Received: 6 April 2023 | Revised: 31 August 2023 | Accepted: 21 November 2023

DOI: 10.1002/cpe.7976

RESEARCH ARTICLE

WILEY

Enabling the execution of HPC applications on public clouds with *HPC@Cloud* toolkit

Vanderlei Munhoz | Márcio Castro

Department of Informatics and Statistics (INE), Distributed Systems Research Lab (LaPeSD), Federal University of Santa Catarina (UFSC), Florianópolis, Brazil

Correspondence
Márcio Castro, Department of Informatics and Statistics (INE), Distributed Systems Research Lab (LaPeSD), Federal University of Santa Catarina (UFSC), R. Delfino Conti, Trindade, Florianópolis–SC, 88040-900, Brazil.
Email: marcio.castro@ufsc.br

Funding information
Amazon Web Services; Conselho Nacional de Desenvolvimento Científico e Tecnológico, Grant/Award Number: 64/2022

Abstract

The advent of cloud computing has made access to computing infrastructure available to millions of users that face resource constraints. In the context of high performance computing (HPC), public cloud resources have emerged as a cost-effective alternative to expensive on-premises clusters. However, there are several challenges and limitations in adopting this approach. This paper proposes *HPC@Cloud* , a provider-agnostic open-source software toolkit that facilitates the migration, testing, and execution of HPC applications in public clouds. The toolkit takes advantage of various fault tolerance technologies to enable the use of inexpensive transient cloud infrastructure, commonly known as "spot" instances. Also, it features integration with singularity containers, allowing users to run complex applications on virtual HPC clusters in a portable and reproducible way. Finally, it provides a data-based empirical approach to estimating cloud infrastructure costs for HPC workloads. The results obtained on two public cloud providers (AWS and Vultr) show that: (i) *HPC@Cloud* can efficiently build virtual HPC clusters on the cloud; (ii) the new adaptive fault tolerance strategy outperforms other existing strategies based on blocking restoration; (iii) the integration of singularity containers into *HPC@Cloud* improves the portability of HPC applications to public clouds with negligible performance penalty to the applications; (iv) the proposed cost prediction approach can estimate the cost of running the applications on AWS and Vultr with up to 93% accuracy on average.

KEYWORDS

cloud computing, fault tolerance, high performance computing, message passing-interface, public cloud, spot instances

1 | INTRODUCTION

The cloud computing paradigm is defined as a *pay-per-use model* that allows convenient on-demand access to a configurable group of computing resources in a hassle-free manner, requiring minimum interaction with the provider.[1] Public cloud platforms allow anyone to access their services over the Internet without requiring a long-term contract. By leveraging economies of scale, the Cloud has democratized access to computing resources, making it available to millions of organizations and individuals. The pay-per-use model significantly lowers the capital barrier required to set up computing infrastructure, reducing investment risk and operational costs.[2] According to Gartner,[3] Amazon Web Services (AWS), Microsoft Azure, Alibaba Cloud, Google Cloud Platform (GCP), and Huawei are the leading public cloud providers today, collectively accounting for 80% of the market share.

Cloud providers can optimize their infrastructure usage by renting out their excess capacity at significant discounts, reclaiming it whenever a more profitable request arises. AWS pioneered this business model in 2009 as a way to sell idle infrastructure and increase revenue. Spot machines, also known as ephemeral instances, are typically used for this purpose. They offer considerably lower hourly costs than standard machines while running on the same type of infrastructure with identical performance characteristics.

In the context of High Performance Computing (HPC), public cloud resources offer an appealing alternative to expensive on-premise clusters. The spot market, in particular, presents exciting opportunities for small research groups to access highly-parallel infrastructures at significantly lower costs, even though applications must be fault-tolerant for spot instances to be viable. Furthermore, building a sustainable HPC cloud platform requires a mature software ecosystem.[4] Cost advisors, large contract handlers, DevOps solutions, automation APIs, and cloud-aware resource managers are among the software gaps that must be addressed to enable a fully functional HPC cloud platform.

Over recent years, cloud providers have launched numerous localized products and services, promoting them as HPC solutions. These offerings typically include infrastructure options featuring high-speed interconnections, such as InfiniBand, although they can be expensive. Cloud providers also provide large Virtual Machine (VM) instances equipped with hundreds of Virtual CPUs (vCPUs) to cater to applications with substantial processing demands. Since the mid-2010s, the rental of Graphical Processing Unit (GPU)-powered machines has gained traction, particularly for accelerating machine learning model training. In software tooling, the most significant contributions have been centered around task scheduling and virtualization technologies.[4] AWS, for example, launched the AWS Parallel Cluster tool[*] in 2016 to assist developers in building clusters using Amazon Elastic Compute Cloud (EC2) instances from the command line. However, despite their utility, these products and tools are exclusive to each provider, making migration between solutions challenging due to the absence of industry standards for cloud computing and HPC.[4]

This paper proposes *HPC@Cloud* : a provider-agnostic software toolkit that enables users to manage virtual HPC clusters in public clouds with minimal effort. Overall, we bring the following contributions[†] to state of the art on Cloud HPC:

1. We introduce a new open-source toolkit that leverages enterprise-grade technologies to help researchers to migrate legacy HPC code to public clouds. It features a flexible command line interface that allows researchers to easily specify the required computing resources and configurations when building the HPC infrastructure on public cloud providers. MPI applications can run seamlessly on the provisioned HPC cluster using TCP/IP or InfiniBand (when available) for communications;
2. We integrate into *HPC@Cloud* all blocking fault tolerance strategies tailored to tightly-coupled Message Passing Interface (MPI) applications running on AWS spot instances proposed in Reference 6;
3. We propose a new adaptive fault tolerance strategy that does not stop the MPI application when AWS revokes spot instances, thus providing better performance than blocking fault tolerance strategies proposed in Reference 6;
4. We introduce support for Singularity containers, allowing more portability when migrating legacy HPC code to the cloud, and evaluate their performance overheads;
5. We propose an empirical approach for estimating job execution costs on public clouds based on the execution of the target application with a small input problem size and gathered execution metrics, then later used to train a regression model for predicting total execution time; and
6. We provide a practical cost-efficiency analysis of executing a set of NAS Parallel Benchmarks (NPB) applications[7] and a parallel solver implementation for Laplace's heat diffusion equations based on a popular alternative Finite Difference Method (FDM) technique called Forward-Time Central Space (FTCS) method[5,8] on two popular public cloud platforms (AWS and Vultr Cloud) across multiple cluster configurations, giving an in-depth analysis of the advantages, main performance bottlenecks, costs, and pitfalls of using public cloud resources for HPC.

Attained results show that *HPC@Cloud* can have a significant impact on costs reduction, be it from the easiness of setting up a cluster in seconds, the cost saving when running workloads with spot infrastructure, or through giving reliable data for decision-making with our costs prediction method. The fault tolerance mechanisms proposed and embedded into *HPC@Cloud* have shown promising results, giving a range of options for the users to handle failures: User-Level Failure Mitigation (ULFM) for users that want the maximum performance possible and are not reluctant to dive into their application code and Berkeley Lab Checkpoint-Restart (BLCR) for running legacy applications that don't support ULFM or have too large of a code base to be easily updated with application-level fault tolerance. When used with spot revocation alarms, ULFM delivers the best results (around two times faster than periodic checkpoint restart with BLCR). Our results also demonstrate that spot instances do not guarantee cost savings, depending on the discount and the fault tolerance technique adopted.

This paper is structured as follows. Section 2 presents the background and motivation for this work. Section 3 discusses related work in this area. Section 4 presents our proposed software toolkit (*HPC@Cloud*) and explains the major trade-offs and the reasoning behind our decisions. In Section 5, we describe our evaluation method and provide details on the gathered experimental data. Section 6 analyzes and discusses the obtained results. Finally, Section 7 concludes the paper by summarizing our findings and outlining future work.

2 | BACKGROUND AND MOTIVATION

2.1 | Cloud computing platforms

Cloud computing provides a *pay-as-you-go* pricing model for computing infrastructure and software services comparable to public utilities, enabling organizations to cut down their information technology costs. This model lowers the entry barrier for small organizations as they can create and dispose of resources as needed, paying solely for what they use. With the ability to dynamically scale resources in real-time as per their application requirements, users can avoid over-provisioning or under-provisioning resources.

Cloud computing platforms are typically categorized into three types: *public*, *private*, and *hybrid*. *Public clouds* are designed to be accessible to anyone via the Internet without needing long-term contracts or direct interaction with the provider. In contrast, *private clouds* are typically accessible only to specific organizations or user groups with appropriate access permissions. Finally, *hybrid clouds* allow organizations to connect public and private clouds, enabling them to leverage both benefits while avoiding the risks of relying solely on a single provider or on-premises infrastructure.

Cloud providers maintain hardware in several data centers, giving users access to compute resources in the form of *instances*. A VM is a type of instance virtually allocated over a shared physical infrastructure and is usually the most commonly available and affordable option. Cloud providers offer various types of instances that vary in characteristics such as hypervisor type, CPU architecture, number of vCPUs, amount of memory, and more, allowing customers to choose the instance that best suits their needs. In the context of HPC, certain public cloud platforms offer specialized solutions specifically designed for demanding applications. These solutions include bare metal[‡] instances, hardware accelerators, and high-speed communications. However, these services are often available in limited capacities and have a considerably higher price tag than standard VMs.

2.2 | Spot markets

Cloud providers can optimize resource usage by renting out spare infrastructure at significant discounts, commonly known as the *spot market*. Spot machines are ephemeral instances that cost considerably less per hour than persistent (standard) machines and run on the same type of infrastructure with identical performance and characteristics. As the market has garnered significant interest from organizations seeking to reduce infrastructure costs, most major cloud providers now offer some type of spot market for Infrastructure as a Service (IaaS). However, the downside is that the cloud provider can revoke spot machines on short notice when capacity needs to be reclaimed for persistent instances, making it the user's responsibility to preemptively save data and recover applications in case of instance repossessions.

Each cloud provider operates under its own unique rules and pricing mechanisms. In this paper, we focus on discussing the AWS spot market. When requesting a spot machine on this platform, users must bid the maximum price they are willing to pay per hour for the requested resources. The provider will only fulfill requests if the specified spare capacity is available at a price lower or equal to the bid value. This model differs significantly from on-demand pricing, where an instance request is met instantly. It is essential to note that spot requests have no guarantee from the provider to be met, making this characteristic a crucial detail in contexts where workloads have a defined deadline for completion. Furthermore, spot prices for each instance type often fluctuate dynamically based on market demand in each availability zone, although this may vary depending on the provider. For example, Oracle offers a fixed discount for their spot machines, regardless of the available capacity.

2.3 | Container technologies

Containerization has become essential in modern software development due to its ability to accelerate and streamline application development and deployment processes.[9] Containers are widely used in cloud computing environments, enabling fast and scalable application deployment. Moreover, containerization tools like Docker and Kubernetes have become fundamental DevOps toolchain components, helping organizations develop and deploy applications more efficiently.[10] Although containers are lightweight compared to VMs, they can still impact performance, which may be a concern for some HPC applications. Nonetheless, the benefits of containerization for most other use cases far outweigh the potential drawbacks.

Over the years, containerization tools tailored for HPC have been developed to address the performance drawbacks of traditional containerization solutions. Sarus[11] and Singularity[12] are two such tools that have gained popularity in the scientific community. Singularity, in particular, was designed specifically for HPC environments and has been proven to have minimal overhead[13] since its containers use fewer resources and have lower latency than containerization technologies like Docker. Singularity allows users to bundle the entire software environment, including the operating system, libraries, and dependencies, into a single portable container that can be run on different HPC clusters.

Singularity offers several key features that make it an excellent fit for HPC environments, including seamless integration with HPC clusters and tools such as job schedulers, MPI, InfiniBand networks, and GPUs. It is also easy to use, even for users unfamiliar with containerization or HPC systems, and is widely adopted for research computing, scientific simulations, and other HPC workloads.

2.4 | Challenges for cloud and HPC convergence

In the context of HPC, the public-utility pricing model of IaaS offerings provides an appealing alternative for budget-constrained researchers and organizations. This is because it eliminates the need for expensive on-premise specialized hardware and substantially reduces the capital and technological entry barriers for HPC. Public cloud platforms have evolved to support the rapid allocation and deallocation of virtualized resources, allowing users to scale infrastructure dynamically based on their needs. As a result, most companies use cloud resources to ensure service availability and disaster recovery capabilities for their enterprise systems. In most cases, these companies do not require low-level knowledge of the underlying hardware or high-speed communication between nodes, both of which are typical requirements for HPC workload.[14] Instead, for most enterprise applications, the primary requirements are resiliency and fault tolerance, which can be achieved through redundancy: replicas of the same service are executed in different geographical locations to avoid single points of failure. The HPC use case, however, tends to be the opposite as nodes are allocated as close as possible to each other to reduce communication latency.

Although spot markets and alternative low-cost cloud providers may be appealing options for small organizations, traditional HPC still has a long way to go before effectively utilizing cheap cloud resources. Due to the possibility of spot instance repossessions during job execution, fault tolerance strategies are imperative for executing stateful applications, which is typical for most HPC workloads. Furthermore, since available capacity varies between different instance types, careful consideration is necessary when selecting instance flavors. The number of instances repossessions can be high and significantly slow down execution, even when using low-overhead fault tolerance mechanisms.

Unfortunately, the software ecosystem required to establish a viable HPC cloud platform is not yet fully developed.[4] There is a deficiency in cost advisors, large contract handlers, DevOps solutions, automation Application Programming Interfaces (APIs), and HPC-aware resource managers. For example, cost advisors commonly used in standard cloud environments must not only be capable of predicting the duration of a user's HPC jobs but also estimate how long an experiment with an unknown number of jobs will take to execute entirely. Additionally, it is necessary to have HPC-aware resource managers that can handle specific HPC workflows and guarantee high levels of performance, which typical cloud resource managers do not offer. Furthermore, large contract handlers and automation APIs that can manage thousands of machines simultaneously are needed to enable HPC workflows at scale. Such solutions are necessary for adopting HPC on cloud platforms and necessitate new solutions to overcome these challenges.

Recent research suggests that the existing pricing models for public cloud services are not tailored to the unique requirements of HPC workloads.[15] Since public cloud resources are shared among multiple users, the performance of HPC workloads can vary substantially each time they are tested, which makes DevOps for HPC a complex and challenging task. Moreover, the lack of specialized tools for managing HPC workloads in the cloud, such as HPC-aware resource managers and automation APIs, makes optimizing the performance and cost of these workloads in the cloud environment challenging. However, state of the art in Cloud HPC is characterized by rapid innovation, with cloud providers continually adding new capabilities and features to support the needs of HPC users. The availability of high-speed interconnects such as InfiniBand and RDMA over Converged Ethernet (RoCE) on the cloud has led to improved performance for parallel computing workloads by allowing low-latency communication between compute nodes. Containerization technology, such as Singularity, is also being used to simplify the deployment and management of HPC applications on the cloud, increasing portability and agility.

Additionally, HPC is being used as a foundation for Machine Learning (ML) and Artificial Intelligence (AI) workloads, with specialized hardware such as Tensor Processing Units (TPUs) offered by cloud providers to accelerate Deep Learning models. Many organizations are adopting hybrid cloud architectures that combine on-premises infrastructure with cloud resources to achieve optimal cost and performance for their HPC workloads. Integrating these cutting-edge technologies propels innovation in HPC and expands its reach to a broader audience.

3 | RELATED WORK

The utilization of public cloud resources for HPC has garnered significant attention recently. Numerous studies have delved into the challenges and opportunities of leveraging cloud resources for HPC workloads, addressing performance, cost, and security concerns. For instance, a recent study by Guidi et al.[16] showed that contemporary high-end cloud computing could provide HPC-competitive performance at moderate scales.

Despite these efforts, there is still a need for provider-agnostic software tools that can make it easier for the HPC community to build HPC clusters with public cloud resources. In this paper, we propose *HPC@Cloud* , a software framework that addresses some of the challenges and limitations of using public cloud resources for HPC workloads. We demonstrate the effectiveness of *HPC@Cloud* through an experimental analysis on two public cloud platforms: Amazon AWS and Vultr Cloud. While our research also employs microbenchmarks to assess performance gaps, it primarily focuses on low-budget instances lacking high-speed memory systems and interconnects featured in prior studies. Additionally, we delve deeper into the actual costs of deploying HPC applications to the cloud, a complex task that may ultimately constitute the most significant expense associated with cloud-based application execution.

Some studies concentrate on providing access to HPC applications on public cloud platforms rather than developing and running them directly in the cloud. Wong et al.[17] introduced a framework for deploying and offering HPC applications as Software as a Service (SaaS) solutions in public clouds. While their research addresses the provisioning of HPC applications, our study investigates the feasibility of executing HPC workloads using public cloud infrastructure. In addition, our study delves deeper into the cost-effectiveness of running these workloads on unreliable shared infrastructure (AWS spot instances).

Numerous other studies focused on developing workflows for HPC in public cloud environments, including the work from Vaillancourt et al.[18] focusing on the portability of HPC applications. Their research employs Docker and Terraform to build clusters on AWS and test multi-VM MPI applications, a methodology similar to our study. However, our work expands upon this by assessing containerization technologies designed for HPC, such as Singularity, and utilizing cost-effective AWS spot instances to reduce expenses further. Furthermore, we introduce and develop a software toolkit that enables users to seamlessly customize their infrastructure according to their requirements, eliminating the need for prior cloud-specific technical expertise.

Sindi et al.[19] shared insights from their implementation of a containerized HPC environment based on Docker for running multi-VM MPI workloads, investigating container migration techniques for fault tolerance. Their primary approach involved using Checkpoint/Restore In Userspace (CRIU) technology to migrate containers to different hosts in anticipation of potential failures. Although their research also addresses the containerization of HPC applications, it does not focus on executing them on public cloud platforms or analyzing the associated workflow costs.

Several studies explore the feasibility of using spot instances for HPC applications. Teylo et al.[20] comprehensively assessed the use of AWS spot instances for scheduling bag of tasks applications. We consider their research complementary to ours, offering valuable insights into employing spot instances. Our study extends their work by introducing a generalized software toolset for migrating any HPC workload to public cloud platforms. Amelie et al.[21] proposed a framework for optimizing costs with spot instances. Again, their research is complementary to ours, as their framework is focused solely on predicting prices in the AWS spot market without considering the use of AWS on-demand instances (or other provider's infrastructure) to minimize costs and the actual execution of HPC workloads on these instances with fault-tolerant mechanisms. Aniruddha et al. [22] researched redundancy for the cost-effectiveness of time-constrained HPC applications on AWS. Nevertheless, their results are not applicable anymore due to the recent modification in AWS spot market rules.[23]

We integrated into *HPC@Cloud* existing fault tolerance strategies to address the unreliability of spot instances.[6] These strategies are built upon existing technologies such as BLCR and the innovative ULFM, which is currently under development by the MPI Forum. Although numerous studies have explored fault tolerance using BLCR and other well-established technologies, they often do not comprehensively analyze the advantages and disadvantages of utilizing more affordable, less reliable cloud instances compared to high-end options in a public cloud and HPC context. We also propose and evaluate a new adaptive fault tolerance strategy based on ULFM that does not stop the MPI application when AWS revokes spot instances, thus providing better performance than blocking fault tolerance strategies proposed in Reference 6.

4 | THE *HPC@CLOUD* TOOLKIT

This section introduces our proposed software toolkit, *HPC@Cloud* , designed to utilize public cloud resources for running HPC applications. While the initial version supports two cloud providers (Amazon AWS and Vultr Cloud), the toolkit is readily extensible to accommodate additional providers. *HPC@Cloud* is an open-source software whose source code is accessible at https://github.com/lapesd/hpcac-toolkit.

4.1 | Overview of the proposed architecture

To facilitate the migration of legacy HPC applications to public cloud platforms, *HPC@Cloud* offers a suite of tools that can be executed on the user's machine. These tools enable users to configure cloud infrastructure, execute jobs, monitor performance, predict costs, and interact with the cloud in an automated and provider-agnostic manner. *HPC@Cloud* is open-source software developed using Rust for the primary command-line application and Python for automation and data-gathering scripts.

Figure 1 depicts a high-level overview of *HPC@Cloud* , presenting the existing integrations between its modules (colored boxes), third-party tools such as Terraform, and Cloud Providers' APIs. Overall, the software architecture comprises the following primary modules, each serving a distinct purpose:

1. A *command-line interface* from which users can send commands and receive information regarding virtual HPC cluster state and running workloads;
2. A *resource manager*, which handles provider integration, instances creation, monitoring, and destruction;
3. An MPI *job manager*, incorporating fault tolerance mechanisms to enable the use of AWS spot instances;
4. An experiment *metadata collection and storage module* designed to compile execution data to be used by the cost prediction module; and
5. A *costs prediction module* in which users can implement analytical or ML models to predict both makespan and runtime costs more accurately based on previous experiments.

In the following sections, we further detail the infrastructure and workload lifecycles showcased in Figure 1.

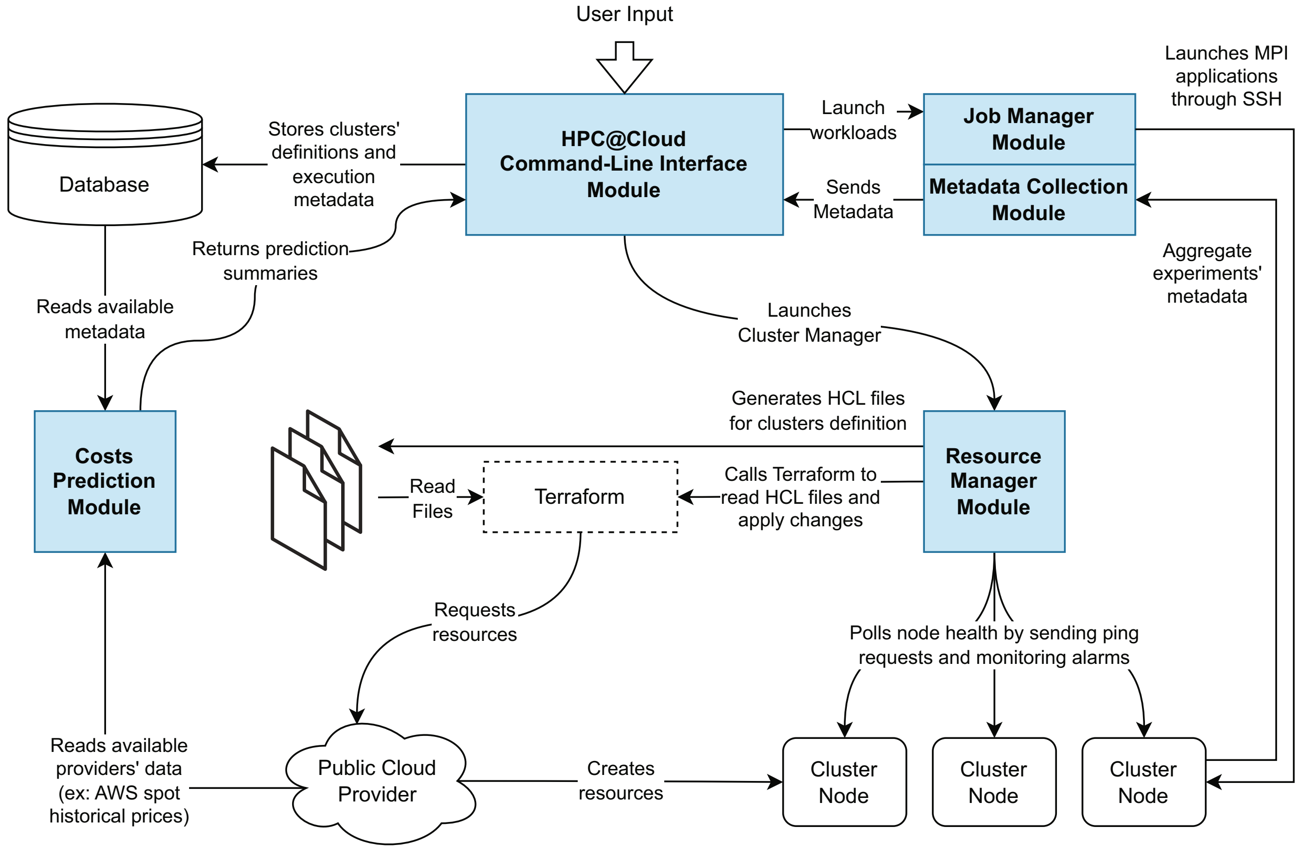
User Input
HPC@Cloud Command-Line Interface Module
Database
Stores clusters' definitions and execution metadata
Returns prediction summaries
Launch workloads
Sends Metadata
Job Manager Module
Metadata Collection Module
Launches MPI applications through SSH
Aggregate experiments' metadata
Reads available metadata
Launches Cluster Manager
Costs Prediction Module
Generates HCL files for clusters definition
Read Files
Terraform
Calls Terraform to read HCL files and apply changes
Resource Manager Module
Requests resources
Polls node health by sending ping requests and monitoring alarms
Reads available providers' data (ex: AWS spot historical prices)
Public Cloud Provider
Creates resources
Cluster Node
Cluster Node
Cluster Node

FIGURE 1 Overview of the *HPC@Cloud* software architecture (modules highlighted in blue color).

4.2 | Managing cloud resources

Despite standardization efforts, configuring and managing cloud resources frequently requires provider-specific knowledge about the platform and its APIs, which may be subject to arbitrary changes by the providers. Repeatable infrastructure management is a sought-after feature in both the HPC and enterprise domains. The current best practice for infrastructure management centers around the concept of Infrastructure as Code (IaC). IaC refers to the practice of configuring computing resources via machine-readable definition files instead of relying on interactive configuration tools.[24]

We have integrated the concept of IaC into the *HPC@Cloud* toolkit for creating, monitoring, and destroying cloud resources. To bridge the gap between cloud computing and HPC, our proposed toolkit aims to utilize pre-existing open-source tools whenever feasible. For infrastructure configuration, we chose Terraform[§], a tool that adopts a declarative approach to defining infrastructure. Terraform allows us to create virtual clusters on numerous cloud providers, including AWS and Vultr Cloud – two cloud platforms evaluated in this paper.

The *resource manager* comprises a command-line application capable of generating Terraform plans written in HashiCorp Configuration Language (HCL), a configuration language visually resembling JavaScript Object Notation (JSON). Users can provide high-level configuration parameters as input, such as the desired number of cluster nodes, instance types, storage, and memory options. The module then generates Terraform plans using template files. Terraform plans can then be applied using commands. When applying changes or creating infrastructure for the first time, Terraform checks the state of the desired cluster and defines which resources are stale, missing, or needing to be re-created with different configurations.

4.3 | Spot instance lifecycle management

The *resource manager* is embedded with a routine script that periodically checks spot nodes' health. We currently consider three strategies for the user when a spot repossession or node failure happens:

No Restoration. It ignores the failed node and continues running the application with the remaining instances. In this strategy, the target workload must be able to handle failures at the application level, usually using ULFM or some other mechanism provided by the user. The *resource manager* takes no action when a failure occurs.

Blocking Restoration. It checkpoints the workload state before a failure occurs (periodically or not) and then reconstructs the failed infrastructure by executing new spot bids. In this strategy, the application is paused until the cluster returns to its pre-defined state. During this period, the remaining healthy instances stay idle and will be billed by the cloud provider.

Adaptive Restoration. In case of spot evictions, it continues running the application with the remaining instances but automatically executes new spot bids to reconstruct the original infrastructure. This strategy enables uninterrupted workload execution (thus, minimizing idle infrastructure expenses) but requires handling failures at the application level, usually using ULFM or some other mechanism provided by the user.

Depending on the cloud provider, we can implement a mechanism to predict imminent failures, saving additional time when creating new instances and executing workloads. Specifically for AWS, we can enable spot repossession alarms that provide a two-minute warning before a spot instance is reclaimed. When the cluster is created with *blocking* or *adaptive restoration* and spot eviction alarms enabled, the routine script also monitors AWS eviction notices and creates new spot bids beforehand.

By default, *HPC@Cloud* provides some fault tolerance mechanisms for working with unreliable cloud infrastructures. These mechanisms are described in detail in Section 4.4.

4.4 | Managing fault-tolerant workloads

Workloads are managed using ubiquitous tools in the HPC world, such as `mpiexec` for MPI applications. This approach enables users familiar with these tools to seamlessly deploy their applications in the cloud, thus simplifying the migration process. Once a cluster is created, the *Job Manager* transfers user-defined application files to the cluster's shared file system and executes them using `mpiexec` over an Secure Shell Protocol (SSH) connection.

To support spot instances and other unreliable infrastructures, the *Job Manager* provides a set of fault tolerance strategies to resume execution and restore spot clusters through new spot requests. We integrated into *HPC@Cloud* two blocking restoration strategies proposed in Reference 6: one based on a system-level rollback restart mechanism using the BLCR package[25] and other based on the novel ULFM approach for checkpoint-restart. Moreover, we also introduce an entirely new adaptive restoration strategy based on ULFM that keeps applications running while waiting for new spot instances to be created. We also propose two variants of each strategy that dictate when checkpoints are triggered (periodically and reactively). The source code of our custom implementations with comments can be found at https://github.com/lapesd/hpcac-toolkit. The following sections briefly present these strategies and discuss their pros and cons.

4.4.1 | Berkeley labs checkpoint-restart (BLCR)

Using the BLCR approach for fault tolerance requires no changes to the MPI application code. To create and restore a checkpoint, the BLCR command-line tool is called by *HPC@Cloud* with the Process Identification Number (PID) of the MPI application launched. To take advantage of spot repossession notifications from AWS, *HPC@Cloud* orchestrates checkpoints preemptively and reactively, launching new spot requests and listening to spot eviction alarms sent by the cloud provider. *Periodic or preemptive checkpoints* are triggered in pre-defined time intervals, while *reactive checkpoints* are triggered only upon notification by AWS. Periodical checkpoints are necessary when spot eviction alarms are unavailable or unreliable, which may be the case for most providers. Some applications may also need more time than a short notice alarm to complete a checkpoint, which may render the reactive strategy moot.

In summary, the BLCR approach has the following limitations:

1. BLCR requires specific versions of Linux Kernel headers to work (older than 3.10.0-514 when testing with CentOS 7.3.16.11), which are difficult to install and have severe unpatched security issues;
2. Executables must be compiled as a static binary for BLCR to be able to checkpoint them;
3. BLCR only supports MPI over TCP/IP connections; thus, it cannot use InfiniBand and other types of high-speed networks;
4. Not all MPI library distributions support BLCR: currently, MVAPICH2 is the only MPI distribution with embedded support for BLCR, limiting options for the users; and
5. BLCR does not support adaptive restoration.

Despite these issues, BLCR is simple enough that almost any MPI application can be made fault-tolerant without requiring any source code changes.

4.4.2 | User-level failure mitigation (ULFM)

As the scale of parallel systems continues to grow, the likelihood of hardware and software failures also increases. This is also true for spot clusters, where the chance of a node eviction increases with the number of VM instances in use. ULFM offers a solution to manage failures by providing a set of extensions to the MPI standard, enabling application developers to handle process failures more resiliently. These extensions allow the detection, reporting, and recovery from process failures within the MPI applications.

Unlike traditional fault tolerance mechanisms focusing on checkpoint/restart or replication techniques, ULFM allows applications to adapt their behavior and continue execution in the presence of process failures, making adaptive restoration possible. However, this requires developers to design and implement their own solutions, tailoring them to their applications' specific requirements and characteristics.

In summary, the ULFM approach has the following limitations:

1. In contrast to BLCR, ULFM does not provide a ready-to-use solution; instead, users must design and implement a fault tolerance strategy of their own, requiring significant source-code modifications to the MPI application and making migration time-consuming and costly;
2. Not all MPI library distributions currently offer comprehensive support for ULFM, though the most popular ones either provide limited support or are working towards implementing it. Among the available options, the development branch of OpenMPI currently delivers the most robust ULFM support;
3. At present, ULFM APIs for MPI are exclusively available for C programs, rendering this strategy inapplicable to code written in other languages.

Despite its limitations, employing ULFM significantly enhances checkpointing efficiency compared to system-level methods such as BLCR and offers users inventive methods for maintaining application continuity. When creating a cluster with *HPC@Cloud* , if the user enables the ULFM support flag, a compatible version of OpenMPI and all necessary system dependencies are installed on the cluster nodes. ULFM has a massive advantage over BLCR in supporting InfiniBand and other high-speed networks, making the use of high-end cloud computing VMs that support this type of specialized hardware possible.

Both ULFM and BLCR strategies proposed in Reference 6 relied on blocking restoration (i.e., the application is suspended when the cloud provider revokes any spot instance). In this paper, we introduce a new *adaptive restoration strategy using ULFM* that allows applications to continue executing without waiting for new spot instances to be created. The proposed solution is depicted in Figure 2. Let us consider that an MPI application is running with eight MPI ranks (*R1–R8*) on four spot nodes (*Node-1–Node-4*). Let us also assume that each spot node has two vCPUs and there are two MPI ranks per node (①), each one running on a different vCPU. Assuming that the cloud provider revokes *Node-4* (②), the *HPC@Cloud* 's *Resource Manager* will immediately request a replacement node (*Node-5*), and all failed MPI ranks will be oversubscribed to healthy nodes (③), resuming the work from the latest checkpoint (④). This allows the application to keep running but with a damped performance, since MPI ranks *R1*/*R7* and *R3*/*R8* are sharing the vCPUs. When eventually *Node-5* is restored, the application is again rebalanced to its original number of MPI ranks per node by killing the oversubscribed MPI processes (sharing the same machine cores as the original ranks) and respawning them on the new instance (⑤). This strategy eliminates idle time waiting for spot requests to be handled by the provider. To the best of our knowledge, *HPC@Cloud* is the only toolkit that features adaptive restoration with ULFM tailored to HPC applications running on cloud spot instances.

4.5 | Machine images, communication, and containerization support

Users can select basic operational system images when spawning VMs on public cloud platforms, which are usually the Long-Term Support (LTS) releases of popular Linux distributions. Most cloud providers also allow customers to save snapshots of modified machine images. As using a snapshot to create a new machine is much faster than installing everything from scratch every time, *HPC@Cloud* leverages this feature to reduce cluster setup time. The user can use pre-defined machine images with the latest packages and libraries needed by *HPC@Cloud* or provide his own machine image.

To execute installations that depend on the configuration of other machines, such as setting up a Network File System (NFS) server for shared storage and connecting clients to it, users of *HPC@Cloud* can define shell scripts to be executed immediately after a VM is instantiated. To minimize makespan, dependencies must be installed beforehand and copied as static binaries or a custom machine image must be utilized.

While creating clusters, users can enable multi-VM containerized execution of MPI applications. If this option is activated, the *Resource Manager* installs Singularity[¶] from the source on all machines, incorporating a compatible distribution of the MPI library, which could be MPICH, OpenMPI, or MVAPICH2. To allow applications inside Singularity containers to communicate with each other, we have implemented a hybrid approach wherein an MPI distribution is installed inside the containers as well, allowing it to communicate seamlessly with the MPI process from the host VM. The MPI program is executed directly on the host VM, like any other traditional MPI application, using `mpiexec` on the `singularity` command itself.

At this point, processes within the container function as they would typically do directly on the host VM. The benefits of this workflow include seamless integration with resource managers and simplicity, as the process closely mirrors natively running MPI applications. However, the

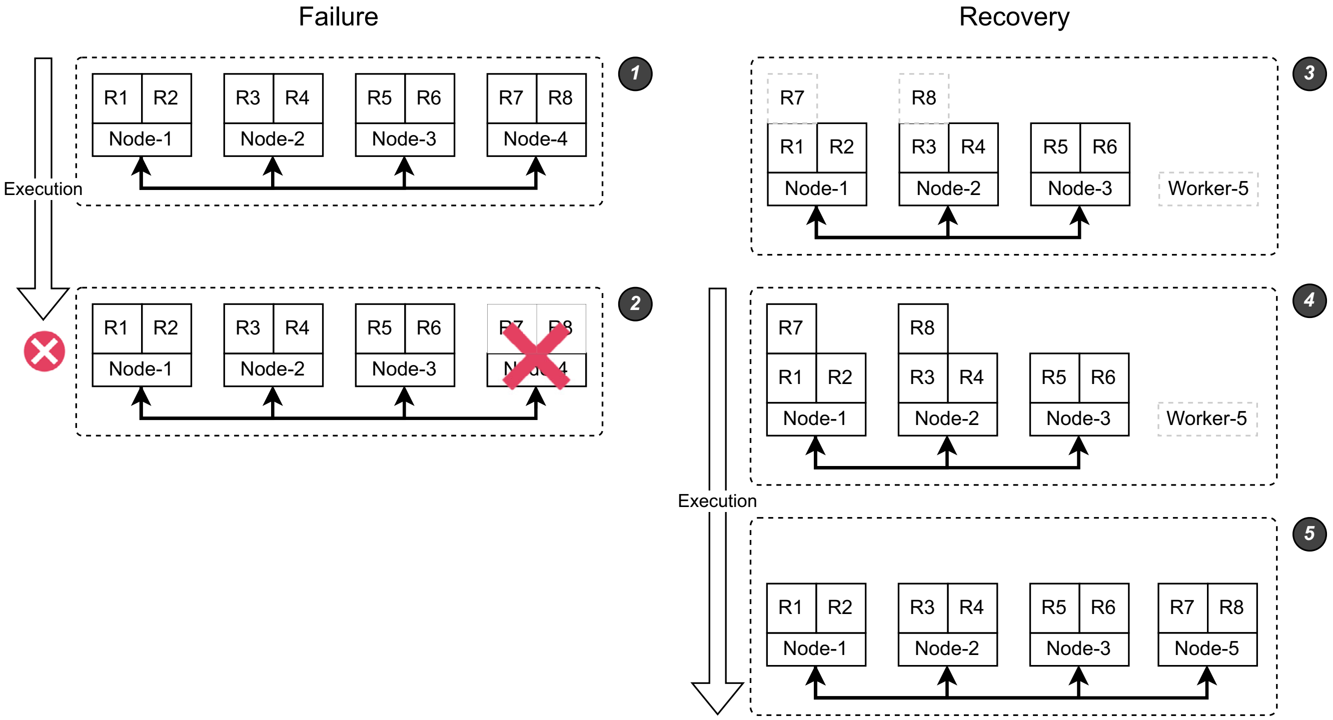

FIGURE 2 MPI oversubscription approach for adaptative ULFM fault tolerance.

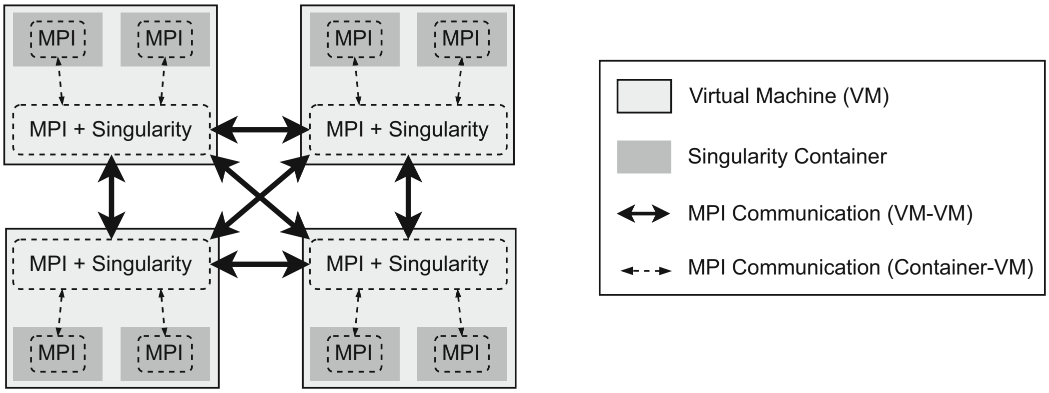

FIGURE 3 MPI hybrid model: Message passing communications between VM hosts and containers.

drawbacks of this workflow involve the need for compatibility between the MPI version within the container and the host's MPI version, as well as the necessity for careful configuration of the container's MPI implementation to ensure optimal hardware usage, especially when performance is crucial.

The hybrid model for MPI communications between Singularity containers operates by establishing a bridge between containers utilizing the MPI distribution present on the VM hosts. This implies that for one container to communicate with another, it must first engage with the host MPI, followed by host-to-host communication. Subsequently, the host containing the target container forwards the information to the container MPI process. This communication hierarchy is illustrated in Figure 3.

4.6 | Metadata gathering and cost prediction

HPC@Cloud automatically gathers metadata from workloads executed using the command-line interface. The collected data encompasses the selected cloud provider, number of nodes, cores per node, memory per node, storage type and size, and node-to-node bandwidth (measured via

Python scripts right after a cluster is created). Additionally, the system records the time to set up the cluster, from the initial request until the complete execution of the Terraform plan generated by *HPC@Cloud*'s *Resource Manager*. The total workload execution time, from the launch of an MPI job until its completion, is also stored. For workloads running on spot clusters, the number of spot instance evictions during execution and the total time spent waiting for spot requests to be fulfilled are also stored. The data collected by *HPC@Cloud* is then used for execution time and cost estimation.

Predicting cloud computing costs has become a significant concern for companies and organizations. Although the initial information technology investments can be avoided, over time, cloud computing expenses may exceed those of operating a physical data center, depending on the scale and types of services utilized.[26] As Netto et al.[4] highlighted, accurate cost predictions remain a significant challenge in developing a fully realized cloud HPC platform.

For multi-VM MPI applications, the total cluster makespan constitutes the most significant cost, as storage and network expenses are negligible compared to compute time prices per hour. Makespan costs are billed by various increments, such as months, hours, minutes, or even seconds, depending on the cloud provider, with most providers offering discounts for long-term purchases. In the case of applications running on AWS, EC2, and storage services have been billed in seconds since 2017. Elastic GPUs are also billed by seconds, albeit with a 1-min minimum.

Estimating costs thus becomes a matter of determining the workload's runtime, which is a complex problem due to the numerous variables involved. For spot instances, despite their variable pricing, the smoother price fluctuations since 2017 have made time series prediction a viable option for accurately predicting the spot price of specific instances.[27] However, public cloud computing resources are inherently shared, resulting in unreliable performance. Identical VMs may behave differently, as the underlying hardware state is unknown.

At least for uniform workloads with minimal time variations, such as grid-based physics simulations (a large portion of traditional HPC), it is possible to estimate the total execution time with acceptable accuracy based on the data collected from sample experiment executions (see Section 6). This approach enables users to test their applications for bugs and performance using cloud computing machines and quickly destroy resources after testing if desired.

To estimate costs, *HPC@Cloud* employs a straightforward yet effective Linear Regression ML model, which is trained using data gathered from previous experiments. Users can expedite the process of building a dataset for prediction by providing *HPC@Cloud* with a scaled-down version of their application, minimizing costs during testing. Also, by running numerous experiments with various cluster combinations, users can accumulate the necessary dataset for training the prediction model. As more experiments are conducted over time, the accuracy of the *HPC@Cloud* model continually improves. The Linear Regression model is trained using the `scikit-learn` implementation.

5 | EVALUATION METHODOLOGY

This section presents our evaluation methodology to assess the most recent version of *HPC@Cloud* . First, we describe the experiment environment used to carry out all experiments (Section 5.1). Then, we briefly describe the applications considered in our study (Section 5.2). Finally, we discuss the metrics and scenarios considered in the experimental evaluation (Section 5.3).

5.1 | Experimental environment

We focus our experiments on low-end instance types with eight vCPUs. The exception is the AWS's `r6idn.16xlarge` instance, which is designed for memory-bound and data-intensive HPC workloads. Our goal with this HPC instance is to evaluate the benefits of its processing power and high-speed network when running our target parallel applications when compared to low-end instances. AWS instances are created at the `us-east-1` availability zone located at Washington DC (USA), and Vultr instances are created at the `dfw` availability zone located at Dallas (USA). All VM instances used in the experiments and their characteristics as informed by each cloud provider are detailed in Table 1. The network interconnection utilizes standard Ethernet, with MPI communications done via TCP/IP.

We prepared a series of cluster configurations as shown in Table 2. Most of the experiments are based on virtual clusters composed of eight nodes. The only exceptions are the experiments with *AWS-r6idn* (single node) and the experiments to evaluate the overhead of containers (16 nodes). All cluster nodes are either based on the same on-demand or spot VM instance, and all resources are isolated inside a single-zone Virtual Private Cloud (VPC) resource. One cluster node is chosen to host an NFS server, and all nodes have a persistent block storage device attached to them to implement a shared file system. We execute one MPI rank per vCPU to avoid core sharing.

5.2 | Applications

We used applications from NPB, a well-known suite of HPC benchmarks designed by the NASA Advanced Supercomputing (NAS) division to evaluate and compare the performance of parallel supercomputers.[7] These benchmarks are derived from Computational Fluid Dynamics (CFD)

TABLE 1 Cloud providers and instances' configurations.

					Instance cost* (USD/h)	
Cloud provider	Instance type	vCPUs	Storage type	Network bandwidth	On-demand	Spot
AWS	`t2.micro`	1	EBS	1 Gbps	0.0058	0.0039
	`t3.2xlarge`	8	EBS	Up to 5 Gbps	0.3328	0.1277
	`c5.2xlarge`	8	EBS	Up to 12.5 Gbps	0.3400	0.1147
	`c6i.2xlarge`	8	EBS	Up to 25 Gpbs	0.3400	0.1151
	`i3.2xlarge`	8	1×1900 NVMe SSD	Up to 10 Gpbs	0.6240	0.1913
	`r6idn.16xlarge`	64	2×3800 NVMe SSD	100 Gpbs	6.2525	0.7024
Vultr	`vhp-intel-8c-16gb`	8	1×350 NVMe SSD	—	0.1430	—
	`voc-c-8c-16gb`	8	1×300 NVMe SSD	—	0.2680	—

*Average costs at the time of the experiments.

TABLE 2 Clusters' configurations.

					Cluster cost* (USD/h)	
Cloud provided	Cluster name	Base instance	Nodes	Total vCPUs	On-demand	Spot
AWS	*AWS-t2*	`t2.micro`	8	8	0.0464	0.0312
	AWS-t3	`t3.2xlarge`	8	64	2.6624	1.0216
	AWS-c5	`c5.2xlarge`	8	64	2.7200	0.9176
	AWS-c6i	`c6i.2xlarge`	8	64	2.7200	0.9208
	AWS-i3	`i3.2xlarge`	8	64	4.9920	1.5304
	AWS-r6idn	`r6idn.16xlarge`	1	64	6.2525	0.7024
Vultr	*Vultr-vhp*	`vhp-intel-8c-16gb`	8	64	1.1440	—
	Vultr-voc	`voc-c-8c-16gb`	8	64	2.1440	—

*Average costs at the time of the experiments.

applications, representing real-world aerospace and engineering applications. By measuring various aspects of performance, such as computational speed and efficiency, NPB provides a standardized way for researchers and developers to assess the capabilities of parallel computing systems. In this paper, we used the most recent reference MPI implementation of NPB provided by the NAS division team (version 3.4.2). In this reference implementation, two applications are implemented in C, and all others are implemented in Fortran.

In addition to the applications from NPB, we considered a parallel solver implementation for Laplace's heat diffusion equations based on the iterative Jacobi method implemented in C. For the experiments, we consider a square region with non-periodic borders as the physical domain to be simulated. This domain is then broken into subdivisions and distributed to each MPI rank for iterative computation, resulting in a Cartesian topology of MPI processes. Subdivisions are made following a one-dimensional decomposition strategy, and global boundaries have a constant temperature value. Each subdivision needs information from its neighbors to compute each iteration at the internal borders. To reduce messaging, we employ a ghost-cell pattern updated at the end of each iteration.[28] Thus, given an initial state, the application will iteratively solve the discrete heat diffusion recurrence formula in each mesh point until a convergence criterion or a defined maximum number of iterations is reached.

Table 3 shows the applications and their configurations. The standard NPB input problem sizes consist of eight classes (S, W, A, B, C, D, E and F), where S and W are useful for quick test purposes, A–C are standard test problems and D–F are large test problems. Considering the available public cloud infrastructure and the monetary costs involved in the experimental evaluation, we selected the appropriate *C class* for the performance evaluation of selected NPB applications (*NPB-EP*, *NPB-FT* and *NPB-IS*) and $n = 2048$ for Laplace's heat diffusion equations solver (*HEAT*). For the cost prediction evaluation (Section 6.4), we used the *A class* for NPB applications and $n = 256$ for *HEAT* to build the training dataset. Overall, the chosen applications belong to different domains, allowing a broad range of test scenarios under different conditions and requirements regarding CPU operations and communications. A complete description of NPB applications can be found in Reference 7.

TABLE 3 Applications considered in the study.

					ULFM
App.	**Description**	**Characteristic**	**Language**	**Input size**	**Variant?**
NPB-EP	Generates pairs of Gaussian random deviates	Embarrassingly parallel, almost no communication	Fortran	**Small**: 2^{28} number pairs (Class A) **Big**: 2^{32} number pairs (Class C)	No
NPB-FT	Discrete 3D fast fourier transform (FFT)	Floating-point computations, all-to-all communications	Fortran	**Small**: 256 × 256 × 128 grid, six iterations (Class A) **Big**: 512 × 512 × 512 grid, 20 iterations (Class C)	No
NPB-IS	Integer sort using bucket sort algorithm	Integer computations, all-to-all communications	C	**Small**: 2^{23} keys, key max. value 2^{19} (Class A) **Big**: 2^{27} keys, key max. value 2^{23} (Class C)	Yes
HEAT	Jacobi-based solver for Laplacian heat diffusion	Floating-point computations, neighborhood communications	C	**Small**: $n = 256$ **Big**: $n = 2048$	Yes

As ULFM only works with C programs, we only developed ULFM-based fault-tolerant variants for *HEAT* and *NPB-IS* applications. In these variants, we included an error hander for broken MPI connections using the ULFM API, blocking execution while the failed node is not restored by the *HPC@Cloud* 's *Resource Manager*. When the cluster is restored, we unblock execution, respawning the lost MPI connections and resuming work from the latest checkpoint. For the adaptative fault-tolerant version, execution is not blocked. Instead, we resume work with the remaining nodes allowing the oversubscription of MPI processes from the failed node to the other machines as described in Section 4.4.2.

When tested with Singularity containers, all applications are compiled using MVAPICH2 version 2.3.7 with flags enabling BLCR and TCP/IP-only communications. For fault-tolerant tests, the *HEAT* and *NPB-IS* applications implemented with ULFM support are compiled using OpenMPI version 5.1.0 built from sources with the *mpi-ext* headers. All applications are either compiled with the C compiler version 7.3.1 20180712 (Red Hat 7.3.1-15) or the GNU Fortran (GCC) 7.3.1 20180712 (Red Hat 7.3.1-15) compiler. We used Singularity Community Edition version 3.11.0 for the experiments with containers.

5.3 | Metrics and scenarios

Our evaluation is based on the following key performance indicators:

1. Efficiency, in terms of idle infrastructure time, which is mainly comprised of:
   a. Time spent spawning VM instances;
   b. Time spent installing dependencies on cluster nodes; and
   c. Time spent waiting for checkpoint/restart (whenever it blocks execution).
2. Overhead of running the MPI applications with containers (hybrid model);
3. Accuracy of the experimental cost prediction method.

In the following paragraphs, we give more details about the scenarios considered in this study.

Resource management efficiency. We evaluate the time needed to spawn VM instances by launching virtual clusters with variable numbers of nodes (1, 2, 4, and 8 nodes). We record the time to create and initialize each instance and the time necessary to install all dependencies on cluster nodes in different scenarios. This includes using both prepared snapshots and installing everything from scratch, which helps us comprehend the overhead involved in each approach.

Fault tolerance efficiency evaluation. We analyze the fault tolerance execution overhead by simulating checkpoint/restart events and measuring the time during the process. We examine all fault tolerance strategies and their variants as discussed in Sections 4.3 and 4.4. The infrastructure is based on eight-node clusters comprised of AWS spot instances described in Table 1. We also execute an experiment with *AWS-r6idn* (single node) for evaluating the cost-effectiveness of using spot clusters with fault tolerance versus on-demand high-end instances. To gain insights into their respective efficiencies and flexibility, we measure the idle time directly caused by the fault tolerance mechanisms and the costs incurred by them.

Containerized execution performance evaluation. We analyze the containerized execution overhead when using the hybrid approach for MPI communication with multi-VMs and Singularity. In these experiments, we used the *AWS-c6i* cluster with on-demand instances running one container per MPI rank. We also included results with 16 nodes (128 MPI ranks/containers) to evaluate the applications' scalability in a more communication-demanding scenario. Results are compared against the same workloads executed without using containers.

Costs prediction evaluation. To evaluate the accuracy of the experimental cost prediction method, we compare the model's cost predictions against actual costs observed in real-world scenarios, using all applications running on six 8-node cluster configurations from AWS and two 8-node cluster configurations from Vultr. By altering the input parameters, such as VM types, instance sizes, and storage configurations, we will assess the model's responsiveness and precision in adapting to varying conditions. These tests will provide insights into the performance of *HPC@Cloud* while running MPI applications on public cloud infrastructures and guide further optimization efforts.

6 | EXPERIMENTAL RESULTS

This section presents results from practical experiments with actual public cloud infrastructure from AWS and Vultr. The devised experiments mainly evaluate the cost-effectiveness of using cloud resources and the contributions the *HPC@Cloud* toolkit brings to the users. Experiment types are divided into resource management efficiency (Section 6.1), fault tolerance strategies evaluation (Section 6.2), containerized execution evaluation (Section 6.3), and cost prediction evaluation (Section 6.4).

6.1 | Resource management efficiency

To test the infrastructure management efficiency, we launch variations of 1-node, 2-node, 4-node, and 8-node homogeneous clusters in AWS using the instances described in Table 1 and we measured the average time spent on all infrastructure management tasks (spawning instances, installing dependencies, and setting up the cluster's shared file system). Figure 4 shows the average time needed to set up the virtual

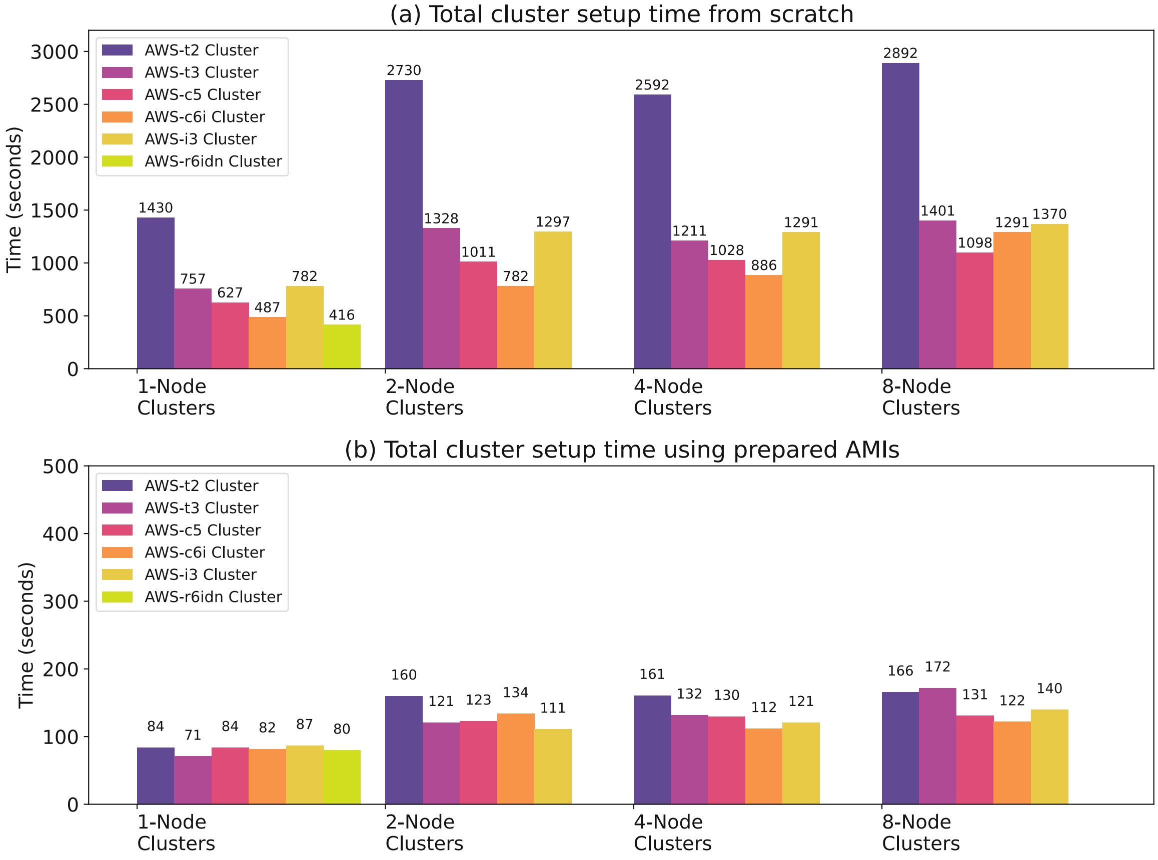

FIGURE 4 Resource management efficiency.

clusters when: (a) all dependencies are build and installed after booting a base Linux image on the VMs; and (b) a pre-built Amazon Machine Image (AMI) with all dependencies installed is used.

Based on the obtained results, we notice that using pre-built machine images is essential to reduce the time needed to launch the cluster and, consequently, reduce costs. Even though larger machines can compile and install dependencies from source quicker, it is still a lot of unproductive time setting up infrastructure. Smaller instances, such as `t2.micro`, can take up to 40 min just to install OpenMPI from sources versus a 2-min setup time when using a prepared AMI. Results show a clear trend that more powerful machines get set up quicker than cheaper and more common instances when installing dependencies on the fly. However, this difference is negligible when using AMIs.

It is evident that the *Resource Manager*'s high scalability is attributed to the parallel creation of worker nodes, with its only limitation being the provider's capacity. Performance tests reveal that cluster setup times remain remarkably consistent, regardless of whether 2 or 8 nodes are created, indicating good scalability. Single-node clusters are created much faster, given that there is no need for setting up a shared NFS.

6.2 | Fault tolerance strategies evaluation

Fault tolerance strategies are evaluated with clusters comprised of AWS instances described in Table 2. Figure 5 presents the execution times of *NPB-IS* and *HEAT* applications when running with on-demand instances (No FT) and with spot instances with different fault tolerance strategies (BLCR and ULFM) and their variations. Since spot instance revocations may not happen during the experiments, we simulate node evictions by artificially destroying nodes during runtime (two evictions per experiment). Cross-hatched sections represent idle time (waiting for instances to be recreated, checkpoint saving, and/or restoration). The values for the analyzed metrics were derived from the average of five executions per experiment.

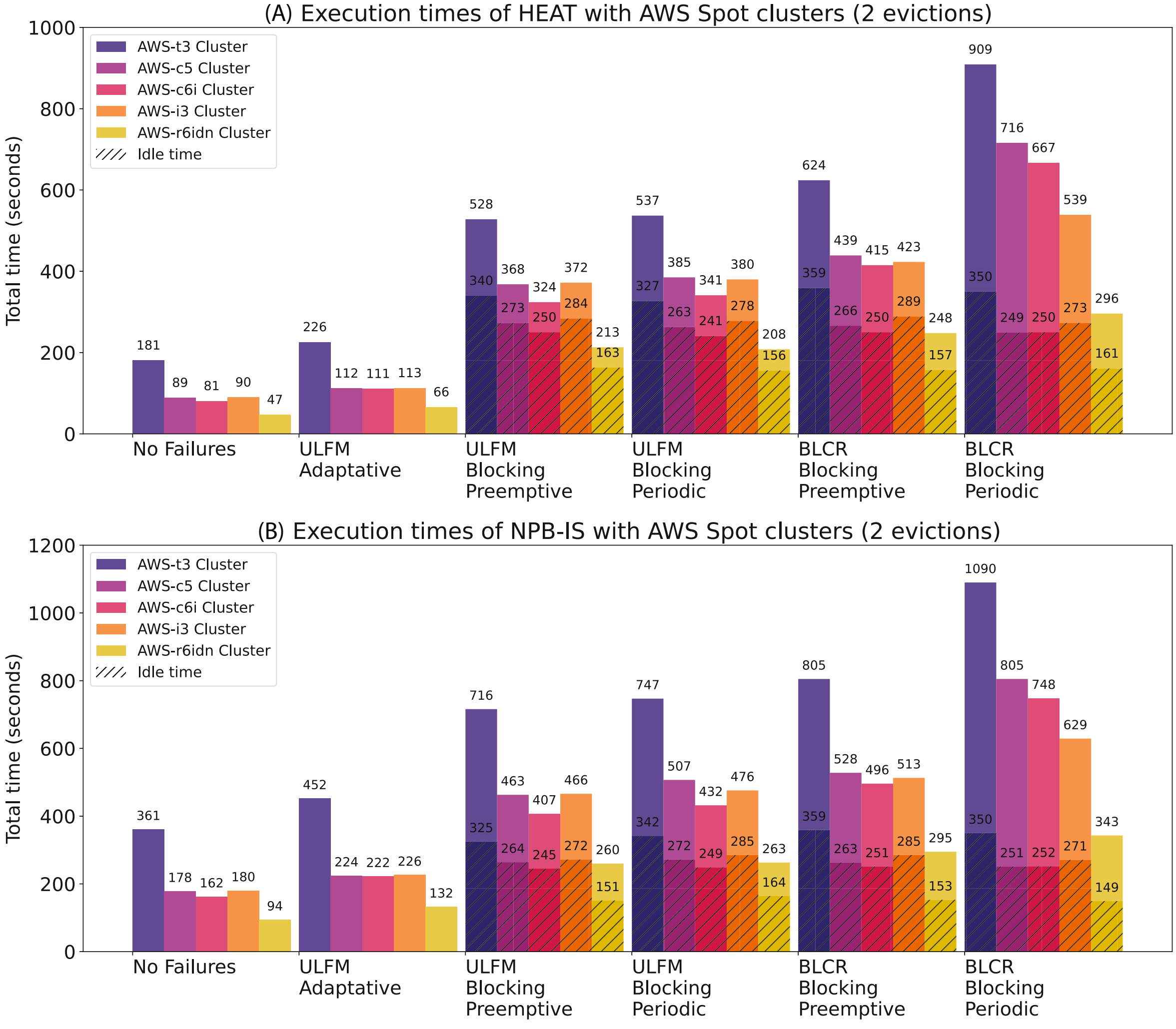

FIGURE 5 Performance evaluation of fault tolerance strategies.

The findings show that the duration spent awaiting infrastructure provisioning becomes considerably more prominent as the workload execution time is smaller. In such instances, spot eviction significantly influences the total execution time compared to on-demand instances. Consequently, spot or unreliable infrastructure is more appropriate for extended running applications, as failures will not drastically increase execution costs by multiple times.

When comparing fault-tolerant approaches, periodic checkpointing has a larger footprint, especially when using system-level methods such as BLCR. Checkpointing time of BLCR is improved when instances with SSDs are used. However, its footprint remains larger than in-memory checkpointing performed using ULFM. When using a preemptive approach, where checkpoints are done only when AWS spot eviction alarms are detected, the performance gap between BLCR and ULFM gets smaller. With a preemptive strategy, only two checkpoints are made, while nine checkpoints are done when adopting a periodic strategy in our experimental workloads. This footprint gap gets even larger as the total workload execution time grows. This suggests that, for long-running workloads, periodic approaches for checkpointing will have significantly degraded performance compared to preemptive approaches. Even though the BLCR strategy has a worse footprint than ULFM, it might be cost-effective when considering the development and migration expenses associated with the ULFM method, especially when adopting a preemptive approach for triggering checkpoints.

Currently, the AWS spot market does not work like a standard auction, where whoever gives the highest bid wins the spot request. The pricing of spot instances remains uniform for all users. The sole variable within our control is the *maximum hourly rate* we are willing to pay for a specific machine type. Setting a higher maximum value decreases the probability of revoking the spot instance. However, this comes at a trade-off, as the cost may increase over time, reaching up to the on-demand price. With a low-latency fault-tolerance method, the user can disregard the bidding strategy and set the maximum hourly rate the same as the on-demand price, as the checkpoint costs overhead will be small, and whatever discount the user can get will be advantageous. However, if the checkpoint method is slow or too expensive, the optimal choice may be executing workloads without fault tolerance on on-demand instances. Low-latency fault tolerance plays a vital role in cost-efficiency when using cloud resources, as the spot discounts can be huge (e.g., the 89.2% discount for spot `r6idn.16xlarge` instances at the time of our experiments, as shown at Table 1).

6.3 | Containerized execution evaluation

To evaluate the containerization support, we executed all applications using Singularity on the *AWS-c6i* cluster with on-demand instances running one container per MPI rank. We also included results with 16 nodes (128 MPI ranks/containers) to evaluate the applications' scalability in a more communication-demanding scenario. Results are compared against the same workloads without using containers. We do not test failure scenarios for containerized executions, as support for it is limited and still under research and development in *HPC@Cloud* .

Figure 6 showcases the results obtained, emphasizing the comparative performance analysis of the applications when run on AWS with and without containers. These experiments aim to determine whether there is a notable overhead cost when using Singularity containers alongside a hybrid MPI approach, as opposed to the standard execution using only VMs. The number of processes employed ranged from 8 to 128, given that each cluster possesses a total of 128 vCPUs (only varying the number of nodes). The values were derived from the average of 5 executions per experiment.

Overall, no significant discrepancies were observed in the execution times when applications were executed with or without containers. Consequently, a preliminary conclusion can be drawn that the overhead cost of utilizing Singularity containers and the hybrid execution approach of MPI is negligible. As anticipated, due to its inherently parallel nature, the *NPB-EP* application displayed favorable scalability. Conversely, the *NPB-IS* and *NPB-FT* applications encountered scalability challenges as the number of nodes increased. We observed that the interconnection network speed was the most significant performance bottleneck for multi-VM MPI.

6.4 | Costs prediction evaluation

Given that *HPC@Cloud* does not possess a predictive mechanism for the timing and frequency of spot evictions, we limit ourselves to estimating a lower-bound for execution costs. This constraint exists as strategies for predicting spot failures, such as those presented by Ghavamipoor et al.,[29] have yet to be integrated into *HPC@Cloud* .

The *HPC@Cloud* costs prediction method is evaluated by running the applications with small input problem sizes (class A for NPB applications and $n = 256$ for *HEAT*) on eight-node clusters of *on-demand* instances to generate datasets for Linear Regression training. After training a model, we scored the predicted total execution time for the larger input problems (Table 3) and computed the total execution costs. Predicted and actual results are presented in Figure 7, with error bars indicating the minimum and maximum costs estimated by our model. The average accuracy across all tested scenarios is 93%.

Cost estimations are computed from each instance's hourly rates multiplied by the predicted execution times. Several factors significantly impact the total execution costs, including the execution time overhead of the checkpointing or fault-tolerant strategy and the availability of spare

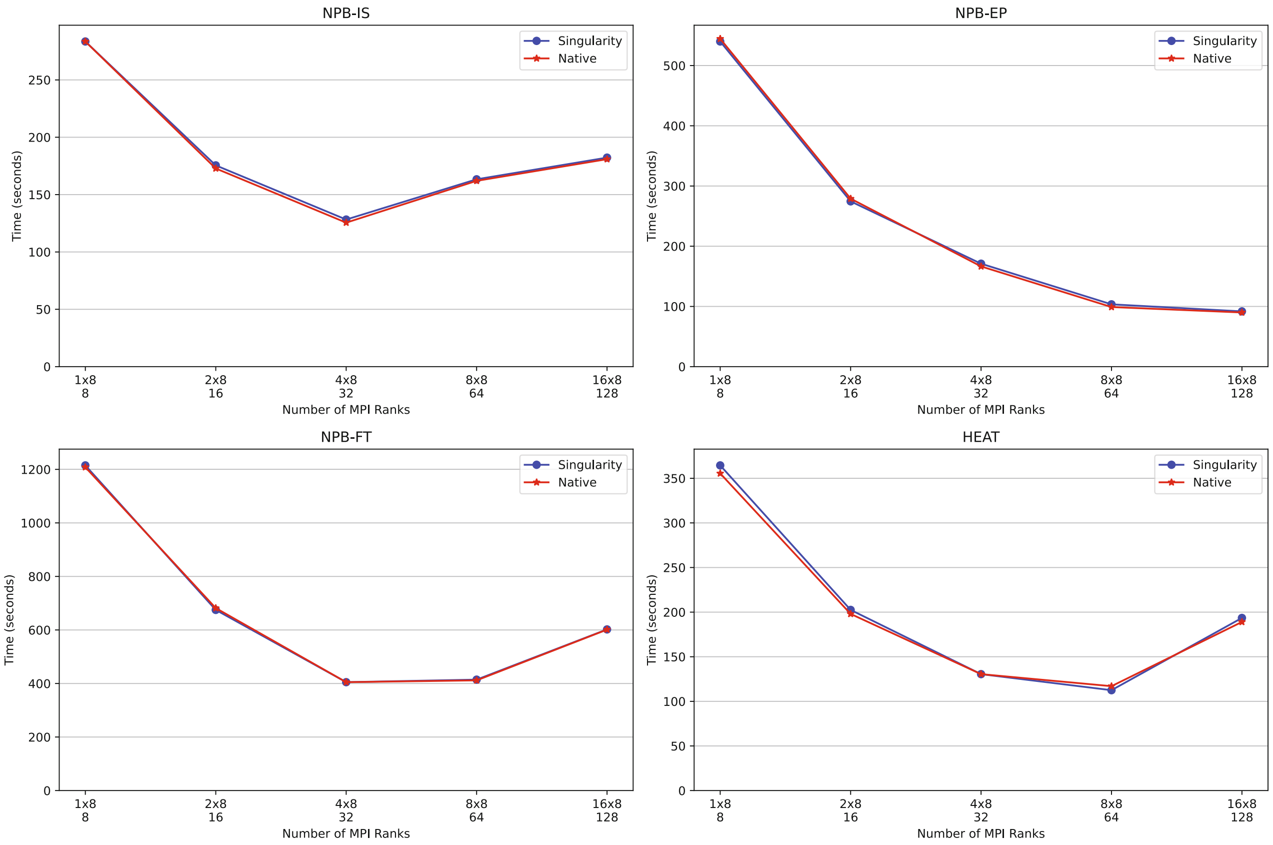

FIGURE 6 Containerized execution performance evaluation.

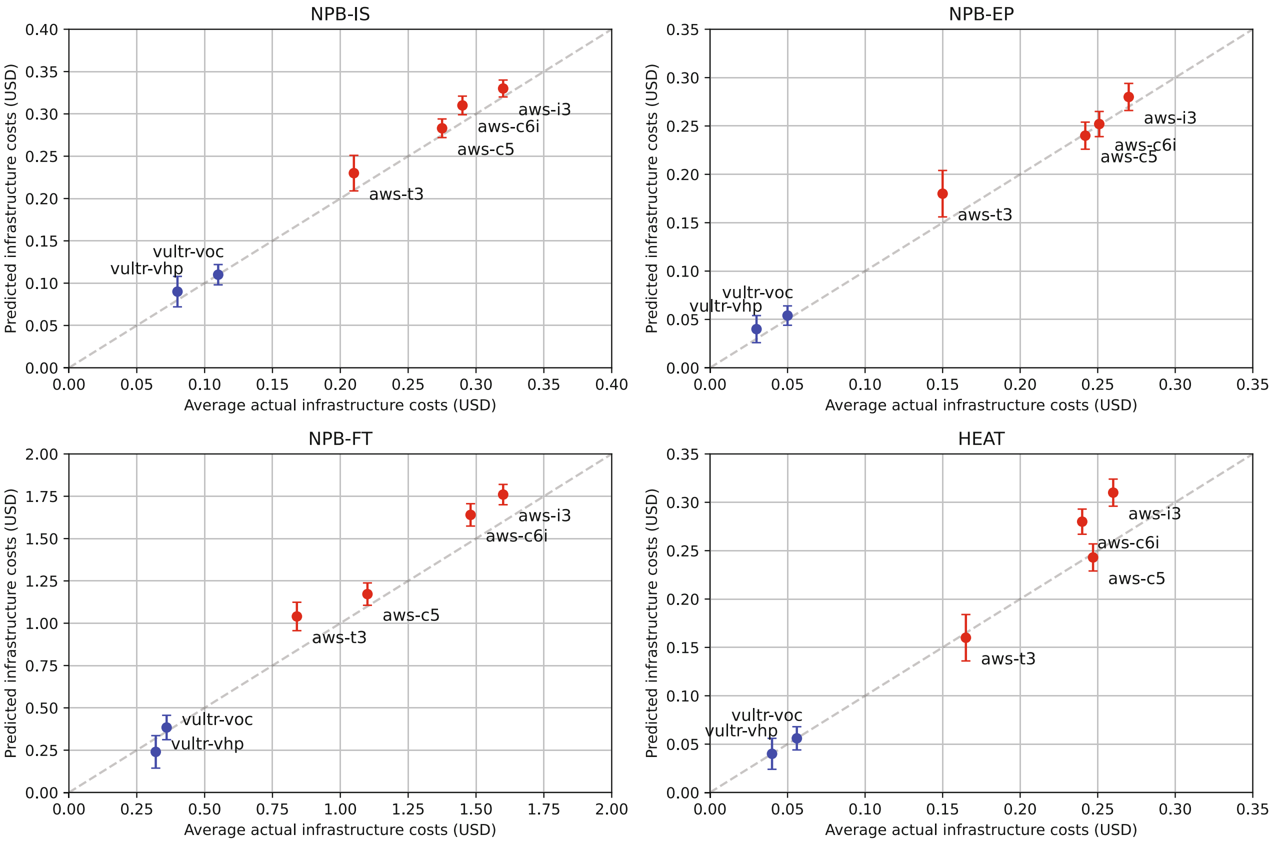

FIGURE 7 Workload costs prediction evaluation (considering only eight-node on-demand clusters).

capacity for spot instances on AWS when they are evicted. Spot hourly rates also play a crucial role, even though they exhibit less variation than in the past, as stated by AWS.[23] To accurately predict costs with spot instances, *HPC@Cloud* currently lacks a component to estimate the occurrence and frequency of AWS spot evictions. Utilizing publicly available data from the provider is not feasible, and generating a dataset for this type of prediction through experiments is not economically viable.

Nevertheless, for fault tolerance techniques with minimal overhead, the total execution time closely resembles that of on-demand instances, rendering our cost estimation relevant in these cases as well. This observation also holds for long-running applications, as the time spent waiting for spot instances and performing fault tolerance tasks is relatively small compared to the total execution time. Figure 7 demonstrates that our regression model exhibits a slight bias towards overestimating costs compared to the actual values.

7 | CONCLUSION

To facilitate the migration of legacy HPC applications to public cloud platforms, we introduced the *HPC@Cloud* toolkit, which consists of software modules for automated virtual environment configuration, experimental application testing, and cost prediction. *HPC@Cloud* features fault tolerance support through BLCR and ULFM for MPI applications, enabling the use of more affordable transient spot resources from AWS. The new ULFM adaptive strategy based on oversubscribing failed MPI ranks to existing nodes avoids blocking the execution of the applications while waiting for spot requests and VMs to be spawned. Our tool also leverages provider-specific notification systems, such as AWS CloudWatch, to preemptively prepare VMs before revocations occur, reducing even further the time needed to restore the execution of the applications. Overall, the ULFM adaptive fault tolerance strategy achieved the best results.

Our experimental data-based costs prediction method demonstrated the ability to estimate the cost of running MPI applications on AWS and Vultr with up to 93% accuracy on average, adding a negligible execution time overhead. Our cost estimations also revealed that utilizing spot resources does not guarantee cost savings when evaluating multiple cloud providers. Persistent Vultr instances exhibited similar performance to some AWS spot instances at a slightly lower cost, however, with the caveat that Vultr is billed by hourly increments, thus being suitable only for long-running applications.

We also concluded that high-end instances could be highly cost-effective if spot options are available. The AWS `r6idn.16xlarge` instance, although more expensive than an 8-node cluster of `c6i.2xlarge` VMs, was available with a high discount. However, using single-node clusters can be fatal for in-memory checkpointing strategies because the fault tolerance strategy must fall back to Elastic Block Storage (EBS) or persistent disk storage in case of instance revocations. We also showed that spot instances could be more expensive than on-demand instances if the fault tolerance strategy is inefficient.

Finally, we added support for the execution of containerized applications through the integration of Singularity in *HPC@Cloud* . Results obtained with the selected applications demonstrated the promising potential of this technology, as the performance of containerized workloads exhibited negligible overhead and was nearly on par with native execution using VMs.

This work can be extended in several directions:

Fault tolerance. We plan to explore other alternatives with support for C++ and Fortran such as Scalable Checkpoint Restart (SCR). We also plan to investigate the feasibility of integrating ULFM with *Charm++*[#] and *StarPU*[‖], two distributed programming models that leverage MPI for inter-process communication that can potentially provide fault tolerance with ULFM in a transparent way. Moreover, we would like to enhance our *Job Manager Module* to accommodate other system-level checkpoint strategies that leverage container migration technologies. Containers can facilitate workload migration, making it more efficient and faster, as container images offer greater portability than VMs.

Support for cloud providers. Thanks to the versatility of Terraform, the *Resource Manager Module* can be easily extended to support new cloud providers. We aim to add support to other public cloud providers in *HPC@Cloud* such as Microsoft Azure, GCP, Alibaba Cloud and IBM Cloud.

Applications. We intend to evaluate the cost-effectiveness of running HPC applications in public clouds using a broader range of benchmarks, such as those from the Standard Performance Evaluation Corporation (SPEC), and real-world applications.

Cost prediction. We intend to integrate existing solutions to estimate when a spot revocation may happen based on historical data into *HPC@Cloud* , such as those proposed by Ghavamipoor et al.[29] Based on these predictions, the *Resource Manager Module* could trigger checkpoints when the probability of a spot instance revocations becomes high, thus being a generic alternative for provider-specific alarm systems. These predictions could also be useful to cleverly choose instance types, or even know beforehand if using spot instances will be cost-effective or not, basing the decision-making on the discount rates, probabilities of eviction and estimated number of failures at runtime.

ACKNOWLEDGMENTS

This work was partially funded by the National Council for Scientific and Technological Development (CNPq) and Amazon Web Services (AWS) through the CNPq/AWS Call Nº 64/2022 – Cloud Credits for Research.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in hpcac-toolkit at https://github.com/lapesd/hpcac-toolkit.

ENDNOTES

[*] https://github.com/aws/aws-parallelcluster.

[†] A preliminary shorter version of this work was published in *Anais do XXIII Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD 2022)*.[5]

[‡] Bare metal instances provide users with direct access to the hardware without any virtualization.

[§] http://terraform.io.

[¶] https://sylabs.io.

[#] https://charmplusplus.org/.

[‖] https://starpu.gitlabpages.inria.fr/.

ORCID

Márcio Castro https://orcid.org/0000-0002-9992-8540

REFERENCES

1. Mell PM, Grance T. *The NIST Definition of Cloud Computing*. Tech. rep. National Institute of Standards & Technology; 2011.
2. Buyya R, Srirama SN, Casale G, et al. A manifesto for future generation cloud computing: research directions for the next decade. *ACM Comput Surv*. 2019;51(5):1-38. doi:10.1145/3241737
3. Gartner. Gartner Says Worldwide IaaS Public Cloud Services Market Grew 41.4% in 2021; 2022. https://www.gartner.com/en/newsroom/press-releases/2022-06-02-gartner-says-worldwide-iaas-public-cloud-services-market-grew-41-percent-in-2021
4. Netto M, Calheiros R, Rodrigues E, Cunha R, Buyya R. HPC cloud for scientific and business applications: taxonomy, vision, and research challenges. *ACM Comput Surv*. 2018;51:1-29. doi:10.1145/3150224
5. Munhoz V, Castro M. HPC@cloud: a provider-agnostic software framework for enabling HPC in public cloud platforms. *Anais Do Simpósio Em Sistemas Computacionais de Alto Desempenho (WSCAD)*. Brazilian Computer Society; 2022:157-168.
6. Munhoz V, Castro M, Mendizabal O. Strategies for fault-tolerant tightly-coupled HPC workloads running on low-budget spot cloud infrastructures. *IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE Computer Society; 2022:263-272.
7. Bailey DH, Barszcz E, Barton JT, et al. The NAS parallel benchmarks. *Int J Supercomput Appl*. 1991;5(3):63-73.
8. Kafle J, Bagale LP, Durga Jang KC. Numerical solution of parabolic partial differential equation by using finite difference method. *J Nepal Phys Soc*. 2020;6(2):57-65. doi:10.3126/jnphyssoc.v6i2.34858
9. Casalicchio E, Iannucci S. The state-of-the-art in container technologies: application, orchestration and security. *Concurr Comput: Pract Exper*. 2020;32(17):e5668. doi:10.1002/cpe.5668
10. Kithulwatta WMCJT, Wickramaarachchi WU, Jayasena KPN, Kumara BTGS, Rathnayaka RMKT. Adoption of Docker containers as an infrastructure for deploying software applications: a review. *Advances on Smart and Soft Computing*. Springer Singapore; 2022:247-259.
11. Boettiger C, Brown J, Seinstra F, et al. Sarus: highly scalable Docker containers for HPC systems. Annual Conference on Extreme Science and Engineering Discovery Environment (XSEDE) ACM. 2015:1-8.
12. Kurtzer GM, Sochat V, Bauer MW. Singularity: scientific Containers for Mobility of compute. *PloS One*. 2017;12(5):e0177459. doi:10.1371/journal.pone.0177459
13. Zhang J, Lu X, Panda DK. Is singularity-based container technology ready for running MPI applications on HPC clouds? International Conference on Utility and Cloud Computing (UCC). ACM, New York, NY, USA. 2017:151-160.
14. Santos MA, Cavalheiro GGH. Cloud infrastructure for HPC investment analysis. *Rev Inf Teór Apl*. 2020;27(4):45-62. doi:10.22456/2175-2745.106794
15. Al-Roomi M, Al-Ebrahim S, Buqrais S, Ahmad I. Cloud computing pricing models: a survey. *Int J Grid Distrib Comput*. 2013;6:93-106. doi:10.14257/ijgdc.2013.6.5.09
16. Guidi G, Ellis M, Buluç A, Yelick K, Culler D. Ten years later: cloud computing is closing the performance gap. Companion of the ACM/SPEC International Conference on Performance Engineering (ICPE). ACM. 2021:41-48.
17. Wong AK, Goscinski AM. A unified framework for the deployment, exposure and access of HPC applications as services in clouds. *Future Gener Comput Syst*. 2013;29(6):1333-1344. doi:10.1016/j.future.2013.01.014
18. Vaillancourt P, Wineholt B, Barker B, et al. Reproducible and portable workflows for scientific computing and HPC in the Cloud. PEARC'20. *Practice and Experience in Advanced Research Computing (PEARC)*. ACM; 2020:311-320.
19. Sindi M, Williams JR. Using container migration for HPC workloads resilience. IEEE High Performance Extreme Computing Conference (HPEC). IEEE. 2019:1-10.
20. Teylo L, Arantes L, Sens P, Drummond LMA. Scheduling bag-of-tasks in clouds using spot and burstable virtual machines. *IEEE Trans Cloud Comput*. 2021;11:984-996. doi:10.1109/TCC.2021.3125426
21. Zhou AC, Lao J, Ke Z, Wang Y, Mao R. FarSpot: optimizing monetary cost for HPC applications in the cloud spot market. *IEEE Trans Parallel Distrib Syst*. 2022;33(11):2955-2967. doi:10.1109/TPDS.2021.3134644
22. Marathe A, Harris R, Lowenthal D, Supinski dBR, Rountree B, Schulz M. Exploiting redundancy for cost-effective, time-constrained execution of HPC applications on amazon EC2. Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing HPDC'14. Association for Computing Machinery, New York, NY, USA. 2014:279-290.
23. Services AW. New Amazon EC2 Spot pricing model: Simplified purchasing without bidding and fewer interruptions; 2023. https://aws.amazon.com/blogs/compute/new-amazon-ec2-spot-pricing/
24. Rahman A, Mahdavi-Hezaveh R, Williams L. A systematic mapping study of infrastructure as code research. *Inf Softw Technol*. 2019;108:65-77. doi:10.1016/j.infsof.2018.12.004
25. Hargrove P, Duell J. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. *J Phys: Conf Ser*. 2006;46:494. doi:10.1088/1742-6596/46/1/067

26. Li Z, Liang H, Wang N, Xue Y, Ge S. Efficiency or innovation: the long-run payoff of cloud computing. *J Global Inf Manag*. 2021;29(6):1-23. doi:10.4018/JGIM.287610
27. Chhetri MB, Lumpe M, Vo QB, Kowalczyk R. On estimating bids for amazon EC2 spot instances using time series forecasting. IEEE International Conference on Services Computing (SCC). IEEE. 2017:44-51.
28. Kjolstad FB, Snir M. Ghost cell pattern. Workshop on Parallel Programming Patterns (ParaPLoP). ACM, New York, NY, USA. 2010.
29. Ghavamipoor H, Mousavi SAK, Faragardi HR, Rasouli N. A reliability aware algorithm for workflow scheduling on cloud spot instances using artificial neural network. Paper presented at: 2020 10th International Symposium on Telecommunications (IST). 2020:67-71.

How to cite this article: Munhoz V, Castro M. Enabling the execution of HPC applications on public clouds with *HPC@Cloud* toolkit. *Concurrency Computat Pract Exper*. 2024;36(8):e7976. doi: 10.1002/cpe.7976