


Modular and Reproducible Simulator Architecture for Composable Cloud Systems

Rubén Luque 


Department of Informatics
University of Oviedo
Gijón, Spain

e-mail: luqueruben@uniovi.es

José Luis Díaz 


Department of Informatics
University of Oviedo
Gijón, Spain

e-mail: jldiaz@uniovi.es

Joaquín Entrialgo 

Department of Informatic Systems
Polytechnical University of Madrid
Madrid, Spain

e-mail: j.entalgo@upm.es

Rubén Usamentiaga 

Department of Informatics
University of Oviedo
Gijón, Spain

e-mail: rusamentiaga@uniovi.es

Abstract—Simulating modern cloud systems requires tools that balance precision, extensibility, and reproducibility. Existing simulators often target specific use cases or rely on monolithic designs, which hinder the integration of alternative models for workload generation, resource allocation, or cost estimation. We present a modular and reproducible architecture for a cloud simulation framework, implemented in a functional prototype, and designed to support composable experimentation through a plugin-based approach. Simulation scenarios are defined declaratively, specifying interchangeable components, such as allocators, load balancers, workload injectors, and cost models. This architecture enables the systematic exploration and evaluation of diverse cloud management strategies, offering full support for event traceability, component reuse, and seamless integration into scientific workflows.

Keywords—Cloud simulation; Discrete-event simulation; Reproducible research; Workload modeling; Plugin-based architecture.

I. INTRODUCTION

Cloud computing has become the dominant paradigm for deploying scalable and elastic services. However, the growing heterogeneity of modern infrastructures, including container orchestration platforms, serverless computing, and hybrid cloud-edge deployments, introduces new challenges for modeling and evaluating such systems in a systematic and repeatable manner. In this context, discrete-event simulation remains a fundamental tool for studying resource allocation policies, autoscaling strategies, load balancing mechanisms, and cost evaluation models.

This paper introduces a declarative, plugin-oriented architecture for cloud simulation and evaluates it using *Nuberu*, an internal prototype that embodies the proposed design.

The main contributions of this paper are:

- The design of a modular and reproducible simulation architecture based on dynamic plugin discovery and decoupled component integration.
- An extensible plugin system that supports declarative simulation configuration through YAML (Yet Another Markup Language) files and static interface validation through Python Protocols.

- A practical validation scenario that demonstrates the framework's support for traceability, component reuse, and reproducibility, and showcases its applicability across diverse runtime configurations.

The remainder of this paper is organized as follows: Section II reviews related simulation frameworks; Section III presents the simulator architecture; Section IV details the plugin system and extensibility model; Section V provides a simple yet comprehensive use case to validate the architectural design; and Section VI concludes the paper, summarizing key findings and outlining directions for future work.

II. RELATED WORK

Simulation has long been a fundamental tool for evaluating cloud infrastructures, as real-world experimentation is often prohibitively expensive, time-consuming, and difficult to reproduce. Numerous simulation frameworks have been developed to support the study of cloud systems, each focusing on specific aspects, such as resource provisioning, scheduling policies, or cost modeling.

CloudSim [1] is one of the most established simulators, providing a general-purpose Java framework for modeling datacenters, Virtual Machines (VMs), and application workloads. Despite its configurability, *CloudSim* lacks a plugin architecture, is tightly coupled to Java workflows, and requires code modification to explore alternative policies, which limits its adaptability and reproducibility.

SimGrid [2] is another mature toolkit for modeling large-scale distributed systems, supporting diverse paradigms, such as High Performance Computing (HPC) and Grid computing. While it enables precise modeling of network and computing resources and has been widely adopted in the systems research community, its focus is broader than cloud infrastructures, and its extensibility relies on low-level Application Programming Interfaces (APIs) rather than composable modules.

Beyond these foundational tools, several recent surveys [3]–[7] systematically review cloud simulation frameworks, identifying common limitations and areas for future research. Mansouri et al. [3] evaluated 33 simulators and concluded

that no single tool covers all required dimensions, calling for improvements in Mobile Cloud Computing (MCC) [3][8], federated environments [9], and emerging paradigms, such as edge, fog, and Internet Of Things (IoT) [10][11]. Other studies [4][5] stress the lack of integrated support for security, dynamic behavior, or complex task prioritization, and emphasize the need for reproducibility, flexibility, and modularity in future frameworks.

More recently, several simulators written in Python have gained attention for their accessibility and extensibility. *Yet Another Fog Simulator (YAFS)* [10] simulates microservice deployments over user-defined network topologies, using the SimPy engine, and supports modular control over service placement and routing policies. Although it exhibits high flexibility for customizing placement, routing, and scheduling strategies, allowing dynamic scenario definition via class extension and functions integration, it lacks an explicit plugin system for external and decoupled integration of new core components. As a result, adding new functionality in YAFS often requires more intrusive modifications to the core codebase. *Cloudy* [12], by contrast, introduces a hybrid discrete-time and event-driven simulator with native Graphics Processing Unit (GPU) support and integration plans for optimization and machine learning (ML) libraries. However, its extensibility depends on manual template duplication, and it lacks a unified declarative configuration system. Finally, *ECLYPSE* [13], a preprint that has not undergone peer review, focuses on simulating composable cloud architectures with an emphasis on reproducibility. Its extensibility is achieved through a highly modular architecture that leverages object-oriented design principles, such as inheritance, and Python's dynamic capabilities, such as decorators, rather than relying on an explicit, separate plugin ecosystem.

These Python-based initiatives highlight the community's growing interest in modern, flexible, and scriptable simulation platforms. However, to the best of our knowledge, none of them adopts a modular, plugin-based architecture as the one we propose for simulating cloud environments. This makes our approach a novel contribution to the field.

Table I summarizes and contrasts key features of representative simulators in the domain, highlighting their support for modularity, configuration mechanisms, extensibility, and reproducibility, along with their limitations and typical application areas. As shown, none of the existing solutions fully meets all desired characteristics, especially in terms of reproducibility and plugin support.

III. SYSTEM ARCHITECTURE

The proposed architecture targets cloud simulation and centers on a discrete-event simulation engine, a global event bus, and a set of pluggable components. This design minimizes coupling between simulation logic and system policies, allowing researchers to prototype, compare, and reproduce complex deployment strategies with minimal implementation effort.

Figure 1 shows a conceptual view of the architecture, structured in layers of abstraction. The uppermost layer corresponds

TABLE I. SIMULATORS COMPARISON

Simulator	Modularity	Declarative Config	Plugin Support	Reproducibility	Limitations	Use Cases
CloudSim	○	○	○	●	(a)	VM scheduling, provisioning
SimGrid	●	●	●	●	(b)	HPC, distributed simulation
YAFS	●	●	○	●	(c)	Fog, IoT strategy evaluation
Cloudy	●	○	○	●	(d)	Cloud + ML, GPU workloads
ECLYPSE	●	○	●	●	(e)	Edge-cloud prototyping
Nuberu (proto)	●	●	●	●	(f)	Reproducible workflows

Legend of symbols:

- Fully supported
- ◐ Partially supported
- Not supported

Limitation notes:

- (a) Rigid architecture, low modularity.
- (b) Low extensibility, low-level abstractions.
- (c) No plugin interface, core modification required.
- (d) No declarative config, manual extension required.
- (e) Tightly coupled modules, no plugin API.
- (f) See Section VI.

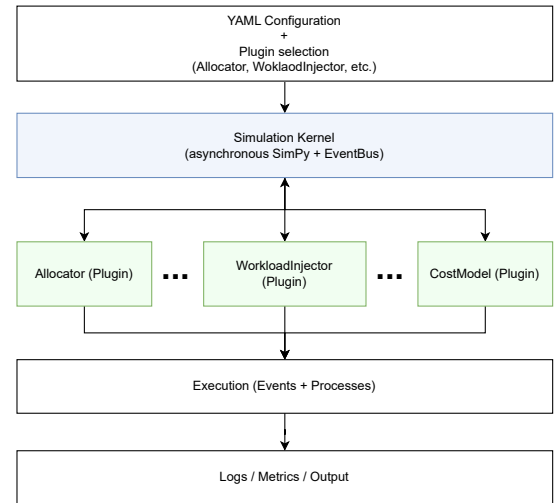


Figure 1. Conceptual high-level architecture

to the declarative experiment definition, where the simulation scenario and plugin selection are specified. The simulation kernel is responsible for orchestrating component instantiation and execution using asynchronous event-driven logic. Plugins encapsulate functional policies and interact only through the EventBus. The bottom layer collects structured outputs, thereby enabling traceability and reproducible analysis.

A. Core Simulation Engine

The simulation kernel follows a discrete-event model similar to SimPy library [14], but adopts Python's native `async/await` syntax instead of generator-based event handling. This design choice significantly improves the readability and maintainability of complex simulation flows, particularly those that involve multiple concurrent components, such as virtual machines, containers, and request dispatchers.

B. Component Model and Plugin Architecture

This architecture distinguishes between two user roles: developers, who create alternative implementations of pluggable components by writing plugins that conform to predefined interfaces, and analysts, who design simulation scenarios by selecting among available plugins without modifying the core system. In practice, a single user may assume both roles, developing custom components and designing simulation scenarios.

The simulation framework distinguishes between core components, which define the structure and control flow of the system, and pluggable components, which encapsulate specific, customizable behaviors. Core components include the discrete-event simulation engine, the communication primitives (e.g., event bus and channels) and essential modules, such as the workload injector, allocator, and infrastructure manager. These components are not pluggable themselves, but delegate critical functionality—such as workload characteristics, allocation strategy, or cost modeling—to user-defined plugins.

The mechanisms for dynamic discovery, registration, and static validation of these plugins are detailed in Section IV.

C. Event Bus and Inter-component Communication

Components communicate through a central event bus, implemented as a publish/subscribe mechanism over asynchronous message queues. Each event is categorized by a predefined topic (e.g., `VM_STARTED`, `REQUEST_COMPLETED`) and includes metadata, such as simulation time, origin, and payload. This decoupled communication model ensures that components remain independent and composable, facilitating experimentation and instrumentation without introducing tight coupling or global state dependencies.

The event bus is not limited to simulation components: additional observers (e.g., loggers, metric collectors, or debugging tools) can be implemented and subscribed to relevant event topics at runtime without modifying existing logic.

D. Simulation Configuration

The simulation runtime is configured declaratively via a YAML specifying parameters such as the simulation duration, the names of the plugins to be loaded for each functional component, and the input data, such as workloads, infrastructure specifications, performance data or allocation strategies. It can also define external data sources, such as workload traces in custom formats, to be parsed and injected at runtime by compatible plugins. This enables integration with external tools, such as cost optimizers, whose solutions can be imported through the appropriate plugin.

The architecture follows a *microkernel-inspired* design, in which the simulation engine acts as a lightweight orchestrator. Pluggable components are dynamically instantiated, operate in isolated asynchronous processes, and communicate exclusively through event-based interactions. This design allows for flexible composability and simplifies the development and integration of experiment-specific logic without entangling it with the simulation kernel.

IV. PLUGIN SYSTEM AND EXTENSIBILITY

A. Plugin Discovery and Registration

The architecture uses the `pluggy` library [15] to support dynamic plugin discovery using Python's `entry_points` mechanism. Each plugin is an installable python package which registers itself in the `pyproject.toml` file under a specific namespace (e.g., `application_model.llm`, `cost_model.default`), which enables the simulation framework to identify the type and logical name of each component. Once installed in the Python environment, plugins are automatically discovered at runtime without requiring any additional code modification.

Multiple plugins of the same type can be installed and selected declaratively through the YAML configuration file.

If a plugin declared in the YAML configuration cannot be found or does not conform to the expected interface, the simulation engine is designed to abort execution and issue a descriptive error. This validation occurs at startup time, before any event execution, ensuring that core simulation behavior remains consistent and reproducible, even when user-defined extensions are used in the configuration. Non-essential third-party plugins, such as auxiliary observers or loggers, may fail gracefully with a warning, allowing the simulation to proceed when their absence does not compromise correctness.

B. Interface Contracts via Protocols

Each pluggable component in the architecture adheres to two complementary interface mechanisms. First, a *hook specification* (`hookspec`) is defined using `pluggy`, which declares the methods that a plugin must implement to be properly registered and invoked at runtime. Second a Python Protocol interface is used for each plugin type, enabling static type checking and improved developer experience. These protocols specify the required methods (e.g., `get_workloads()`, `apply_allocation()`, `compute_cost()`) and allow for static verification using tools, such as `mypy`.

This dual-layer interface ensures runtime compatibility via `pluggy`, while also providing static guarantees, editor support, and better documentation through `Protocol`. Together, these mechanisms improve reliability, reduce integration errors, and facilitate the rapid development of new components.

V. EXPERIMENTAL VALIDATION: COMPARING OPTIMIZED ALLOCATIONS WITH SIMULATED BEHAVIOR

To demonstrate how the proposed architecture supports rigorous, scenario-driven evaluation, we present a case study executed with *Nuberu*, a prototype that instantiates our design. The goal is to show how the framework can expose hidden assumptions in external decision-making tools, such as mathematical optimizers, and thus guide their refinement.

Optimizers based on mathematical models, such as linear programming, often rely on idealized assumptions about workload, resource performance, and system behavior. This section investigates to what extent such optimized allocations remain effective when deployed in a more realistic simulated

TABLE II. PERFORMANCE IN REQUEST PER SECOND (RPS) OF EACH CONTAINER CLASS ON EVERY VM INSTANCE CLASS

C. Class VM I. Class	cc0app0	cc0app1	cc1app0	cc1app1	cc2app0	cc2app1
c5.2xlarge	2.10	0.46	4.30	0.96	6.35	1.63
c5.large	2.10	0.46	4.30	0.96	6.35	1.63
c5.xlarge	2.10	0.46	4.30	0.96	6.35	1.63
c6i.2xlarge	2.29	0.50	4.71	1.02	6.82	1.76
c6i.large	2.29	0.50	4.71	1.02	6.82	1.76
c6i.xlarge	2.29	0.50	4.71	1.02	6.82	1.76

environment. By simulating the deployment plan produced by the optimizer under multiple runtime conditions, we aim to identify discrepancies, stress points, and potential modeling oversights. This not only validates the practical viability of the computed solution but also highlights the role of simulation as a complementary tool for refining optimization strategies.

A. Description of the scenario to simulate

The scenario to be simulated is the output of an optimizer, *Conlloovia* [16], that solves a linear programming problem to allocate container replicas on VMs to minimize cost while ensuring the throughput of each application reaches or exceeds its 95th percentile over the forecast load trace. Inputs include:

- VM instance classes (including cost, cores and memory),
- container classes (defining CPU/memory requirements),
- and throughput performance matrices for each container class/VM instance class pair (see Table II).

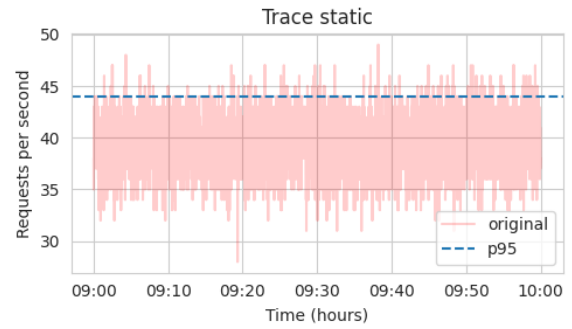
As an example, we analyze a one-hour segment from one of the scenarios presented in Section 5.4 of [16]. It involves two deployed applications, app0 and app1, each with a one-hour request trace that exhibits different dynamics: app0 maintains a stable average load of 39 rps with a 95th-percentile (p95) of 44 rps, whereas app1 shows a variable load whose average rate changes over time, with a p95 of 117 rps (see Figure 2). The optimizer's solution deploys 38 VMs across three instance types and 126 container replicas from three classes (one for app0 and two for app1), as depicted in Figure 3.

Using a custom plugin, the simulator can read this allocation directly from the files generated by *Conlloovia* and use it to: bootstrap the VMs, start the containers, inject traffic (using a user selected mode), and route requests through a configurable load balancer. Each container simulates service time based on the performance data, and metrics are collected throughout.

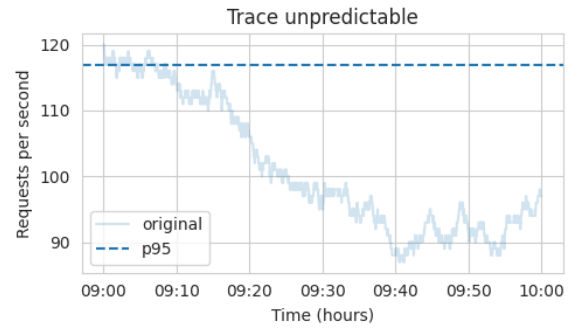
B. Experimental design

To assess the flexibility and analytical power of the simulator, we simulate the same scenario under 16 configurations combining four binary dimensions

- 1) **Load injection (Load)**: either from the original trace (replaying realistic variability) or as a synthetic Poisson process which ensures the same p95 throughput value used by the optimizer.



(a) Workload for app0



(b) Workload for app1

Figure 2. Plot of the workloads for each application

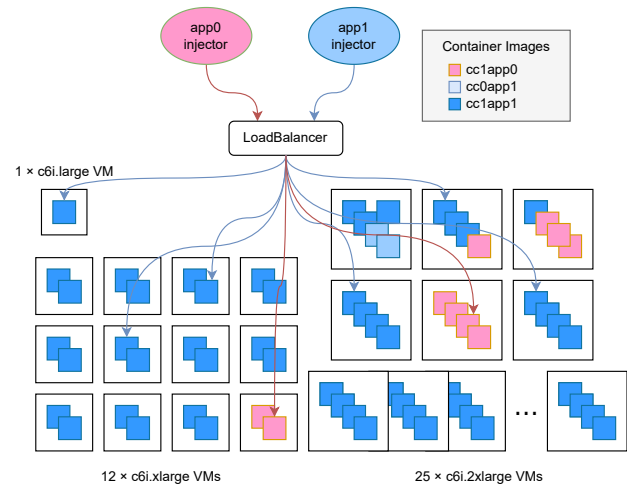


Figure 3. Scenario to simulate

- 2) **Load Balancing (LB)**: either a simple Round-Robin (RR) or a Smooth Weighted Round-Robin (SWRR), as the one used in nginx [17], which takes into account the performance differences between containers to assign appropriate weights.
- 3) **Queuing model (Q)**: either none (requests are dropped if busy) or bounded queues of size 1000 per container.
- 4) **Termination policy (Term)**: either 'hard' (containers are terminated immediately) or 'drain' (containers are kept alive to complete queued requests).

TABLE III. SUMMARY OF KEY METRICS (SUCCESS RATE AND TOTAL COST) FOR THE 16 SIMULATION SCENARIOS.

Q	Term	LB	Load	app0	app1	cost
0	drain	RR	poisson	82.6%	95.0%	\$10.66
			trace	100.0%	98.5%	\$10.63
		SWRR	poisson	82.6%	94.8%	\$10.66
			trace	100.0%	94.4%	\$10.63
	hard	RR	poisson	82.6%	94.9%	\$10.62
		SWRR	trace	100.0%	98.5%	\$10.62
1000	drain	RR	poisson	100.0%	99.8%	\$16.56
			trace	100.0%	99.8%	\$16.53
		SWRR	poisson	100.0%	100.0%	\$10.66
			trace	100.0%	100.0%	\$10.63
	hard	RR	poisson	100.0%	99.2%	\$10.62
		SWRR	trace	100.0%	100.0%	\$10.62

This design allows us to evaluate how an optimized deployment responds under diverse execution settings and policies.

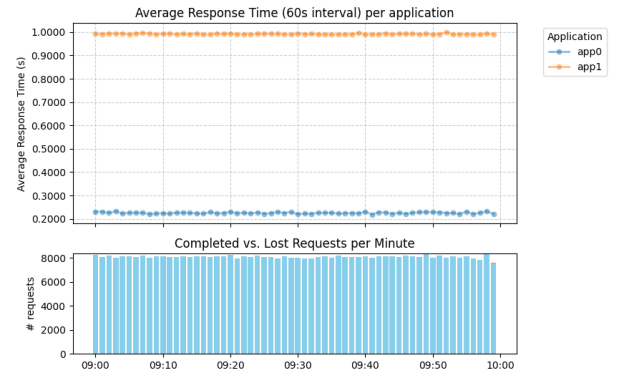
Each of the 16 simulations is defined through a YAML file that declares the scenario parameters, input data sources (e.g., system specification and optimal allocation), and the plugin components responsible for parsing external formats, such as Conlloovia. All experiment definitions, input traces and simulation results used to create the tables and figures in this paper are available in a public repository [18].

C. Discussion

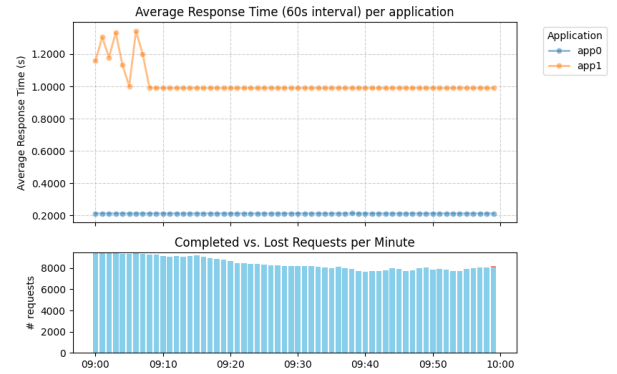
Table III summarizes two key metrics obtained from the 16 simulated scenarios: the percentage of completed requests and the total simulated cost. The results provide a compact overview of how different combinations of runtime parameters affect system performance. Configurations that include queueing, and SWRR load balancing consistently deliver the highest completion rates. By contrast, in scenarios with no queues, only the ones which use the actual traces achieve high completion rates. Poisson arrivals degrade performance, because the optimal solution generated by Conlloovia relies on very high container utilization, which in turn presupposes perfectly synchronized request arrivals.

Queues absorb demand spikes and improve request completion, though at the cost of higher response times. The drain policy avoids loss of in-flight or queued requests but prolongs VM usage and increases cost.

Interestingly, the simulated costs match exactly the optimizer's predictions in all scenarios using hard termination, since containers are shut down precisely as scheduled. However, in scenarios with drain termination, VMs remain active longer to complete pending requests, resulting in slightly higher costs. The RR scheduler results in the highest cost because it ignores container performance, leading to long queues



(a) Synthetic workload with Poisson arrivals



(b) Trace based workload

Figure 4. Response time and number of requests completed for the scenarios with SWRR balancing, large queues and ‘hard’ termination (last two rows of Table III)

of pending requests in the slower containers. These take longer to drain at the end of the simulation, thereby increasing the cost. SWRR balancing proves superior in these scenarios by distributing the load more proportionally across containers with heterogeneous performance, resulting in shorter queues.

Figure 4 shows the evolution of average response times and request completion rates for the SWRR load balancer under a ‘hard’ termination policy. Subfigure 4a corresponds to a synthetic workload generated as a Poisson arrival process with $\lambda = 34.563$ rps for app0 and $\lambda = 100.718$ rps for app1, ensuring a p95 of 44 rps and 117 rps, respectively, matching the throughput guaranteed by Conlloovia’s solution. Subfigure 4b uses a trace-based workload from [16], where the number of requests per second varies over time and is read from Comma Separated Values (CSV) files. In this case, the request rate can be at times above the p95 throughput expected by the solver. This is most noticeable for app1, which experiences pressure during the initial minutes, resulting in increased response times. In contrast, app0 remains stable throughout, even during short periods when its demand exceeds the p95 threshold.

Together, these results confirm the value of simulation not just for performance validation but as a diagnostic tool to uncover modeling assumptions that may not hold under

realistic or adverse conditions.

VI. CONCLUSION AND FUTURE WORK

This paper has introduced a modular, extensible architecture for cloud simulation frameworks that is explicitly designed to support reproducible and composable experimentation. Based on decoupled components, dynamic plugin discovery, and declarative configuration, the design enables researchers to prototype and compare alternative models for workload generation, resource allocation, and cost evaluation without modifying the simulation core.

By capturing the experimental setup in version-controlled configuration files and generating structured simulation traces, the proposed architecture aligns with the FAIR principles, Findable, Accessible, Interoperable, and Reusable [19]. This foundation enables both local reproducibility and broader community validation of alternative orchestration strategies.

The framework is under active development, with future releases providing curated plugins and scenarios. This work serves as a foundation for reproducible and extensible cloud simulation. Although the current prototype does not yet simulate network communication, I/O operations, or energy consumption, and no validation against real cloud deployments has been performed, it can already handle hundreds of VMs and thousands of requests with acceptable overhead. A complete evaluation of scalability and runtime efficiency is planned as part of future work. Upcoming extensions will enable more complex simulation scenarios. Firstly, we plan to incorporate models for network and I/O operations to support richer and more realistic simulations. Secondly, we will expand the plugin ecosystem with curated modules for common use cases, including auto-scalers, Large Language Model (LLM) serving patterns, spot-instance strategies, and multi-tenant execution. Finally, we will validate the architecture through large-scale comparative studies and evaluate its suitability for hybrid cloud-edge deployments.

ACKNOWLEDGMENT

This research was funded by the project PID2021-124383OB-I00 of the Spanish National Plan for Research, Development and Innovation from the Spanish Ministerio de Ciencia e Innovación.

REFERENCES

- [1] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. D. Rose, and R. Buyya, "Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, pp. 23–50, 2011. DOI: <https://doi.org/10.1002/spe.995>.
- [2] H. Casanova, A. Legrand, and M. Quinson, "SimGrid: A generic framework for large-scale distributed experiments," *Tenth International Conference on Computer Modeling and Simulation (uksim 2008)*, pp. 126–131, 2008. DOI: [10.1109/UKSIM.2008.28](https://doi.org/10.1109/UKSIM.2008.28).
- [3] N. Mansouri, R. Ghafari, and B. M. H. Zade, "Cloud Computing simulators: A comprehensive review," *Simulation Modelling Practice and Theory*, vol. 104, p. 102144, Nov. 2020, ISSN: 1569-190X. DOI: [10.1016/j.simpat.2020.102144](https://doi.org/10.1016/j.simpat.2020.102144).
- [4] I. Bambrik, "A survey on Cloud Computing simulation and modeling," *SN Computer Science*, vol. 1, no. 5, p. 249, Aug. 2020, ISSN: 2661-8907. DOI: [10.1007/s42979-020-00273-1](https://doi.org/10.1007/s42979-020-00273-1).
- [5] S. Lata and D. Singh, "Cloud simulation tools: A survey," *AIP Conference Proceedings*, vol. 2555, no. 1, p. 030003, Oct. 2022, ISSN: 0094-243X. DOI: [10.1063/5.0109181](https://doi.org/10.1063/5.0109181).
- [6] F. Fakhfakh, H. H. Kacem, and A. H. Kacem, "Simulation tools for cloud computing: A survey and comparative study," *2017 IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS)*, pp. 221–226, 2017. DOI: <https://doi.org/10.1109/ICIS.2017.7959997>.
- [7] S. V. Margariti, V. V. Dimakopoulos, and G. Tsoumanis, "Modeling and simulation tools for fog computing - a comprehensive survey from a cost perspective," *Future Internet*, vol. 12, p. 89, 2020. DOI: <https://doi.org/10.3390/fi12050089>.
- [8] M. Shiraz, A. Gani, R. H. Khokhar, and E. Ahmed, "An extendable simulation framework for modeling application processing potentials of smart mobile devices for mobile cloud computing," in *2012 10th International Conference on Frontiers of Information Technology*, Dec. 2012, pp. 331–336. DOI: [10.1109/FIT.2012.66](https://doi.org/10.1109/FIT.2012.66).
- [9] A. Núñez *et al.*, "iCanCloud: A flexible and scalable cloud infrastructure simulator," *Journal of Grid Computing*, vol. 10, no. 1, pp. 185–209, Mar. 2012, ISSN: 1572-9184. DOI: [10.1007/s10723-012-9208-5](https://doi.org/10.1007/s10723-012-9208-5).
- [10] I. Lera, C. Guerrero, and C. Juiz, "YAFS: A simulator for IoT scenarios in fog computing," *IEEE Access*, vol. 7, pp. 91745–91758, 2019. DOI: <https://doi.org/10.1109/ACCESS.2019.2927895>.
- [11] X. Zeng *et al.*, "IOTSim: A simulator for analysing IoT applications," *Journal of Systems Architecture, Design Automation for Embedded Ubiquitous Computing Systems*, vol. 72, pp. 93–107, Jan. 2017, ISSN: 1383-7621. DOI: [10.1016/j.sysarc.2016.06.008](https://doi.org/10.1016/j.sysarc.2016.06.008).
- [12] A. Siavashi and M. Momtazpour, "Cloudy: A Pythonic cloud simulator," *2024 32nd International Conference on Electrical Engineering (ICEE)*, pp. 1–5, 2024. DOI: <https://doi.org/10.1109/ICEE63041.2024.10667881>.
- [13] J. Massa *et al.*, "ECLYPSE: A Python framework for simulation and emulation of the cloud-edge continuum," *ArXiv*, vol. abs/2501.17126, pp. 1–16, 2025. DOI: <https://doi.org/10.48550/arXiv.2501.17126>.
- [14] SimPy Development Team, "Simpy: Discrete event simulation for python," Version 4.1.2, 2025, Available from: <https://simpy.readthedocs.io/> [retrieved: July, 2025].
- [15] pytest-dev, "Pluggy: A minimalist production ready plugin system," Version 1.5.0, 2025, Available from: <https://pluggy.readthedocs.io/> [retrieved: July, 2025].
- [16] J. Entrialgo, M. García, J. García, J. M. López, and J. L. Díaz, "Joint autoscaling of containers and virtual machines for cost optimization in container clusters," *Journal of Grid Computing*, vol. 22, pp. 1–24, 2024. DOI: <https://doi.org/10.1007/s10723-023-09732-4>.
- [17] M. Dounin, "Upstream: Smooth weighted round-robin balancing," Commit, nginx source tree, May 2012, Available from: <https://github.com/nginx/nginx/commit/27e94984486058d73157038f7950a0a36ecc6e35> [retrieved: July, 2025].
- [18] R. Luque, J. L. Díaz, J. Entrialgo, and R. Usamentiaga, "Nuberu simulation results – experiments repository," 2025, Available from: <https://github.com/asi-uniovi/nuberu-experiments-results> [retrieved: July, 2025].
- [19] M. D. Wilkinson *et al.*, "The FAIR guiding principles for scientific data management and stewardship," *Scientific Data*, vol. 3, p. 160018, 2016. DOI: [10.1038/sdata.2016.18](https://doi.org/10.1038/sdata.2016.18).