Deploying Cloud-Native HPC Clusters on HPE Cray EX

Felipe A. Cruz
Swiss National Supercomputing Centre
Lugano, Switzerland
felipe.cruz@cscs.ch

Manuel Sopena Ballesteros
Swiss National Supercomputing Centre
Lugano, Switzerland
manuel.sopena@cscs.ch

Alejandro J. Dabin
Swiss National Supercomputing Centre
Lugano, Switzerland
alejandro.dabin@cscs.ch

***Abstract*—The software stack that manages a High-Performance Computing (HPC) cluster is a collection of applications and services put together by multiple engineers. Integrating all the software components is often complex and challenging. Therefore, the engineering effort frequently focuses on minimizing service disruption over the value delivery of new features. In this work, we introduce a cloud-native architecture for delivering an HPC cluster on top of HPE Cray EX that streamlines the development, operation, maintenance, and administration of the many services that compose an HPC cluster. Under a cloud-native approach, an HPC cluster is architected as a collection of small, loosely coupled services that can be independently delivered. Moreover, we leverage an on-prem cloud platform deployment that enables a self-service model for engineers to introduce controlled changes to the cluster while streamlining service and infrastructure automation. The presented cloud-native architecture is a starting point for delivering HPC clusters that are more resilient and scalable to operate.**

***Index Terms*—Cloud-native, HPC, clusters, services, operations, containers, system, Cray EX, CSCS**

I. Introduction

A High-Performance Computing (HPC) cluster consists of specialized hardware, facilities for hundreds to thousands of servers connected by a high-performance network, and software enabling the cluster to perform large complex scientific calculations extremely fast. Often, the nodes that form the clusters can be designed to have different combinations of hardware accelerators, huge memory capacities, large storage capacities, and low-latency networks. These provisioned nodes fulfill different roles with their software stack statically configured to work as either access nodes, job scheduling servers, filesystems servers, support services nodes, or work together as compute nodes within the cluster to solve a particular task.

Building the software stack that runs and manages the system is challenging:

- The stack is composed of multiple services, libraries, modules, and tools
- Different components of the software stack are developed and managed by multiple independent engineering teams that need coordination
- The delivery of the software stack has to contend with mutable components and applications with hard-wired parameters
- The process of integrating, validating, and deploying all the components can be labor intensive

Moreover, it is not uncommon for the software stack to be statically composed into a system image, often comprised of multiple layers built and installed on top of one another. The base layer contains the operative system (OS) and is the foundation for the rest. Subsequent layers extend features and functionality to the software stack, for example, these can range from administrator tools, workload manager services, and automated pipelines to compute environment libraries, compilers, and debuggers.

It is important to note that the software is installed in layers which creates interdependencies that can affect system performance. As the system infrastructure becomes larger and more complex, there is a greater potential for interactions between the components to display errors or conflicts that can impact system stability and reliability. Therefore, managing the interdependencies often requires careful planning and coordination among all stakeholders building the HPC system software stack. It is then the role of system administrators to manage the hardware and software complexity while also satisfying the many needs of stakeholders of the HPC system. Consequently, the HPC software stack solution often focuses on a monolithic process that builds a centrally-managed global system environment, with heavy system administrator involvement, to resolve all dependency conflicts and gatekeep any installations requiring privileged operations. Moreover, trying to automate the integration work can result in opaque automation due to the the breath and depth of the tasks, as such, only a handful of engineers have a full understanding of the context and impact of local and global changes.

As seen above, delivering the HPC cluster software stack is an exercise of compromise between managing the complexity of the system and the flexibility of introducing frequent changes to answer the needs of the many stakeholders. Moreover, this challenge grows non-linear with the number of stakeholders involved as HPC system administrators have to contend with the following:

- The system software stack is hard to validate due to far-reaching dependencies
- The change process can be resource intensive in effort and time
- To prevent service disruptions, updates are only allowed a few times per year, slowing down the rate of feature innovation

- Due to the many compromises made to build a stable system, advanced/niche features can not be adopted quickly
- The most challenging use cases follow a one-system per need, a solution approach that does not scale well concerning the system and human resources

Consider now the disruptive technological progress brought forward by the 'cloud' in their need to resolve the challenges brought by computing at scale. It is in this 'cloud' context that Infrastructure as Code (IaC) was born, modeling infrastructure and application lifecycle with code using software development best practices. Cloud technologies have focused on enabling developers to deploy scalable applications rapidly. A cloud-ish set of capabilities to manage HPC infrastructure is developed as part of the Cray System Management (CSM) [7], providing an application programming interface (API) to the HPE Cray EX supercomputer to configure the infrastructure dynamically. CSM allows CSCS to leverage a single large common infrastructure that can be customized via software to fulfill many needs. In turn, and to leverage cloud infrastructure, modern software development has shifted toward microservice architecture to increase scalability and reduce the complexity of services and applications. Cloud Native is a paradigm for deploying services that leverage cloud infrastructure capabilities to manage services at scale while reducing manual effort.

We will now discuss how a Cloud-Native approach can leverage HPE Cray EX supercomputer capabilities to manage the growing complexity of delivering HPC cluster services.

II. Cloud-Native HPC Clusters

Cloud-Native, as defined by the Cloud Native Computing Foundation [1], refers to a paradigm and architecture that leverages technologies like microservices, containerization, continuous integration and continuous delivery (CICD), and DevOps practices that have been optimized to maximize the benefits enabled by the dynamic environment capabilities provided by cloud platforms and infrastructure, allowing engineering teams to deliver software frequently and predictably with minimal effort.

Applying a cloud-native approach to HPC means that the many services that form an HPC cluster follow a microservice architecture, where services are broken down into a collection of smaller components that are loosely coupled and can be independently delivered from one another. Using microservices instead of a monolithic architecture enables engineering teams to work more efficiently since services are isolated and can be developed, maintained, and updated independently.

Whenever possible, the microservice architecture is supported by a containerization technique. The containerization of microservices aims to package the services into containers so that services are independent and isolated. Containerized microservices have their instance of operating system (OS) and dependencies independent from the host OS within the container engine where they are executed. Containerized microservices are lightweight, portable, and easily managed through a cloud platform, like Hashicorp's Nomad [3], used in this work.

Nomad, developed by Hashicorp, is a flexible cloud platform for managing containerized and non-containerized services that are deployed on clusters by its scheduler. The Application Programming Interface (API) provided by Nomad allows service developers to manage their services as code through declarative configurations. In this way, we observe multiple benefits:

- Automating tasks such as service deployments, service update rollout, and self-healing
- Simplifies the dynamic configuration of large number of nodes
- Improve efficiency by automatically managing resource utilization across the infrastructure resources
- Provides visibility and control over all managed services running on the infrastructure
- Presents a standard interface that can be leveraged for detecting issues with services
- Provides a central interface for managing services and their automation

To ensure service quality, the software development for the HPC services follows a Continuous Integration and Continuous Delivery (CICD) process that automates the services' builds, tests, deployments, and releases of its version-controlled artifacts. The goal is to detect bugs early during the service development, ideally before changes are introduced into production. This enables service developers to introduce changes frequently without being constrained by lengthy release cycles or slow deployment processes that restrict the ability to react quickly and efficiently to customers' needs. Moreover, leveraging CICD can maintain a consistent code base across cluster instances while enforcing quality control through automation.

The availability of modern interfaces to infrastructure and platforms like HPE's Shasta and Hashicorp's Nomad provides HPC engineering teams with access to modern development environments and tools for building, testing, deploying, automating, and managing their services under DevOps practices. In DevOps, an approach combining software engineering and service operation roles in a single team, automation provides the highest potential improvement for fast and efficient management of cloud-native HPC clusters. DevOps automation helps service teams to automate non-trivial tasks, reducing human errors during interventions or repetitive tasks, improving consistency and reliability during service delivery, and increasing productivity by automating manual tasks.

III. Cloud Interfaces

Using the CSM on HPE Cray EX supercomputer with Hashicorp's Nomad platform allows us to implement a cloud-native architecture. The infrastructure and platform-level capabilities serve needs that are managed by different engineering teams. As seen in Figure 1, this combination of capabilities and roles enables service scalability and flexibility on top of a unified infrastructure, i.e., one system that can fulfill many needs.

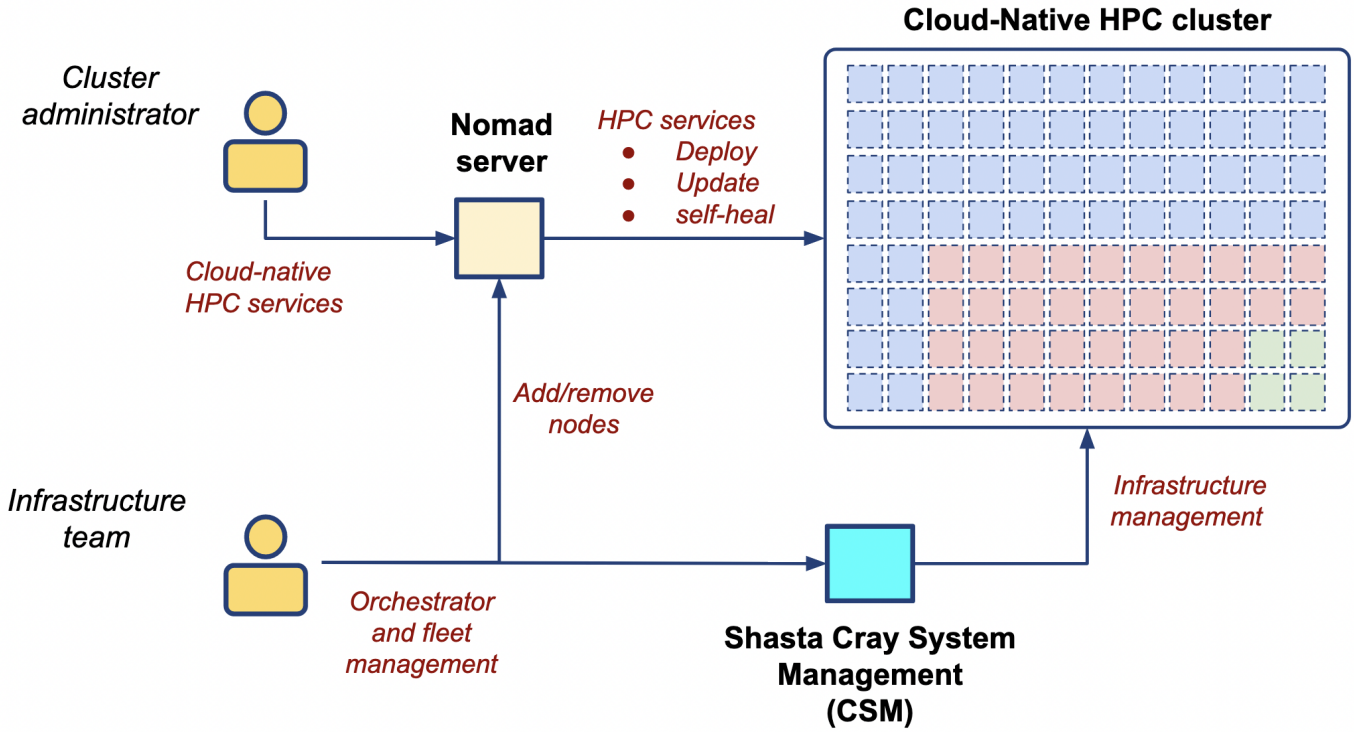

Fig. 1. Cloud-Native paradigm for managing an HPC cluster. Cloud capabilities at the infrastructure level provided by CSM for HPE Cray EX supercomputer and the platform level provided by Hashicorp's Nomad enabled us to implement a cloud-native approach to deliver HPC clusters on top of a unified infrastructure.

An infrastructure team uses the CSM interfaces to perform various tasks on the HPE Cray EX supercomputer, from building the base system image of nodes to power cycling and image booting. The CSM interfaces are also used for configuring nodes for resource access, like network configurations and filesystem availability. Once configured, nodes are logically grouped and managed as fleets by the Nomad orchestrator. The infrastructure team is then responsible for managing fleet membership for resource allocation and infrastructure health management, i.e., monitoring infrastructure health, removing faulty nodes, and adding healthy ones. Members of a fleet managed by the Nomad platform can then run cluster services as assigned by the platform orchestrator.

A cluster administration team can focus on building and deploying the cluster services. Using the Nomad platform enables engineers to introduce controlled changes to the cluster via quick redeployment of individual services, improving flexibility, scalability, and fault isolation while making the cluster more resilient, manageable, and observable. Furthermore, we leverage cloud platform interfaces in two ways: First, for the provisioning and delivery of services, this enables automation that can be used to streamline frequent granular releases of changes with reduced toil; Second, to partially manage infrastructure operations, for example, dynamic allocation of node roles according to business needs like batch compute, high-throughput computing, or cluster service nodes.

IV. Cloud-native cluster architecture

Let us now consider the architecture of a single cluster, where we have both Nomad servers and clients, see Figure 2. All cluster nodes are configured as Nomad clients that booted to the same base state consisting of the base OS, a container runtime, and the nomad agent.

Nomad servers manage client nodes that form the cluster, take cluster administrator requests for service deployments, and orchestrate service deployments in the cluster. A small set of nodes run nomad server instances for redundancy, these servers control all services placement and lifecycle in the cluster.

The nodes that form the HPC cluster run as Nomad clients by running a Nomad agent communicating with the Nomad servers. In this way, cluster nodes register themselves with the Nomad servers, wait for new tasks, and report any updates to the servers on the status of active tasks. Nomad servers then use the available resources reported by the Nomad agents to schedule services and assign work to clients.

Cluster administrators submit services for deployment in the cluster via a command line interface (CLI) or the Nomad application programming interfaces (API). These services are designed to be transient, i.e., impermanent with no long-term state changes, and are described within a *nomad job* that specifies the desired state of the service, artifacts to be used, and extra information that constraints the appropriate infrastructure resources to be used. The Nomad server then

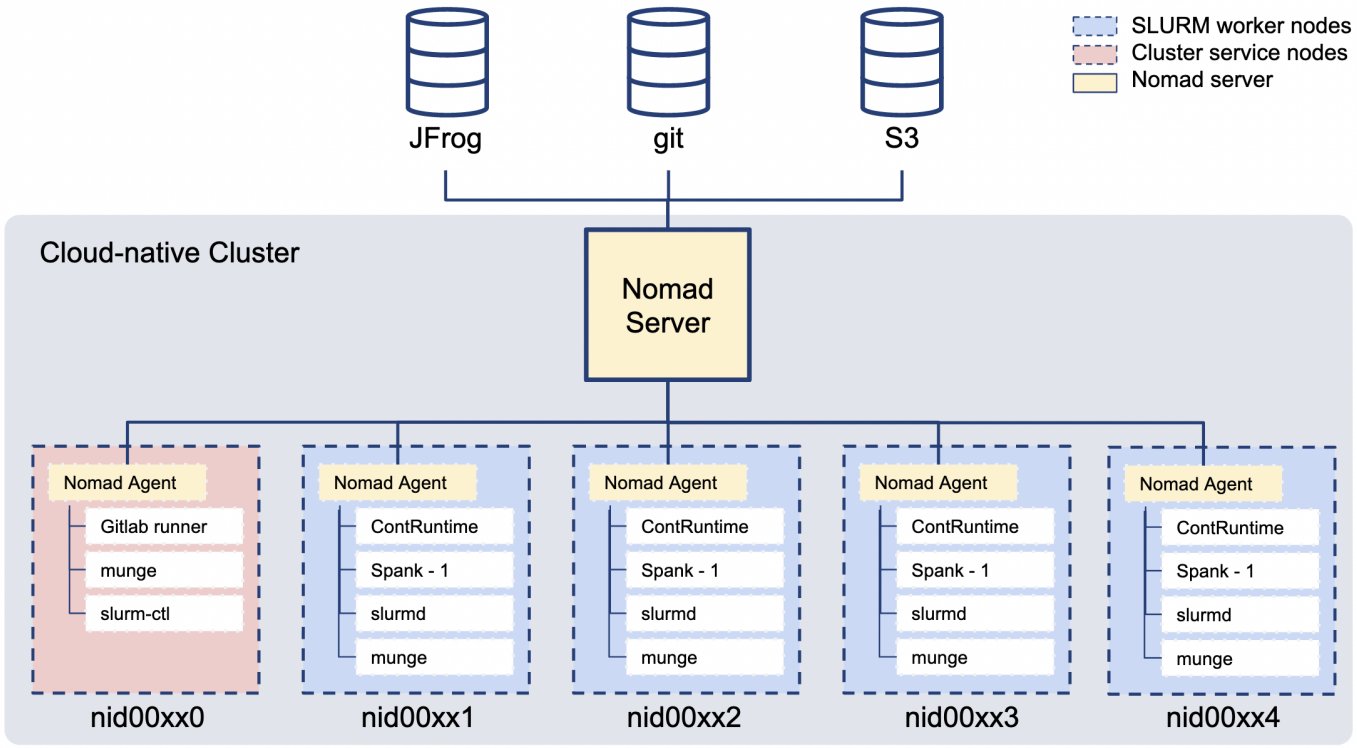

Fig. 2. Orchestrating the HPC services of a single HPC cluster with Hashicorp's Nomad. We have both Nomad servers and clients, with all cluster nodes configured as Nomad clients booted to the same base state. Cluster nodes register themselves with the Nomad servers, wait for new tasks, and report any updates to the servers on the status of active tasks. Nomad servers then use the available resources reported by the Nomad agents to schedule services and assign work to clients.

uses the service description and constraint information to decide the placement and schedule the service for deployment on the cluster.

A. *Service containerization challenges*

Despite the advantages mentioned earlier, using a cloud-native architecture for HPC clusters does not come without challenges. Consider that not all traditional HPC services are equally amenable to being deployed and delivered as microservices. Among the difficulties when containerized, we found:

- Not-easily containerizable applications composed of multiple components with complex interprocess communications
- Conflicts of cgroups between platform and application
- Conflicts of Linux namespaces managed by the application and container-runtime
- Complex management of direct access to infrastructure resources from a container
- Challenging to run privileged applications
- Challenging to run long-lived stateful applications built with no-dynamic redeployment assumptions
- Some services are daemon-less, only providing transient tools on the host
- Nuances of Linux namespace management for services that spawn child processes

The difficulties with containerizing applications like Slurm [8] have driven the choice of Nomad for our cloud-native architecture. Nomad is a simple and flexible workload orchestrator that deploys and manages containers and, more crucial for our use-case, non-containerized applications. Through Nomad, cluster administrators can use a cloud-native paradigm to manage and deploy traditional HPC services that are not easily containerizable alongside cloud-native services.

B. *Containerized User Environments*

The user environment is the layer of the software stack with everything needed to support users' workflows for development, debugging, testing, and job execution. As such, environments can include compilers, libraries, environment variables, and command-line tools. User environments meet additional challenges: the need to provide stability and flexibility in large systems with thousands of users; as user environments are coupled and built over the system software layer, they are subject to overall system cadence and validation with all that this entails.

To adapt the delivery of user environments to a cloud-native model, we leverage and extend HPC container technologies like Pyxis [6], Enroot [5], and Sarus [9] to deliver the user-environment software stack exclusively, quickly, and transparently via containers to users while seamlessly integrated with the capabilities of an HPE Cray EX system. A Container-first

approach for HPC improves flexibility toward specific user needs, the fast rollout of fixes, minimal system dependencies, and transparent utilization. Moreover, containers provide system decoupling, enabling consistent and stable workflows across system updates.

C. HPC services

In its most simplified form, when a cluster administrator defines HPC services to run in a cloud-native cluster, it does so using the *Nomad job* specification. Nomad jobs are specified in HCL, a specification language designed to be friendly for humans and machines. A Nomad job is a declarative specification for a task the Nomad cluster should run. Service deployments in the cluster follow the architecture described in Section IV. However, the actual Nomad job execution is provided by a *Task driver*, allowing Nomad clients to execute a task and provide resource isolation. Among the task drivers available, we find:

- The *Podman* [11] task driver plugin for Nomad uses the Pod Manager (podman) daemon-less container runtime for executing Nomad tasks
- The *raw_exec* driver executes a command for a task without any isolation and can be started as the same user as the Nomad process
- The *exec* driver executes a particular command for a task. However, unlike raw_exec it uses kernel isolation to limit the task's access to resources

Let us briefly introduce the service deployment strategy for a cluster administrator of cloud-native HPC clusters. The default approach to deploying a service is to containerize it and use the Podman task driver to execute it. With this strategy, we have successfully deployed a range of services that go from gitlab [4] runners to APIs to HPC, like Firecrest [10]. Moreover, we are exploring the development of many other well-suited services for containerization.

For services that are not easily containerized for reasons like the ones discussed in Section IV-A, we use *raw_exec* driver. The usage of this driver is an exception rather than the rule. It mostly applies to core node services like enabling the *container engine* for running containerized user environments or deploying a workload manager like Slurm, a use-case we discuss in detail in the next section.

D. Deploying Slurm as microservice

Consider now Slurm, one of the most challenging services to deploy under a cloud-native HPC cluster. Slurm is not container-friendly and suffers from many difficulties discussed in Section IV-A. As such, we opted to deploy Slurm using Nomad's *raw_exec* task driver. To describe Slurm in three microservices:

1) Slurm controller. The orchestrator of Slurm operations, which include job scheduling, compute node monitoring, and resource allocation. See Listing 1
2) Munge. The credential authentication service is used by all compute nodes of the slurm cluster. See Listing 2
3) Slurm Daemon. Daemon service For Slurm runs on all compute nodes. The Slurm Daemon monitors all running tasks on the node while also accepting, starting, and killing tasks on request from the controller. See Listing 3

The HCL service descriptions for the Slurm controller, Munge, and Slurm Daemon can be seen in Listings 1, 2, 3. To deploy these services with Nomad, the Cluster administrator can use the Nomad cli to first plan the changes to the system using `Nomad job plan` and then execute them if the changes are approved with `Nomad job apply`.

Points to take notice of from all the mentioned listings for Slurm:

1) use of `driver="raw\_exec"` task driver
2) specification of the user to execute the task can be set to any, e.g., `user="munge"` or `user="root"`
3) the command to execute and the argument to pass are defined by the parameters `command` and `args`

By reviewing the HCL for the Slurm services in more detail, we observe that the *type* of service for the slurm controller is different than for munge and slurm daemon. In Listing 1 line 4, we specify `type="service"` as we request Nomad only to instantiate the slurm controller service once. Moreover, we also specify the actual node where the service will start by using the constraints stanza as seen in Listing 1 line 18. Please note that this is not the only way to direct Nomad to instantiate services, and among others, it is possible to allow Nomad to do the node placement on its own.

```
1  job "slurm-ctl" {
2   priority = 95
3   datacenters = ["${var.datacenter}"]
4   type = "service"
5   group "slurm-ctl" {
6    task "slurmctld" {
7     driver = "raw_exec"
8     user = "root"
9     config {
10      command = "/usr/sbin/slurmctld"
11      args = ["-D"]
12     }
13    }
14    network {
15     port "slurmctl" {
16      static = 6817 # host linked port
            ↪ to TCP 6817
17     }}}
18  constraint {
19    attribute = "${attr.unique.hostname
        ↪ }"
20    value = "${var.slurm-ctld-host}"
21  }}
```

Listing 1. With filename 'slurmctld.hcl'. Contains a Nomad job description of slurm controller service for Slurm

```
job "munge" {
  priority = 95
  datacenters = ["${var.datacenter}"]
  type = "system"
  group "munge" {
    task "munge" {
      driver = "raw_exec"
      user = "munge"
      config {
        command = "/usr/sbin/munged"
        args = ["--foreground", "--
            ↪ syslog"]
      }
    }
  }
}
```

Listing 2. With filename 'munge.hcl'. Contains a Nomad job description of Munge service for Slurm

In the case of the services for Munge and the Slurm daemons, these need to run on every compute node of the slurm cluster. Moreover, the services must start in order, with Munge starting first and the slurm daemon starting next. We can easily achieve the service start order with another automation tool like Terraform [2]. While in order to achieve for the jobs to start on all compute nodes, we request Nomad to deploy the services as `type="system"`, this ensures that the services are started only once per node.

V. Benefits from cloud-native

We will now discuss some benefits of delivering an HPC cluster using the presented cloud-native architecture.

A. Cluster bootstrapping

As the services necessary to deploy an HPC cluster are described in the Nomad specification, we can manage the cluster in consistent and repeatable ways as the service descriptions, configurations, and artifacts are versioned, enabling reuse and sharing. In this way, once acquiring compute resources, a cluster administrator can quickly deploy a new cluster instance by requesting Nomad to orchestrate the instantiation of the cluster from the description of the services. Moreover, this is quick as all nodes are already in a live state and waiting for work. Thus, the cluster startup time is dominated by the time services take to start: Munge, Slurm, and others.

B. Service reconfiguration on a live cluster

Consider now a well-designed microservice built so that its parameters and capabilities can be modified with minimal impact on the cluster end-users.

For instance, let us take as a thought experiment a microservice based on the Slurm workload manager. Slurm uses *slurm.conf* as a configuration file that can be managed as versioned artifacts stored in a repository. Configuration changes, like scheduling parameters, can be made by updating the artifact and the corresponding service definition to Nomad

```
job "slurm-cn" {
  priority = 95
  datacenters = ["${var.datacenter}"]
  type = "system"
  group "slurmd-cn" {
    # Each task is scheduled on a
        ↪ different node
    constraint {
      operator = "distinct_hosts"
      value = "true"
    }
    task "slurmd" {
      driver = "raw_exec"
      user = "root"
      config {
        command = "/usr/sbin/slurmd"
        args = ["-D", "-Z", "--conf-
            ↪ server", "${var.slurm-ctld
            ↪ -host}", "--conf", "
            ↪ Feature=compute"]
      }
    }
    network {
      port "slurmd" {
        static = 6818 # host linked port
            ↪  to TCP 6818
}}}}
```

Listing 3. With filename 'slurmd.hcl'. Contains a Nomad job description of Slurm daemon on compute nodes for Slurm

and redeploying the Slurm controller without necessarily disrupting any active Slurm worker.

In an analogous way to our Slurm example, HPC services and automation have to be carefully designed to take advantage of the platform functionality to provide an experience to end-users that is as seamless as possible.

C. Service update on a live cluster

System updates are one of the core responsibilities of a cluster administrator, and this can cover scenarios that range from minor microservice version updates to the rollout of major security vulnerability upgrades.

As such, the implementation of a service update can vary depending on the required service disruption. For minor service updates, rolling updates might be possible, allowing for the update to take place with zero downtime by automatic and incremental redeployment of service instances with updated ones.

D. Node management

Rearranging or reassigning nodes between different target usage can also be done using the Nomad platform features to specify deployment constraints. In this way, nodes can be assigned to different work pools: batch computing, high-throughput computing, interactive computing, cluster services, or others.

E. Base node update

Upgrades to the Linux kernel or any components in the base node state, like the container engine or the orchestrator, are made by rebuilding the base image of the node and rebooting nodes to the new state. Note that this type of information can be passed as *node attributes* to Nomad, which can use them as scheduling constraints, enabling jobs to target specific node configurations. The update of the nodes is done over the CSM layer by the infrastructure team.

VI. Conclusion

The presented cloud-native architecture can be seen as a starting point to build and deliver HPC clusters that are more resilient and scalable to operate on top of a common infrastructure enabled by HPE Cray EX supercomputer. The proposed cloud-native architecture breaks down the HPC cluster into smaller, independent services that enable different engineers to work on different components without interfering with each other. It also gives different engineering teams greater autonomy when operating and managing services and infrastructure. The common infrastructure and platform interfaces provided by CSM and Nomad enable standard automation of management tasks improving reproducibility and simplifying the services integration processes while facilitating collaboration among teams. Moreover, fast troubleshooting and self-healing are possible using a cloud platform and tools with improved monitoring and observability of the state of the cloud-native cluster and its services. Consequently, a cloud-native HPC cluster is a novel approach for managing the complexity of an HPC system's operation, maintenance, and administration.

References

[1] Cloud Native Computing Foundation. "Who we are. Cloud Native Definition." https://www.cncf.io/about/who-we-are/ (accessed April 17, 2023)

[2] Hashicorp. "Introduction to Terraform". https://developer.hashicorp.com/terraform/intro (accessed April 17, 2023)

[3] Hashicorp. "Nomad documentation". https://developer.hashicorp.com/nomad/docs (accessed April 17, 2023)

[4] Gitlab. Gitlab Runners. https://docs.gitlab.com/runner/ (accessed April 17, 2023)

[5] NVIDIA. Enroot github repository. https://github.com/NVIDIA/enroot (accessed April 17, 2023)

[6] NVIDIA. Pyxis github repository. https://github.com/NVIDIA/pyxis (accessed April 17, 2023)

[7] HPE Cray. Cray System Management Documentation. https://cray-hpe.github.io/docs-csm/en-10/ (accessed April 17, 2023)

[8] Yoo, A.B., Jette, M.A., and Grondona, M. SLURM: Simple Linux Utility for Resource Management. JSSPP 2003. Lecture Notes in Computer Science, vol 2862.

[9] Benedicic, L., Cruz, F.A., Madonna, A. and Mariotti, K., 2019, June. Sarus: Highly Scalable Docker Containers for HPC Systems. In International Conference on High Performance Computing (pp. 46-60). Springer, Cham.

[10] F. A. Cruz et al., 'FirecREST: a RESTful API to HPC systems,' 2020 IEEE/ACM International Workshop on Interoperability of Supercomputing and Cloud Technologies (SuperCompCloud), Atlanta, GA, USA, 2020, pp. 21-26, doi: 10.1109/SuperCompCloud51944.2020.00009

[11] Podman. "What is Podman?". https://docs.podman.io/en/latest/ (accessed April 17, 2023)