?: a comprehensive literature review on the cloud computing research. *International Journal of Information Management* **36(4)**:635–644 DOI 10.1016/j.ijinfomgt.2016.04.006.

Bild DE, Bluemke DA, Burke GL, Detrano R, Diez Roux AV, Folsom AR, Greenland P, Jacobs Jr DR, Kronmal R, Liu K, Nelson JC, O'Leary D, Saad MF, Shea S, Szklo M, Tracy RP. 2002. Multi-ethnic study of atherosclerosis: objectives and design. *American Journal of Epidemiology* **156(9)**:871–881 DOI 10.1093/aje/kwf113.

Candela L, Castelli D, Pagano P. 2013. Virtual research environments: an overview and a research agenda. *Data Science Journal* **12**:GRDI75–GRDI81.

CityCloud. 2019. *Available at http://citycloud.com* (accessed on 16 May 2019).

Cloud-init. 2019. *Available at https://cloud-init.io* (accessed on 16 May 2019).

Cloudflare. 2019. *Available at https://www.cloudflare.com* (accessed on 16 May 2019).

Cloudfuse. 2019. *Available at https://github.com/redbo/cloudfuse* (accessed on 16 May 2019).

CSC cloud. 2019. *Available at https://research.csc.fi/cloud-computing* (accessed on 16 May 2019).

Cyvoct P. 2018. How to deploy an EFK stack to Kubernetes. *Available at https://blog.ptrk.io/how-to-deploy-an-efk-stack-to-kubernetes* (accessed on 23 August 2018).

D'Agostino D, Roverelli L, Zereik G, Luca AD, Salvaterra R, Belfiore A, Lisini G, Novara G, Tiengo A. 2017. A microservice-based portal for X-ray transient and variable sources. *PeerJ PrePrints* **5**:e2519.

Dahlö M, Scofield DG, Schaal W, Spjuth O. 2018. Tracking the NGS revolution: managing life science research on shared high-performance computing clusters. *GigaScience* **7(5)**:Article giy028.

de.NBI cloud. 2019. *Available at https://www.denbi.de/cloud* (accessed on 16 May 2019).

Duhrkop K, Shen H, Meusel M, Rousu J, Bocker S. 2015. Searching molecular structure databases with tandem mass spectra using CSI:FingerID. *Proceedings of the National Academy of Sciences of the United States of America* **112**:12580–12585 DOI 10.1073/pnas.1509788112.

Elia IA, Antunes N, Laranjeiro N, Vieira M. 2017. An analysis of OpenStack vulnerabilities. In: *2017 13th European Dependable Computing Conference (EDCC)*. 129–134.

Emami Khoonsari P, Moreno P, Bergmann S, Burman J, Capuccini M, Carone M, Cascante M, De Atauri P, Foguet C, Gonzalez-Beltran AN, Hankemeier T, Haug K, He S, Herman S, Johnson D, Kale N, Larsson A, Neumann S, Peters K, Pireddu L, Rocca-Serra P, Roger P, Rueedi R, Ruttkies C, Sadawi N, Salek RM, Sansone S-A, Schober D, Selivanov V, Thévenot EA, Van Vliet M, Zanetti G, Steinbeck C, Kultima K, Spjuth O. 2018. Interoperable and scalable data analysis with microservices: applications in metabolomics. *Bioinformatics* **35(19)**:3752–3760 DOI 10.1093/bioinformatics/btz160.

EMBL-EBI Cloud. 2019. *Available at http://www.embassycloud.org* (accessed on 16 May 2019).

FeatureFinderMetabo. 2018. *Available at https://abibuilder.informatik.uni-tuebingen.de/archive/openms/Documentation/RC/2.3.0/html/TOPP_FeatureFinderMetabo.html* (accessed on 06 April 2018).

Gao J, Sadawi N, Karaman I, Pearce J, Mereno P, Larsson A, Capuccini M, Elliott P, Nicholson JK, Ebbels T, Glen RC. 2019. Metabolomics in the cloud: scaling computational tools to big data. ArXiv preprint. arXiv:1904.02288.

GlusterFS. 2019. *Available at https://www.gluster.org* (accessed on 16 May 2019).

Goecks J, Nekrutenko A, Taylor J, Galaxy Team. 2010. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biology* **11(8)**:Article R86 DOI 10.1186/gb-2010-11-8-r86.

Google Cloud Filestore. 2019. *Available at https://cloud.google.com/filestore* (accessed on 16 May 2019).

Google Cloud Run. 2019. *Available at https://cloud.google.com/run* (accessed on 16 May 2019).

Hao J, Astle W, De Iorio M, Ebbels TMD. 2012. BATMAN—an R package for the automated quantification of metabolites from nuclear magnetic resonance spectra using a Bayesian model. *Bioinformatics* **28(15)**:2088–2090 DOI 10.1093/bioinformatics/bts308.

Haug K, Salek R, Conesa Mingo P, Hastings J, Matos P, Rijnbeek M, Mahendraker T, Williams M, Neumann S, Rocca-Serra P, Maguire E, Gonzlez-Beltrán A, Sansone S-A, Griffin J, Steinbeck C. 2012. MetaboLights—an open-access general-purpose repository for metabolomics studies and associated meta-data. *Nucleic Acids Research* **41(D1)**:D781–D786.

Helm. 2019. *Available at https://github.com/kubernetes/helm* (accessed on 16 May 2019).

Herman S, Khoonsari P, Tolf A, Steinmetz J, Zetterberg H, Akerfeldt T, Jakobsson P-J, Larsson A, Spjuth O, Burman J, Kultima K. 2018. Integration of magnetic resonance imaging and protein and metabolite CSF measurements to enable early diagnosis of secondary progressive multiple sclerosis. *Theranostics* **8(16)**:4477–4490 DOI 10.7150/thno.26249.

Hindman B, Konwinski A, Zaharia M, Ghodsi A, Joseph AD, Katz RH, Shenker S, Stoica I. 2011. Mesos: a platform for fine-grained resource sharing in the data center. In: *NSDI, vol. 11.* NSDI'11 Proceedings of the 8th USENIX conference on Networked systems design and implementation, 22–22. *Available at https://www.usenix.org/legacy/events/nsdi11/tech/full_papers/Hindman.pdf* .

Javed A, Heljanko K, Buda A, Främling K. 2018. CEFIoT: a fault-tolerant IoT architecture for edge and cloud. In: *2018 IEEE 4th world forum on internet of things (WF-IoT)*. IEEE, 813–818.

Jupyter. 2019. *Available at https://jupyter.org* (accessed on 16 May 2019).

Karakoyunlu C, Kimpe D, Carns P, Harms K, Ross R, Ward L. 2013. Toward a unified object storage foundation for scalable storage systems. In: *Cluster computing (CLUSTER), 2013 IEEE international conference on*. IEEE, 1–8.

Karaman I, Ferreira DLS, Boulangé CL, Kaluarachchi MR, Herrington D, Dona AC, Castagné R, Moayyeri A, Lehne B, Loh M, De Vries PS, Dehghan A, Franco OH, Hofman A, Evangelou E, Tzoulaki I, Elliott P, Lindon JC, Ebbels TMD. 2016. Workflow for integrated processing of multicohort untargeted 1H NMR metabolomics data in large-scale metabolic epidemiology. *Journal of Proteome Research* **15(12)**:4188–4194 DOI 10.1021/acs.jproteome.6b00125.

Khan A. 2017. Key characteristics of a container orchestration platform to enable a modern application. *IEEE Cloud Computing* **4(5)**:42–48 DOI 10.1109/MCC.2017.4250933.

KubeNow GitHub organization. 2019. *Available at https://github.com/kubenow* (accessed on 16 May 2019).

Kubespray. 2019. *Available at https://github.com/kubernetes-incubator/kubespray* (accessed on 16 May 2019).

Kurtzer GM, Sochat V, Bauer MW. 2017. Singularity: scientific containers for mobility of compute. *PLOS ONE* **12(5)**:e0177459 DOI 10.1371/journal.pone.0177459.

Lampa S, Dahlö M, Olason PI, Hagberg J, Spjuth O. 2013. Lessons learned from implementing a national infrastructure in Sweden for storage and analysis of next-generation sequencing data. *Gigascience* **2(1)**:Article 2047-217X-2-9 DOI 10.1186/2047-217X-2-9.

Laure E, Edlund Å. 2012. The e-infrastructure ecosystem: providing local support to global science. *Large-Scale Computing Techniques for Complex System Simulations* **80**:19–34.

Luigi. 2018. *Available at https://github.com/spotify/luigi* (accessed on 07 January 2018).

Manousis A, Ragsdale R, Draffin B, Agrawal A, Sekar V. 2016. Shedding light on the adoption of Let's Encrypt. ArXiv preprint. arXiv:1611.00469.

Manu AR, Patel JK, Akhtar S, Agrawal VK, Murthy KNBS. 2016. A study, analysis and deep dive on cloud PaaS security in terms of Docker container security. In: *2016 international conference on circuit, power and computing technologies (ICCPCT)*. 1–13.

Marathon. 2018. *Available at https://mesosphere.github.io/marathon* (accessed on 06 April 2018).

Naik N. 2016. Building a virtual system of systems using Docker Swarm in multiple clouds. In: *Systems Engineering (ISSE), 2016 IEEE International Symposium on*. Piscataway: IEEE, 1–3.

Neal F. 2018. *The state of microservices maturity*. O'Reilly Media, Inc.

Netto HV, Lung LC, Correia M, Luiz AF, De Souza LMS. 2017. State machine replication in containers managed by Kubernetes. *Journal of Systems Architecture* **73**:53–59 DOI 10.1016/j.sysarc.2016.12.007.

NIP.IO. 2019. *Available at http://nip.io* (accessed on 16 May 2019).

Novella JA, Emami Khoonsari P, Herman S, Whitenack D, Capuccini M, Burman J, Kultima K, Spjuth O. 2018. Container-based bioinformatics with Pachyderm. *Bioinformatics* **35(5)**:839–846.

Open Container Initiative. 2016. The 5 principles of standard containers. *Available at https://github.com/opencontainers/runtime-spec/blob/master/principles.md* (accessed on 16 May 2019).

OpenStack Manila. 2019. *Available at https://wiki.openstack.org/wiki/Manila* (accessed on 16 May 2019).

OpenStack Zun. 2019. *Available at https://docs.openstack.org/zun* (accessed on 16 May 2019).

Pachyderm. 2019. *Available at https://pachyderm.io* (accessed on 16 May 2019).

Packer. 2019. *Available at https://www.packer.io* (accessed on 16 May 2019).

Pathan A-MK, Buyya R. 2007. A taxonomy and survey of content delivery networks. Technical Report, 4. Grid Computing and Distributed Systems Laboratory, University of Melbourne.

Peters K, Bradbury J, Bergmann S, Capuccini M, Cascante M, De Atauri P, Ebbels TMD, Foguet C, Glen R, Gonzalez-Beltran A, Günther UL, Handakas E, Hankemeier T, Haug K, Herman S, Holub P, Izzo M, Jacob D, Johnson D, Jourdan F, Kale N, Karaman I, Khalili B, EmamiKhonsari P, Kultima K, Lampa S, Larsson A, Ludwig C, Moreno P, Neumann S, Novella JA, O'Donovan C, Pearce JTM,

Peluso A, Piras ME, Pireddu L, Reed MAC, Rocca-Serra P, Roger P, Rosato A, Rueedi R, Ruttkies C, Sadawi N, Salek RM, Sansone S-A, Selivanov V, Spjuth O, Schober D, Thévenot EA, Tomasoni M, VanRijswijk M, VanVliet M, Viant MR, Weber RJM, Zanetti G, Steinbeck C. 2018. PhenoMeNal: processing and analysis of metabolomics data in the cloud. *GigaScience* **8(2)**:Article giy149.

Ranninger C, Schmidt LE, Rurik M, Limonciel A, Jennings P, Kohlbacher O, Huber CG. 2016. Improving global feature detectabilities through scan range splitting for untargeted metabolomics by high-performance liquid chromatography-Orbitrap mass spectrometry. *Analytica Chimica Acta* **930**:13–22 DOI 10.1016/j.aca.2016.05.017.

Roth B, Hecht R, Volz B, Jablonski S. 2011. Towards a generic cloud-based virtual research environment. In: *Computer software and applications conference workshops (COMPSACW), 2011 IEEE 35th annual*. Piscataway: IEEE, 267–272.

Sariyar M, Schluender I, Smee C, Suhr S. 2015. Sharing and reuse of sensitive data and samples: supporting researchers in identifying ethical and legal requirements. *Biopreservation and Biobanking* **13(4)**:263–270 DOI 10.1089/bio.2015.0014.

Shimel A. 2016. Docker becomes de facto Linux standard. *Available at http://www. networkworld.com/article/2226751/opensource-subnet/docker-becomes-de-facto-linux-standard.html* (accessed on 16 May 2019).

Terraform. 2019. *Available at https://terraform.io* (accessed on 16 May 2019).

Traefik. 2019. *Available at https://traefik.io* (accessed on 16 May 2019).

Travis CI. 2019. *Available at https://travis-ci.org* (accessed on 16 May 2019).

Thönes J. 2015. Microservices. *IEEE Software* **32(1)**:116–116.

Toor S, Lindberg M, Falman I, Vallin A, Mohill O, Freyhult P, Nilsson L, Agback M, Viklund L, Zazzik H, Spjuth O, Capuccini M, Möller J, Murtagh D, Hellander A. 2017. SNIC science cloud (SSC): a national-scale cloud infrastructure for Swedish Academia. In: *2017 IEEE 13th international conference on e-science (e-Science)*. Piscataway: IEEE, 219–227.

Vaughan-Nichols SJ. 2016. Containers vs. virtual machines: how to tell which is the right choice for your enterprise. *Available at https://www.networkworld.com/article/3068392/cloud-storage/containers-vs-virtual-machines-how-to-tell-which-is-the-right-choice-for-your-enterprise.html* (accessed on 29 June 2018).

Vayghan LA, Saied MA, Toeroe M, Khendek F. 2018. Deploying microservice based applications with Kubernetes: experiments and lessons learned. In: *2018 IEEE 11th international conference on cloud computing (CLOUD)*. IEEE, 970–973.

Vixie P, Thomson S, Rekhter Y, Bound J. 1997. Dynamic updates in the domain name system (DNS UPDATE). Technical report, RFC 2136.

Weerasiri D, Barukh MC, Benatallah B, Sheng QZ, Ranjan R. 2017. A taxonomy and survey of cloud resource orchestration techniques. *ACM Computing Surveys (CSUR)* **50(2)**:Article 26.

Williams CL, Sica JC, Killen RT, Balis UG. 2016. The growing need for microservices in bioinformatics. *Journal of Pathology Informatics* **7**:Article 45 DOI 10.4103/2153-3539.194835.

Wu C, Tobar R, Vinsen K, Wicenec A, Pallot D, Lao B, Wang R, An T, Boulton M, Cooper I, Dodson R, Dolensky M, Mei Y, Wang F. 2017. DALiuGE: a graph execution framework for harnessing the astronomical data deluge. *Astronomy and Computing* **20**:1–15 DOI 10.1016/j.ascom.2017.03.007.

Zhao D, Mohamed M, Ludwig H. 2018. Locality-aware scheduling for containers in cloud computing. *IEEE Transactions on Cloud Computing* Epub ahead of print Jan 16 2018 DOI 10.1109/TCC.2018.2794344.