Operational Lessons from Chameleon

Kate Keahey
keahey@anl.gov
Argonne National Laboratory
Lemont, IL

Jason Anderson
The University of Chicago
Chicago, IL
jasonanderson@uchicago.edu

Paul Ruth
Renaissance Computing Institute
Chapel Hill, NC
pruth@renci.org

Jacob Colleran
The University of Chicago
Chicago, IL
jakecoll@uchicago.edu

Cody Hammock
Texas Advanced Computing Center
Austin, TX
hammock@tacc.utexas.edu

Joe Stubbs
Texas Advanced Computing Center
Austin, TX
jstubbs@tacc.utexas.edu

Zhuo Zhen
The University of Chicago
Chicago, IL
zhenz@uchicago.edu

ABSTRACT

Chameleon is a large-scale, deeply reconfigurable testbed built to support Computer Science experimentation. Unlike traditional systems of this kind, Chameleon has been configured using an adaptation of a mainstream open source infrastructure cloud system called OpenStack. In this paper, we discuss operational challenges for experimental testbeds and explain what impact they have on the profile of the operating team. We then discuss methods we developed to alleviate the operational burden and show how they can be used in practice. We conclude with a discussion of our interaction with the user community and describe how experimental platforms create a high potential for community involvement.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; **Reliability**; **Availability**; **Maintainability and maintenance**; *Reconfigurable computing*; *Heterogeneous (hybrid) systems.*

KEYWORDS

cloud computing, cloud operations, DevOps

ACM Reference Format:
Kate Keahey, Jason Anderson, Paul Ruth, Jacob Colleran, Cody Hammock, Joe Stubbs, and Zhuo Zhen. 2019. Operational Lessons from Chameleon. In *Humans in the Loop: Enabling and Facilitating Research on Cloud Computing (HARC '19), July 29, 2019, Chicago, IL, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3355738.3355750

1 INTRODUCTION

Computer Science experimental testbeds allow investigators to explore a broad range of different state-of-the-art hardware options, assess scalability of their systems, and provide conditions that allow deep reconfigurability and isolation so that one user does not impact the experiments of another. An experimental testbed is also in a unique position to support methods facilitating experiment analysis and improve repeatability and reproducibility of experiments. Providing these capabilities at least partially within a commodity framework improves the sustainability of systems experiments and thus makes them available to a broader range of experimenters.

Chameleon [9, 24] is a large-scale, deeply reconfigurable testbed built specifically to support the features described above. It currently consists of almost 20,000 cores, a total of 5PB of total disk space hosted at the University of Chicago and TACC, and leverages 100 Gbps connection between the sites. The hardware includes a large-scale homogenous partition to support large-scale experiments, as well as a diversity of configurations and architectures including Infiniband, GPUs, FPGAs, storage hierarchies with a mix of HDDs, SDDs, NVRAM, and high memory as well as non-x86 architectures such as ARMs and Atoms. To support systems experiments, Chameleon provides a configuration system giving users full control of the software stack including root privileges, kernel customization, and console access.

Unlike traditional experimental infrastructures such as GENI [5], Grid'5000 [8], Emulab [22], or CloudLab [35] which provide experimental capabilities by developing in-house infrastructures, Chameleon created an approach that provides similar and partially enhanced capabilities by building on and extending a mainstream open source Infrastructure-as-a-Service implementation: OpenStack [32]. Configured with OpenStack's Ironic [20] component that supports bare metal provisioning, and enhanced with new experimental features such as the Bring-Your-Own-Controller (BYOC) functionality [6, 36], Chameleon gives users full control of the software stack including root privileges, kernel customization, console access, as well as the ability to experiment with software defined networking (SDN). Using OpenStack as a foundational layer allows us to leverage the effort of the extensive OpenStack developer community and greatly increases the pool of potential operators of the testbed as many have had experience with OpenStack. The benefit of this approach is akin to using well-supported libraries;

HARC '19, July 29, 2019, Chicago, IL, USA

ACM ISBN 978-1-4503-7279-4/19/07…$15.00
https://doi.org/10.1145/3355738.3355750

we as operators benefit from patches submitted by authors or other members of the wider community, and we know that others are actively testing the same code we are using on a daily basis to run our infrastructure. It also allows us to give back: Chameleon is a core contributor to the OpenStack Blazar system [7], which is used to provide advanced reservations of physical hosts, network segments, and IP addresses [23].

Chameleon has been used to support many projects in research on operating systems, virtualization, power management, networking, high-performance computing (HPC), cloud computing, data science, machine learning, and others. A sizable proportion of Chameleon users also leverage the system for education in computer science. To date, Chameleon has supported 3,000+ users working on 500+ projects.

In this paper, we first discuss the challenges of operating an experimental platform like Chameleon and compare them with traditional scientific research platforms as well as clouds. We discuss what operating this type of infrastructure means to people and describe ways of streamlining these operations we developed to alleviate their burden. Finally, we discuss our interactions with users and explain why experimental platforms create a larger opportunity for community engagement.

2 OPERATIONAL CHALLENGES

2.1 Clouds versus HPC Resources

The cloud paradigm is still a relative newcomer in scientific datacenters most of which are operated as traditional HPC centers; to outline the challenges associated with running them we therefore briefly highlight the differences in the offering these two approaches present to the user, and the resulting differences in the complexity and cost of their operations.

On a typical HPC resource, users submit jobs via a batch queue which are run on an on-availability basis [26] in non-privileged accounts (e.g., no sudo access, limited access to various file systems, etc.). This approach to resource management thus optimizes provider concerns, i.e., emphasize utilization and relative simplicity of management against flexibility and user control: traditional HPC workloads sit in queues, sometimes for hours, and run once node resources are available. Another feature of this approach is that it reduces a significant complexity of resource provisioning and scheduling to a relatively simple and constrained user interface effectively trading-off user flexibility for a simpler and more manageable implementation.

Cloud resources on the other hand, emphasize support for interactive resources that satisfy hard user constraints for availability at a given time [23] – as well as flexibility in terms of configuration. To fulfill those requirements, users are given temporary time-controlled ownership of resources containers (typically provided via virtual machines) that are configured in a way of their choosing and give them high level of privilege (i.e., root). Users may devise many complex configurations across compute, networking, and storage resources, effectively requiring the system to correctly (and securely) configure virtual clusters or complex distributed environments.

These differences in offering and emphasis result in a range of differences in how these types of resources are used, but more importantly in their complexity, the associated operations cost, and the dynamics between the operators and user community. First, cloud systems are more complex because they solve a significantly more complex problem. One of the most significant differences is in networking: HPC job schedulers do not need to manage networks to correctly connect new user resources ("instances") to potentially multiple networks; much of the complexity of cloud computing comes from the fact that such networking needs to be reliably and dynamically managed. Further, correct management of security becomes significantly more complex in the clouds because the attack surface is larger and types of potential attacks more varied: users typically get root permissions to deployed instances that they configure themselves with all the potential for risk that this involves.

Another difference comes with management user artifacts. While cloud users in principle want to manage their own images, they typically also expect providers to maintain at least some popular configurations on hand as a startup base – together with associated artifacts such as orchestration or contextualization templates that allow users to deploy complex configuration such as virtual clusters. This implies not only additional system administration support but also the advisability of supporting community mechanisms that would allow users to share such configuration artifacts which calls for a tighter integration of the dialogue between users and operators. Last but not least, having been around for significantly less time, cloud computing is a less mature paradigm than HPC systems; this implies not only more operations costs but more "unknown unknowns" in operations and problem solving that typically require a higher level of expertise to address.

2.2 Clouds as Experimental Facilities

The configuration of Chameleon grew out of the desire to test the hypothesis that experimental facilities for Computer Science systems research can be configured as modern clouds. An experimental facility for Computer Science systems research has to support deep reconfigurability and both system and performance isolation [23], that is, it has to support the creation of isolated user environments (such as e.g., can be achieved by virtual machines and are available in the mainstream via commercial clouds) and offer direct access to hardware and firmware to support experiments that cannot be supported via a layer of virtualization to e.g., explore performance variability or power management. Chameleon uses OpenStack with Ironic (a component that implements bare metal deployment) to provide these capabilities. While this provides the bulk of capabilities that an experimental platform for Computer Science research needs to support, it does not provide them all; we've had to extend this base to support additional features such as allocatable resource management (i.e., the Blazar service referenced in Introduction), and add new ones such as support for networking experiments. Last but not least, in order to stay relevant, experimental platforms need to evolve as new research questions open up which means that an experimental platform is in constant state of development.

Operating at Bare Metal Level. Operating at bare-metal level increases the complexity of cloud management (and consequent operational demands) significantly. First, giving users access to bare metal resources and the ability to make changes to components

such as the BIOS significantly increases the complexity of security management. Also, offering resources via virtual machines or other types of container provides an additional layer of abstraction that is not present when resources are provided at bare metal, increasing the complexity of both developing/providing some of the features and operating the testbed.

While infrastructure maturity for cloud infrastructure is relatively underdeveloped, it is worse for bare metal: although most OpenStack components have now been integrated with Ironic, some important capabilities (e.g., Ironic's use for Cinder, the OpenStack remote block storage component, snapshotting, etc.) are still not available. It is also clear that the Ironic integration gets proportionally less attention (in terms of performance management for example) than more commonly used components of OpenStack: we have observed performance bottlenecks for scheduling of an instance on a bare metal machine, during provisioning of networks on physical switches, during the PXE-based instance provisioning itself, or even for user interface response.

These shortcomings made it necessary for us to develop a variety of adaptations. An example of issues created by the missing hypervisor abstraction layer is that security groups are unsupported for bare metal – yet, they are even more critical to support here, given the higher security management requirements. We overcame this by creating Firewall-as-a-Service [15]. Missing features, such as snapshotting, were implemented by our team albeit in a different form than is supported by OpenStack for virtual machines. While those adaptations allow us to provide a more complete offering to our users they do increase the operational complexity as infrastructure immaturity needs to be overcome or improved, new solutions need to be researched and developed, features need to be either contributed or patched with every upgrade.

Networking. Since cloud resources are by nature remote, networking is an important aspect of cloud computing experimentation. Most public cloud providers have rolled out advanced networking services that are simple to access by any cloud user, e.g. routing between regions and private networking spaces within a cloud. However, access to low-level, externally facing cloud network services such as AWS Direct Connect [10], Azure ExpressRoute [14], and Google Dedicated Interconnect [19] is difficult, expensive, or even simply impossible to most researchers without complicated support by campus IT staff, as well as national and regional network providers. As a result, most research on these services is being done by a few select scientists or campus IT staff themselves. Reducing the hurdles that prevent individual researchers from accessing these services enables a wide array cloud experiments, as well as provides expanded training to the next generation of campus IT staff in the use of these otherwise inaccessible services. Using networks for Computer Science experimentation is thus in a class by itself in terms of the demands it places on operators and the capabilities it offers to users.

The most significant challenge to increasing access to direct cloud network connection services is automation of key provisioning steps that currently require IT staff intervention. Public cloud providers provide a set of connection points (that we call "stitchports" [4]) in various geographic locations. Attaching a campus facility to a public cloud using a direct cloud connection requires a provisioning a series of private network circuits between the campus and a cloud connection point. Each circuit in the series will be provisioned by a different regional or national network transit provider. A typical direct cloud connection between a public cloud and a campus might make use of a shared cloud connection point provided by a national transit providers (e.g., Internet2). This national provider will provision a circuit between the cloud connection point and the campus' regional network provider. The regional provider will, in turn, provision a circuit between the national provider and the campus. Once in the campus, IT staff can connect the circuit to the desired local facility. Each of these circuits and connection points requires intervention by IT staff authorized to provision the appropriate infrastructure. In some cases (e.g. Internet2 and ESnet) network providers have APIs that the researcher, or more commonly the campus IT staff, can use to deploy the required infrastructure on their own. However, most regional providers and campus infrastructure have no such API and require manual intervention by trained IT staff.

Recent additions to Chameleon have enabled users to design and run experiments using AWS Direct Connect, Azure ExpressRoute, and/or Google Dedicated Interconnect without involving institutional IT staff. These experiments can connect Chameleon resources with the public cloud direct connections. Users can mimic existing or imagined campus infrastructure by deploying large scale experiments using Chameleon hardware and directly connect these "campuses" to public clouds. These connections take advantage of Chameleon's direct stitching capabilities and it's isolated user controlled OpenFlow networks using Corsa DP2000 series switches.

Evolution. A very significant complexity factor in operating a Computer Science research platform is the fact that by nature an experimental instrument has to change as the research frontier evolves. New research problems and new opportunities to solve them mean not only new hardware, and thus horizontally porting the infrastructure to new hardware types, but also the need for new vertical capability development. For example, in Chameleon porting our infrastructure to ARMs and now also various types of IoT devices required additional horizontal investment, while supporting new types of networking experiments to keep pace with the filed required additional vertical development. Finally, the idea of Computer Science testbeds itself is relatively young and evolving: for example, we are still in the process of understanding the potential of testbeds as a platform for repeatability and replication which drives discovery and feature development around that topic. This type of evolution is thus an inherent feature of an experimental testbed; to account for it development has to be a significant component of the operation of any infrastructure of this type.

2.3 People Dynamics

The increased operational demands of testbeds place a demand on operations specialists in two important ways. First, they require a significant level of expertise. This expertise needs to be broad and range across topics from systems administration, cloud infrastructure (i.e., OpenStack), to networking and security. While teams can of course include specialists in each area (such as e.g., dedicated network engineers or hardware specialists) at least some of the operators in integrative roles have to have cross-disciplinary expertise in all operational concerns. Further, experience in development

is critical, both with respect to DevOps operation style and to account for the need for adaptation and evolution described above. This need for a specialized mix of high expertise means that the on ramping process is complex and requires significant time for individuals joining the team.

Second, the complexity of the system also makes operating testbeds – including configuration, upgrades, as well as problem diagnosis and resolution – more complex, effort intensive, and time consuming. We employ three broad strategies to mitigate that. First, building on a mainstream open source infrastructure significantly cuts down on the development costs as we can integrate features developed by the broader community as well as rely on their support to some extent; though the cost of additional features persists. Second, much effort can be saved by developing streamlined operations models described in sections below; though those models themselves take significant time to develop which means that the effort invested there takes time to amortize. Finally, packaging the approach and making it available to others for easy deployment and integration creates the potential for better amortization of this work.

While all infrastructures can benefit from closer interaction with user community such as e.g., engaging users in community support activities, experimental testbeds have a stronger community interaction component than traditional infrastructures. This is due largely to the fact that there is a potentially large base of digital artifacts such as images and orchestration templates that can be shared among all of the community. In this case, successfully engaging the community in sharing them will ensure increased relevance on one hand (as the existence of the most current images will make the infrastructure more useful), and cut down on operations costs on the other. In testbeds, this potential for sharing gets even broader – but it also gets more important as it supports fundamental scientific patterns such as repeatability and replication. Finally, in testbeds understanding user needs goes beyond being able to provide incremental improvements such as ease of use and means keeping the hand on the pulse of evolving research needs.

3 STREAMLINING OPERATIONS

Over time we have made efforts to drive the cost of operating Chameleon down as low as possible by combining a strong "base" of commodity systems with layers of automation around error detection and resolution. What this means in practice is that, whenever possible, an error should be correctable in either an automated or well-documented process to both ease the burden and the expertise required of the operator.

3.1 Keeping the System Running

Achieving such automation is nontrivial in an environment like Chameleon, which consists of many OpenStack services, their dependent system services, additional bespoke applications, and a heterogeneous inventory of user-provisionable nodes. To tackle this problem, it is helpful to work backwards. The end goal of automation ("automatically resolve this problem") implies we have detected the problem. To detect the problem, we must have been able to measure some symptoms expressed by the system at large. We discuss each of these steps below.

Monitoring. Performing triage across the system is difficult because each piece of the Chameleon infrastructure has its own way of expressing errors. Bare metal servers may indicate via their baseboard management controller that a voltage reading is outside of bounds or that a cable appears to be unplugged, while an OpenStack service may be returning an HTTP error code or logging output to its error log. The nodes running the Chameleon services may themselves have underlying issues, such as a disk partition running out of space or the resources being overloaded. In the past, we used Nagios [29] to centrally collect and display this data, however, we found that this system worked best if you knew what questions to ask ("is the load on this machine high? Is the OpenStack API down?"), and we found that this is often not obvious. We needed a more holistic view of the normal operating state of the system so we could understand baselines and identify aberrant behavior.

To improve this, we adopted an approach of collecting metrics based on symptoms of failure, not failure itself. We deployed the Prometheus [34] monitoring system to provide operators with real-time performance metrics with fine granularity and broad coverage and implemented several custom metric exporters for Chameleon-specific symptoms (e.g., number of failed provisions.) We additionally configured all OpenStack services, which predominantly communicate their error state via logging, to log to one standard location, and then implemented automatic ingestion of all logs via Fluentd [16] into a searchable Elasticsearch [13] database. We integrated both the output of Prometheus and Elasticsearch with the Grafana [17] visualization tool to give operators an overview of the health of the system and make it easier to diagnose abnormal conditions, such as a large spike of network traffic or a large number of failed provisions that could be symptomatic of undetected errors.

In addition, we maintain a suite of black-box Jenkins [21] tests that periodically test various "happy paths" through the system, e.g. reserving a node, provisioning the node with a Chameleon-provided disk image, assigning a public IP to the node, and ensuring that external SSH connectivity works and that experiment metrics are being automatically pushed from the node once it is running. Depending on the scope of the test, they may run hourly (to give quick feedback on a particular system that may be in an error state), or daily (to act as an overall "smoke test" in to the health of the system from a typical user's point of view.) These tests serve two purposes beyond catching high-level errors: they expose holes in our fine-grained monitoring infrastructure, and also provide data points to allow us to correlate such errors to any detectable proximate causes.

Detection. Once enough data about the system was being collected via our monitoring infrastructure, we were able to more clearly understand the "shape" of some of our failure cases, e.g. provisioning errors increasing when a particular staging directory for disk images ran too full. This is where the benefits of symptoms-based metrics shined: we used the query language built in to Prometheus to write targeted alerts on our symptoms, which were then broadcasted via an Alertmanager [1] system to a visible communications channel (in our case, a Slack [38] channel dedicated to alerts.) Similarly, having all logs consolidated in a database allowed us to write custom queries that could detect certain failure cases we were aware of. This same approach was used to detect

and alert on system event log (SEL) messages from our fleet of bare metal nodes.

Resolution. Detecting a problem is not very useful if there are no discrete actions an operator can take to fix it. While certain mitigations might be known to an experienced operator, this knowledge takes time to accrue and means that juniors (or indeed, operators simply new to Chameleon's systems) have significant barriers to productivity. Therefore, we took steps to ensure that when an alert was written, it is linked to a corresponding runbook detailing more information about what the alert is expressing, what the user impact is, and what some possible (or known) mitigation steps are. The effect is that when an alert is seen, e.g. in a Slack channel, there is an easy path for an operator to find more information and get to work resolving the issue.

Some errors are predictable enough in their regularity that we can reliably fix the symptom, even if the root cause may go untreated. Typically these are issues where various OpenStack systems have not converged to a steady state, and a simple adjustment is often all that is needed, e.g. restarting the service or updating a database to make it consistent. While it is true that such issues should theoretically not arise, our experience has told us that this is the reality of running a testbed at this scale, and the simple solution of whacking the system in a few precise areas does the trick, especially as the time to investigate and solve the root cause exceeds the cost of implementing the automated workaround. We maintain a set of scripts, called "hammers" [18], that can fix up known bad states in the system quickly. In addition to fixing common errors, we also use hammers to prevent problems with system usage (e.g., detect conditions when the system is used in ways that are not consistent with our policies). Hammers are currently implemented as periodic Jenkins jobs, but can also be run ad-hoc by operators as needed; over time we hope to integrate them to automatically respond on-demand when the system enters a known error state.

3.2 System Maintenance

System upgrades. As the system grew in complexity, it became untenable to maintain a manual release process. Upgrading the OpenStack services in particular was a task that only expert operators could tackle, due to the experience necessary around the roles of each service and the interdependencies. This was further complicated by the configurations between multiple Chameleon sites getting out of sync with each other, leading to confusing situations where a certain error would appear in one site but not another. We tackled these problems by settling on an approach that automatically can handle upgrades, configuration changes, and new service deployments. For OpenStack services, we leverage the Kolla-Ansible [25] project, which consists of a set of Ansible scripts that can provision OpenStack as Docker containers [12]. Our monitoring infrastructure and custom applications are similarly deployed with a set of Ansible scripts we maintain. There are numerous benefits for this approach, which ultimately requires expressing all Chameleon infrastructure as code: it allows us to have a reliable process that easy to manually invoke if need be, supports automation as it is hands-off, and, most importantly, allows us to effectively package all of Chameleon as one artifact, deployable in a variety of environments.

Appliance releases. We rely on a series of tools that automate most of our disk image release process. In particular, we use DiskImageBuilder [33], a tool provided by OpenStack: it allows us to express our disk image build process as a sequence of provisioning scripts, which can then be run on top of an existing base image. This helps us reliably create images, yet we still must take care to release upgrades as the base image e.g. CentOS or Ubuntu distributions are updated. We solved this with another set of periodic jobs that, as new upstream OS releases are released, automatically trigger rebuilds of our disk images bound to that OS. These images are then released to the testbed as the latest version of the OS image. We also periodically test that these images deploy correctly on our various hardware configurations; for each type of machine, we schedule a test provision every week, for each disk image Chameleon provides.

4 PACKAGING OPERATIONS

Chameleon's utility increases with every deployment: more sites mean more specialized hardware, more capacity for experimentation, and more experiments. We have thus worked to package Chameleon (and the automated operational structure described above) such that others can deploy it with minimal configuration. Reproducing Chameleon using vanilla OpenStack components is possible, but challenging. For one, Chameleon maintains several forks of OpenStack services with minimal patch sets that are custom-tailored to the needs of a testbed; while most of them have been contributed it sometimes takes a long time for a contribution to be accepted. Furthermore, configuring a functional testbed that supports bare metal provisioning requires significant understanding of the underlying configuration of all the various OpenStack components. The human challenge then becomes: how do we make Chameleon feasible to install for as large community of operators without requiring them to absorb what is complex and specialized knowledge? In order to address it, we have worked to generalize the internal deployment of Chameleon such that it can be released and deployed by any user; we call this CHI-in-a-Box.

We boiled down the interface for CHI-in-a-Box is a set of configuration files. The site operator must define the inventory, or which physical machines will host which components of Chameleon, a globals YAML [40] configuration file, which defines important variables such as subnets and which parts of Chameleon should be enabled, and finally an encrypted passwords YAML configuration file, which contains secrets such as MySQL [28] passwords for each system user. By default, such passwords can be automatically generated using random strings if the operator wishes.

The deployment of CHI-in-a-Box, and indeed with the primary Chameleon sites, is handled via a commodity tool developed by the OpenStack community called Kolla-Ansible. Ansible is used to coordinate the deployment of pre-built Docker images on the operator's physical infrastructure. Rolling upgrades are supported, solving one of the biggest pain points with maintaining an OpenStack deployment. CHI-in-a-Box also automatically installs the automated testbed monitoring and operations tools described above allowing CHI-in-a-Box sites to reduce their operational costs.

Chameleon has chosen to "dogfood" CHI-in-a-Box internally. This means that any bug fixes or features that Chameleon operators add to the internal deployment of Chameleon can immediately

be leveraged by any other operators running Chameleon as an associate site.

5 INTERACTING WITH USERS

We interact with our users via a variety of communication channels starting with daily support interactions through a variety of training and information channels such as blogs, webinars, and the web page, and finally relatively rare events such as surveys and User Meetings.

In daily interactions, Chameleon users interact with staff by submitting support tickets [39]. The submitted tickets are processed by University of Chicago and TACC staff on alternating weeks; Chameleon is thus operated as a "single instrument" with ticketmaster staff having sufficient level of privilege on both sites to handle most problems. The ticket categories range between routine user profile management (comprising PI requests, allocation review and renewals, etc.), user questions (e.g., resulting from imperfect understanding of the system), system problems (e.g., network, hardware, software, or configuration failures), to allocation adjustment requests. The latter comprise a variety of special resource requests (e.g., lease extensions, FPGA access, or early user access to newly released capabilities) as well as fair sharing requests, e.g., the assignment of public IPs which may become unavailable [37] due to users' failure to release them.

Interacting via support ticket submission was selected over support via mailing list in anticipation of the scale of the system. The advantages of this mode of support are that it provides reliable ticket tracking and thereby ensures that no user problem is left behind. It also greatly facilitates weekly reviews of tickets which are one of our primary user feedback mechanisms and motivated the development of many new features. On the other hand, it prevents users from interacting directly with each other as they would have if support was given via a mailing list which inhibits the development of the community as users might share experiences relevant to not only direct support questions but also insights related to experimental methodology and approach. While we try to compensate by providing other communication channels such as the Chameleon blog, and periodic User Meeting, ultimately we may have to revisit this decision for community building purposes.

To facilitate sharing of experimental artifacts, such as images and orchestration templates we provide an appliance catalog where users can enter appliance description and link it to different versions of their experimental appliances. While some users contributed appliances effectively providing packaging of systems supported by their groups such as e.g., MPI [2, 3] or specific experiments [11, 27, 30, 31] we find that relatively few users use it to ensure repeatability of their experiments. We expect that this is partly due to still relatively few incentives for repeatable and replicable research and partly to continuing search for the format: our new integration with Jupyter notebooks is more promising in that respect and we plan to support it with better sharing mechanisms.

Overall, we find that a proper functioning of the system is a collaboration between the operations team and the users: a knowledgeable user community, sensitive to the need for fair sharing as well as operational needs of the system can significantly cut down on operations costs. Specifically, we are often indebted to our users for pointing out system shortcomings ranging from errors to insufficient documentation to desire for new features. On the other hand, much additional work is sometimes generated when the system is used improperly, or when best practices are ignored. In particular, proper management of shared resources, such as e.g., releasing nodes or public IP addresses or abstaining from stacking leases, is clearly required to implement fair sharing on the testbed; failure to do so often results in a significant support burden on the operators as they arbitrate the resources between users.

6 CONCLUSIONS

In this paper we have discussed the operational experience behind Chameleon, a large-scale, deeply reconfigurable testbed for Computer Science research.

Experimental platforms, such as Chameleon, expose a much broader and more complex interface to users than traditional scientific resources and consequently require both a higher level of experience and skill as well as potentially higher level of time commitment from the human operators. In addition, since they are designed to evolve in the set of supported use cases to follow the evolving research frontier, the operations of an experimental system will also require development skills. We described how streamlining operations can alleviate these costs and transform the system into one that lends itself better to support by a "devops co-op" where operations staff with high expertise packages operations in such a way that it can be operated with potentially lesser degree of expertise and skill making experimental platforms cheaper to operate and thus hopefully more easily accessible.

An important lesson learned from Chameleon is that experimental platforms include a higher potential for sharing and community participation than traditional systems. This is largely because many system-related artifacts such as images or orchestration templates, Jupyter notebooks, etc. can be relevant to broader sections of the community. In addition, experimental platforms serve as a "common denominator" for many experiments eliminating much of the complexity that goes into systematic experimentation, as well as repeating, replicating and producing variations on existing experiments.

ACKNOWLEDGMENTS

Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation. This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357.

REFERENCES

[1] Prometheus Alertmanager. [n. d.]. https://prometheus.io/docs/alerting/alertmanager/
[2] Chameleon MPICH 3.3.1 Appliance. [n. d.]. https://www.chameleoncloud.org/appliances/71/
[3] MPI Bare-metal Cluster (MPICH3) Appliance. [n. d.]. https://www.chameleoncloud.org/appliances/33/
[4] Ilya Baldin, Jeff Chase, Yufeng Xin, Anirban Mandal, Paul Ruth, Claris Castillo, Victor Orlikowski, Chris Heermann, and Jonathan Mills. 2016. ExoGENI: A Multi-domain Infrastructure-as-a-service Testbed. In *The GENI Book*. Springer, 279–315.
[5] Mark Berman, Jeffrey S Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. 2014. GENI: A Federated Testbed for Innovative Network Experiments. *Computer Networks* 61 (2014), 5–23.

[6] Divyashri Bhat, Jason Anderson, Paul Ruth, Michael Zink, and Kate Keahey. 2019. Application-based QoE Support with P4 and OpenFlow. In *Proceedings of the IEEE Conference on Computer and Networking Experimental Research using Testbeds (CNERT 2019)*. IEEE Press.

[7] OpenStack Blazar. [n. d.]. https://docs.openstack.org/blazar

[8] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, Noredine Melab, et al. 2006. Grid'5000: A Large Scale and Highly Reconfigurable Experimental Grid Testbed. *The International Journal of High Performance Computing Applications* 20, 4 (2006), 481–494.

[9] Chameleon. [n. d.]. https://www.chameleoncloud.org

[10] AWS Direct Connect. [n. d.]. https://aws.amazon.com/directconnect/

[11] Ryu OpenFlow Controller. [n. d.]. https://www.chameleoncloud.org/appliances/54/

[12] Docker. [n. d.]. https://www.docker.com/

[13] Elasticsearch. [n. d.]. https://www.elastic.co/

[14] Azure ExpressRoute. [n. d.]. https://azure.microsoft.com/en-us/services/expressroute/

[15] OpenStack Firewall-as-a-Service. [n. d.]. https://docs.openstack.org/neutron/latest/admin/fwaas.html

[16] Fluentd. [n. d.]. https://www.fluentd.org/

[17] Grafana. [n. d.]. https://grafana.com/

[18] Chameleon Hammers. [n. d.]. https://github.com/ChameleonCloud/hammers

[19] Google Dedicated Interconnect. [n. d.]. https://cloud.google.com/interconnect/docs/concepts/dedicated-overview

[20] OpenStack Ironic. [n. d.]. https://docs.openstack.org/ironic

[21] Jenkins. [n. d.]. https://jenkins.io/

[22] David Johnson, Tim Stack, Russ Fish, Daniel Montrallo Flickinger, Leigh Stoller, Robert Ricci, and Jay Lepreau. 2006. Mobile Emulab: A Robotic Wireless and Sensor Network Testbed. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*. IEEE, 1–12.

[23] Kate Keahey, Pierre Riteau, Jason Anderson, and Zhuo Zhen. 2019. Managing Alocatable Resources. In *Proceedings of The IEEE International Conference on Cloud Computing*. IEEE Press.

[24] Kate Keahey, Pierre Riteau, Dan Stanzione, Tim Cockerill, Joe Mambretti, Paul Rad, and Paul Ruth. 2019. Chameleon: a Scalable Production Testbed for Computer Science Research. In *Contemporary High Performance Computing: From Petascale toward Exascale* (1 ed.), Jeffrey Vetter (Ed.). Chapman Hall/CRC Computational Science, Vol. 3. CRC Press, Boca Raton, FL, Chapter 5, 123–148.

[25] OpenStack Kolla-Ansible. [n. d.]. https://docs.openstack.org/kolla-ansible

[26] Feng Liu, Kate Keahey, Pierre Riteau, and Jon Weissman. 2018. Dynamically Negotiating Capacity between On-demand and Batch Clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 38.

[27] Security Onion Network Monitoring. [n. d.]. https://www.chameleoncloud.org/appliances/59/

[28] MySQL. [n. d.]. https://www.mysql.com/

[29] Nagios. [n. d.]. https://www.nagios.org/

[30] ExoGENI Stitchable Network. [n. d.]. https://www.chameleoncloud.org/appliances/55/

[31] OpenFlow BYOC Network. [n. d.]. https://www.chameleoncloud.org/appliances/58/

[32] OpenStack. [n. d.]. https://www.openstack.org/

[33] OpenStack Diskimage-builder. [n. d.]. https://docs.openstack.org/diskimage-builder/latest/

[34] Prometheus. [n. d.]. https://prometheus.io/

[35] Robert Ricci, Eric Eide, and The CloudLab Team. 2014. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *USENIX* 39, 6 (Dec. 2014). https://www.usenix.org/publications/login/dec14/ricci

[36] Paul Ruth, Mert Cevik, Kate Keahey, and Pierre Riteau. 2019. Wide-area Software Defined Networking Experiments using Chameleon. In *Proceedings of the IEEE Conference on Computer and Networking Experimental Research using Testbeds (CNERT 2019)*. IEEE Press.

[37] Use Fewer IPs Save the Planet. 2019. https://www.chameleoncloud.org/blog/2019/02/27/save-planet-use-fewer-ips/

[38] Slack. [n. d.]. https://slack.com/

[39] Chameleon Tickets. [n. d.]. https://consult.tacc.utexas.edu

[40] YAML. [n. d.]. https://yaml.org/