A DevOps Approach to Integration of Software Components in an EU Research Project

Mark Stillwell
m.stillwell@imperial.ac.uk

Jose G. F. Coutinho
gabriel.figueiredo@imperial.ac.uk

Imperial College London
United Kingdom

ABSTRACT

We present a description of the development and deployment infrastructure being created to support the integration effort of HARNESS, an EU FP7 project. HARNESS is a multi-partner research project intended to bring the power of heterogeneous resources to the cloud. It consists of a number of different services and technologies that interact with the OpenStack cloud computing platform at various levels. Many of these components are being developed independently by different teams at different locations across Europe, and keeping the work fully integrated is a challenge. We use a combination of Vagrant based virtual machines, Docker containers, and Ansible playbooks to provide a consistent and up-to-date environment to each developer. The same playbooks used to configure local virtual machines are also used to manage a static testbed with heterogeneous compute and storage devices, and to automate ephemeral larger-scale deployments to Grid'5000. Access to internal projects is managed by GitLab, and automated testing of services within Docker-based environments and integrated deployments within virtual-machines is provided by Buildbot.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.9 [**Software Engineering**]: Software configuration management

Keywords

DevOps, Configuration Management, Automated Testing, Ansible, Vagrant, Docker, OpenStack, GitLab, BuildBot

1. INTRODUCTION

In the past most academic software was developed for specific purposes by individuals or small teams of developers. However, current funding policies of agencies such as the EPSRC, EU FP7 commission, and HORIZON 2020 encourage both multi-partner coalitions and development of high-quality software intended for distribution and reuse [14, 15]. While these trends are encouraging in terms of increasing knowledge exchange and reducing wasted or repeated effort, meeting the new requirements poses challenges for project leaders who need to ensure that work undertaken by independent organizations is coordinated to an appropriate degree in order to assure quality. Furthermore, a recent study [19] compared four research projects conducted by academics over the span of nine years, and concluded that there is a strong correlation between publication output and the software development effort invested three years prior to publication, thus highlighting the importance of software engineering practices to boost the number of papers stemming from large research projects.

In this paper we describe how developers on the HARNESS project [7] address this issue through an approach based on version-controlled configuration management, automated software deployment, and continuous integration. While software development in industry faces similar difficulties, there are differences in the objectives, incentives, and measures of success between commercial and academic research projects, and this work is intended primarily to address the needs of the latter. HARNESS is an EU FP7 research project with the objective of making it easier for cloud providers and consumers alike to take advantage of heterogeneous resources, including, computational accelerators (GPGPUs and FPGAs), programmable routers and heterogeneous storage devices. The key motivation for incorporating heterogeneity is to offer a richer context for price/performance trade offs, and to bring wholly new degrees of freedom to the cloud resource allocation and optimization problem.

A key challenge of the HARNESS project, and indeed most EU research projects [17], is that it requires bringing together specialists from a number of geographically distributed partner institutions in academia and industry. Individual components addressing different classes of heterogeneous resources or requirements can be developed independently, but there is a need to coordinate effort and ensure that API specifications are adhered to and consistently interpreted, so that components can be deployed in a such a way as to provide a coherent distributed computing infrastructure. While responsibility for the quality of the individual components is shared among a number of lead institutions and is managed by the work package leader for each component, there is a need to provide a way to test and evaluate the fully integrated deployment as well. Finally, as this is an academic research project, deployments and benchmarking experiments should be made as reproducible as possible in order to facilitate verification of research results. To address these challenges we present in this paper a development and operations (DevOps) workflow that allows: **(a)** teams of developers to work autonomously on specific parts of the software architecture; **(b)** automated testing of individual projects as well as the integrated system deployments; **(c)** reproducible automated deployment on heterogeneous and large-scale testbeds.

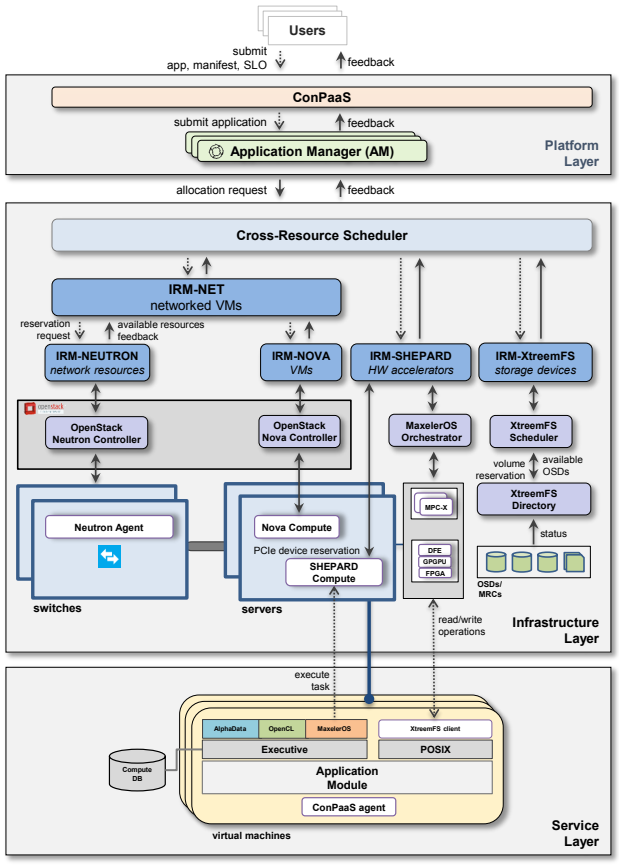

Figure 1: The HARNESS cloud architecture consists of a suite of loosely coupled distributed services that interact with each other through an HTTP/REST API. This microservice architecture puts considerable overhead in managing the development and deployment processes.

The remainder of this paper is structured as follows. In Section 2 we describe the HARNESS cloud architecture. and explain why a DevOps approach is needed to maintain high quality output. In Section 3 we present a high-level view of the HARNESS DevOps workflow. In Section 4 we describe how various deployment tools help achieve reproducibility and why this is important for testing and quality assurance. In Section 5 we report our automated testing infrastructure and our novel methodology for testing full systems deployments. In Section 6 we describe two platforms where HARNESS is being deployed. Finally, we conclude with a summary and plans for future work in Section 7.

2. HARNESS CLOUD ARCHITECTURE

The HARNESS cloud architecture is divided into three main parts: a *platform layer* in charge of managing applications that works on behalf of the cloud tenant, an *infrastructure layer* responsible for managing resources that works on behalf of the cloud provider, and a *service layer* where cloud applications are deployed and executed. An example HARNESS cloud platform, currently being developed as a proof-of-concept, is presented in Figure 1. This platform combines (a) VM and network resources managed by OpenStack [10], (b) networked FPGA devices managed by MaxelerOS Orchestrator [9], (c) OpenCL accelerators managed by SHEPARD [21], and (d) heterogeneous storage devices managed by XtreemFS [23].

The HARNESS architecture is designed to be open and modular, allowing arbitrary types of resources (compute, storage and network) to be integrated and leased to cloud users. Each resource type has a specially designed *Infrastructure Resource Manager* (IRM) that understands its internal semantics while presenting a uniform API to the *Cross-Resource Scheduler* component (CRS). The CRS acts as the central scheduling module and has a global view of all resources available in the data-center. This structure, wherein a larger project is made up of a suite of fine-grained collaborative services, each running on its own process and communicating with each other through a well-defined HTTP/REST API, is commonly referred to as a *microservice* software architecture [18]. As a consequence of this design decision, services are loosely coupled, allowing different developer teams to work and maintain each service autonomously. As an alternative to the microservice architecture, there is the *monolithic* architecture, in which applications can be realized into a single logical executable.

Cloud tenants submit their applications and performance/cost objectives through the ConPaaS frontend interface [4]. ConPaaS is responsible for providing an entry point to cloud users, as well as handling user authentication and creating a context for application management. An *application manager* instance is generated whenever an application is submitted on the HARNESS platform, and is responsible for overseeing the life-cycle of the application. For instance, the application manager automatically runs a suite of micro-benchmarks to generate a performance profile for a submitted application, and then works with the CRS module to determine the best way to run it, taking into account both user-specified performance and cost objectives and the demands of competing users. Once resources have been provisioned by the CRS, the application manager deploys the application on allocated VMs. The application can access other resources (such as GPGPUs, FPGAs and heterogeneous storage) allocated in the previous step by interacting with management systems (daemons) running on the VMs.

While there are many benefits to following a microservice architecture, it does require each service to be built and tested individually, and then deployed with other services in order to ensure that all these modules stay up and collaborate with each other. Hence, managing and rolling out all these services puts considerable overhead on the development and operations processes, requiring a high degree of deployment automation to ship and configure the integrated system. While this problem is also commonly encountered in real world systems, dealing with it is still an area of active development and research and there are no widely accepted standard solutions. In the next section, we describe our development and operations processes in the context of the HARNESS project.

3. DEVOPS WORKFLOW

Figure 2 illustrates the HARNESS DevOps workflow, which captures the process of developing the HARNESS cloud system from version control to release. Our key goal is to allow different teams of programmers to develop and maintain each service autonomously, while also enabling collaboration between teams and providing feedback as soon as possible about the flow of changes and their impact on the combined system. The HARNESS project has four *development* teams and one *integration* team. Each development team is responsible for maintaining a specific part of the architecture belonging to one of four technical areas: compute, network, storage and platform. The integration team is responsible for ensuring that all these services are properly tested when combined and deployed on the HARNESS testbeds (see Section 6).

All the development and integration teams share a single HARNESS **GitLab** server. GitLab is a web-based Git repository manager

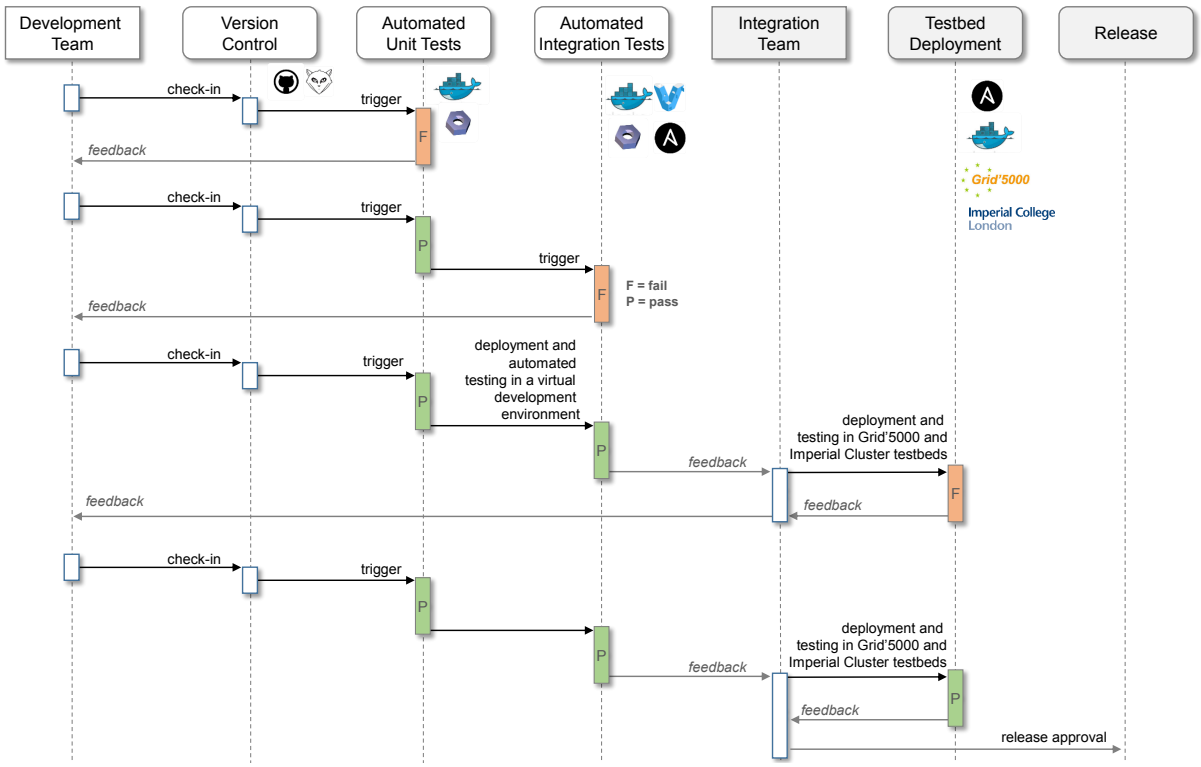

Figure 2: The HARNESS DevOps workflow.

that supports multiple users, groups, and owner-specified access controls for repositories. GitLab provides an open-source alternative to GitHub that can be installed on private infrastructure. With GitLab, each developer team stores a specific project (such as an architecture component or an automated deployment project) in its own **Git** repository, with all HARNESS software contributions aggregated into a single central server acting as the authoritative reference. In HARNESS, each project has one *owner* who is responsible for maintaining the project and has exclusive access to the master branch. The *contributors* (other members of the development team) can only make changes, such as adding experimental features or fixing bugs, by forking the master branch. Hence, rather than logging an issue, contributors can fork (copy) the repository, make updates, and then submit a pull (merge) request to the project owner. The project owner can then review the changes and accept or reject the merge request by exploiting GitLab's advanced tracking of the relationship between forks and its code reviewing facilities.

Whenever the owner of a project pushes a commit to GitLab or merges in changes, it triggers a set of automated tests (see Section 5). First, unit tests associated with each project are executed. If any of these tests fail, then the project owner is notified. Otherwise, automated integration tests are queued to run at specific times. In this case, an integration deployment project pulls the latest version of all HARNESS components from various master branches, deploys the HARNESS software stack on virtual machines, and runs tests that aim to exercise all critical features of the system. If any integration test fails, then both the integration team and the project owner (whose commit triggered the integration tests) are notified by email.

Periodically, the integration team deploys and tests the HARNESS cloud system in two HARNESS testbeds (see Section 6), namely Grid'5000 and the Imperial Cluster, depending on the types of updates submitted by the development teams. This process is manually initiated but almost entirely automated. In the case of Grid'5000, deploying HARNESS requires requesting nodes from the batch scheduler and provisioning them with a base Ubuntu 14.04 operating system [16], but after that point the deployment infrastructure can take over to install and configure all of the HARNESS software and its dependencies without human intervention. If any deployment test fails, then the integration team writes one or more integration tests that can flag a particular fault the next time the tests are executed. In addition, the integration team also notifies the development team of any bug, who in turn can write one or more unit tests to flag the problem at the component level.

4. REPRODUCIBLE DEPLOYMENT

One of the defining characteristics of a DevOps-based approach is that the testing environments should reliably reflect the production environments. That is, the environments themselves should be, to the extent feasible, *stateless* and *reproducible*. Statelessness in this case refers simply to the idea that the runtime environment should not change over time in a way that might affect the behavior of running applications. It is difficult, if not impossible, to achieve true statelessness on real-world machines, but many of the same benefits can be achieved by isolating running services from each other and the host environment through either virtualization or operating system specific methods of containerization. In order to achieve reproducibility we focus on automation and version control. Automation allows us to ensure that all configuration steps are fully documented, while version control lets us see how configurations have evolved over time, and potentially to review the differences in configuration between running systems. An additional benefit of automation is that it can ease deployment to new target systems, potentially increasing the number of operating production systems and reducing downtime in the event of catastrophic failure.

4.1 Containerized Services

An operating system container is an isolated environment provided by an operation system kernel rather than a hypervisor. While hypervisors achieve isolation between environments through high-overhead techniques like intercepting interrupts in order to provide the appearance of a physically isolated machine, containers provide a lighter-weight approach. In the case of the Linux operating system, containerization is primarily achieved by replicating various kernel data structures to provide separate *namespaces* to running processes. From the perspective of the operating system, processes running within containers are no different from processes running outside of containers, they just have a more constrained view of the system. Thus, containerized services can run in isolation at native or near-native performance (there may be some small overheads due to an extra layer of abstraction for some operations) [24]. Containers have an additional benefit in that, since services can be launched directly from the host, and links can be made easily between containers or between a container and the host [5], there is no need to deploy a complete stack of running services in every container, which improves efficiency as compared with full virtual machines. While some work may be needed to give containers direct access to hardware devices, it is nonetheless simpler to do so than to implement similar functionality for virtual machines. Some drawbacks of containers relative to virtual machines are that environments must all share the same kernel version, and there may be some difficulty in management of access to the kernel module space.

Docker is a software technology for managing containerized software deployments. As with OpenStack, Docker provides an interface and various databases to track conceptual objects, while leaving implementation primarily up to other lower-level technologies [5]. Application specific software runtime environments are described in a "Dockerfile" that can be committed to the software repository or maintained in a separate project. Dockerfiles give instructions for reproducibly creating an image from a standard binary base. There are base images available representing the environments provided by most of the mainline Linux distributions, though it should be noted that applications may not function in precisely the same way within a container as on the equivalent full operating system. In particular, through experience we have learned that in the standard Ubuntu image the upstart service does not work correctly, so daemons need to be started either directly or by using a third-party application.

The main advantages of deploying services within containers rather than directly on the host system are 1) that the services themselves are implemented and tested within the same environment, which includes all dependencies and so there is no need to worry about the configuration of the host and 2) deploying services does not affect the host's operating environment, and so there is no concern, for example, that deploying a service B will result in breaking some unrelated service A because of dependencies on incompatible libraries. This latter benefit also means that services can be un-installed and reinitialized cleanly, without worrying that they are leaving behind old versions of data or configuration files that may affect the running of future service deployments. Within HARNESS, many components are implemented as python daemons that need to talk or be available on different network interfaces, which is the ideal situation for docker based deployment. In recent versions of the platform we are moving away from simply running these daemons directly on the host to having Dockerfiles embedded within the projects and having docker as the preferred means of deployment.

4.2 Service Orchestration and Configuration

In recent years there has been a movement toward increasing the use of automation for systems administration and configuration management tasks. This has been motivated by a number of factors, including: the difficulty of tracking configuration changes across multiple systems, the need to ensure configurations are applied consistently to ephemeral cloud-based systems, and the desire to make configurations reproducible across platforms. Leading technologies in this area include Puppet [11], Chef [3], Salt [12], and Ansible [1], among others. While each of these has its advantages and disadvantages, **Ansible** stands out for its relatively low barrier to entry for new projects: client systems need only ssh and Python; there is no need to install a client service or manage a separate trusted certificate registry as with Puppet or Chef; and it is conceptually simple: nodes are listed and categorized into groups within an "inventory" (usually a static configuration file, but potentially a dynamic script), while configuration changes are described within "playbooks" as sequences of tasks applied in parallel to one or more nodes or groups of nodes, with checks implemented within modules to ensure idempotency (that is, if a configuration change is applied once then it should not be applied a second time, even if the same set of Ansible tasks are run multiple times on the same system) [1]. In Puppet, by contrast, there is a need to fully describe dependencies between configuration directives to ensure that changes are consistently applied in the same order when there are possible side-effects [20].

As discussed in the previous section, container-management technologies like Docker are extremely useful for creating reproducible environments for individual services, but there is still a need for higher-order orchestration and configuration management. For one thing, not every service can be deployed within a Docker container: examples include Docker itself (which can run in a container, but there first needs to be an installation on the base operating system) and services that need to cross standard container boundaries, like OpenStack Neutron, which must be able to control the host networking interface. It should be emphasized that services like Neutron *can* run in containers, but getting them to function correctly requires significant effort and is not yet widely supported. Another reason that an orchestration is required is that even if all services are containerized, there is still a need to place them on particular hosts and make sure that required configuration information (particularly secret information, like passwords) is distributed correctly to the services that need it.

The HARNESS deployment project contains several Ansible playbooks and related configuration files describing how the various components are instantiated on different nodes within the distributed system. There is an inventory file for each deployment target, including the automated testing environment. Each inventory groups hosts in the target environment by services run in the deployment and sets deployment-specific configuration variables. Currently, all of the deployment targets make use of the same set of playbooks. The "getreqs.yml" playbook first fetches related projects, or roles, each of which describes the tasks required to instantiate a number of standard services: mysql (database), rabbitmq (messaging), docker (container management), keystone (OpenStack authentication and identity), glance (OpenStack virtual machine image service), nova controller (OpenStack virtual machine frontend API and management services), neutron controller (OpenStack virtual network frontend API and management services), neutron network (OpenStack virtual network gateway service) and nova compute (OpenStack virtual machine management service). The source code and Dockerfiles for each of the HARNESS services are also fetched, so that these can ultimately be synced to appropriate target nodes in order to build the required Docker images. The "deploy.yml" playbook contains instructions for actually deploying the HARNESS platform, while the "test.yml" playbook should be run after the deployment to ensure that the integrated system functions as expected.

4.3 Virtual Machine Environments

Yet another advantage of automated, reproducible deployment is that it makes it possible for developers to create personal testbeds so that they can see how their individual components function within the larger system and make changes without fear of causing problems for others. The most practical way to go about this is to provision the full system in a virtual machine based environment on the developer workstation—this way the environment can be destroyed and recreated in a pristine state relatively quickly, without having to worry about reconfiguring the base operating system on a physical system. Of course, setting up virtual machines, particularly multiple machines connected to each other by virtual network links, is in itself a complicated process, but fortunately one that is also amenable to a certain level of abstraction and automation.

Vagrant is used to manage the creation and configuration of virtual machines for testing environments (see Section 5). Vagrant is a software tool that provides a simple and consistent configuration and command line interface for developers to manage virtual machines [13]. For each project, a "Vagrantfile" is created within the root directory of the software project and can be included within the source control system. After this, developers can simply run "vagrant up" to have all of the virtual machines required for local testing made available, with networking and local file synchronization set up automatically. Vagrant has hooks that are configured to call Ansible to configure a full HARNESS software deployment automatically. Critically, Vagrant can be used to manage the creation of both local virtual machines in VirtualBox on developer workstations and virtual machines that run within the testing environment.

5. AUTOMATED TESTING

Automating tests is a fairly standard procedure with the aim of minimizing the number of defects of a software system within a standard environment. These tests need to ensure that not only individual services or packages function correctly, but that the individual parts work together as part of a coherent whole. In this context, development and integration teams can be automatically notified by email or access a web service interface to verify if their projects are passing—or failing—the automated tests.

There are a number of standard solutions for automating testing and continuous integration. Perhaps the most popular and widely deployed of these is Jenkins [8]. However, the normal way to configure Jenkins and set up projects is through its interactive web console, whereas we wished to take a DevOps-oriented approach and manage the configuration and deployment of the testing environment using Ansible. For this reason, along with its Python implementation, we use **Buildbot** to provide an automated testing service for the HARNESS project.

Figure 3 illustrates the HARNESS Buildbot architecture. With Buildbot, it is possible to define a number of automated tests to run, and to have those tests invoked whenever changes are made to a specified repository. The architecture of Buildbot consists of one or more *master* nodes which monitor repositories and generate tasks, and multiple *buildslave* nodes to run queued tasks [2].

For the HARNESS unit tests, the Buildbot master monitors specified projects. When changes are detected, Buildbot checks them out, and looks for a `run_tests.sh` script in the root directory. If such a file is found then the script is run within a **Docker** container using the standard Ubuntu 14.04 image as illustrated in Figure 3(a). The buildslaves, in this case, run the unit tests on standard virtual machines deployed within the Imperial DoC cloud infrastructure.

The integration tests, on the other hand, require all the HARNESS cloud services to be deployed, including OpenStack, incurring a

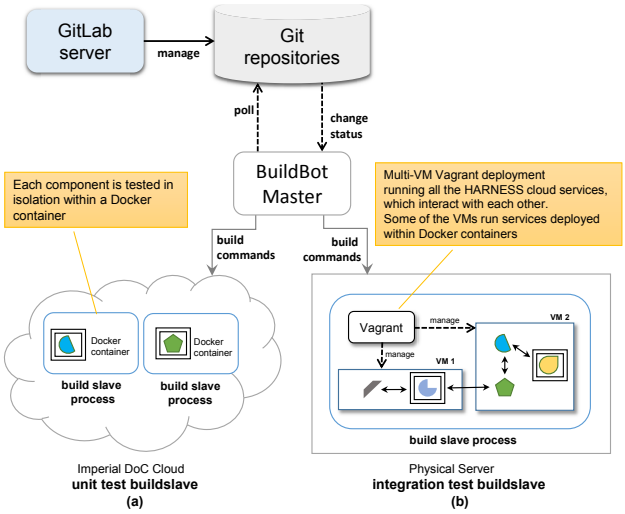

Figure 3: Automated testing workflow using Buildbot. Individual components tested include the services illustrated in Figure 1, including the Application Manager, the Cross-Resource Scheduler, and the IRM components which interface various resource managers.

higher runtime overhead than running isolated unit tests. For this reason, we run the integrated tests on dedicated physical nodes as illustrated in Figure 3(b). In this case, the Buildbot service runs Vagrant with the same configuration file as used by developers to create and test systems locally, and the output is monitored to ensure both that the Ansible configuration and testing scripts run successfully, and that the implementation is idempotent (that is, that subsequent runs of the `vagrant provision` command do not result in changes to the configuration of the deployed virtual machines). Our current automated integration tests run on top of multiple Vagrant spawned VMs, where some of these VMs run HARNESS cloud services within Docker containers. These integration tests are triggered and queued whenever a change is pushed to any project on which the integrated environment depends. However, due to high overheads, these builds are only run when all of the required projects have passed their unit tests, to a maximum of a few times per hour.

6. HARNESS TESTBEDS

There are two main deployment testbeds for the HARNESS cloud platform: the Imperial Cluster, which offers a relatively small-scale static testbed with heterogeneous compute and storage devices, and Grid'5000, which is a large-scale research testbed to support parallel and distributed computing experiments.

The Imperial Cluster testbed infrastructure, which is partly managed by the Custom Computing Group at Imperial College London, consists of a total of 6 compute nodes, 16 CPU cores, 2 GPGPUs, heterogeneous storage devices (HDD and SSD), and 3 MPC-X boxes harboring a total of 24 Dataflow Engines (DFEs). A Dataflow Engine is a general purpose reconfigurable device using an FPGA at its core and RAM for bulk storage. In our testbed, DFEs are co-located in MPC-X appliances which are in turn connected to select hosts via an Infiniband network. Dataflow computing technology [22] has been used successfully in fields such as oil and gas exploration and financial risk analytics, while research has been conducted in scientific areas as diverse as fluid dynamics and quantum chemistry.

The other main deployment target, Grid'5000, is a large-scale research testbed to support parallel and distributed computing exper-

iments. This testbed is distributed across 10 sites (mostly in France), with 1000 compute nodes and 8000 cores. It features a diverse set of technologies, including 10G Ethernet, Infiniband, GPUs, Xeon PHI, and data clusters. One key capability of Grid'5000 is that it is highly reconfigurable, providing bare-metal deployment that allows a fully customized software stack (including the operating system) and isolation at the network layer [16].

Deploying HARNESS to Grid'5000 requires an almost fully automated approach. The allocation is ephemeral, so there is an emphasis on speed of deployment and a need to dynamically generate an inventory. The dynamic nature of Grid'5000 means that there is no single set of static nodes that form the test bed. Rather, developers must request sets of nodes and other resources from the OAR batch scheduler. The following commands show an example of how HARNESS can be deployed on Grid'5000:

```
% oarsub -t deploy -I \
         -l slash_22=1+cluster=1/nodes=3,walltime=4:00:00
% kadeploy3 -f $OAR_NODE_FILE -e ubuntu-x64-1404 -k
% ansible-playbook -i inventories/g5k.sh deploy.yml
```

The first command (`oarsub`) requests a reservation of three Grid'5000 nodes, all on the same cluster, with a /22 subnet, for 4 hours. The second command (`kadeploy3`) puts a fresh install of Ubuntu 14.04 on the reserved nodes. Finally, the third command (`ansible-playbook`) deploys the HARNESS cloud to the reserved nodes. The `g5k.sh` script, which is passed as an argument, is a shell script that reads environment variables set by the batch job scheduler to dynamically generate an inventory based on the user's reservation.

7. CONCLUSION

In this paper we have described the development and deployment infrastructure created to support the integration effort of the FP7 HARNESS project. This infrastructure addresses a number of challenges commonly found in EU research projects that aim to develop high-quality software intended for distribution and reuse, with a focus on dissemination activities (such as proof-of-concept demonstrations and experiments for research papers) rather than following the imperatives of commercial clients. Currently, the primary deployment targets for HARNESS consist of a static testbed hosted at Imperial College London and Grid'5000, which presents a larger-scale but ephemeral environment with direct access to machine hardware. Future plans for the project include deploying HARNESS to the EGI Federated Cloud, a multi-site cloud computing infrastructure for research within the European Union [6].

While it is difficult to fully quantify how our DevOps workflow has affected our development process, and we are only now starting to collect metrics, we can see a qualitative difference in the way teams operate as compared to the previous year, before putting in place automated testing and deployment. In particular, in the beginning of this year considerable changes in the HARNESS cloud architecture where planned with the introduction of a new API specification and updated features such as monitoring and feedback, which required extensive modifications across the whole cloud software stack. We found development teams more able to operate autonomously, while simultaneously being more willing to accept changes and providing more frequent updates. There also seemed to be a reduction in communication overhead for making coordinated changes. The combination of automated testing and deployment appears to have contributed greatly to improving the speed and efficiency with which we converge toward the project milestones.

8. ACKNOWLEDGMENTS

This work was supported by the European Union Seventh Framework Programme under Grant agreement number 318521.

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see `https://www.grid5000.fr`).

9. REFERENCES

[1] Ansible: DevOps made simple. `http://ansible.com`.

[2] Buildbot: The continuous integration framework. `http://buildbot.net`.

[3] Chef: Automation for web-scale IT. `http://chef.io`.

[4] ConPaaS Project. `http://www.conpaas.eu/`.

[5] Docker. `http://docker.io`.

[6] European grid infrastructure federated cloud. `http://www.egi.eu/infrastructure/cloud/`.

[7] FP7 HARNESS project. `http://www.harness-project.eu/`.

[8] Jenkins: An extensible open source continuous integration server. `http://jenkins-ci.org`.

[9] Maxeler Technologies. `http://www.maxeler.com`.

[10] OpenStack: Open source software for creating private and public clouds. `http://openstack.org`.

[11] Puppet Labs. `http://puppetlabs.com`.

[12] SaltStack. `http://saltstack.com`.

[13] Vagrant: Development environments made easy. `http://vagrantup.com`.

[14] European Commission: Software technologies, the missing key enabling technology, 2012. `http://cordis.europa.eu/fp7/ict/docs/istag-soft-tech-wgreport2012.pdf`.

[15] EPSRC: Software as an infrastructure, 2015. `https://www.epsrc.ac.uk/newsevents/pubs/software-as-an-infrastructure/`.

[16] D. Balouek et al. Adding Virtualization Capabilities to the Grid'5000 Testbed. In *Cloud Computing and Services Science*, volume 367, pages 3–20. 2013.

[17] A. Bubeck et al. Implementing Best Practices for Systems Integration and Distributed Software Development in Service Robotics. In *IEEE/SICE Inter. Symp. on System Integration (SII)*, pages 609–614, Dec 2012.

[18] M. Fowler. Microservices. `http://martinfowler.com/articles/microservices.html`.

[19] D. Groen et al. Software Development Practices in Academia: A Case Study Comparison. *CoRR*, abs/1506.05272, 2015.

[20] R. Harrison. How to avoid Puppet dependency nightmares with defines. `http://www.webcitation.org/6a1doDLla`.

[21] E. O'Neill et al. Cross resource optimisation of database functionality across heterogeneous processors. In *Proc. IEEE on Parallel and Dist. Processing with Applications*, 2014.

[22] O. Pell et al. Maximum Performance Computing with Dataflow Engines. In *High-Performance Computing Using FPGAs*, pages 747–774. Springer, 2013.

[23] F. Schintke. XtreemFS & Scalaris. *Science & Technology*, (6):54 – 55, 2013.

[24] M. G. Xavier et al. Performance Evaluation of Container-based Virtualization for High Performance Computing Environments. In *Proc. of Euromicro Inter. Conf. on Parallel, Distributed and Network-Based Processing (PDP)*, pages 233–240, 2013.