

UNIVERSIDADE FEDERAL FLUMINENSE

Daniel Marcondes Bougleux Sodré

**Being SMART: A multi-objective framework for
executing Bag-of-Tasks workloads in the Cloud**

NITERÓI

2026

Daniel Marcondes Bougleux Sodré

Being SMART: A multi-objective framework for executing Bag-of-Tasks workloads in the Cloud

This Master's Dissertation was presented to the Postgraduate Program in Computing of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Master of Science. Research Area: Computer Science.

Advisor:

Lúcia Maria de Assumpção Drummond

Co-Advisor:

Eugene Francis Vinod Rebello

NITERÓI

2026

Ficha catalográfica automática - SDC/BCPV
Gerada com informações fornecidas pelo autor

S679b Sodr , Daniel Marcondes Bougleux
Being SMART: A multi-objective framework for executing Bag-
of-Tasks workloads in the Cloud / Daniel Marcondes Bougleux
Sodr . - 2026.
75 f.: il.

Orientador: L cia Maria de Assump  o Drummond.
Coorientador: Eugene Francis Vinod Rebello.
Disserta  o (mestrado)-Universidade Federal Fluminense,
Instituto de Computa   , Niter i, 2026.

1. Computa    em nuvem. 2. Computa    paralela e
distribuida. 3. Escalonamento Bi-objetivo. 4. Produ   
intelectual. I. Drummond, L cia Maria de Assump  o,
orientadora. II. Rebello, Eugene Francis Vinod, coorientador.
III. Universidade Federal Fluminense. Instituto de
Computa   . IV. T tulo.

CDD - XXX

Bibliotec rio respons vel: Debora do Nascimento - CRB7/6368

Daniel Marcondes Bougleux Sodré

Being SMART: A multi-objective framework for executing Bag-of-Tasks workloads in
the Cloud

This Master's Dissertation was presented to
the Postgraduate Program in Computing of
the Universidade Federal Fluminense in partial
fulfillment of the requirements for the
degree of Master of Science. Research Area:
Computer Science.

Approved on September 16th, 2026.

EXAMINATION BOARD

Prof. Lúcia Maria de Assumpção Drummond, D.Sc. - Advisor, IC/UFF

Prof. Eugene Francis Vinod Rebello, Ph.D. - Co-Advisor, IC/UFF

Prof. José Viterbo Filho, D.Sc., IC/UFF

Prof. Aletéia Patrícia Favacho de Araújo von Paumgartten, D.Sc.,
DCC/UNB

Niterói
2026

Declaration

In accordance with the guidelines laid out in the CNPq Directive 2.664/2026 of March 6th, 2026, I, the author of this dissertation, declare that during the preparation of this work, I have used the following AI tools: Overleaf’s spellchecker, Microsoft Copilot (<https://copilot.microsoft.com/>), and Language Tool (<https://languagetool.org/>) for the sole purpose of improving the writing and spell-checking specific passages. Following the use of these tools, I fully reviewed, edited, and validated all content, and assume full responsibility for the accuracy and integrity of the information presented in this document.

I further declare that this dissertation was composed by myself and that the work reported herein is my own except where indicated. Some of the material presented in this dissertation has already been published in the following conference papers which are available online from their respective copyright holders:

- A. L. Nunes, **D. M. B. Sodr **, C. Boeres, J. Viterbo, L. M. A. Drummond, V. E. F. Rebello, L. Teylo, F. A. Portella, P. J. B. Estrella and R. Q. Malini, “A Framework for Executing Long Simulation Jobs Cheaply in the Cloud”, 2024 IEEE International Conference on Cloud Engineering (IC2E), Paphos, Cyprus, 2024, pp. 233-244, doi: 10.1109/IC2E61754.2024.00033.
- **D. M. B. Sodr **, L. Serrano, M. de Lima, C. Boeres, L. M. A. Drummond and V. E. F. Rebello, “Spotting the Right Cloud Instances with Multiple AWS EC2 Fleets”, 2025 IEEE/SBC 37th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Bonito, Mato Grosso do Sul, Brazil, 2025, pp. 69-79, doi: 10.1109/SBAC-PAD66369.2025.00017.
- F. Souza, **D. M. B. Sodr **, J. Paulo, R. Macedo, C. Boeres, V. E. F. Rebello, F. A. Portella, P. J. B. Estrella and R. Q. Malini, “Phoning Home: Repatriating output from cloud jobs with a user-level, provider-agnostic data transfer tool”, 2026 IEEE/SBC 38th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Madrid, Spain, 2026, *To appear*.

Daniel Marcondes Bougleux Sodr 

Acknowledgements

To my advisors, Lúcia and Vinod, I am deeply grateful for their support, for the opportunity to carry out this work, and for the knowledge they shared with me throughout its development. I also thank Professor Cristina Boeres for her valuable contributions throughout this work.

To my fiancée, Paula, I am grateful for your love and companionship. None of this work would be possible without your support. Thank you for always believing in me.

To my friends and laboratory colleagues, Miguel, Douglas, Matheus, Lucas, Alan, and Bernardo, thank you for the countless cups of coffee, the feedback exchanges, the advice, and the many friendly discussions. Meeting you all has certainly been the highlight of this journey.

I thank Professor Aletéia Araújo and Professor José Viterbo for accepting the invitation to be on the examination board and for their excellent contributions to this work. I would also like to thank all the other colleagues, lecturers, and mentors who were part of this journey. In particular, I am grateful to Miguel, Lucas, and Fabio for their collaboration on the recently published papers.

I thank Petrobras for funding this work through the projects “4892 – Otimização de Custos de HPC em Nuvem” and “5244 – Otimização de Custos de HPC em Nuvem para Aplicações de Reservatório e Geofísica”, both through research grants and by providing the necessary resources to conduct the experiments carried out on the Amazon Web Services cloud.

Resumo

Recentemente, a computação em nuvem tornou-se a principal plataforma para a execução de aplicações científicas e industriais, por oferecer acesso a uma ampla variedade de recursos computacionais por meio de modelos de precificação sob demanda. Apesar dessas vantagens, a comunidade de Computação de Alta Performance (HPC) permanece relutante em migrar para a nuvem, pois mesmo os contratos de longo prazo (mais baratos) da nuvem são considerados caros em comparação com os custos associados às infraestruturas locais. Uma alternativa potencialmente mais barata é o uso de instâncias *Spot*, por meio das quais provedores de nuvem oferecem capacidade ociosa com descontos significativos (até 90%), em contrapartida, podendo revogá-las sempre que a capacidade for necessária em outro lugar. Ainda assim, a adoção prática da nuvem por usuários de HPC continua limitada por desafios como a heterogeneidade de tipos de instância, disponibilidade variável de capacidade, interrupções imprevisíveis e a sincronização dados entre a nuvem e o armazenamento local. Para lidar com tais desafios, esta dissertação propõe um *framework* de três fases para **S**elecionar e escalonar instâncias, **M**onitorar e migrar aplicações HPC, levando em conta a disponibilidade (**A**vailability) das instâncias, **R**epatriando resultados e buscando atender o Objetivo (**T**arget) de custo-desempenho do usuário (o *framework* SMART). Demonstrado por um estudo de caso com simulações de reservatório, uma fase inicial constrói modelos de desempenho e custo através do perfilamento *offline* eficiente da aplicação, capaz de reduzir os custos de *benchmarking* em 90,8%. A segunda fase utiliza esses modelos para identificar configurações Pareto-ótimas na nuvem, das quais o usuário escolhe a opção que melhor atende aos seus objetivos de tempo e custo. O *framework* então aloca o conjunto ideal de instâncias, de acordo com a disponibilidade. Resultados apontam um desvio inferior a 35% em relação à configuração alvo, mesmo sob restrição de capacidade. Por fim, durante a fase de execução, o SMART adapta o escalonamento inicial, utilizando um preditor de tempo de execução, e um mecanismo de sincronização de dados que diminui o tempo de encerramento da aplicação, para que o estado da execução seja preservado em caso de revogação de instâncias *Spot*.

Palavras-chave: *Framework* SMART, Computação em nuvem, Cargas de trabalho científicas, Escalonamento de *jobs*, Otimização multi-objetivo.

Abstract

Over the past decade, cloud computing has become a dominant platform for running scientific, industrial, and enterprise applications, offering elastic access to a wide range of computational resources and capacities through flexible pay-as-you-go pricing models. Despite these advantages, the High Performance Computing (HPC) community remains reluctant to migrate to the cloud, as HPC applications have historically relied on large, tightly coupled local infrastructures, and even the cloud’s cheaper long-term contracts are often considered expensive when compared with the costs of maintaining local computing facilities. A potentially cheaper alternative is to use Spot instances, where cloud providers offer spare capacity at significantly discounted prices (up to 90%) but can reclaim the capacity whenever it is needed elsewhere. As a result, practical cloud adoption by HPC users remains limited by challenges such as heterogeneous instance types, variable instance availability, unpredictable interruptions, and the overhead of synchronizing data between the cloud and on-premises storage. To address such challenges, this dissertation proposes a three-phase framework to effectively **S**elect and schedule instances, **M**onitor and migrate HPC applications, while accounting for instance **A**vailability, **R**epatriating results, and attempting to meet the user’s **T**arget cost-performance objective (the SMART framework) using both On-Demand and Spot instances. Demonstrated by a case study on reservoir simulations, an initial phase builds performance and cost models for the user’s application through efficient offline profiling, capable of reducing the benchmarking costs by 90.8%. The second phase uses these models to identify Pareto-optimal cloud configurations, from which the user selects an option that best matches their desired execution time and cost. The framework then allocates the ideal set of instances given the availability. Results show a deviation of less than 35% from the target configuration, even when under capacity constraints. Finally, during the execution phase, SMART adapts the initial scheduling, using an execution time predictor and a data synchronisation mechanism that reduces the application’s shutdown time, ensuring that the execution state is preserved in the event of Spot instance revocation.

Keywords: SMART Framework, Cloud Computing, Scientific Workloads, Job Scheduling, Multi-Objective Optimization.

Contents

List of Figures	ix
List of Tables	xi
1 Introduction	12
1.1 Motivation and Objectives	13
1.2 Contributions	16
1.3 Organisation	17
2 Related Work	18
2.1 Cloud Computing	18
2.1.1 AWS EC2	20
2.2 Finding the Ideal Cloud Configuration	22
2.3 Cloud-to-On-Premises Data Synchronization	24
2.4 Summary and Comparative Analysis	25
3 A SMARTer Framework	26
3.1 The Offline Profiling Phase	28
3.2 Multi-Objective Optimization	29
3.3 Static On-line Selection Phase	31
3.4 Dynamic On-line Selection Phase	35
3.5 Execution Management Phase	37
3.6 The Sync Engine	38

3.7	The Sync Engine Design	40
3.7.1	Remote Synchronization with <i>Rsync</i>	41
4	A Reservoir Simulation Case Study	43
4.1	CMG's Reservoir Simulators	43
4.2	History Matching	44
4.3	Predicting the Execution Time of Reservoir Simulations	46
4.3.1	Extrapolating Performance	47
4.4	CMG's Simulations Offline Profiling Phase	50
4.4.1	Finding the Set of Feasible Instances	51
4.4.2	Efficient Profiling of CMG Simulations	53
4.5	Multi-objective Instance Allocation	55
4.5.1	Dynamic Allocation under Availability Constraints	57
4.6	Sync Engine Results	60
4.6.1	Experimental Setup	60
4.6.2	Comparing Persistence Approaches	61
4.6.3	Experimental Methodology and Metrics	61
4.6.4	Results	62
4.6.4.1	On-premises Data Availability Latency	63
4.6.4.2	Close Phase Duration and Spot-eviction Resilience	64
4.6.4.3	Monetary Costs	65
5	Conclusions and Future Work	67
5.1	Future Work	68
5.2	Final Remarks	68
	REFERENCES	69

List of Figures

1	Worldwide cloud market share (as of Q1 2026). Source: statista.com	19
2	Cheapest Spot instance from a set of 35 different 48-CPU instance types in sa-east-1 and us-east-1.	21
3	C5.24xlarge instance Spot price on each us-east-1 availability zones from June to August 2026.	22
4	A typical cloud-bursting workflow.	26
5	Architectural model for SMART cloud-bursting management.	27
6	Number of instances allocated in sa-east-1 on different times of the day. . .	36
7	Sync engine operation on a typical cloud-bursting model.	42
8	A typical history matching workflow.	45
9	Pre Sal 200×200 - TSP of one year - (a) and (b) depict two Time Windows at different stages of the simulation and their respective prediction errors. .	49
10	Changing estimates of the total execution time by the four predictors during the simulation's execution.	50
11	Memory consumption, Execution time and efficiency for the Pre-Salt $200 \times$ 200 simulation model running in a C7.48xlarge instance.	52
12	Execution time vs threads for categories m, r, and c.	55
13	Solution Space and Pareto Optimal set for the execution of 100 Pre-Salt 200 Jobs (Based on prices collected on August 19th, 2026).	56
14	Solution Space and Pareto Optimal set for the main changing points of the solution 2 allocation path.	58
15	Data availability time.	63
16	Close phase duration.	64

17	Average execution cost for each approach, in USD.	65
----	---	----

List of Tables

1	Available instance types for the c5 family.	20
2	Number of CPUs for each possible instance size of the AWS's c, m and r families.	51
3	Instance types and families used on the profiling phase.	53
4	Execution time of the full and short versions of the Pre-Salt 200×200 model. The presented values are an average of three executions.	54
5	Pareto-optimal solutions for 100 Pre-Salt 200 simulation jobs.	57
6	Evolution of solutions 1, 2, 3, and 4 under availability restriction.	57
7	Evolution of the lowest-cost and best-performance solutions under availability restriction.	59
8	Linux Kernel Tunings Parameters.	60
9	Data Availability Times (DATs) (in seconds) for different configurations of the two SYNC ENGINE approaches.	63

1 Introduction

Over the past decade, cloud computing has become the dominant platform for running scientific, industrial, and enterprise applications, offering elastic access to computational resources through flexible pay-as-you-go pricing models. Due to their increasing revenue, major Cloud Providers (CP) such as Amazon Web Services ([Amazon Web Services, 2026a](#)), Microsoft Azure ([Microsoft Azure, 2026](#)), and Google Cloud ([Google Cloud, 2026](#)) are able to offer the most recent technologies at aggressively low prices, enabling organizations to migrate sporadic or light-to-medium workloads to the cloud infrastructure, reducing the need for on-premises hardware and accelerating development cycles.

To reach a broader range of users, cloud providers offer service models with varying levels of abstraction and control over the characteristics of the underlying infrastructure. Recently, the adoption of models in which users do not need to manage the infrastructure directly, such as Software-as-a-Service (SaaS) ([Red Hat, 2022](#)), Platform-as-a-Service (PaaS) ([Red Hat, 2026b](#)) or Function-as-a-Service (FaaS) ([Red Hat, 2020](#)), has grown significantly. Still, one of the oldest and most widely used service models is the low-level *Infrastructure-as-a-Service* (IaaS) ([Red Hat, 2026a](#)). In the IaaS model, users typically allocate units of cloud resources by selecting one or more specific *instance* (or *Virtual Machine* (VM)) configurations; each configuration determines the specific types and amounts of computing resources, such as CPU, memory, storage, and network. Furthermore, a decision must be made about where within the cloud infrastructure the instances should be hosted. Cloud infrastructure is typically distributed across multiple geographically separate regions, each of which normally hosts multiple (at least three) availability zones for redundancy. Each availability zone may offer different types of VM configurations, and the cost of a specific configuration may differ between zones. Regional cloud installations in different countries are subject to local legislation and taxation.

In public clouds, computing costs usually follow a pay-as-you-go model in which each instance type has a predefined hourly price (but billing is based on the minimum billable unit of time, e.g., per hour or per second). Cloud providers expose this price through

different purchasing options, or markets, that determine how resources are billed and managed (Dimitri, 2020). The most common option is the On-Demand market, where, in principle, instances are available on request, and users pay a fixed hourly rate for the duration of use they need. This cost model offers predictable pricing and stable availability. In addition to the On-Demand market, cloud providers also normally offer a Spot market, where unused capacity is offered at significantly reduced rates (for example, up to 90% but updated hourly). While types of instances available in the Spot market (also known as Spot instances or preemptible instances) have the same VM configurations as those offered in the On-Demand market (On-Demand instances), their availability is not guaranteed. Even when such an instance is successfully acquired, its lease to the user may be cancelled at any moment by the provider, typically when capacity is needed elsewhere. This requires the user’s application to be robust to interruptions, and the total cost will only be known when the instance’s use has terminated.

Under the IaaS service model, the user must not only choose the most appropriate VM configuration but also the region (and optionally the availability zone) and pricing model for their workload, and manage the entire lifecycle of the allocated virtual machines, from provisioning to shutdown. Major cloud providers offer hundreds of virtual machine instance types, organised into different families and sizes. Considering the many combinations of CPU, memory, storage, accelerators, and other characteristics, there can be thousands of options to choose from. Furthermore, both Spot prices and instance availability fluctuate heavily over time. As a result, users must constantly navigate a dynamic, highly heterogeneous environment to ensure their workloads run efficiently and reliably.

1.1 Motivation and Objectives

Computational applications have proven crucial to advancing research across various scientific fields (Badia; Ayguade; Labarta, 2017). From decoding genetic sequences to modelling climate change, these applications have facilitated innovation by automating procedures and analysing complex data, thereby improving the efficiency of scientific research.

In this era of exascale computing, the demand for computational resources from such applications has grown considerably as scientists increase the accuracy of their computational models (Ferreira da Silva; Badia, 2023). In this context, the field of High Performance Computing (HPC) has become the foundation driving innovation across various

fields of knowledge, in both academia and industry (Ezell; Atkinson, 2016). This computing paradigm involves the use of supercomputers and parallel processing techniques to solve complex problems through modelling, simulation, and data analysis. HPC applications have historically used large local infrastructures, consisting of *clusters* of interconnected computers organised to operate as a single high-performance system, designed for workloads that require massive parallelism, low-latency, high-bandwidth communication between nodes, and support for accelerators (Ali; Syed, 2013).

Scientific research in the natural sciences generally follows a cyclical process consisting of two phases. In the theoretical phase, data is collected, and mathematical models and analytical solutions are developed to explain physical phenomena. In the experimental phase, experiments are conducted to validate theoretical predictions through computational simulations. As a result, HPC infrastructure use exhibits seasonal patterns, with peaks in demand followed by periods of low activity (Landré; Philippe; Pierson, 2024). Since local clusters are sized to meet peak demand, workload variability in HPC environments has led to low average utilisation of available computational resources.

To increase average utilisation and reduce idle resources, data centres have adopted hybrid strategies that combine on-premises clusters with public cloud resources to handle demand spikes. This strategy reduces hardware acquisition costs, since on-premises clusters can be scaled to meet average demand. When on-premises infrastructure fails to meet demand, the cloud’s elasticity absorbs usage spikes, preventing jobs from accumulating in queues and reducing overall response time (Gupta *et al.*, 2013).

Despite the apparent simplicity, efficiently executing parallel applications in the cloud remains a significant technical challenge, making compute-intensive users, such as the HPC community, reluctant to migrate entirely to the cloud. Many new cloud users tend to adopt traditional *lift-and-shift* strategies, where applications are moved to the cloud without architectural changes. Because many HPC workloads are designed for tightly coupled, low-latency on-premises clusters, these strategies may fail to deliver expected results, resulting in high cloud bills and underwhelming performance.

In contrast, loosely-coupled (or embarrassingly parallel) applications are a class of HPC applications characterised by the minimal effort required to divide their workload into a set of independent tasks. This class of applications is commonly used to solve scientific problems in various fields, such as Bioinformatics, Machine Learning, Chemistry, and Astronomy (Gunarathne *et al.*, 2010) (Hwang *et al.*, 2015). The lack of communication between tasks allows these workloads to be executed across different regions and avail-

ability zones, making them especially well suited to cloud environments, where hundreds or thousands of compute nodes can process tasks concurrently.

A particularly common class of loosely-coupled applications is the Bag-of-Tasks (BoT) model (Borin *et al.*, 2023), in which the workload consists of a collection of independent jobs that share no data dependencies and can be executed in any order. BoT workloads are naturally suited to distributed environments. However, the diversity of available instance types, performance variability, and multiple pricing markets, such as On-Demand and Spot, create a complex decision space. On-Demand instances provide stable and predictable performance but at a higher cost. Spot instances, in contrast, offer substantially lower prices but may be interrupted at any time. For massively parallel workloads that can tolerate interruptions or task reallocation, Spot instances can significantly reduce execution costs. Yet achieving these savings without compromising reliability requires mechanisms that can handle pre-emptions, synchronise data efficiently, and reallocate tasks dynamically.

Furthermore, because of their high resource demands, HPC users must leverage multiple cloud regions simultaneously to use fully utilise the cloud’s potential. Because of data sovereignty requirements, many organisations must ensure sensitive data remains within specific geographic boundaries or storage systems. This introduces a major new obstacle: data movement. HPC applications often run on large datasets that must move between on-premises storage systems and cloud environments. Depending on the application’s data consumption and generation patterns, on-premises-to-cloud latency can become a substantial bottleneck to application performance. With Spot instances, this challenge becomes even more acute, turning data movement into a time-constrained synchronisation problem: if an instance is revoked, the application must save its data within the *eviction window* before termination.

Given this scenario, the efficient execution of HPC applications in the cloud requires a solution capable of addressing the following challenges: Selecting the ideal set of instances according to the user’s cost and performance objectives; exploiting heterogeneous cloud markets while overcoming availability constraints; adapting job-instance allocations by appropriately migrating HPC jobs when necessary; repatriating results and output data efficiently without coupling storage and computing; and achieve all of these goals without requiring any modification to the underlying application.

Motivated by these challenges, this research proposes a framework for executing BoT applications in the cloud, leveraging both Spot and On-Demand instances within a hybrid

cloud-bursting architecture. Unlike traditional approaches that aim to identify a single “best” configuration, the proposed framework adopts a multi-objective optimization strategy to generate a set of Pareto-optimal solutions, enabling users to choose among trade-offs between execution time and financial cost. This approach acknowledges that no single configuration is universally optimal, as small increases in cost may yield substantial reductions in execution time, while modest extensions in runtime may significantly reduce overall expenses. By presenting multiple efficient alternatives, the framework empowers users to make informed decisions tailored to each workload’s specific requirements at a given moment.

1.2 Contributions

The main contribution of this work is the design and preliminary evaluation of a three-phase framework that enables HPC users to explore cloud markets efficiently. The proposed framework supports informed **S**election and scheduling of instances, continuous **M**onitoring and migration of HPC applications, explicit handling of instance **A**vailability, safe **R**epatriation of results, and adherence to the user’s **T**arget cost-performance, forming the SMART framework. The approach is initially tailored for Bag-of-Tasks (BoT-like) workloads, an important class of HPC applications that can benefit substantially from cloud elasticity.

In the first phase, SMART performs one-off offline profiling in which the user’s application runs on a curated set of cloud instances to build performance and cost models. Beyond selecting the appropriate subset of instances to benchmark, this work introduces a technique for generating a curtailed version of the application that reduces profiling costs while preserving representative behaviour. This technique is evaluated using a reservoir simulation application, though further testing could generalise it to other scientific workloads.

In the second phase, the modelling data is used to identify a set of Pareto-optimal cloud configurations. From this frontier, users, whether engineers or system managers, can select the configuration that best matches their current cost-performance requirements. This step is essential because HPC Quality of Service (QoS) needs vary not only across applications but also across stakeholders. The selected QoS target then guides SMART in allocating the ideal set of instances, taking into account real-time cloud availability at the moment of execution.

Finally, in the execution phase, SMART dynamically adapts the schedule to maintain

the user's QoS target in the presence of spot instance revocations or unexpected changes in application behaviour. To achieve this, the framework integrates a runtime estimator capable of predicting the remaining execution time of jobs and a synchronisation mechanism that overlaps computation with data transfer, called Sync Engine. This mechanism ensures that application state is safely preserved, supports data sovereignty requirements, and enables rapid recovery when spot instances are revoked. This dissertation focuses on the first and second phases, while the integration and validation of the execution phase remains as future work.

1.3 Organisation

The remainder of this dissertation is organised into four chapters. Chapter 2 introduces fundamental concepts of cloud computing and reviews prior work in two areas central to this dissertation: cloud-instance selection and cloud-to-on-premises data synchronisation. It summarises key approaches to performance modelling, multi-objective scheduling, and spot-instance reliability, as well as techniques for efficient and resilient data transfer in hybrid environments. By connecting these research threads, the chapter identifies the gap addressed by this dissertation – the need for an integrated solution that combines multi-objective, availability-aware instance selection and reliable synchronisation for Bag-of-Tasks workloads.

Chapter 3 describes the architecture of the proposed framework. It details the design and operation of its main components, the Multi-Objective Scheduler, the Job Manager, the Host Manager, and the Sync Engine, explaining how these modules interact to support efficient cloud execution. This chapter also outlines the mechanisms used for instance selection, execution-time prediction, and data synchronisation.

Chapter 4 presents a reservoir simulation case study used to evaluate the framework. It introduces the industry-standard simulators employed in the experiments, characterises their computational behaviour, and describes the profiling methodology used to model execution time. The chapter then presents the application of the proposed framework's instance selection algorithms, examines the effectiveness of the Execution Time Predictor and evaluates the Sync Engine through a series of experiments involving different persistence strategies, Spot-eviction scenarios, and cost analyses. Chapter 5 concludes the dissertation by summarising the main contributions, discussing the implications of the results, and outlining directions for future research.

2 Related Work

The problem of finding the ideal VM configuration across multiple cloud providers, regions, instance types and markets is one of the most fundamental problems in the cloud resource management area. Efficient cloud deployment therefore requires understanding how an application behaves under different resource profiles and how these choices interact with cost models and instance availability. For many workloads, especially those that must return results to on-premises environments, the efficiency of a deployment also depends on how data is moved between cloud and local infrastructure.

This chapter first presents the challenges of efficiently allocating cloud instances, exemplified by AWS EC2 ([Amazon AWS, 2006](#)). Then, it reviews prior work in two areas central to this problem. Initially, we discuss techniques for predicting application performance and searching large configuration spaces to identify suitable VM types. Then, we examine approaches for reducing the overhead of cloud-to-on-premises data synchronization. Finally, we conclude the chapter with considerations about the mentioned work’s relation with this dissertation.

2.1 Cloud Computing

Cloud computing has evolved from the broader concept of utility computing, in which computational resources are delivered as metered services similar to electricity or water ([Rappa, 2004](#)). Instead of acquiring and maintaining physical hardware, users can access remote compute, storage, and networking resources on demand, paying only for what they consume. The National Institute of Standards and Technology (NIST) ([Mell; Grance, 2011](#)) formalizes this model by defining cloud computing as:

“a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction.”

Together, these properties allow cloud platforms to support a wide range of workloads, from interactive web services to large-scale scientific applications, while abstracting the complexity of the underlying infrastructure.

When deciding how to run their workload in the cloud, the first decision a user must make is which cloud provider to use. Although multiple cloud providers exist today, Statista ([Statista, 2026](#)) reports that the top three companies account for 63% of total market share: Amazon Web Services (28%), Microsoft Azure (21%), and Google Cloud (14%). Although there are many different companies, most of them offer the same set of services. Modern cloud providers tend to organize their offerings into distinct service models, each exposing different degrees of control, flexibility, and operational responsibility.

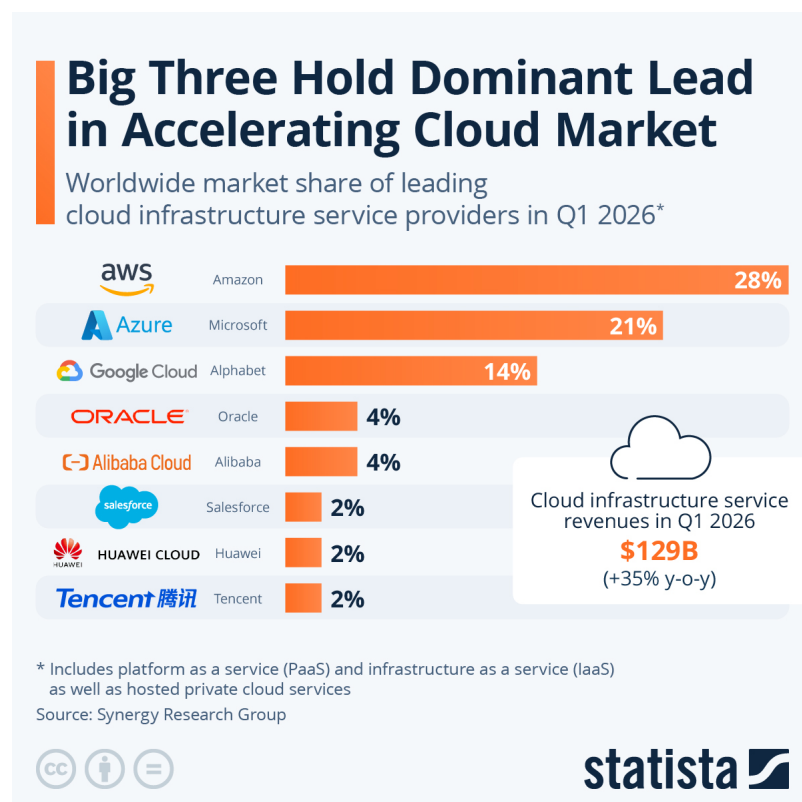


Figure 1: Worldwide cloud market share (as of Q1 2026). Source: statista.com

Amazon Web Services (AWS) holds the post of most widely adopted cloud platform (as seen in Figure 1) and the IaaS model is the most commonly used by HPC users. With this in mind, this research focuses on the IaaS model, specifically on Amazon Elastic Compute Cloud (EC2), but the results presented here are applicable to any cloud provider that offers similar services.

2.1.1 AWS EC2

EC2 exposes a plethora of predefined and fixed instance configurations, each representing a specific combination of CPU architecture, number of CPUs, memory capacity, storage type, and network performance, rather than offering fine-grained control over hardware characteristics. This catalogue allows users to select instance families optimized for general-purpose computing, compute-intensive workloads, memory-bound applications, accelerated computing, or high-throughput storage.

To identify each offered configuration, AWS uses a structured naming convention to encode the characteristics of each instance type. A typical instance name, such as `m5.large` or `c7g.2xlarge`, contains information about the instance family, generation, and size. The family prefix (e.g., `m`, `c`, `r`, `g`) indicates the workload profile the instance is optimized for: general purpose (`m`), compute optimized (`c`), memory optimized (`r`), among many others. The generation number (e.g., `5`, `7g`) reflects the hardware generation or processor architecture, with suffixes such as `g` denoting Graviton-based ARM processors. The size designation (e.g., `large`, `2xlarge`, `12xlarge`) determines the specific resource allocation within the family, typically scaling CPU count, memory capacity, and network bandwidth linearly with the instance size. Table 1 shows the AWS’s `c5` family instance configurations. As of July of 2026, AWS offers more than 1320 instance types divided across 19 families.

Table 1: Available instance types for the `c5` family.

Family/Type	Number of vCPUs	Memory (GiB)
<code>c5.large</code>	2	4
<code>c5.xlarge</code>	4	8
<code>c5.2xlarge</code>	8	16
<code>c5.4xlarge</code>	16	32
<code>c5.9xlarge</code>	36	72
<code>c5.12xlarge</code>	48	96
<code>c5.18xlarge</code>	72	144
<code>c5.24xlarge</code>	96	192

Beyond instance types, EC2 also offers several purchasing models. On-Demand instances provide predictable performance and guaranteed availability at a fixed hourly price. Reserved Instances and Savings Plans offer substantial discounts for users willing to commit to sustained usage over one or three years. Spot instances, in contrast, grant access to unused EC2 capacity at significantly reduced prices but may be interrupted when AWS reclaims resources to serve higher-priority customers. When a revocation is imminent, the provider issues an eviction warning. The interval between this notice and the

actual termination, known as the *eviction window*, varies across clouds: AWS provides a two-minute window, whereas Azure and Google Cloud offer roughly thirty seconds. Each purchasing model introduces different trade-offs between cost, flexibility, and reliability, further expanding the decision space users must navigate.

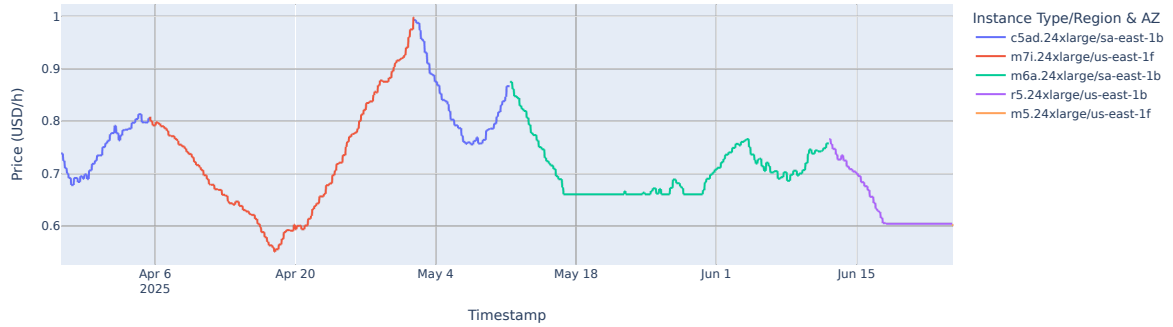


Figure 2: Cheapest Spot instance from a set of 35 different 48-CPU instance types in sa-east-1 and us-east-1.

The complexity of EC2 provisioning is amplified by AWS’s global infrastructure. EC2 resources are distributed across geographic regions, each containing multiple availability zones (AZs). Regions differ in pricing, hardware availability, and network latency. Figure 2 illustrates the price variation of the cheapest Spot instance among 35 different 48-CPU instance types in AWS’s sa-east-1 (São Paulo) and us-east-1 (North Virginia) regions over the period from April to June 2025. Although on-demand prices in sa-east-1 are typically around 50% higher than their counterparts in us-east-1, the cheapest Spot instance is often found in the São Paulo region, which highlights the importance of considering multiple regions when allocating spot instances. Furthermore, availability zones within a region are isolated from one another to provide fault tolerance, but they also differ in price, capacity and Spot interruption behaviour.

Figure 3 presents the Spot price variation for the c5.24xlarge instance across all availability zones in the us-east-1 region from June to August 2026. As shown in the figure, Spot prices of the same instance type can fluctuate significantly between availability zones, often exceeding 50% price difference within the same region. As a result, selecting an instance is not merely a matter of choosing a type and size. Users must also decide in which region and AZ the instance should run.

Taken together, these factors create a remarkably large and intricate configuration space. Users must interpret naming conventions, compare dozens of instance families, evaluate multiple purchasing models, and select among regions and availability zones,

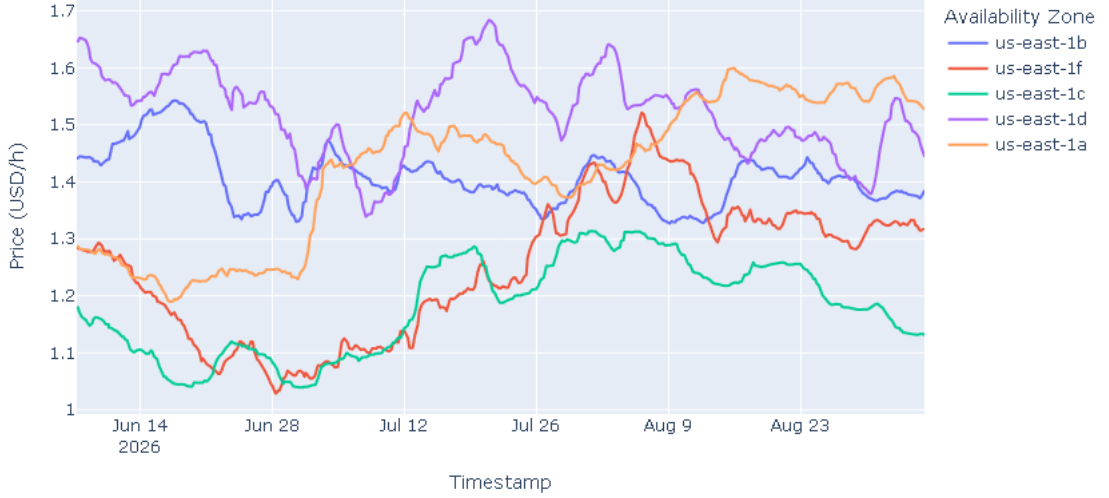


Figure 3: C5.24xlarge instance Spot price on each us-east-1 availability zones from June to August 2026.

each with its own pricing, capacity constraints, and reliability profile. AWS currently offers hundreds of distinct instance types across dozens of regions and multiple purchasing models, resulting in thousands of possible combinations. Instead of simplifying resource allocation, this abundance of choice often overwhelms users, making it difficult to identify configurations that balance cost and performance effectively. This complexity motivates the need for systematic approaches to resource selection, as the one explored in this research.

The research on this topic is typically focused on two aspects: (a) estimating an application’s performance across different cloud configurations, and (b) efficiently searching for the ideal configuration. Characterizing an application’s behaviour in the cloud may involve executing it on all possible cloud configurations, which is often prohibitively expensive or impractical. To reduce costs, research has focused on performance modelling, transfer learning, and sample-efficient search techniques. Section 2.2 explores the literature on both aspects (a) and (b).

2.2 Finding the Ideal Cloud Configuration

A first line of work builds predictive performance models that predicts how an application will behave on unseen VM types using only a small number of measurements. PARIS (Yadwadkar *et al.*, 2017), for example, decouples VM characterization from work-

load characterization by performing a one-time offline benchmarking of many VM types and then fingerprinting the user workload on only two reference VMs. The system uses Random Forest models to predict mean and tail performance across all VM types, achieving accurate runtime and cost estimates while avoiding exhaustive exploration.

Another important research direction uses Bayesian Optimization (BO) ([Mockus, 1989](#)) to guide the search for good configurations. CherryPick ([Alipourfard *et al.*, 2017](#)) is a seminal system in this space. It models execution cost using Gaussian Processes and iteratively selects promising configurations using Expected Improvement. This allows it to find near-optimal configurations with only a handful of samples. BO-based methods treat the application as a black box, requiring no prior knowledge of its internal behaviour, but they can be fragile when the search space is large or when cloud performance variability introduces noise. To address these limitations, recent work augments search-based methods with low-level system metrics. Arrow ([Hsu; Nair; Freeh, *et al.*, 2018](#)) demonstrates that incorporating metrics such as CPU utilization, memory pressure, and I/O wait into the model makes BO more stable and sample-efficient.

A complementary strategy is transfer learning, where models trained on historical workloads help guide the search for new ones. SCOUT ([Hsu; Nair; Menzies, *et al.*, 2018](#)) exemplifies this approach by using pairwise ranking models trained on past optimization runs and enriched with low-level metrics. Instead of predicting absolute performance, SCOUT predicts which configuration is likely to be better than another, enabling faster convergence.

It is important to note that these systems differ not only in how they search the configuration space, but also in what they consider to be the best configurations. Paris, Cherrypick and Arrow for example, optimize a single user-defined objective, such as minimizing runtime or cost. This type of approach may leave out important configurations that compromise in between both objectives.

In contrast, some systems treat configuration selection as a multi-objective optimization problem. MOHEFT ([Durillo; Prodan, 2014](#)), for instance, extends the classical HEFT ([Topcuoglu; Hariri; Wu, 2002](#)) scheduling algorithm to produce Pareto-optimal alternatives rather than a single solution. Instead of collapsing performance and cost into one metric, MOHEFT identifies a set of configurations that represent different trade-offs, allowing users to choose according to their priorities.

The PORSCHE ([Soares, 2023](#)) methodology differentiates itself from other works by proposing multiple Pareto-optimal execution models that define not only which instance

configurations to use, but also how jobs are scheduled inside each allocated VM. These models are supported by multiple profiling experiments performed in an offline profiling stage, where the most efficient scheduling models are identified.

Many of these works implicitly assume that cloud resources are effectively limitless and therefore overlook the availability constraints of the configurations they select. Sim@Cloud (Nunes *et al.*, 2024) and Fleet-MR (Sodré *et al.*, 2025), on the other hand, rely on offline profiling to identify the cost-performance relationship between different vm configurations, producing an ordered preference list that optimizes a single-objective (either execution time or cost) which is then used to allocate instances under variable availability. CMI (Monge *et al.*, 2020) adopts another perspective by integrating multi-objective reasoning into an online instance allocation process that balances execution time, cost and resilience to spot revocations.

2.3 Cloud-to-On-Premises Data Synchronization

Besides selecting the most efficient cloud configuration to run their jobs, users must also consider the costs and challenges of repatriating the application’s output for local analysis or to satisfy data-sovereignty requirements. When applications produce large volumes of data, the cost and latency of transferring results back to on-premises environments become a dominant factor in overall execution time.

Managed data-movement services such as AWS DataSync (Amazon Web Services, 2024) can automate transfers between cloud and on-premises endpoints. However, DataSync operates independently of the application runtime, functioning as a periodic synchronization tool. Because it does not integrate with the application’s I/O path, it cannot mitigate the synchronous blocking penalties that occur when an active application writes large payloads over high-latency connections.

To address these issues, prior work has explored buffering, staging, and tiered placement techniques to reduce data-movement costs across heterogeneous storage layers. The Hermes tool (Kougkas; Devarajan; Sun, 2018) uses a multi-tier distributed buffering system to absorb I/O bursts and move data asynchronously through deep storage hierarchies. In cloud settings, Mutant (Yoon *et al.*, 2018) demonstrates that policy-driven placement across storage tiers can balance cost, latency, and durability as workloads evolve.

Another approach reduces I/O overhead by performing asynchronous persistence and caching data close to the application. HDF5 Cache VOL (Zheng *et al.*, 2022) integrates

caching and staging mechanisms directly into the HDF5 I/O stack, allowing data to be written first to node-local storage and later transferred to a parallel file system. Prototype implementations using HDF5 VOL connectors show that asynchronous I/O with background threads can effectively overlap computation and data movement, reducing the impact of high-latency storage systems.

2.4 Summary and Comparative Analysis

The literature reviewed in this chapter spans two major areas: efficient cloud VM allocation and on-premises-to-cloud data synchronization. In the instance-selection space, most existing approaches optimize a single objective, such as execution time or cost, which limits their ability to explore richer trade-off solutions. Although some works consider multiple objectives, CMI is the only system that explicitly incorporates availability concerns across different cloud markets.

On the data management side, prior research frequently employs I/O tiering to mitigate high-latency connections between local and cloud storage. However, none of these approaches address the challenge of synchronizing data under strict time constraints, which is essential when exploring the Spot market. To the best of our knowledge, our work is the first to combine efficient offline profiling, multi-objective instance allocation, and time-constrained data synchronization into a unified system capable of executing bag-of-tasks-like HPC applications in the cloud.

3 A SMARTer Framework

Cloud bursting is a hybrid computing strategy in which on-premises infrastructure temporarily expands into the public cloud when local resources are insufficient to run applications. Instead of over-provisioning hardware to handle peak demand, the system “bursts” excess workloads to cloud instances, leveraging the elasticity and scale of the cloud providers. This approach allows organizations to avoid queues and maintain availability during spikes while keeping baseline costs low. When implemented well, it extends local infrastructure, enabling applications to scale without permanent investment in additional hardware. This chapter describes the SMART framework architecture and its components. The proposal is evaluated within a cloud-bursting setup, but the architecture is not limited to hybrid environments and can be used by any HPC user running jobs in the cloud.

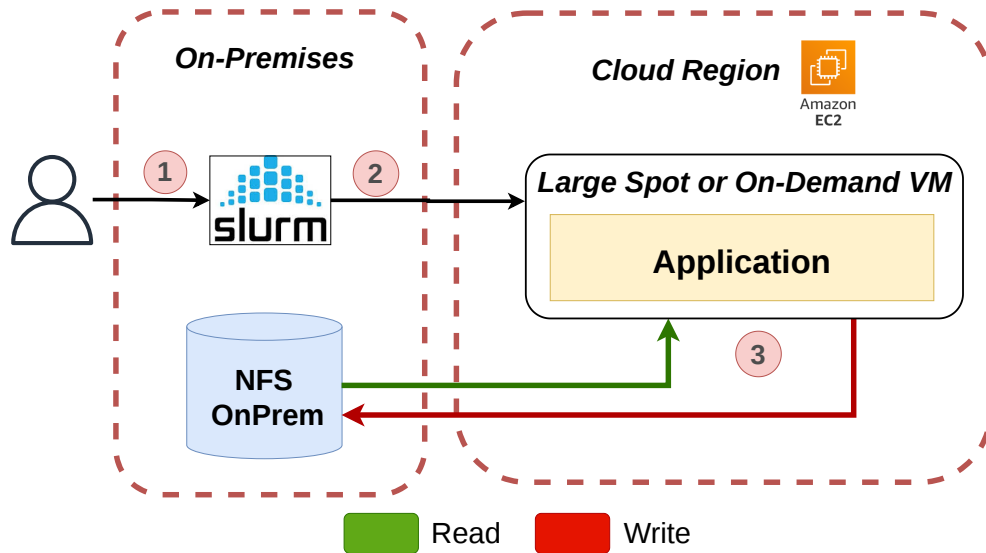


Figure 4: A typical cloud-bursting workflow.

Figure 4 shows a typical cloud-bursting workflow, where jobs originally submitted to an on-premises HPC environment are executed in a public cloud. Step ① shows the user submitting a job to the local scheduler, in this case Slurm (SchedMD LLC., n.d.)(Yoo;

Jette; Grondona, 2003), running on the front-end node of the on-premises HPC cluster. Instead of allocating the job to a cluster node, it can opt to provision a cloud virtual machine instance in a chosen cloud region at step ②. The application then runs on the allocated (Spot or On-Demand) VM, reading input data and writing simulation output files back to the shared on-premises NFS storage file system during step ③. In this work, we propose the SMART framework, which can be easily incorporated into existing cloud-bursting architectures, sits between the local scheduler and the cloud infrastructure, and manages the execution of BoT workloads.

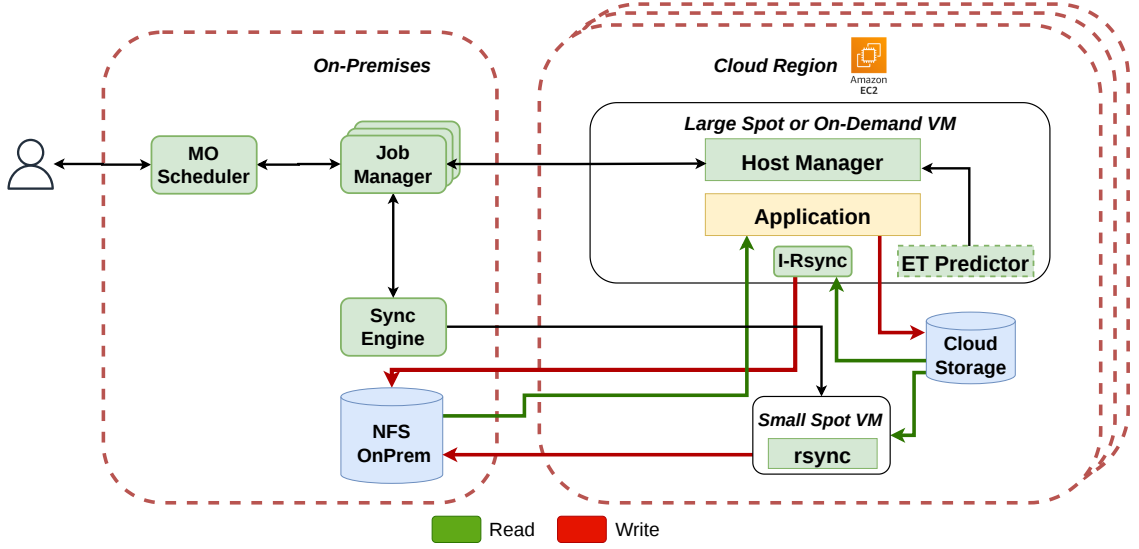


Figure 5: Architectural model for SMART cloud-bursting management.

As shown in Figure 5, the Multi-Objective Scheduler (MOS) selects and allocates cloud instances based on user requirements and optimization criteria. It also continuously receives monitoring information about each running job, allowing it to adjust allocations as needed to meet performance, cost, and reliability goals. For each instance the MOS provisions, it launches a dedicated Job Manager. The Job Manager oversees application execution inside the cloud VM and reports to the MOS when unexpected events occur, such as Spot revocations or deviations from predicted execution time.

Inside the cloud instance, the Host Manager coordinates the runtime environment. It launches the application, monitors resource usage and Spot eviction signals, and initializes the Execution Time Predictor (ETP), when available. The Host Manager also loads the Sync Engine, which transparently redirects the application’s output to a cloud-local storage while asynchronously mirroring data back to the on-premises NFS system.

The SMART framework is organized into three phases: profiling, static and dynamic selection, and execution management. Each phase contributes a distinct capability: the

profiling phase builds the performance and cost models for each desired application; the static selection phase identifies efficient cloud configuration trade-offs and incorporates user preferences; the dynamic selection phase ensures that the chosen configuration is respected as closely as possible under real-world availability constraints; and the execution management phase supervises the running jobs through the Job Manager, Host Manager, and Sync Engine.

3.1 The Offline Profiling Phase

HPC applications often impose strict constraints on the resources they can use. Some exhibit different numerical behaviour when executed with varying numbers of threads, potentially producing incorrect results or even crashing if the thread count is different from what was validated during development. Applications also may fail when the available memory is insufficient for the workload, or when the underlying processor architecture does not match the one for which the application was compiled. To avoid these issues, users frequently restrict the number of threads to a specific set of previously tested values and limit the execution to machines that satisfy the memory requirements and architectural compatibility. In addition to application-level constraints, organizational policies may also impose limits on minimum CPU utilization to avoid wasting resources, with monitoring systems alerting administrators whenever an instance operates below the required utilization level.

In this phase, the user provides a representative benchmark of their application along with any relevant execution constraints such as the permissible number of threads, maximum memory consumption and any architectural requirements. Based on these constraints, the MOS constructs a feasible-instances list, composed of EC2 instance types that fit the desired application.

To ensure that only valid configurations are considered during the scheduling, Algorithm 1 filters the available cloud instances according to the application's constraints. For each candidate instance family, the algorithm first checks whether the instance has enough memory to support the workload's maximum consumption, on lines 4 and 5. If the instance passes the memory check, the algorithm then identifies which of the feasible number of threads will maximize the instance's CPU usage without overcommitting threads to CPUs, on lines 6 to 12. After determining this value, the algorithm evaluates the CPU utilization that such configuration would achieve, allowing it to enforce orga-

Algorithm 1 Generating the feasible instance configurations List

Input: *FeasibleNThreads*, *MaxMemConsume*, *FeasibleInstFamilies*, *MinCpuUtilization*
Output: *FeasibleList* (A list with each possible instance size and the number of threads the application should use on it)

```

1:  $FL \leftarrow []$ 
2:  $FeasibleNThreads \leftarrow FeasibleNThreads$  in ascending order
3: for each  $inst \in FeasibleInstFamilies$  do
4:    $availMem \leftarrow getMem(inst)$ 
5:   if  $availMem > MaxMemConsume$  then
6:      $nopus \leftarrow getCpus(inst)$ 
7:      $bestFitNthr \leftarrow -1$ 
8:     for each  $nthr \in FeasibleNThreads$  do
9:       if  $nthr \leq nopus$  then
10:         $bestFitNthr \leftarrow nthr$ 
11:      end if
12:    end for
13:     $utilization = bestFitNthr / nopus$ 
14:    if  $bestFitNthr > 0$  and  $utilization \geq MinCpuUtilization$  then
15:       $FL \leftarrow FL \cup \langle inst, bestFitNthr \rangle$ 
16:    end if
17:  end if
18: end for
19: return  $FL$ 

```

nizational rules that require a minimum utilization threshold on line 14. Only instances that satisfy all of these conditions are added to the feasible list returned by the algorithm.

Once the feasible list is defined, an offline profiling stage is performed, in which the representative benchmark is executed on each selected instance type using the pre-defined number of threads. This process collects execution times, memory usage, and other relevant performance metrics, producing a detailed characterization of how the application behaves across different resource configurations. The resulting dataset forms the basis for the multi-objective model used in the static and dynamic selection phases, enabling the MOS to reason about trade-offs between cost and performance using accurate, workload-specific measurements rather than assumptions or heuristics.

3.2 Multi-Objective Optimization

When selecting cloud instances, the MOS must simultaneously consider cost, performance, availability, and even interruption risk. These objectives interact in non-trivial ways: reducing cost may require choosing smaller or less reliable instance types, improving performance often demands more expensive configurations or increasing reliability may exclude

Spot instances entirely. As these metrics cannot be collapsed without losing information, the MOS uses multi-objective optimization to identify a set of efficient allocations.

In multi-objective optimization, solutions are evaluated across several criteria at once, and the goal is not to identify a single “best” configuration. Instead, the aim is to determine which configurations represent efficient trade-offs among the objectives. This leads to the concept of Pareto optimality, which characterizes solutions that cannot be improved in one objective without degrading another. According to (Marler; Arora, 2004), Pareto optimality can be defined as follows:

A point $x^* \in X$ is Pareto optimal if there does not exist another solution, $x \in X$, such that $F(x) \leq F(x^*)$ and $F_i(x) < F_i(x^*)$ for at least one function.

That means a configuration is Pareto-optimal if no alternative configuration exists that is strictly better in at least one objective while being at least as good in all others. Pareto-optimal solutions thus represent the most balanced and rational choices available within a given configuration space.

To find the Pareto-optimal configurations, this work uses the notion of dominance, also defined by (Marler; Arora, 2004) as follows:

A vector of objective functions, $F(x^*) \in Z$, is non-dominated if there does not exist another vector, $F(x) \in Z$, such that $F(x) \leq F(x^*)$ with at least one $F_i(x) < F_i(x^*)$. Otherwise, $F(x^*)$ is dominated.

That means, given two configurations, A and B, configuration A is said to dominate configuration B if A is no worse than B in all objectives and strictly better in at least one. Dominated configurations are inherently inefficient as they are inferior to another configuration across the entire objective space. Non-dominated configurations, by on the other hand, form the Pareto frontier, a set of solutions that collectively represent the trade-off surface of the problem. Each point on the frontier corresponds to a different balance between cost, performance, reliability, and other relevant criteria.

The strength of Pareto-based approaches lies in their ability to preserve the structure of trade-offs rather than imposing arbitrary weights or collapsing objectives into a single score. This is particularly important in a scenario, where user preferences vary widely across workloads. Some users may prioritize cost above all else while others require strict performance guarantees. Pareto optimality accommodates these differences without prescribing a single universal solution.

3.3 Static On-line Selection Phase

The static on-line selection phase uses the profiling data to build a decision space and identifies the most efficient resource configurations available to the user. Using the execution times, memory usage, and feasible-instance list produced during profiling, the MOS enumerates all valid configurations that satisfy the application’s architectural and resource constraints. For each configuration, the MOS computes its execution time and monetary cost, resulting in a complete set of candidate solutions that represent the trade-off landscape for the user’s workload.

Given that bag-of-tasks executions typically consist of many similar and independent tasks, we assume that the execution time and cost metrics obtained through the profiling apply uniformly across tasks. Although PORSCHE (Soares, 2023) shows that executing multiple tasks on the same instance can improve efficiency, this strategy also increases the impact of Spot revocations, as a single interruption could terminate multiple tasks simultaneously. This would significantly complicate the task termination and recovery logic, and the exact benefits of such a strategy remain unclear for our workload model. For this reason, we assume that each task runs on its own cloud instance (though not necessarily the same instance type allocated to other tasks) and leave more sophisticated multi-task scheduling strategies for future work. To support the enumeration of candidate solutions, we clarify two concepts that will be used throughout the selection phase: the instance configuration and the solution.

Instance Configuration. An instance configuration represents all the information required to launch a virtual machine on AWS EC2. It is defined as a tuple containing:

- **Instance type** – the AWS EC2 hardware specification (vCPUs, memory, architecture), in the form of an instance family and size, e.g., c5.12xlarge;
- **Region** – the EC2 code that represents the geographic location where the instance is deployed, e.g., us-east-1;
- **Availability zone** – the specific datacenter within the region, e.g. us-east-1a;
- **Market** – the pricing model used (On-Demand or Spot).

Solution. A solution describes a complete resource-allocation plan for executing the workload. It is composed of:

- One or more **instance configurations** (depending on the availability);

- The **number of instances** of each configuration to be allocated;
- The **price** associated with each configuration.

Each solution has an associated cost, the sum of the costs of its instance configurations, and an execution time, the time of the slowest instance configuration. Furthermore, each solution may be heterogeneous, composed of multiple instance configurations with different execution-time and cost characteristics; or homogeneous, consisting of one or more identical configurations. Because all jobs and instance availabilities are initially assumed to be equivalent, no heterogeneous solution can dominate a homogeneous one. The rationale for this property is demonstrated below.

Claim 1. In a scenario where all instance configurations have the same availability, all Pareto-optimal solutions should be homogeneous.

Proof. Consider a homogeneous solution S_1 composed of k instances of type m_1 . Now consider a heterogeneous solution S_2 , composed of $k - n$ instances of type m_1 and n instances of type m_2 , where $1 \leq n \leq k - 1$. Assume that all instance configurations have identical availability. We will show that if S_2 is not dominated by S_1 , then there exists a homogeneous solution S_3 , composed only of instances of type m_2 , that dominates S_2 . This implies that no heterogeneous solution can be Pareto-optimal unless all instance types involved are equivalent.

Let t_{m_1} and t_{m_2} denote the execution times of the application on a single instance of types m_1 and m_2 , respectively, and let c_{m_1} and c_{m_2} denote their unit costs. Since the execution time of a solution is the maximum execution time among the instances it contains, we have

$$T(S_1) = t_{m_1}, \quad T(S_2) = \max(t_{m_1}, t_{m_2}).$$

Therefore, $T(S_2) \geq T(S_1)$ regardless of the value of t_{m_2} . We now analyze the possible relationships between m_1 and m_2 :

1. **If $c_{m_2} > c_{m_1}$, or if $c_{m_2} = c_{m_1}$ and $t_{m_2} > t_{m_1}$:**

In this case, m_2 is strictly worse in cost or time. Hence, S_1 dominates S_2 .

2. **If $c_{m_2} = c_{m_1}$ and $t_{m_2} = t_{m_1}$:**

The two instance types are equivalent. Consequently, S_1 and S_2 are equivalent solutions.

3. **If** $t_{m_2} < t_{m_1}$ **with** $c_{m_2} = c_{m_1}$, **or if** $c_{m_2} < c_{m_1}$:

In this case, S_2 is not dominated by S_1 . However, consider the homogeneous solution

$$S_3 = (k \text{ instances of type } m_2).$$

Since all instances in S_3 are of type m_2 , we have

$$T(S_3) = t_{m_2} \leq T(S_2), \quad C(S_3) = k \cdot c_{m_2} \leq C(S_2).$$

At least one inequality is strict, because m_2 is strictly better in either execution time or cost. Therefore, S_3 dominates S_2 .

Thus, in every case, a heterogeneous solution can only be Pareto-optimal if all instance types involved are equivalent in both execution time and cost. Otherwise, there always exists a homogeneous solution that dominates it.

□

Based on this claim, Algorithm 2 constructs the complete solution space by enumerating all valid homogeneous solutions that the MOS can consider during the static on-line selection phase. The algorithm begins by initializing an empty list of candidate solutions, it then iterates over the full set of feasible instance configurations, identified by Algorithm 1 during the profiling phase. For each instance type, it explores every region, availability zone, and market combination permitted by the user's configuration, which is received as input. For each combination, the algorithm queries the cloud provider for the corresponding price. If the price is available, meaning the instance type can be launched in that region, zone, and market, the algorithm constructs a solution consisting of N_{jobs} instances of that configuration. Each candidate solution is then appended to the solutions list and returned by the algorithm.

With this candidate set defined, the MOS applies the dominance concept to determine the Pareto frontier, retaining only those configurations that are not dominated in any objective. A configuration is considered Pareto-optimal when no alternative offers strictly better performance or lower cost without worsening the other objective. The resulting Pareto frontier captures the spectrum of optimal choices, ranging from low-cost but slower configurations to high-performance but more expensive ones.

Algorithm 3 evaluates the candidate solution set and extracts the Pareto frontier by performing pairwise comparisons between all solutions. For each solution sol_1 , the algo-

Algorithm 2 Generate the solution space set

Input: $nJobs$, $regionSet$, $regionAZ$, $markets$, $feasibleInstList$ **Output:** $candidateSols$ (A list with possible solutions)

```

1:  $candidateSols \leftarrow []$ 
2: for each  $instType \in feasibleInstList$  do
3:   for each  $region \in regionSet$  do
4:     for each  $az \in regionAZ[region]$  do
5:       for each  $mkt \in markets$  do
6:          $price \leftarrow getPrice(instType, region, az, mkt)$ 
7:         if  $price \neq NULL$  then
8:            $instConfig \leftarrow \langle instType, region, az, mkt, price \rangle$ 
9:            $candidateSols \leftarrow candidateSols \cup \langle nJobs, instConfig \rangle$ 
10:        end if
11:      end for
12:    end for
13:  end for
14: end for
15: return  $candidateSols$ 

```

rithm checks whether any other solution sol_2 offers meaningfully better cost or execution time. This comparison relies on two tolerance-aware formulas that model the natural performance variability of multi-tenant cloud instances. The *isGreater* function determines whether one value is significantly larger than another by requiring

$$v_1 > v_2 \cdot (1 + \text{tolerance}),$$

which ensures that only differences exceeding the expected variance margin are treated as meaningful improvements. Conversely, the *isEqual* function treats two values as effectively identical when v_1 lies within a symmetric tolerance around v_2 , defined by

$$v_1 \geq (1 - \text{tolerance}) \cdot v_2 \quad \mathbf{and} \quad v_1 \leq (1 + \text{tolerance}) \cdot v_2.$$

These formulas ensure that dominance is established only when differences exceed typical cloud noise. If any solution sol_2 dominates sol_1 under these rules, sol_1 is discarded; otherwise, it is retained. By exhaustively applying this criterion across all pairs, the algorithm identifies the complete set of non dominated solutions.

The Pareto frontier is then presented to the user through the MOS interface. The user decision concludes the static selection phase and establishes the target configuration that the MOS will attempt to realize during dynamic selection. By separating the identification of optimal configurations from the user's final choice, the static selection phase ensures

Algorithm 3 Generate the Pareto set**Input:** *candidateSols*, *tolerance***Output:** *paretoFront* (A list of non-dominated solutions)

```

1: paretoFront  $\leftarrow \emptyset$ 
2: outerLoop:
3: for each  $sol_1 \in candidateSols$  do
4:   for each  $sol_2 \in candidateSols$  do
5:      $c_1, c_2 \leftarrow getCost(sol_1, sol_2)$ 
6:      $s_1, s_2 \leftarrow getET(sol_1, sol_2)$ 
7:     if  $isGreater(c_1, c_2, tolerance)$  and  $isGreater(s_1, s_2, tolerance)$  then
8:       continue outerLoop
9:     else if  $isEqual(c_1, c_2, tolerance)$  and  $isGreater(s_1, s_2, tolerance)$  then
10:      continue outerLoop
11:     else if  $isEqual(s_1, s_2, tolerance)$  and  $isGreater(c_1, c_2, tolerance)$  then
12:       continue outerLoop
13:     end if
14:   end for
15:    $paretoFront \leftarrow paretoFront \cup sol_1$ 
16: end for
17: return paretoFront

```

transparency, flexibility, and alignment with workload-specific preferences.

3.4 Dynamic On-line Selection Phase

As the spot market relies on the cloud provider's spare capacity, the availability of instances can fluctuate significantly depending on the instance type, region, and availability zone. This variability directly affects the feasibility of large-scale allocations of a single instance type. Figure 6 illustrates this behaviour through an experiment in which 200 instances were requested for a set of 24 different 48-core instance types at three different times of day (09:00, 13:00, and 18:00), in the sa-east-1 region. The results reveal substantial oscillations in the number of instances that could actually be obtained, with availability ranging from fewer than 20 instances to more than 120 depending on the time of day. These fluctuations highlight the inherent unpredictability of the spot market and show that, for large-scale allocations, relying on a single region is often insufficient to meet the desired capacity. Because of this instability, the MOS must dynamically adapt the solution selected by the user to the changing availability conditions, progressively selecting alternative configurations when the preferred ones cannot satisfy the required number of jobs.

Algorithm 4 implements the dynamic allocation phase of the MOS. Once the user

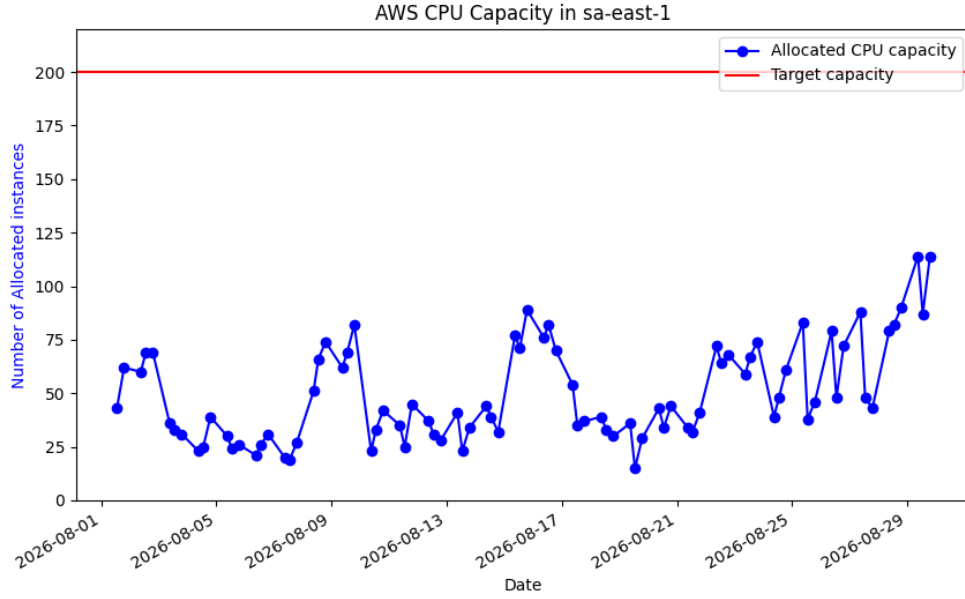


Figure 6: Number of instances allocated in sa-east-1 on different times of the day.

chooses a Pareto-optimal configuration, the MOS attempts to allocate the corresponding instance type for all pending jobs (lines 1 and 2). However, the desired amount of instances of that instance type may not be available. When this occurs, the MOS recomputes the solution space and the Pareto frontier on-the-fly, incorporating the set of instances that have already been successfully allocated (lines 3 to 8). This updated frontier reflects the current state of the cloud market while preserving the multi-objective structure of the decision space.

To avoid requiring continuous input from the user, the MOS automatically selects, within the update Pareto frontier, the configuration that is the closest to the user’s original choice. Similarity is quantified using Euclidean distance in the objective space, with the smallest distance indicating the most faithful alternative. To ensure that cost and execution time contribute equally to this comparison, both metrics are first normalized with respect to the cheapest configuration’s cost and the fastest configuration’s execution time, respectively. This ensures that the newly selected configuration remains as faithful as possible to the user’s preferences regarding cost and performance.

The MOS then attempts to allocate the remaining amount of instances of this new configuration. To prevent repeated attempts to allocate the same instance types, the algorithm keeps track of all previously tried configurations and excludes them when re-building the solution space. This process repeats until the desired number of instances

Algorithm 4 Dynamic Instance Allocation**Input:** $Njobs$, $initialSol$ **Output:** $allocated$ (List of allocated instances)

```

1:  $instConfig \leftarrow getInstanceConfigurations(initialSol)$ 
2:  $allocated \leftarrow AllocInstances(instConfig, Njobs)$ 
3:  $triedConfigs \leftarrow [instConfig]$ 
4:  $baseSol \leftarrow createSolution(allocated)$ 
5:  $solutionSpace \leftarrow generateSolutionSpace(Njobs, triedConfigs, baseSol)$ 
6: while  $|allocated| < Njobs$  and  $|solutionSpace| > 0$  do
7:   // Find the new Pareto front
8:    $paretoFront \leftarrow generateParetoSet(solutionSpace, tolerance)$ 
9:   // Automatically selects the solution closest to initialSol
10:   $selectedSol \leftarrow autoSelectSol(initialSol, paretoFront)$ 
11:  // Makes sure that only new configurations are allocated
12:   $instConfig \leftarrow getConfig(selectedSol) \setminus triedConfigs$ 
13:   $remaining \leftarrow Njobs - |allocated|$ 
14:   $obtainedInst \leftarrow AllocInst(instConfig, remaining)$ 
15:   $allocated \leftarrow allocated \cup obtainedInst$ 
16:   $triedConfigs \leftarrow triedConfigs \cup instConfig$ 
17:   $baseSol \leftarrow createSolution(allocated)$ 
18:   $solutionSpace \leftarrow generateSolutionSpace(Njobs, triedConfigs, baseSol)$ 
19: end while
20: return  $allocated$ 

```

has been allocated or no new instance configurations remain to be tried. Through this adaptive mechanism, the dynamic on-line selection phase maintains alignment with user intent while navigating the unpredictable nature of cloud resource availability.

3.5 Execution Management Phase

The execution management phase oversees the full lifecycle of the application once all jobs have been successfully deployed in the cloud. At this stage, the MOS hands control to the runtime components responsible for monitoring, coordinating, and safeguarding the execution. Inside each VM, the Host Manager supervises the local execution environment, handling Spot-interruption signals, collecting resource-usage metrics, and activating the Execution Time Predictor (ETP), when available. When AWS issues a Spot-interruption notice, the Host Manager initiates the creation of an application checkpoint, ensuring that progress is preserved before the instance is terminated. This mechanism allows the MOS to recover gracefully from revocations and maintain forward progress even under volatile market conditions.

The ETP is a plugin-based module designed to dynamically monitor the application's

progress and estimate its remaining execution time. Because different applications exhibit distinct runtime behaviours, the ETP can be implemented specifically for each workload. When an application-specific ETP plugin is present, it provides real-time predictions that help anticipate whether the job can complete within the expected window. If no plugin is available, the MOS falls back to the execution-time measurements obtained during the profiling phase, ensuring that execution management remains functional even without dynamic prediction support.

For each job, the MOS spawns a Job Manager process, which continuously communicates with the Host Manager to track the progress of the application. The Job Manager reports anomalies such as Spot revocations, unexpected slowdowns, or deviations from the predicted execution time, enabling the MOS to react promptly to runtime events. To ensure data persistence and compliance with on-premises workflows, the Sync Engine is also active during this phase. In this context, the Sync Engine relies exclusively on I-Rsync, which periodically synchronizes output files from cloud-local storage back to the on-premises NFS system. This approach allows the application to write at cloud-local speeds while maintaining consistent and timely propagation of results to the on-premises environment. Although both the Sync Engine and the Execution Time Predictor are implemented in this work, a full evaluation of the execution-management phase is left for future work.

3.6 The Sync Engine

As shown previously by Figure 4, cloud-bursting workflows typically rely on on-premises NFS storage file systems to store the application’s input and output files when running in the cloud. Operational constraints drive this design choice, since simulation artifacts must be persistently stored on-premises to meet data sovereignty requirements and to preserve continuity with existing reservoir engineering workflows, which are centered on the on-premises storage namespace.

This coupling of the execution with the on-premises storage can become a dominant performance bottleneck, especially for regions with high RTT (Round Trip Time) latencies relative to the on-premises location. While US regions are often attractive because of the earlier availability of newer instance types, lower pricing, and higher available resource capacities, these benefits can be negated by slow data transfers that increase the effective runtime of jobs, in some cases making public cloud use impractical due to higher-than-

expected monetary costs.

As stated before, one way to keep cloud costs down is to leverage the heavily discounted Spot VM instances, though they carry the risk of being interrupted. While cloud providers (CPs) will give prior warning of an impending VM eviction, gracefully terminating the application and returning all output files to on-premises storage within the notice period can be challenging, especially if last-minute checkpoints are used to save the application's current state and allow it to continue execution on another VM in the future without having to recompute previous states. Given the uncertainty surrounding the frequency of such evictions, checkpoints may need to be created frequently, increasing the application's runtime and storage requirements. Furthermore, HPC applications usually maintain a large in-memory amount of data, resulting in checkpoints in the gigabyte range that may be difficult to save within the CP's eviction notification period.

Cloud providers (AWS, Google Cloud, and soon Azure) offer a Spot eviction notification period of up to 2 minutes. To maximize the utilization of the Spot VM, it would therefore be advantageous to use this interval to interrupt and checkpoint the application only at this moment, rather than incur repeated delays to write checkpoint files at regular intervals during the application's execution. While such an approach can lead to shorter execution times and lower monetary costs, it is not trivial and requires solving three challenges:

- **Compute–Storage coupling.** Persisting outputs to a remote on-premises file system couples cloud execution to network and storage latency, turning write-intensive phases into additional wall-clock overheads;
- **Guarantee successful checkpoint completion within the eviction notification period.** Depending on a single last-minute checkpoint is severely risky, given that the time to persist it to on-premises storage may exceed the notification period, wasting computation and forcing the application to be restarted from an earlier checkpoint, or worse, the beginning;
- **Transparent implementation.** Solving the previous challenges without any changes to the application.

3.7 The Sync Engine Design

One of the main modules of the proposed SMART framework is the Sync Engine, which is designed to support environments that require outputs to be stored on-premises. Sync Engine is designed to transparently overlap computation with high-latency remote writes, without any application modifications, regardless of the storage vendor, and to be resilient to Spot instance evictions, thereby facilitating data synchronization between the cloud and on-premises environments.

With the Sync Engine, the user's job continues to be submitted normally but write operations, are redirected to a cloud-local storage to reduce the overhead of compute-storage coupling. In parallel, Sync Engine propagates selected output files back to the on-premises environment in the background, through data synchronization which leverages rsync ([The Samba Team, 2026](#))([Johnson, 2025](#)), a well-established UNIX file transfer utility. To characterize the proposed approach, we partition the write activity of a job's execution into three phases:

- **Progress Phase:** The application is actively executing and periodically updating its output files on local storage, while Sync Engine asynchronously replicates these files to on-premises storage in the background;
- **Close Phase:** This phase refers to the period from the end of normal execution (or a graceful shutdown) until all pending writes are flushed and open output files are closed. On Spot VMs, this may include persisting a checkpoint file. When an eviction notice is received, this phase is time-constrained: failure to persist the necessary data on-premises before the instance is reclaimed may prevent the job from resuming on a replacement instance. The required write throughput during this phase can be substantially higher than during the Progress Phase;
- **Post-Execution Phase:** A final verification pass ensures that all required output files have been correctly and completely mirrored on-premises. This step is necessary because expensive HPC or transient Spot cloud instances may be terminated before all write operations to remote storage have been acknowledged, leaving the on-premises file system in a potentially inconsistent state.

3.7.1 Remote Synchronization with *Rsync*

System administrators often need to synchronize or back up files and directories between local and remote locations without transferring entire files each time. *rsync* is a highly effective tool for this purpose since it uses a delta synchronization scheme to transfer only the modified portions of a file. To achieve this, *rsync* divides the two files being synchronized into corresponding pairs of chunks, computes and compares their respective checksums, and transfers only the chunks whose checksums differ. To further optimize this process, *rsync* pipelines the operations of reading files, calculating checksums, and transmitting distinct chunks to minimize the time resources, such as CPU, disk bandwidth, and network capacity, are idle. This approach has recently received renewed attention for improvements (Zhang *et al.*, 2026), particularly given its adoption for large-scale synchronization across multiple regions, similar to file transfers between cloud and on-premises infrastructures.

The *rsync* tool supports two mechanisms for establishing a connection to a remote host: through a remote-shell (*e.g.*, SSH, RSH), or via a direct TCP connection to an *rsync daemon*. While SSH provides a secure, encrypted channel, it also introduces computational overhead from cryptographic operations. On the other hand, using the *rsync daemon* mode requires launching a daemon process on the cloud instance that listens for incoming *rsync* requests on a designated port, which can expose sensitive data if not properly secured. Thus, when using daemon mode, it may be necessary to protect the connection through a VPN. Both *rsync* mechanisms were evaluated in this research. Furthermore, the following *rsync* options are enabled: archive mode, which preserves permissions, timestamps, symbolic links, and device files; and compression during data transfer to save bandwidth, when necessary.

The Sync Engine employs two *rsync* instances at distinct points in the job lifecycle. Figure 7 represents a simplified version of the SMART architecture, containing only the elements related to the sync engine, for the sake of simplicity. As it is shown in Figure 7, when the job scheduler opts to create a Cloud VM at Step 2, Sync Engine configures the *rsync* to be initialized with the application on the VM. When the application begins, it enters the Progress Phase, Step 3, where it is configured to write or update its files directly to a cloud storage that is accessible after the VM is shut down (*e.g.* An EFS). Concurrently with the application, Sync Engine periodically runs *rsync* on these output files, synchronizing their current states with the on-premises storage in the directories specified in the original job submission. In this approach, named I-Rsync,

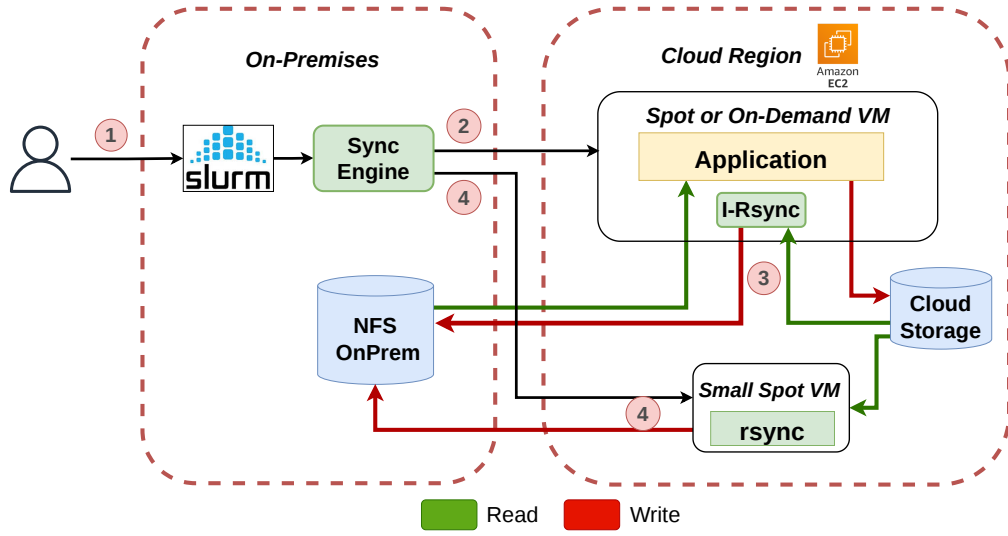


Figure 7: Sync engine operation on a typical cloud-bursting model.

rsync is executed over a TCP or SSH connection at runtime-configurable time intervals of I_{rsync} , which are long enough to ensure that each invocation completes before the next is triggered.

When the application completes, i.e., reaches the Close Phase, and all output files have been closed on the local cloud storage, the VM will be shut down. If the application has not been completed and an eviction notice is received, a checkpoint is recorded during the Close Phase. Once the application and the VM have been terminated, in the Post-execution Phase, Step 4, a final *rsync* synchronization is performed on a new, but smaller VM, to ensure file persistence between the on-premises and local cloud storage.

4 A Reservoir Simulation Case Study

One of the biggest users of computational simulations is the area of reservoir engineering ([Meng *et al.*, 2025](#)). Reservoir simulation is a central tool in petroleum engineering, used to analyse and predict the behaviour of subsurface oil and gas reservoirs. Because reservoirs cannot be directly observed or experimented on at scale, engineers rely on computational models to represent their physical characteristics and simulate fluid flow over time. These models support decisions throughout the development and production phases of an oil field, including well placement, injection strategies, production forecasting, and uncertainty assessment.

A reservoir simulator solves a set of physical equations that model mass conservation, thermodynamic relationships, multiphase flow, and heat transfer over a three-dimensional grid that represents the geological structure of the reservoir. Each grid cell contains rock and fluid properties such as porosity, permeability, pressure, and saturation. Realistic models often contain millions of cells, and their numerical solution requires substantial computational resources. Many engineering workflows also involve running large ensembles of simulations, either to calibrate the model against historical production data or to explore alternative operational strategies.

This chapter presents a case study using CMG reservoir simulations to evaluate the effectiveness of the framework introduced earlier. The goal is to demonstrate how the components of the framework handle realistic workloads. The case study provides a practical view of how long, computationally intensive simulations behave in cloud environments and how the framework's modules may interact to support their execution.

4.1 CMG's Reservoir Simulators

The Computer Modelling Group (CMG) ([Computer Modelling Group Ltd., n.d.](#)) provides one of the most established suites of reservoir simulation tools in the oil and gas industry. Its simulators are designed to model different classes of reservoirs and production mech-

anisms, allowing engineers to select the tool that best matches the physical behaviour of the fluids and the objectives of the study. In this case study, the focus is on two simulators that represent distinct modelling approaches: IMEX, which implements the black-oil formulation; and GEM, which implements a compositional formulation.

IMEX is CMG's simulator for black-oil models, a formulation widely used in conventional reservoirs where fluid properties vary smoothly with pressure and temperature. In this approach, the reservoir fluids are represented by three phases: oil, gas, and water. Their thermodynamic behaviour is described using simplified pressure-volume-temperature relationships. Because the composition of each phase is assumed to remain relatively stable, the governing equations are less complex than those used in compositional models.

On the other hand, GEM is CMG's compositional simulator, designed for reservoirs where fluid composition changes significantly over time or where phase behaviour is complex. This includes volatile oil reservoirs, gas condensate systems, and reservoirs operating near the critical point. Compositional models require solving larger and more complex systems of equations than black-oil models. As a result, GEM simulations typically require more memory and longer execution times than IMEX simulations. They are also more sensitive to numerical controls, solver tolerances, and the quality of the underlying geological model.

From a computational perspective, both IMEX and GEM are characterized by substantial memory requirements and a reliance on shared-memory parallelism. Up to recent versions, these simulators use OpenMP to parallelize computations within a single node, which means that a single simulation cannot be distributed across multiple nodes. As a result, their performance depends strongly on the characteristics of the compute instance used, particularly memory capacity, memory bandwidth, and CPU architecture.

4.2 History Matching

The purpose of a reservoir simulation is to predict future behaviour (production rates, failure risks, etc) of the target oil reservoir. Engineers use them to evaluate the effects of different exploitation strategies using only a fraction of the necessary cost of implementing them in the real world. Generally, the objective of production studies fall in one of three different categories:

- To identify the optimal reservoir exploitation strategy which consists of the num-

ber, positions and types of well, injection flows, etc;

- To make predictions about future production rates (oil, gas and water flows), pressure and saturation inside the reservoir, the composition of fluids, etc;
- To match historical performance with simulation results in order to calibrate the flow model parameters such as porosity and permeability.

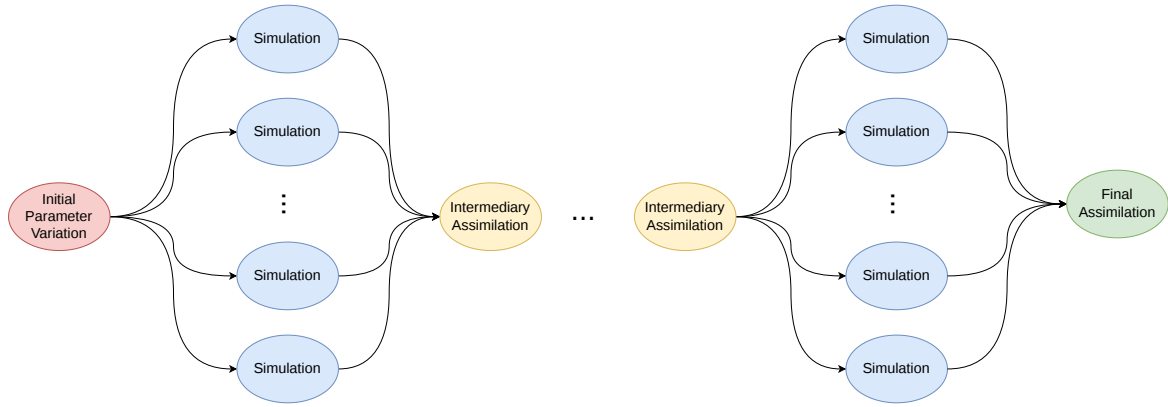


Figure 8: A typical history matching workflow.

The execution time of reservoir simulations grows rapidly according to the resolution (size) of the employed geological model. Because of this, every reservoir model carries a certain degree of uncertainty, as the fine-grained geological properties collected by exploration studies need to be aggregated and simplified into coarser models, in a process called upscaling. History matching (HM) mitigates these uncertainties by assigning values to the parameters of the model in such a way that the mathematical model of the reservoir reproduces the behaviour observed during the production period, leading to a better forecasting. The process of history matching is a big, computationally expensive, bag-of-tasks workflow, as it requires the execution of hundreds of simulations of the same reservoir model.

The History Matching workflow, shown in Figure 8, is composed of two recurring phases: an evaluation phase and an assimilation phase. In the evaluation phase, hundreds of simulations are executed, each one containing small variations in its geological model. On the other hand, the assimilation phase processes the results of the simulations and generates new geological models based on the simulations that were closer to the real production behaviour. This process is executed a predetermined number of times and the final result is a geological model that better reflects the production history.

To evaluate the SMART capabilities we used history matching as a bag-of-tasks use-case. For the simulation jobs, we used a semi-synthetic reservoir simulation model, known as Pre-Salt as the workload. This model represents the complex pre-salt reservoirs of the Brazilian coast, currently being explored by Petrobras. There is a version of this model for each of the CMG's simulators (GEM and IMEX), each with different model resolutions: Pre-Salt 100X100 is a heavier model that is representative of real-world simulations and Pre-Salt 200x200, is a smaller version that is used for benchmarking.

4.3 Predicting the Execution Time of Reservoir Simulations

Engineers may be interested in the production of a given oil field during a few months or even decades into the future, for example, for the years that a company holds the lease for that field. Thus, the execution time of a single simulation can vary significantly from several minutes to hundreds of hours, depending on characteristics related to both the simulation and the computing environment.

The behaviour of reservoir simulations can be affected by a number of factors, e.g., the particular type of simulation (*e.g.*, black oil (IMEX) or compositional (GEM)) and the version of the simulation software tool used (*e.g.*, GEM 2017.10); the sparse matrix solver adopted and its numerical parameters (*e.g.*, the maximum number of simulation days per time step); the size and resolution of the geological model; the reservoir fluid model; and the number, positioning, and dates of well openings and closings. Although simulations typically terminate when they reach a user-defined date, they may terminate sooner if certain stop conditions are met, for example, if the execution time, the pressure change in a time step, or the number of time steps should exceed user-defined thresholds. All of these factors can make predicting a simulation's execution time, with a certain degree of precision, extremely difficult. However, this information may be fundamental in deciding how to schedule a simulation while best meeting the user's quality-of-service criteria given the finite set of available resources.

The goal of the ETP module is to estimate the (remaining) execution time with enough accuracy while the simulation is running, in a manner that is agnostic to the choices made by the engineer listed above. Note that in terms of the accuracy required of the proposed solution, in the context of job scheduling, the estimates provided only need to be sufficiently precise enough to allow the respective decisions to be made correctly.

4.3.1 Extrapolating Performance

Simulation tools iterate through models one step at a time, with each "time step" potentially having a different duration for efficiency and accuracy (Gasparini; Rodrigues, *et al.*, 2021). Larger time steps reduce the total number of iterations and tend to shorten the *Total Execution Time (TET)* of the simulation. However, large time steps may sacrifice precision, necessitating recalculations with smaller time steps. Additionally, simulation execution times are influenced by the hardware architecture, CPU cores, memory, and I/O bandwidth due to parallelization. The Execution Time prediction (ETP) module estimates the remaining run time of a reservoir simulation based on the application's log generated during its execution. This log file provides the user with key information about the state of the simulation model after each completed time step. Because the tool is executed periodically in parallel with the simulation, it has been designed to be lightweight, using only simple linear extrapolation techniques to provide estimates quickly.

To evaluate the progress rate of a simulation and predict the *TET*, the ETP adopts the concept of a sliding time window to represent a sample temporal period of the simulation. Associated with each window of approximately N simulation days is its corresponding execution time. Note that the number of time steps may vary between different windows. To derive the execution time for a window, each completed time step and its respective completion time are collected during the execution.

Associated with each time step i is the *Simulation Period* $SP(i)$ elapsed since the start of the simulation, which is calculated by scanning the output log file and aggregating the number of simulation days processed during each completed time step. The *Current Execution Time* (in seconds) $CET(i)$ is the difference between the time at which the simulation started and the time at which the timestep i was completed. To calculate how fast the simulation is evolving in a certain window, the ETP computes the average progress rate at of a window that starts at timestep i and ends at a timestep j , $PR(i,j)$ as shown in Equation 4.1:

$$PR(i,j) = (CET(j) - CET(i)) / (SP(j) - SP(i)) \quad (4.1)$$

The *Estimated Remaining Time*, $ERT(i,j)$, depends on the number of remaining simulation days, so the ETP presumes that the Equation 4.2 will be an acceptable approximation, given that the simulator obtains TSP from the simulation's stop condition.

$$ERT(i,j) = PR(i,j) \times (TSP - SP(j)) \quad (4.2)$$

Extracting the Total Simulation Period (TSP) from the simulation configuration files is not a trivial task, as it may not be explicitly indicated or defined at a specific location. Since the version released in 2019, the CMG simulator has provided a web interface to indicate the percentage progress of the simulation to the user and a web socket server that provides updates after the completion of each time step k . The web interface estimates the current progress of the simulation, $CP(k)$, based on one of three stop conditions identified by the use of the respective keyword in the simulation's configuration files: the last simulation date before the keyword STOP; a maximum number of simulation time-steps determined by the keyword MAXSTEPS; and a maximum wall clock time specified by the keyword MAXTIME. Thus, if not provided explicitly, ETP estimates the Total Simulation Period TSP , at every time step k , as $\frac{SP(k)}{CP(k)}$. Furthermore, instead of polling the output file for updates, the tool simply logs the time that the web socket signals the completion of each time step.

The blue lines in Figures 9a and 9b present the progress (on the x-axis) of a one-year Pre Sal 200×200 simulation over time (on the y-axis), while the red line is an extrapolation of the progress rate during a given time window to estimate the simulation's finishing time. This technique shows that, for simulations that exhibit a well behaved progress rate, prediction errors can be quite small, with errors typically diminishing as later time windows are used.

However, in practice, simulations do not exhibit constant rates of progress during their execution, and therefore this approach is highly susceptible to time-varying, non-negligible prediction errors. To help address this, N should be sufficiently large so that there is enough historical data to identify cyclic events and sustained progress rates but short enough to adjust its estimate quickly without extenuating fluctuations in its predictions. Furthermore, instead of providing a single ERT value, several variations of this linear extrapolation approach are applied simultaneously to return an interval for the estimate. The smaller the interval, the higher the confidence one might have in the tool's prediction. ETP currently employs four variations of the sliding-time window concept, while others can be easily added later:

- *Naive Window* where $PR(i)$ is determined over the total period of the simulation completed so far. If $PR(i)$ slows over time, this prediction can be used as a lower bound;
- *Fixed Sliding Window* calculates $PR(i)$ based on the behaviour over the last N simulation days;

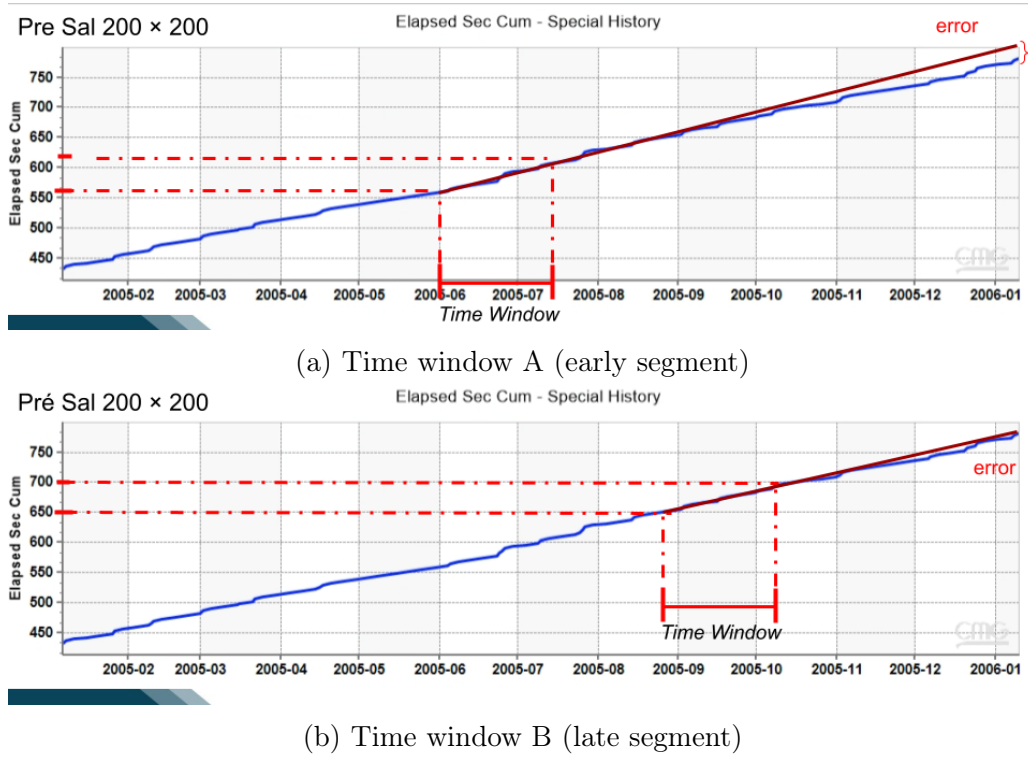


Figure 9: Pre Sal 200×200 - TSP of one year - (a) and (b) depict two Time Windows at different stages of the simulation and their respective prediction errors.

- *Adaptive Sliding Windows* - Before calculating $PR(i)$ for a given window, the window's interval and duration may be adjusted, should temporary but significant slowdowns be detected. To detect such changes in the simulation's behaviour, the tool analyses the progress rate between successive time steps. The following variants are currently utilized: *The Optimist* reduces the interval of the window by ignoring periods where slowdown events occur; *The Pessimist* reduces the time interval of the window so that it begins at the time step at which the slowdown in the progress rate was first detected.

Figure 10 presents the behaviour of the four predictors for Pre-Salt 100×100 (from the year 2000 to 2048), during its execution of $\approx 33,575.88$ seconds. Each dot represents an individual estimate of TET by a single predictor; the closer to the blue line (the actual total execution time), the better.

The further the simulation progresses (*i.e.*, moves to the right), the more accurate each of the predictors becomes. As expected, Figure 10 shows that the predictions during the first $\approx 2,500$ simulation days are not as precise as the others since, during this period, the simulation exhibits higher progress rates than during the rest of the simulation, which leads to the predictor underestimating the TET . Unfortunately, these progress rates can

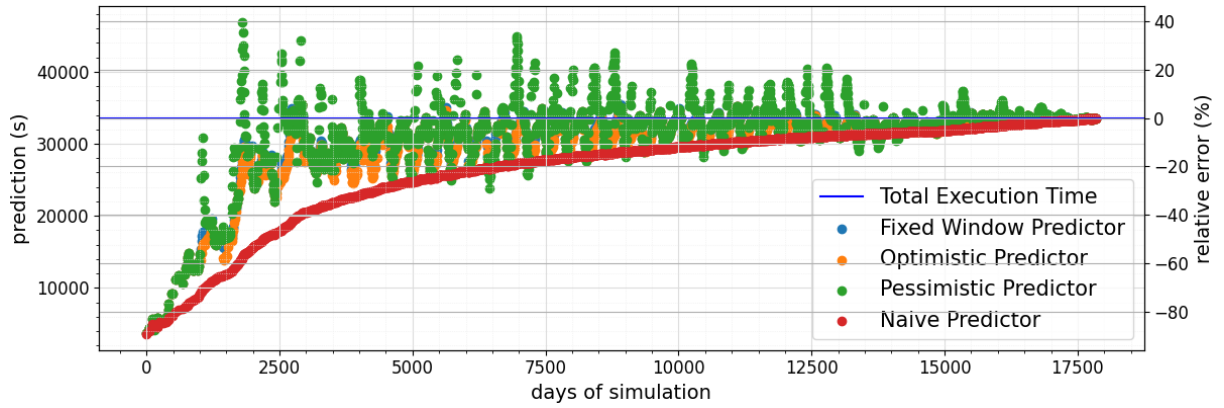


Figure 10: Changing estimates of the total execution time by the four predictors during the simulation's execution.

vary over time as they depend on the characteristics of the simulation model. Experimental evaluations with a variety of distinct models have provided positive indications of the predictor's potential. For example, the ETP predictions are within 10% of the total execution time of the Pre Sal 100×100 49-year simulation for 82% of its duration.

4.4 CMG's Simulations Offline Profiling Phase

Parallel reservoir simulators typically rely on domain decomposition, where the computational mesh representing the reservoir is partitioned into multiple subdomains. Each subdomain is assigned to a parallel thread (or MPI rank, depending on the implementation), which independently solves the fluid-flow equations for its region. After each timestep, the simulator resolves the inter-domain fluxes and pressure interactions, ensuring consistency across the global solution.

Because reservoir meshes are often highly irregular, especially in real-world geological models with faults, pinch-outs, and heterogeneous rock properties, the partitioning produced by different thread counts can vary significantly. Changing the number of threads alters the domain boundaries, which in turn changes the load distribution, communication patterns, and sometimes even the numerical behaviour of the solver. In many commercial and academic simulators, this can lead to non-reproducible results (different numerical paths due to floating-point ordering), imbalanced partitions that degrade performance or even execution failures.

Because of these characteristics, engineers running reservoir simulations on on-premises clusters typically validate their models using a restricted set of thread counts that are known to produce correct results while maximizing CPU utilization and minimizing exe-

cution time. This validation step is essential because each cluster has different hardware configurations, which influence how well a given thread count performs.

The Pre-Salt benchmark model, however, is intentionally designed to be numerically stable across different numbers of threads. As a hardware evaluation benchmark, it must run correctly under any thread count, ensuring that performance comparisons across machines are meaningful and not confounded by numerical factors.

4.4.1 Finding the Set of Feasible Instances

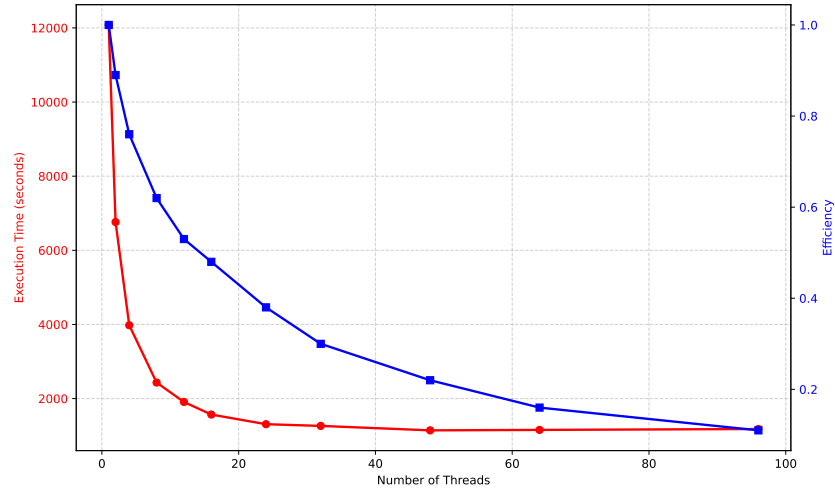
AWS instance types provide a fixed number of CPUs, depending on the instance size. Table 2 lists the possible CPU counts for all AWS’s c, m and r families instance sizes. As the first step of the offline profiling phase (which is usually done by the user), we executed the Pre-Salt 200 model using every thread count that fully utilizes the CPUs of each AWS instance size to test scalability and collect the memory usage.

Instance Size	CPUs
large	1
xlarge	2
2xlarge	4
4xlarge	8
8xlarge	16
9xlarge	18
12xlarge	24
16xlarge	32
18xlarge	36
24xlarge	48
32xlarge	64
48xlarge	96

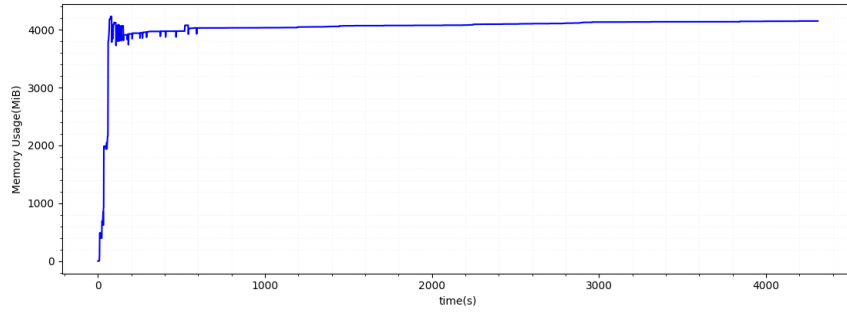
Table 2: Number of CPUs for each possible instance size of the AWS’s c, m and r families.

Figure 11a presents the average execution time of three executions of the Pre-Salt 200 benchmark on a 96-CPU c7i.48xlarge instance for each tested thread count. The results indicate that the application scales reasonably well: the execution time decreases from 12079 seconds down to 1569 seconds as the number of threads increases up to 16. Beyond 18 threads, however, the performance gains become marginal, reaching it’s lowest at 48 threads with an execution time is 1.142 seconds. For 64 and 96, threads the execution time begins to degrade.

Regarding memory usage, reservoir simulations must keep the entire reservoir grid state resident in memory throughout the execution to maintain performance. Because of



(a) Execution time and efficiency across increasing number of threads.



(b) Memory consumption during the execution.

Figure 11: Memory consumption, Execution time and efficiency for the Pre-Salt 200×200 simulation model running in a C7.48xlarge instance.

this, the reservoir's grid size and resolution has a strong influence on memory consumption, making the profiling phase essential for understanding resource requirements. For the Pre-Salt 200 benchmark, the memory usage remained nearly constant at approximately 4GiB during the entire run, as shown by figure 11b.

Based on the profiling results, the feasible thread counts were defined as all values listed in Table 2, except for the 64- and 96-threads configurations, which exhibited performance degradation. The memory requirement for the benchmark was set to 4GiB, and the minimum acceptable CPU utilization was defined as 80%. Because the CMG simulator used in this study is compiled exclusively for x86 processors, the feasible instance families were restricted to x86-based compute-optimized (c), general-purpose (m) and memory-optimized (r) AWS instance families. These parameters were provided as input to Algorithm 1, which produced the feasible instances list shown in Table 3.

The only instance types affected by the memory requirement were the compute-optimized family's *large* sizes, which provide exactly 4GiB of ram, leaving no room for the

Table 3: Instance types and families used on the profiling phase.

Category	Instance Family	CPU Model
compute	c6a	AMD EPYC 7R13
	c5a, c5ad	AMD EPYC 7R32
	c7a	AMD EPYC 9R14
	c5n	Intel Xeon Platinum 8124M
	c5, c5d	Intel Xeon Platinum 8275CL
	c6i, c6id, c6in	Intel Xeon Platinum 8375C
	c7i	Intel Xeon Platinum 8488C
general-purpose	m6a	AMD EPYC 7R13
	m7a	AMD EPYC 9R14
	m5, m5dn, m5n	Intel Xeon Platinum 8259CL
	m6i, m6in	Intel Xeon Platinum 8375C
	m7i	Intel Xeon Platinum 8488C
memory	r6a	AMD EPYC 7R13
	r7a	AMD EPYC 9R14
	r5ad	AMD EPYC 7571
	r5, r5b, r5d, r5dn, r5n	Intel Xeon Platinum 8259CL
	r6i, r6id, r6idn, r6in	Intel Xeon Platinum 8375C
	r7i	Intel Xeon Platinum 8488C

simulation and the operating system to execute at the same time. The architecture restriction excluded all of the AWS’s Arm-based graviton instances. The newer 8th-generation instance families were not included in the evaluation, and due to restrictions in the execution environment, several instance families could not be used. These excluded families are highlighted in bold in Table 3. Finally, the minimum CPU utilization requirement removed all configurations with 64 (32xlarge) and 96 (48xlarge) CPUs.

4.4.2 Efficient Profiling of CMG Simulations

Even after applying all restrictions, the algorithm identified 204 feasible instance types across the selected families. Executing the full benchmark simulation on every one of these configurations may be prohibitively expensive for many users. To reduce profiling costs, this work proposes a method for constructing a curtailed version of the simulation that faithfully represents the behaviour of the full execution, using checkpoint and restore.

Reservoir engineers frequently need to re-run specific portions of a simulation to evaluate parameter variations or test alternative operational strategies without restarting the entire model. To support this workflow, reservoir simulators such as CMG provide built-in checkpoint and restore mechanisms. The challenge is then determining which

subsection of the simulation is the most representative of the overall execution. To address this, we rely on the progress rate (Equation 4.1) of the simulation. This rate is primarily determined by the amount of numerical work performed by the linear solver at each timestep, the processing capacity of the underlying hardware, and the number of threads used. Although the progress rate may vary throughout the run, each subsection of the simulation exhibits a characteristic average rate. Building on this idea, we developed a method to sample the execution of a reservoir simulation and identify a subsection of length n seconds whose average progress rate matches that of the full simulation. Once this representative interval is found, we create a checkpoint at its beginning and terminate the run at its end, effectively producing a "sub-simulation" that preserves some of the characteristics of complete model. After executing the curtailed simulation, we compute its average progress rate ($P(CS)$) and then estimate the total execution time of the full model by multiplying $P(CS)$ by the total simulation period (TSP) of the full simulation.

Table 4: Execution time of the full and short versions of the Pre-Salt 200×200 model. The presented values are an average of three executions.

Instance	Threads	Full ET	Benchmark ET	Estimated ET	Error
c5.24xlarge	10	3755.82	343.27	3393.90	-9.64%
	20	2455.57	220.16	2187.28	-10.93%
	40	1952.76	192.53	1916.04	-1.88%
c6a.24xlarge	10	2649.00	241.75	2396.16	-9.54%
	20	1868.17	164.47	1638.01	-12.32%
	40	1576.72	153.33	1529.60	-2.99%
m7i.24xlarge	10	1838.11	166.07	1654.23	-10.00%
	20	1289.24	111.30	1116.86	-13.37%
	40	1200.75	116.06	1163.96	-3.06%

To evaluate this approach, a shortened version of the Pre-Salt 200 model (Short Pre-Salt 200) was created with 10% of its execution time, following the proposed technique. Both the full and shortened benchmarks were executed with different thread counts (10, 20, and 40) on three different instance types (c5.24xlarge, c6a.24xlarge, and m7i.24xlarge), each offering 48 CPUs. Every configuration was executed three times, and the average results are presented in Table 4. While the execution-time estimates derived from the short benchmark exhibited an average error of 8%, the Short Pre-Salt benchmark consumed 90.8% less CPU time than the full simulation, substantially reducing the profiling costs.

To complete the profiling process, the Short Pre-Salt 200 model was executed on every instance in the feasible list. Each configuration was run three times for consistency,

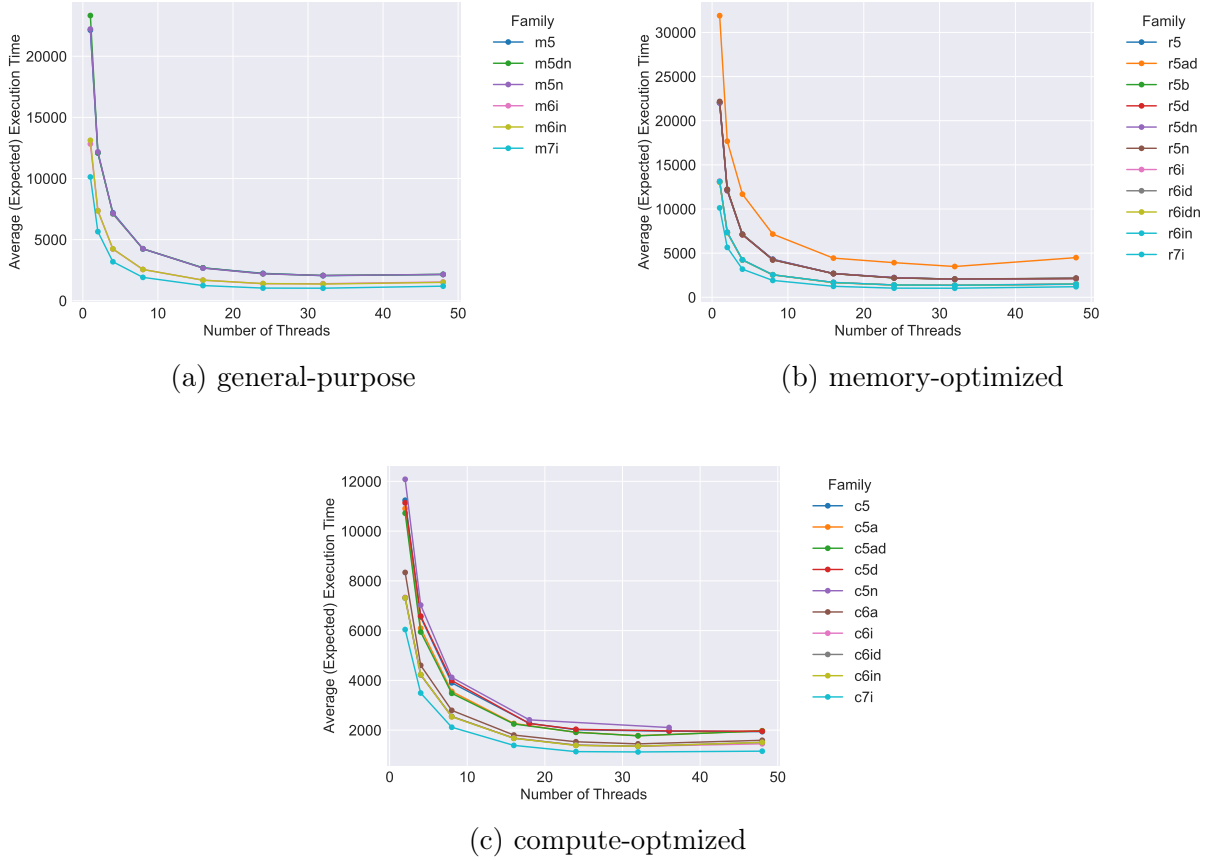


Figure 12: Execution time vs threads for categories m, r, and c.

although repeated executions are not required by the methodology. Figure 12 presents the average estimated execution time for each configuration. The simulation’s scalability exhibited similar behaviour across all instance families, with newer generations achieving faster execution times (as expected, since simulation performance is dominated by CPU capabilities). As clearly illustrated in Figure 12b (and applicable to all categories), only four distinct lines appear in the graph even though the benchmark was executed on all eleven listed families. This occurs because instance types sharing the same CPU model tend to deliver nearly identical execution times. Based on this observation, the profiling phase could be restricted to instances with different CPU models, reducing the number of evaluated instance types from 204 to 60, further lowering profiling costs.

4.5 Multi-objective Instance Allocation

Using the data generated during the profiling phase, SMART proceeds to the static selection phase. In this stage, the execution-profiling results are used as input to the Algorithm 2 to define the solution space, and the Algorithm 3 is applied to identify the

set of Pareto-optimal solutions. Figure 13a illustrates the solution space generated for executing 100 Pre-Salt 200 simulation jobs, considering Spot prices collected on August 19th, 2026. The colour scale represents each solution's price discount relative to its On-Demand counterpart: dark purple points correspond to On-Demand-based solutions, while yellow points indicate Spot-based solutions with the highest discounts. Although the feasible list contained only 204 instance types, the resulting solution space included 2,941 solutions, highlighting how region, availability zone, and market variations significantly increase the complexity of the decision process.

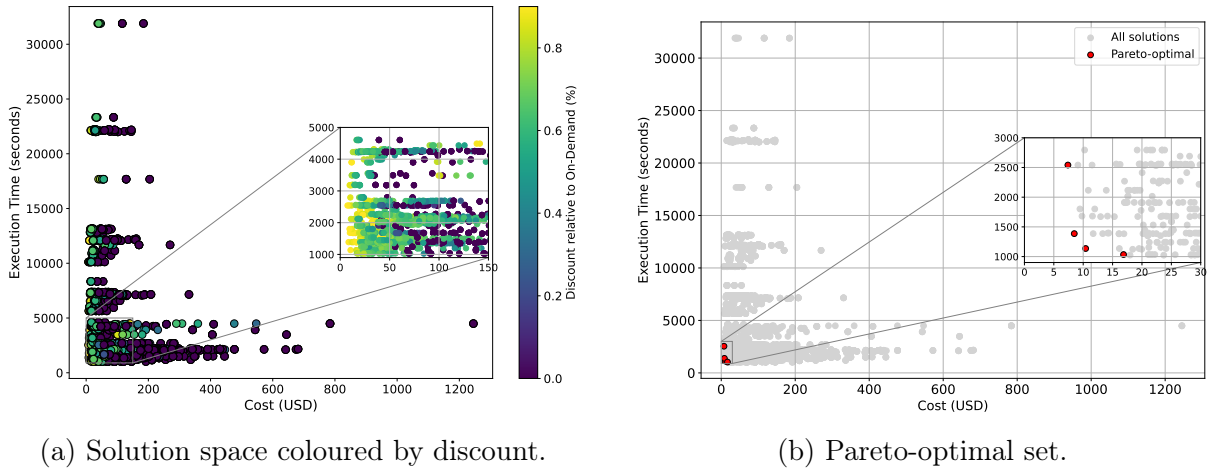


Figure 13: Solution Space and Pareto Optimal set for the execution of 100 Pre-Salt 200 Jobs (Based on prices collected on August 19th, 2026).

Figure 13b shows the Pareto-optimal solutions identified by Algorithm 3 using a tolerance of 5%. Despite the 2,941 candidate solutions, only four were considered non-dominated and, therefore, classified as Pareto-optimal. Table 5 presents an overview of the four Pareto-optimal configurations, ordered by cost, including their estimated execution time, instance type, market, region, and availability zone. These results highlight the importance of presenting multiple configuration options to the user. Solution 1 offers the lowest cost, while Solution 4 delivers the shortest execution time. When comparing Solutions 1 and 2, we observe that although Solution 1 is only 12% cheaper, it is 1.83 times slower. Likewise, comparing Solutions 2 and 4 shows that Solution 4 is 25% faster, but nearly twice as expensive.

At first glance, the intermediate trade-off solutions may appear to be the most balanced choices, but the ideal configuration depends entirely on the user's priorities. In some situations, a reservoir engineer may need results as quickly as possible to support a time-critical production decision that could influence a multi-million-dollar operational outcome. In such cases, paying more for a faster configuration is strategically necessary.

Table 5: Pareto-optimal solutions for 100 Pre-Salt 200 simulation jobs.

Sol.	Exec. Time (s)	Cost (USD)	Instance Type (market)	Region / AZ
1	2540.79	7.40	c6i.4xlarge (spot)	sa-east-1c
2	1386.35	8.48	c7i.8xlarge (spot)	sa-east-1a
3	1135.77	10.42	c7i.12xlarge (spot)	sa-east-1a
4	1034.11	16.86	m7i.12xlarge (spot)	us-east-1f

In other scenarios, the engineer may submit a job on Friday and only require the results by Monday, making the lowest-cost configuration far more attractive. The value of the Pareto set lies precisely in enabling the user to choose the configuration that best aligns with the urgency and financial impact of each decision.

4.5.1 Dynamic Allocation under Availability Constraints

Once the user chooses the solution that best matches their requirements, SMART attempts to allocate the cloud instances that will produce an execution as close as possible to the chosen configuration, following algorithm 4. To evaluate this process, we simulated a scenario with restricted cloud availability, where only 10% of the total required instances cloud be obtained at each allocation attempt. Table 6 presents the evolution of the four Pareto-optimal initial solutions under this constraint.

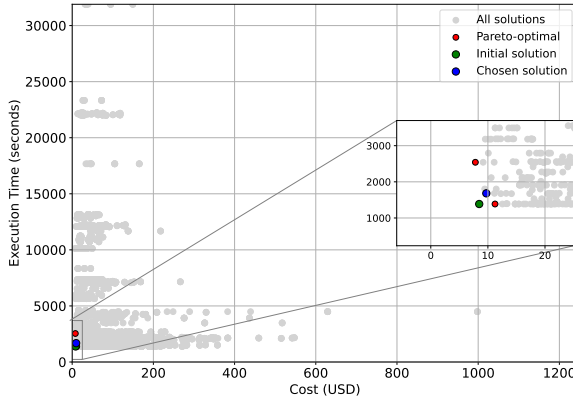
Table 6: Evolution of solutions 1, 2, 3, and 4 under availability restriction.

Attempt	Solution 1		Solution 2		Solution 3		Solution 4	
	Time	Cost	Time	Cost	Time	Cost	Time	Cost
1	2540.79	7.40	1386.35	8.48	1135.77	10.42	1034.11	16.86
2	2540.79	8.37	1386.35	10.22	1386.35	8.67	1034.11	17.86
3	2540.79	8.87	1681.39	9.72	1386.35	11.25	1034.11	20.35
4	2540.79	9.11	1801.69	9.48	1386.35	12.38	1034.11	20.48
5	2540.79	9.32	1801.69	10.76	1395.73	12.89	1135.77	13.97
6	2540.79	9.63	1801.69	10.82	1395.73	12.98	1135.77	16.84
7	2540.79	9.81	1801.69	10.93	1395.73	13.68	1135.77	17.74
8	2540.79	10.00	1801.69	11.22	1395.73	13.68	1135.77	18.07
9	2540.79	10.02	1801.69	11.30	1395.73	13.69	1135.77	18.16
10	2540.79	10.03	1801.69	11.38	1395.73	13.76	1135.77	18.33

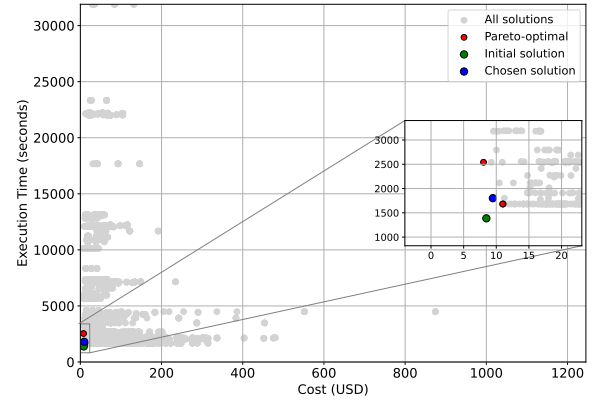
As expected (demonstrated in claim 1 [Section 3.3]), all final allocations exhibit some

degradation in either execution time or cost. This occurs because the algorithm is forced to introduce heterogeneity into the solution as certain instance types become unavailable. Even under these constraints, however, all solutions remained within a 35% deviation from their original objectives. Notably, none of the final allocations had on-demand instances.

Solution 1 is the most stable in terms of execution time, maintaining the same runtime across all attempts while experiencing a cost increase of approximately 35%. This behaviour arises because Solution 1 can incorporate any instance type with execution time equal to or faster than its original configuration without worsening its time, greatly increasing the available options.



(a) Solution space for solution's 2 third allocation attempt.



(b) Solution space for solution's 2 fourth allocation attempt.

Figure 14: Solution Space and Pareto Optimal set for the main changing points of the solution 2 allocation path.

Solution 2 exhibits the largest degradation in both execution time and cost, ending with a configuration that is 30% slower and 34% more expensive than the initial one. The main changing points occur during attempts 3 and 4, illustrated in Figures 14a and 14b. In attempt 3, the algorithm must choose between maintaining the original execution time at the expense of a 37% cost increase, or accepting a 21% slowdown with only a 15% cost increase. Because similarity is computed using Euclidean distance in the normalized objective space, the algorithm selects the more balanced alternative. A similar trade-off happens on attempt 4. Solution 3 follows the same pattern, with the algorithm consistently preferring balanced deviations over extreme changes in a single metric.

Solution 4 achieved the best overall stability, only a 10% increase in execution time and 9% increase in cost, when compared to the initial solution. This outcome reflects the higher availability of similar instance types in the solution space, which reduces the need for aggressive substitutions. The degree of stability observed in Solution 4 may vary

depending on Spot market conditions at the time of allocation.

Although the final allocations depend heavily on real-time price and availability fluctuations, it is noteworthy that the relative ordering of the solutions remains largely preserved. Solutions that were originally faster tend to remain faster, and solutions that were originally cheaper tend to remain cheaper, even after multiple substitutions. This indicates that SMART’s dynamic allocation process maintains the structural relationships defined in the static selection phase.

Table 7: Evolution of the lowest-cost and best-performance solutions under availability restriction.

Attempt	Lowest-Cost		Best-Performance	
	Time	Cost	Time	Cost
1	2540.79	7.40	1034.11	16.86
2	10122.01	7.95	1034.11	17.86
3	10122.01	7.68	1034.11	20.50
4	10122.01	8.245	1034.11	20.37
5	10125.48	8.20	1034.11	20.75
6	10125.48	8.11	1034.11	20.99
7	10125.48	8.46	1034.11	22.37
8	10125.48	8.29	1034.11	21.79
9	10125.48	8.21	1034.11	22.84
10	10125.48	8.34	1034.11	22.78

To contextualize these results, Table 7 compares SMART’s dynamic allocation with two single-objective strategies commonly found in the literature (Nunes *et al.*, 2024) (Sodré *et al.*, 2025): selecting the lowest-cost configuration and selecting the best-performance configuration. On large solution spaces, these approaches behave poorly under availability constraints. The lowest-cost strategy suffers severe performance degradation, with execution time increasing by nearly 300% after the first substitution. The best-performance strategy maintains execution time but incurs substantial cost inflation, reaching values 35% higher than the original estimate. In contrast, SMART maintains a balanced trajectory, avoiding extreme deviations in either dimension.

This comparison highlights the importance of multi-objective selection in real cloud environments. Availability fluctuations are inherent to Spot markets, and any strategy that optimizes a single metric is vulnerable to pathological behavior when the preferred instance type becomes unavailable. By continuously steering the allocation toward the closest feasible point in the Pareto set, SMART provides a more stable and predictable execution plan, even under restrictive or volatile market conditions.

4.6 Sync Engine Results

In this section, the integration of Sync Engine with the typical cloud-bursting workflow and the application of its data-persistence implementation (I-Rsync) are evaluated to assess the benefits and limitations of this strategy, in particular, in terms of temporal and financial impact.

4.6.1 Experimental Setup

All simulation jobs executed in the experiments ran on an AWS `m7i.24xlarge` instance, allocated in the `us-east-1` region. This is a high-performance, general-purpose computing instance with 96 vCPUs, 384 GiB of RAM, and up to 50 Gbps of network bandwidth. Furthermore, Hyper-threading was disabled, and only 48 physical cores were considered. This instance type and configuration was selected based on internal benchmarking and workload profiling of reservoir simulation jobs. A `c5.2xlarge` instance was chosen for persistence approaches that require the use of a small VM during their *Post-execution Phase*.

Table 8: Linux Kernel Tunings Parameters.

Parameters	Values
<code>net.core.rmem_max</code>	2147483647
<code>net.core.wmem_max</code>	2147483647
<code>net.ipv4.tcp_rmem</code>	4096 16777216 1073741824
<code>net.ipv4.tcp_rmem</code>	4096 16777216 1073741824
<code>net.ipv4.tcp_congestion_control</code>	bbr
<code>net.core.default_qdisc</code>	fq
<code>vm.dirty_bytes</code>	3221225472 (3 GB)
<code>vm.dirty_background_bytes</code>	536870912 (512 MB)
<code>sunrpc.tcp_slot_table_entries</code>	128

For these experiments, we simulated Spot instance eviction using on-demand instances to ensure that the eviction notifications were always sent at exactly the same moment. The NFS client version 3, with `rsize` and `wsize` set to 256 MB, was used to mount the on-premises namespace due to the backend storage configuration. The network connection between the cloud provider and the on-premises facilities uses AWS Direct Connect ([Amazon Web Services, 2026b](#)).

Red Hat Enterprise Linux 8.10 was used, with kernel tunings applied to the VM's im-

age to improve data transfer over a high-latency NFS connection. Table 8 lists the parameters and the values that were changed to improve TCP Auto-Tuning, TCP send/receive, TCP Congestion Algorithm Control, VM Dirty Page Cache, and RPC Slot Table.

4.6.2 Comparing Persistence Approaches

Four approaches were evaluated under identical workload conditions and with the same read path from on-premises storage:

- **Baseline:** The simulator writes only to cloud-local storage, specifically to the Amazon Elastic File System (EFS). No data is returned to on-premises storage during execution. This configuration is not a viable production solution; it serves solely to define a lower bound on write overhead;
- **Current:** The usual production configuration, in which each simulation job writes output directly to the on-premises storage via NFS, without the use of cloud local storage;
- **Naive:** A naive **post-hoc copy** strategy, transferring data to the remote server with the command-line utility `scp` ([The OpenBSD Project, n.d.](#)). In this scenario, after a **Baseline** job completes, the user creates another instance in the cloud to manually `scp` the output data from the cloud storage back on-premises;
- **I-RSYNC:** In the SYNC ENGINE approach, the simulation job writes directly to EFS (cloud-local storage); independently, `rsync` periodically synchronizes the output files in their current state to on-premises storage.

4.6.3 Experimental Methodology and Metrics

All experiments were run under the same job submission configuration, typical of the production simulation jobs run daily using the commercially licensed CMG software. The test job was a curtailed version of a compositional (GEM) reservoir simulation of a semi-synthetic Pre-Salt model at 100×100 meter resolution, a model representative of real-world models with their high I/O intensity and large output files. For the chosen cloud instance, the simulation jobs were submitted with 40 cores, which matches the typical number of cores that engineers use when submitting this reservoir simulation model.

The primary metric assessed is the *Data Availability Time (DAT)*, defined as the total elapsed time since the actual start of the simulation until the whole set of output files is

fully available on the on-premises storage (the exception being the **Baseline** scenario, in which case, the *DAT* is simply the simulation’s local execution time). To help interpret the *DAT* values, the duration of the **Close Phase** has also been measured and corresponds to the elapsed time from the end of the execution of the simulation job until the actual closing of the files (including a checkpoint due to Spot revocation).

The evaluation was carried out under two scenarios: (i) where simulation jobs configured for the different persistence approaches (defined in Subsection 4.6.2) were submitted to the on-premises job scheduler for execution on the chosen cloud instance without the occurrence of evictions, and; (ii) where a simulated Spot VM eviction notice is sent to the job to initiate the simulation’s graceful termination, which includes creating a checkpoint. In both cases, the information from the job’s log file is used to determine the total execution time.

The results presented for each approach are the average of five executions. In the case of the I-RSYNC approach, the execution parameters were varied in order to find the best configuration setup. For each of the two `rsync` execution modes: `SSH` and `Daemon`, three values for the time interval (I_{rsync}) were chosen: 5, 10, and 15 minutes. Each combination was also executed with and without `rsync`’s built-in compression option, in which the `zlib` (Gailly; Mark, 1995) algorithm is used as standard

4.6.4 Results

This section first presents the results of a comparison of the average *Data Availability Times* (*DATs*) for different configurations of the SYNC ENGINE’s I-RSYNC approach. This is followed by a comparison of different persistence approaches to assess their relative performance and the overhead of persisting data on-premises.

Table 9 presents the *DATs* for the I-RSYNC approach, for each proposed configuration, without (NC) and with compression (C). Note that, for I-RSYNC-C, compression did not provide any real benefit in terms of *DATs*. Among the simulation’s outputs is a 12 GB HDF5 (Folk; Cheng; Yates, 1999) file that is already highly compressed. As `rsync` applies compression to all files in the `workdir` directory being synchronized, it spends time fruitlessly compressing the HDF5 file and unnecessarily increasing the duration of its **Post-execution Phase**. The best configurations, highlighted in bold, were compared with other persistence approaches in the following experiment.

Figure 15 shows the average *Data Availability Time* for each approach. Each bar

Table 9: Data Availability Times (DATs) (in seconds) for different configurations of the two SYNC ENGINE approaches.

	No Compression (NC)			With Compression (C)		
I-RSYNC	I_{rsync} (min)					
	5	10	15	5	10	15
SSH	3621.6	3720.2	3632.8	3696.8	3774.4	3695.0
Daemon	3535.0	3588.2	3559.4	3755.4	3764.2	3819.8

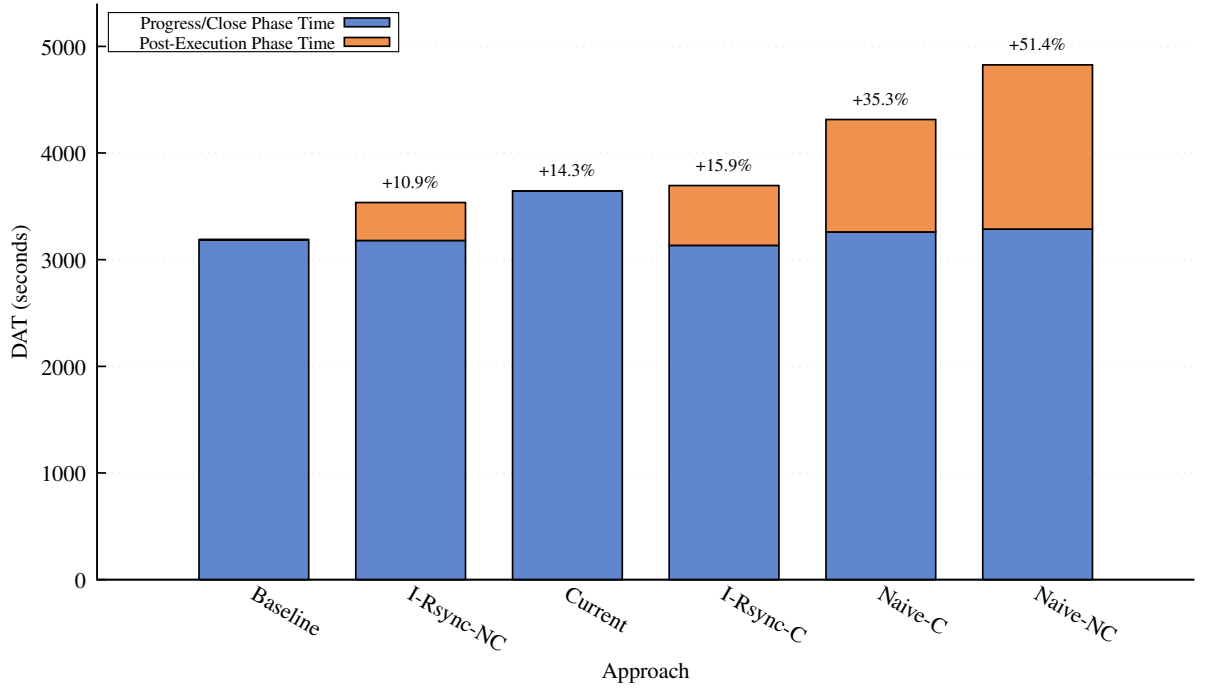


Figure 15: Data availability time.

portrays the *Progress Phase* and *Close Phase* time (in blue) with the *Post-execution Phase* (in orange). Above each bar is how much slower each approach was to execute the simulation and persist all output data on-premises, as a percentage, in relation to the **Baseline**’s cloud-only execution.

Figure 16 presents the duration of just the *Close Phase* for each respective approach in relation to AWS’s two-minute eviction notification window.

4.6.4.1 On-premises Data Availability Latency

The primary metric, *DAT*, measures the total elapsed time required to fully replicate the job’s output files to the on-premises NFS namespace. As shown in Figure 15, *I-RSYNC without compression* (denoted as I-RSYNC-NC) incurs the lowest write overhead

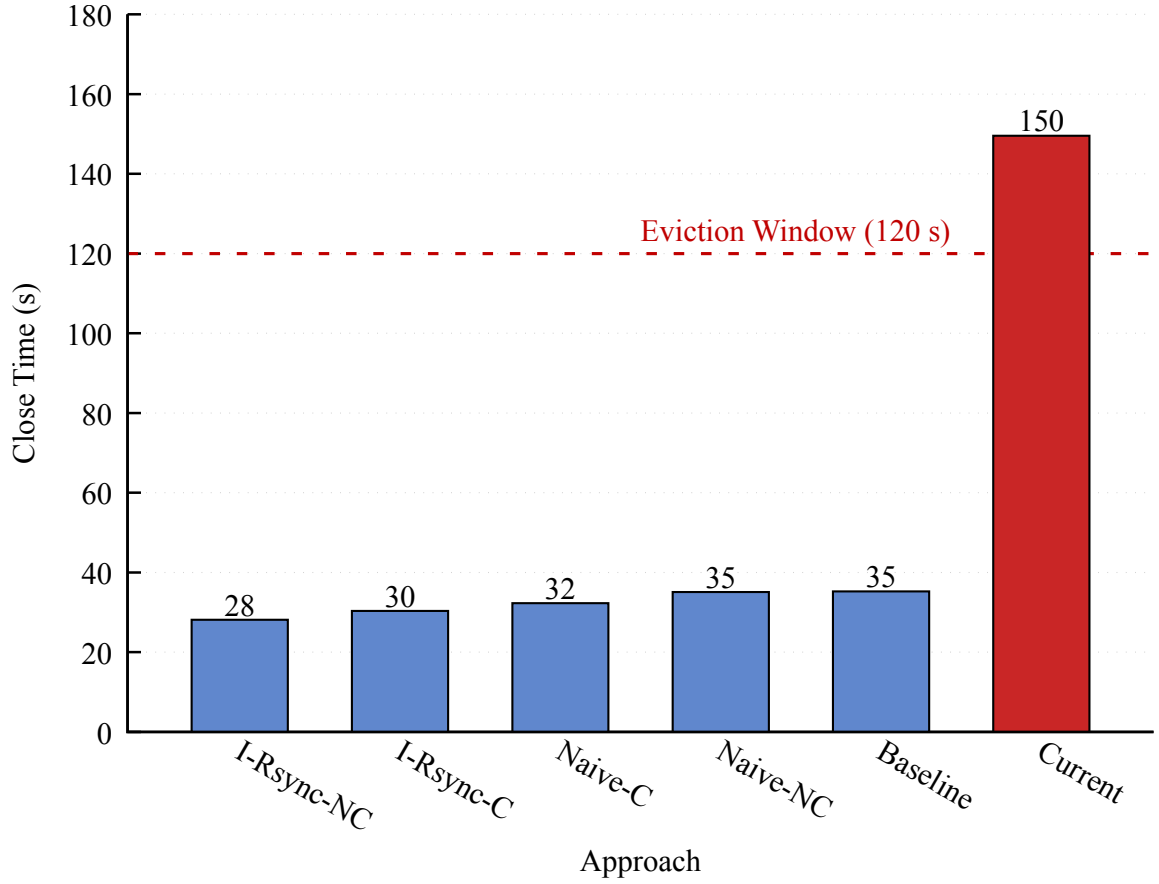


Figure 16: Close phase duration.

across all approaches, being **+10.9%** above the **Baseline**'s lower bound. While both I-RSYNC solutions can match the simulation execution times of the **Baseline** approach, they both have *Post-execution Phases* that incur overheads of **+10.9%** (for NC) and **+15.9%** (with compression). The **Current** approach writes directly to the on-premises NFS storage during the simulation's execution, causing an additional **+14.3%** delay. The worst results are those related to the **Naive** approaches, reaching up to overheads of **+35.3%** (Naive-C) and even **+51.4%** (Naive-NC) to transfer the entire volume of output post-execution. Although compression helps reduce the data transfer times, these results indicate that overlapping data mirroring with computation is essential to avoid paying for additional cloud VM usage while synchronizing outputs over a high-latency path.

4.6.4.2 Close Phase Duration and Spot-eviction Resilience

In addition to the write overhead, another crucial aspect is the fact that executing a job on a Spot instance introduces a hard deadline when an eviction notice is issued. To

avoid losing the previous computation, the application is required to write a consistent checkpoint to some persisted storage during its *Close Phase*. It is therefore beneficial that the duration of this phase be shorter than the eviction notification period.

A comparison of the durations of the *Close Phases* of the different approaches is shown in Figure 15. All of the approaches, except CURRENT, completed their *Close Phases* within this eviction notification period, with I-RSYNC-C achieving the best performance of 28s. Overall, the joint interpretation of Figures 15 and 16 is that RSYNC-NC is the only approach in the comparison that simultaneously keeps *DAT* the nearest to the **Baseline** and meets the eviction notification period for the use case considered in this work

4.6.4.3 Monetary Costs

Since cloud billing is proportional to the length of time the VM is in use, reductions in *DAT* directly translate into lower monetary costs for the workload. Furthermore, during the *Post-execution Phase* executed by SYNC ENGINE, an additional VM might be necessary. Nonetheless, despite the additional cost, these VMs are usually very cheap relative to VM instance required by the HPC application.

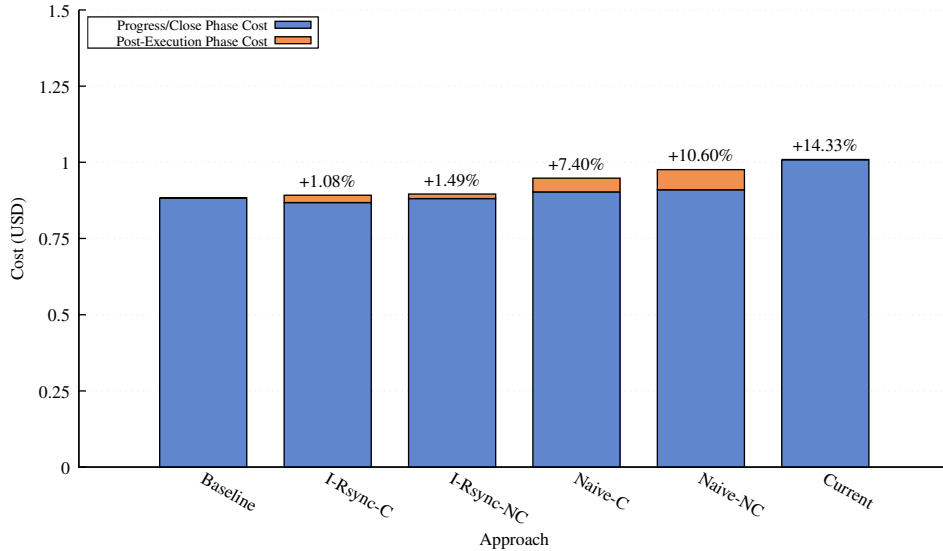


Figure 17: Average execution cost for each approach, in USD.

Figure 17 compares the total cost of executing the simulation job across the approaches under comparison. This cost was based on the AWS US-east-1 Spot prices at 13:40 GMT on June 2nd, 2026. The percentage above each bar indicates the cost difference relative to the **Baseline** cost of US\$0.883. It reveals that I-RSYNC-NC and I-RSYNC-C incur

lower costs than the other approaches. This is due to the fact that the faster simulation execution under the I-RSYNC approaches compensates for the cost of having to allocate an additional VM to execute the *Post-execution Phase*.

I-RSYNC-NC reduces the **Current** per-simulation cost from \$1.009 to \$0.896. While this seems like a small reduction at first glance, notice that, as stated in Section 4.6.3, the experiments used a 1-hour curtailed version of a 12-hour simulation. To demonstrate the significance of the cost savings, one can extrapolate the costs to the scenario of 1000 full reservoir simulations, a scale typical of iterative ensemble workflows using the Ensemble Kalman Filter (EnKF) (Emerick; Reynolds, 2013), used for history matching. In this scenario, our solution would reduce the total expense to US\$10752, yielding a savings of 11.2% (US\$1356) for standard engineering workloads.

5 Conclusions and Future Work

This dissertation presented SMART, a framework designed to support the execution of Bag-of-Tasks workloads in cloud environments. The framework integrates three core capabilities: an offline profiling phase that characterizes application behaviour across cloud configurations; a multi-objective instance-selection mechanism that identifies efficient cost–performance trade-offs; and a data-persistence module capable of synchronizing output files to on-premises storage under high-latency conditions and within strict Spot-eviction deadlines.

The carried evaluation on reservoir simulations demonstrated a profiling methodology capable of reducing benchmarking costs while preserving representative execution behavior. Using the curtailed Pre-Salt benchmark, profiling costs were reduced by 90.8% while maintaining an average execution-time estimation error of approximately 8%. These results indicate that using a smaller, representative, version of an application might enable characterizing large configuration spaces without incurring prohibitive profiling costs.

The multi-objective selection phase produced a complete solution space that reflects the diversity of regions, availability zones, and markets. Although the feasible list contained 204 instance types, the solution space expanded to 2,941 valid configurations and only four of them were identified as Pareto-optimal. The evaluation also showed that single-objective strategies, such as selecting the lowest-cost or fastest configuration, behave poorly under availability constraints and large search spaces, exhibiting degradation in either execution time or cost. In contrast, SMART maintained balanced deviations and preserved the structural relationships among solutions even when availability was restricted.

The Sync Engine addressed the performance limitations introduced by high-latency on-premises storage. The I-Rsync mechanism overlapped computation with asynchronous data mirroring, reducing write overhead and ensuring that checkpoint creation remained within AWS’s two-minute eviction window. The best-performing configuration, I-Rsync-

NC, achieved a Data Availability Time overhead of only +10.9% relative to the cloud-only baseline, outperforming both the current production configuration (+14.3%) and naïve post-hoc copy strategies (+35.3% to +51.4%). Furthermore, I-Rsync-NC reduced the per-simulation cost from US\$1.009 to US\$0.896, corresponding to an 11.2% reduction.

5.1 Future Work

Although the proposed framework provides a solid foundation for executing Bag-of-Tasks workloads in cloud environments, several paths remain open for future research and development.

Execution-Management Phase Integration. This dissertation evaluated the Execution Time Predictor and Sync Engine independently, but a full evaluation of the execution-management phase remains as future work. This includes automated checkpoint restoration, adaptive rescheduling based on real-time predictions and recovery under Spot revocations.

Multi-Task Execution Models. Executing multiple tasks on the same instance may improve resource utilization and execution efficiency but increases the risk associated with Spot revocations. Future work will explore how to expand this work to integrate custom multi-task schedulings.

Multi-Cloud Support. SMART capabilities can be easily extended to additional cloud providers, which could improve the execution results in terms of cost, performance and availability. Considering additional CPs, such as Azure or Google Cloud, is left as future work.

5.2 Final Remarks

The results presented in this dissertation demonstrate that efficient, reliable, and cost-effective cloud execution of HPC workloads is achievable when resource selection, runtime prediction, and data synchronization are treated as interconnected challenges. SMART provides a principled foundation for this integration, and the reservoir simulation case study highlights its practical value for real-world engineering environments.

REFERENCES

ALI, Amjad; SYED, Khalid Saifullah. Chapter 3 - An Outlook of High Performance Computing Infrastructures for Scientific Computing. *In: MEMON, Atif (ed.). **Advances in Computers***. [S. l.]: Elsevier, 2013. v. 91. (Advances in Computers). p. 87–118. DOI: <https://doi.org/10.1016/B978-0-12-408089-8.00003-3>. Available from: <https://www.sciencedirect.com/science/article/pii/B9780124080898000033>.

ALIPOURFARD, Omid *et al.* Cherrypick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. *In: PROCEEDINGS of the 14th USENIX Conference on Networked Systems Design and Implementation*. Boston, MA, USA: USENIX Association, 2017. (NSDI'17), p. 469–482. ISBN 9781931971379.

AMAZON AWS. **Amazon Elastic Compute Cloud (EC2)**. [S. l.: s. n.], 2006. <https://aws.amazon.com/ec2/>. Accessed: september, 2026.

AMAZON WEB SERVICES. **Amazon Web Services**. [S. l.: s. n.], 2026. Accessed: August, 2026. Available from: <https://aws.amazon.com/>.

AMAZON WEB SERVICES. **AWS DataSync: Automated, Accelerated Online Data Transfer**. [S. l.: s. n.], 2024. Accessed: August, 2026. Available from: <https://aws.amazon.com/datasync/>.

AMAZON WEB SERVICES. **What is AWS Direct Connect?** [S. l.: s. n.], 2026. Accessed: Jun, 2026. Available from: <https://docs.aws.amazon.com/directconnect/latest/UserGuide/Welcome.html>.

BADIA, Rosa; AYGUADE, Eduard; LABARTA, Jesus. Workflows for Science: a Challenge when Facing the Convergence of HPC and Big Data. **Supercomput. Front. Innov.: Int. J.**, South Ural State University, Chelyabinsk, RUS, v. 4, n. 1, p. 27–47, Mar. 2017. ISSN 2409-6008. DOI: [10.14529/jsfi170102](https://doi.org/10.14529/jsfi170102). Available from: <https://doi.org/10.14529/jsfi170102>.

BORIN, Edson *et al.* Why Move HPC Applications to the Cloud? *In: HIGH Performance Computing in Clouds : Moving HPC Applications to a Scalable and Cost-Effective Environment*. [S. l.]: Springer, 2023. p. 1–5.

COMPUTER MODELLING GROUP LTD. **CMG**. [S. l.: s. n.]. Accessed: August, 2026. Available from: <https://cmgl.ca>.

DIMITRI, Nicola. Pricing cloud IaaS computing services. **Journal of Cloud Computing**, v. 9, n. 1, p. 14, 2020. DOI: [10.1186/s13677-020-00161-2](https://doi.org/10.1186/s13677-020-00161-2). Available from: <https://doi.org/10.1186/s13677-020-00161-2>.

DURILLO, J.J.; PRODAN, R. Multi-objective workflow scheduling in Amazon EC2. **Cluster Comput**, v. 17, p. 169–189, 2014. DOI: <https://doi.org/10.1007/s10586-013-0325-0>.

EMERICK, Alexandre A.; REYNOLDS, Albert C. Ensemble smoother with multiple data assimilation. **Computers & Geosciences**, Elsevier, v. 55, p. 3–15, 2013. DOI: [10.1016/j.cageo.2012.03.011](https://doi.org/10.1016/j.cageo.2012.03.011). Available from: <https://doi.org/10.1016/j.cageo.2012.03.011>.

EZELL, S. J.; ATKINSON, R. D. **The Vital Importance of High-Performance Computing to U.S. Competitiveness**. [S. l.], Apr. 2016. Available from: <https://itif.org/publications/2016/04/28/vital-importance-high-performance-computing-us-competitiveness/>.

FERREIRA DA SILVA, Rafael; BADIA, Rosa et. al. **Workflows Community Summit 2022: ARoadmap Revolution**. [S. l.], Mar. 2023. DOI: [10.5281/zenodo.7750670](https://doi.org/10.5281/zenodo.7750670).

FOLK, Mike; CHENG, Albert; YATES, Kim. HDF5: A file format and I/O library for high performance computing applications. *In: PROCEEDINGS of supercomputing*. [S. l.: s. n.], 1999. v. 99, p. 5–33.

GAILLY, Jean-loup; MARK, Adler. **Zlib: A Massively Spiffy Yet Delicately Unobtrusive Compression Library**. [S. l.: s. n.], 1995. Available from: <https://zlib.net/>.

GASPARINI, Leonardo; RODRIGUES, José R.P., *et al.* Hybrid parallel iterative sparse linear solver framework for reservoir geomechanical and flow simulation. **Journal of Computational Science**, Elsevier, v. 51, p. 101330, 2021.

GOOGLE CLOUD. **Google Cloud Platform**. [S. l.: s. n.], 2026. Accessed: August, 2026. Available from: <https://cloud.google.com/>.

GUNARATHNE, Thilina *et al.* Cloud computing paradigms for pleasingly parallel biomedical applications. *In: PROCEEDINGS of the 19th ACM International Symposium on High Performance Distributed Computing*. [S. l.: s. n.], 2010. p. 460–469.

GUPTA, Abhishek *et al.* The Who, What, Why, and How of High Performance Computing in the Cloud. *In: 2013 IEEE 5th International Conference on Cloud Computing Technology and Science*. [S. l.: s. n.], 2013. v. 1, p. 306–314. DOI: [10.1109/CloudCom.2013.47](https://doi.org/10.1109/CloudCom.2013.47).

HSU, Chin-Jung; NAIR, Vivek; FREEH, Vincent W., *et al.* Arrow: Low-Level Augmented Bayesian Optimization for Finding the Best Cloud VM. *In: 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. [S. l.: s. n.], 2018. p. 660–670. DOI: [10.1109/ICDCS.2018.00070](https://doi.org/10.1109/ICDCS.2018.00070).

HSU, Chin-Jung; NAIR, Vivek; MENZIES, Tim, *et al.* Scout: An Experienced Guide to Find the Best Cloud Configuration. **ArXiv**, abs/1803.01296, 2018.

HWANG, Eunji *et al.* Resource allocation policies for loosely coupled applications in heterogeneous computing systems. **IEEE Transactions on Parallel and Distributed Systems**, IEEE, v. 27, n. 8, p. 2349–2362, 2015.

JOHNSON, Richard. **Rsync Solutions: Definitive Reference for Developers and Engineers**. [S. l.]: HiTeX Press, 2025.

KOUGKAS, Anthony; DEVARAJAN, Hariharan; SUN, Xian-He. Hermes: A Heterogeneous-Aware Multi-Tiered Distributed I/O Buffering System. *In: PROCEEDINGS of HPDC '18*. [S. l.: s. n.], 2018. DOI: [10.1145/3208040.3208059](https://doi.org/10.1145/3208040.3208059). Available from: <https://dl.acm.org/doi/epdf/10.1145/3208040.3208059>.

LANDRÉ, Damien; PHILIPPE, Laurent; PIERSON, Jean-Marc. Seasonal study of user demand and IT system usage in datacenters. *In: 2024 IEEE 30th International Conference on Parallel and Distributed Systems (ICPADS)*. [S. l.: s. n.], 2024. p. 693–702. DOI: [10.1109/ICPADS63350.2024.00095](https://doi.org/10.1109/ICPADS63350.2024.00095).

MARLER, R. T.; ARORA, J. S. Survey of multi-objective optimization methods for engineering. **Structural and Multidisciplinary Optimization**, v. 26, n. 6, p. 369–395, Apr. 2004. ISSN 1615-1488. DOI: [10.1007/s00158-003-0368-6](https://doi.org/10.1007/s00158-003-0368-6). Available from: <https://doi.org/10.1007/s00158-003-0368-6>.

MELL, Peter; GRANCE, Timothy. The NIST definition of cloud computing. **Special Publication, National Institute of Science and Technology**, SP 800-145, n. 2011, 2011.

MENG, Xiangling *et al.* A Review of Parallel Computing for Large-scale Reservoir Numerical Simulation. **Archives of Computational Methods in Engineering**, v. 32, n. 7, p. 4125–4162, Oct. 2025. ISSN 1886-1784. DOI: [10.1007/s11831-025-10263-2](https://doi.org/10.1007/s11831-025-10263-2). Available from: <https://doi.org/10.1007/s11831-025-10263-2>.

MICROSOFT AZURE. **Microsoft Azure Cloud Computing Platform**. [S. l.: s. n.], 2026. Accessed: August, 2026. Available from: <https://azure.microsoft.com>.

MOCKUS, Jonas. **Bayesian Approach to Global Optimization: Theory and Applications**. 1. ed. Dordrecht: Kluwer Academic Publishers, 1989. p. 270. (Mathematics and its Applications). ISBN 978-94-010-6898-7. DOI: [10.1007/978-94-009-0909-0](https://doi.org/10.1007/978-94-009-0909-0).

MONGE, David A. *et al.* CMI: An online multi-objective genetic autoscaler for scientific and engineering workflows in cloud infrastructures with unreliable virtual machines. **Journal of Network and Computer Applications**, v. 149, p. 102464, 2020. ISSN 1084-8045. DOI: <https://doi.org/10.1016/j.jnca.2019.102464>. Available from: <https://www.sciencedirect.com/science/article/pii/S1084804519303248>.

NUNES, Alan L. *et al.* A Framework for Executing Long Simulation Jobs Cheaply in the Cloud. *In: 2024 IEEE International Conference on Cloud Engineering (IC2E)*. [S. l.: s. n.], 2024. p. 233–244. DOI: [10.1109/IC2E61754.2024.00033](https://doi.org/10.1109/IC2E61754.2024.00033).

RAPPA, M. A. The utility business model and the future of computing services. **IBM Systems Journal**, v. 43, n. 1, p. 32–42, 2004. DOI: [10.1147/sj.431.0032](https://doi.org/10.1147/sj.431.0032).

RED HAT. **What is Function-as-a-Service (FaaS)?** [S. l.: s. n.], 2020. Accessed: september, 2026. Available from: <https://www.redhat.com/topics/cloud-native-apps/what-is-faas>.

RED HAT. **What is IaaS?** [S. l.: s. n.], 2026. Accessed: september, 2026. Available from: <https://www.redhat.com/en/topics/cloud-computing/what-is-iaas>.

RED HAT. **What is PaaS?** [S. l.: s. n.], 2026. Accessed: september, 2026. Available from: <https://www.redhat.com/topics/cloud-computing/what-is-paas>.

RED HAT. **What is SaaS?** [S. l.: s. n.], 2022. Accessed: september, 2026. Available from: <https://www.redhat.com/en/topics/cloud-computing/what-is-saas>.

SCHEDMD LLC. **Slurm Workload Manager**. [S. l.: s. n.]. Accessed: September, 2026. Available from: <https://slurm.schedmd.com/overview.html>.

SOARES, Diego Machado da Conceição. **Choosing the Right Cloud Configuration for You and your Workload**. 2023. MA thesis – Universidade Federal Fluminense.

SODRÉ, Daniel M.B. *et al.* Spotting the Right Cloud Instances with Multiple AWS EC2 Fleets. *In: 2025 IEEE/SBC 37th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. [S. l.: s. n.], 2025. p. 69–79. DOI: [10.1109/SBAC-PAD66369.2025.00017](https://doi.org/10.1109/SBAC-PAD66369.2025.00017).

STATISTA. **"Big Three Hold Dominant Lead in Accelerating Cloud Market"**. [S. l.: s. n.], 2026. <https://www.statista.com/chart/18819/worldwide-market->

[share-of-leading-cloud-infrastructure-service-providers/](#). Accessed June 2026.

THE OPENBSD PROJECT. **OpenSSH scp (Version 9.8)**. [S. l.]. Available from: <https://openssh.com>.

THE SAMBA TEAM. **rsync(1) — Remote Sync**. [S. l.]. Available from: <https://linux.die.net/man/1/rsync>. Visited on: 27 Sept. 2026.

TOPCUOGLU, H.; HARIRI, S.; WU, Min-You. Performance-effective and low-complexity task scheduling for heterogeneous computing. **IEEE Transactions on Parallel and Distributed Systems**, v. 13, n. 3, p. 260–274, 2002. DOI: [10.1109/71.993206](https://doi.org/10.1109/71.993206).

YADWADKAR, Neeraja J. *et al.* Selecting the Best VM across Multiple Public Clouds: A Data-Driven Performance Modeling Approach. *In: PROCEEDINGS of the 2017 Symposium on Cloud Computing*. Santa Clara, California: Association for Computing Machinery, 2017. (SoCC '17), p. 452–465. ISBN 9781450350280. DOI: [10.1145/3127479.3131614](https://doi.org/10.1145/3127479.3131614). Available from: <https://doi.org/10.1145/3127479.3131614>.

YOO, Andy B; JETTE, Morris A; GRONDONA, Mark. Slurm: Simple Linux utility for resource management. *In: SPRINGER. WORKSHOP on job scheduling strategies for parallel processing*. [S. l.: s. n.], 2003. p. 44–60.

YOON, Hobin *et al.* Mutant: Balancing Storage Cost and Latency in LSM-Tree Data Stores. *In: PROCEEDINGS of SoCC '18*. [S. l.: s. n.], 2018. DOI: [10.1145/3267809.3267846](https://doi.org/10.1145/3267809.3267846). Available from: <https://doi.org/10.1145/3267809.3267846>.

ZHANG, Zhihao *et al.* {SkySync}: Accelerating File Synchronization with Collaborative Delta Generation. *In: 24TH USENIX Conference on File and Storage Technologies (FAST 26)*. [S. l.: s. n.], 2026. p. 461–476.

ZHENG, Huihuo *et al.* HDF5 Cache VOL: Efficient and Scalable Parallel I/O through Caching Data on Node-local Storage. *In: PROCEEDINGS of IEEE/ACM CCGrid*. [S. l.: s. n.], 2022. DOI: [10.1109/CCGrid54584.2022.00015](https://doi.org/10.1109/CCGrid54584.2022.00015). Available from: <https://doi.org/10.1109/CCGrid54584.2022.00015>.