A Modular and Reproducible Framework Based on Plugin Composition for Simulating 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.entrialgo@upm.es

Rubén Usamentiaga
Department of Informatics
University of Oviedo
Gijón, Spain
e-mail: rusamentiaga@uniovi.es

***Abstract*—Simulating cloud systems requires reproducible execution models that capture latency dynamics and network contention. This paper presents a discrete-event simulation framework designed to enforce experimental reproducibility through a modular, plugin-based architecture. A key innovation is its execution engine, which replaces traditional callback-based simulation processes with asynchronous coroutines (async/await), enabling structured concurrency and simplified exception handling for complex control flows. We incorporate a communication layer using full-duplex channels with configurable directional latencies, allowing the explicit modeling of end-to-end round-trip times. Simulation scenarios are defined declaratively, treating experimental conditions as versioned data. We validate the framework through a case study on container lifecycle management, comparing timing-sensitive 'drain mode' termination policies under stochastic workloads. Results demonstrate that accounting for network delays and bounded queuing reveals significant discrepancies in compliance with service-level objectives compared to network-agnostic simulations, even when operational cost estimations remain similar. We also show that naive load balancing in cost-optimized heterogeneous deployments induces queue imbalance and increased tail latency, resulting in persistent latency objective violations despite sufficient aggregate capacity, thus revealing a blind spot in cost-driven optimizers.**

***Keywords*-Cloud simulation; Discrete-event simulation; Reproducible research; Plugin-based architecture; Network-aware modeling**

I. Introduction

The growing heterogeneity of cloud 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.

To address these limitations, we propose a modular simulation framework designed to enforce reproducibility by design. Unlike monolithic predecessors, our framework, internally named *Nuberu*, decouples the simulation kernel from the policy logic using a strict plugin system. Scenarios are defined declaratively, treating experimental conditions as versioned data rather than code. This approach allows researchers to systematically explore the impact of diverse architectural inputs, from resource allocators to network topologies, without modifying the underlying engine.

This article is an extended version of the work presented in [1], where the foundational principles of the declarative plugin architecture were introduced. In this extended manuscript, we expand the framework's capabilities and evaluation. Specifically, we introduce: (1) a simulation kernel based on Python's native asynchronous coroutines, replacing the traditional generator-based approach to improve the modeling of complex concurrent flows; (2) a communication layer based on full-duplex channels that accounts for network latency and protocol overheads; (3) an extended experimental evaluation that analyzes the impact of stochastic network delays and load balancing strategies on optimized resource allocations; and (4) the framework's full source code, now made publicly available to support reproducible research [2].

The remainder of this paper is organized as follows: Section II reviews related simulation frameworks. Section III presents the architectural foundations, focusing on the native async/await simulation kernel, the component model, and the duplex channel communication primitives; Section IV describes the plugin system, specifically the discovery mechanisms, the interface contracts via protocols and the development workflow. Section V validates the design by comparing termination policies and load balancing strategies under stochastic workloads. Section VI concludes the paper, summarizing key findings and outlining directions for future work.

II. State of the Art

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 [3] 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 [4] 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 [5]–[9] systematically review cloud simulation frameworks, identifying common limitations and areas for future research. Mansouri et al. [5] evaluated 33 simulators and concluded that no single tool covers all required dimensions, calling for improvements in Mobile Cloud Computing (MCC) [5][10], federated environments [11], and emerging paradigms, such as edge, fog, and Internet Of Things (IoT) [12][13]. Other studies [6][7] 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)* [12] 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* [14], 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. Similarly, *ECLYPSE* [15] focuses on composable cloud architectures but bypasses a formal plugin ecosystem entirely. Instead, it ties extensibility directly to Python's internal mechanisms, specifically inheritance and decorators. While this utilizes language features, it requires users to navigate the object hierarchy rather than interacting with decoupled extension interface.

Moreover, a significant limitation persists regarding experimental reproducibility. Existing frameworks typically adopt a *code-as-configuration* approach, where scenario definitions are embedded within imperative Python scripts or rely on unversioned external libraries. While YAFS supports JSON for topology definition, dynamic policies remain coupled to the codebase [12]. Similarly, ECLYPSE scenarios are constructed by programmatically instantiating configuration objects. This practice complicates compliance with FAIR (Findable, Accessible, Interoperable, and Reusable) principles [16], as the experimental setup is not an immutable artifact. In contrast, a declarative configuration model decouples the simulation kernel from the experiment definition, treating scenarios as static, auditable data.

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.

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.

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. ARCHITECTURAL FOUNDATIONS

The proposed framework targets cloud simulation and is based 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 presents the concrete architecture of the framework. Driven by a declarative YAML configuration, the system relies on a Simulation Kernel that integrates the `aSimPy` environment (a custom fork of the SimPy discrete-event engine adopting an `async/await`-based syntax. See Section III-A)

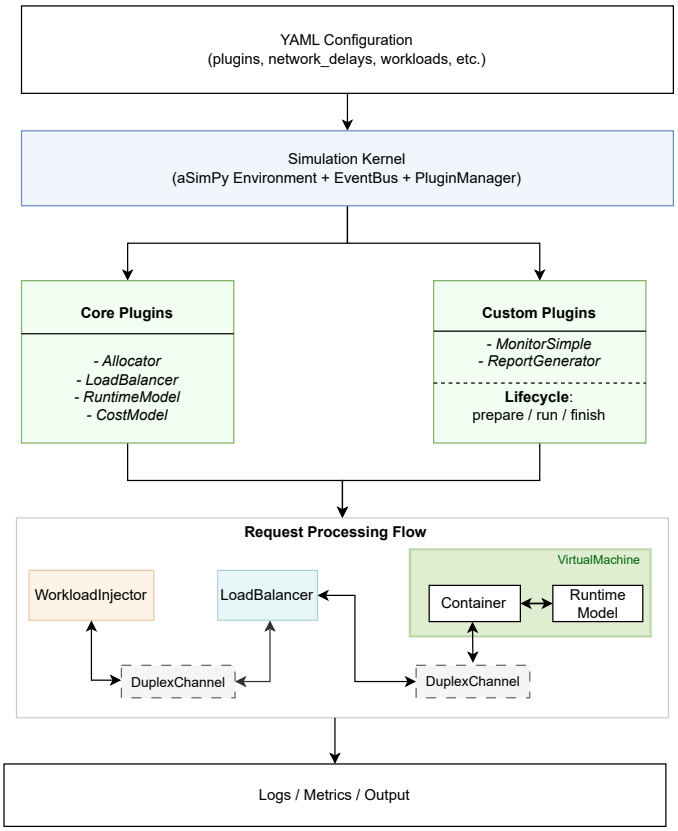


Figure 1. Detailed architecture of the Nuberu framework.

for asynchronous execution, the EventBus for decoupled control signaling, and the PluginManager for dynamic extension loading.

The diagram distinguishes between Core Plugins (which provide essential logic like allocation or load balancing) and Custom Plugins (focused on observability and lifecycle management). Furthermore, it illustrates the Request Processing Flow, visualizing how the Workload Injector, Load Balancer, and Virtual Machines interact through the DuplexChannel primitives to route traffic and generate simulation outputs.

A. Core Simulation Engine: Native Async/Await

The simulation kernel follows a discrete-event model built upon a custom fork of the `SimPy` library [17], referred to here as `aSimPy`. The fork replaces `SimPy`'s generator-based process definitions with Python's native async/await syntax, while preserving the original event scheduling semantics and execution model.

The primary motivation for this refactoring is to improve the expressiveness and clarity of simulation code. In scenarios involving multiple interacting components—such as virtual machines, containers, and request dispatchers—the async/await syntax enables a more linear and compositional representation of process logic than explicit yield-based suspension.

From a semantic perspective, `aSimPy` preserves `SimPy`'s execution model. Although simulation processes are expressed as coroutines, their execution remains governed by the underlying discrete-event scheduler. Suspension and resumption points correspond to simulated events rather than to concurrency primitives managed by Python's native asynchronous runtime.

B. Component Model and Plugin Architecture

This framework supports 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 and the allocator. 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. Communication Primitives

To achieve realistic modeling of distributed systems, the architecture moves beyond idealized method calls between objects. Instead, it employs two distinct communication patterns: a global Event Bus for global signals and Full-Duplex Channels for data plane interactions.

1) Event Bus and Inter-component Communication: Global system events, such as VM_STARTED or SIMULATION_FINISHED, are propagated via an asynchronous EventBus. This provides strict decoupling: auxiliary components, such as metric collectors, trace loggers or user custom plugins, can subscribe to REQUEST_COMPLETED events without the Workload Injector or Load Balancer being aware of their existence. This mechanism models an out-of-band communication mechanism, preventing observability instrumentation from introducing artificial blocking in the operational flow of requests.

2) Duplex Channels with Configurable Latency: Operational request-response interactions use explicit point-to-point *Duplex Channels*. The proposed framework models network propagation delays as configurable parameters. A duplex channel D connecting two components A and B is composed of two unidirectional buffered queues:

$$D_{A \leftrightarrow B} = \langle C_{\text{fwd}}(\delta_{\text{fwd}}), C_{\text{bwd}}(\delta_{\text{bwd}}) \rangle \quad (1)$$

where each channel component C represents a directional path, forward ($A \rightarrow B$) or backward ($B \rightarrow A$), and is configured with an independent transmission delay δ. This abstraction specifically supports the modeling of asymmetric network conditions (where $\delta_{\text{fwd}} \neq \delta_{\text{bwd}}$) and distinct

Listing 1: send() method in Channel

```
async def send(self, message) -> None:
  await self.env.timeout(self.delay) # Enforces configured
                                     # propagation delay
  await self.store.put(message)
```

infrastructure overheads per direction. The implementation enforces these delays asynchronously using the simulation environment's timeout primitives, as shown in Listing 1.

a) Formal Response Time Model: The total response time T_{response} for a request traversing the system is formally defined as the aggregation of network transit, queuing, and service time:

$$T_{\text{response}} = T_{\text{network}} + T_{\text{queue}} + T_{\text{service}} \quad (2)$$

where:

- $T_{\text{network}} = \delta_{wi \to lb} + \delta_{lb \to vm} + \delta_{vm \to lb} + \delta_{lb \to wi}$ represents the accumulated network round-trip time (RTT). Note that forward and backward delays are configured separately in the scenario.
- T_{queue} is the waiting time in the runtime model's buffer.
- T_{service} is the nominal service time provided by the performance model or benchmark.

T_{queue} is an *emergent variable* determined by the system's runtime state (congestion) and the queuing policy, whereas T_{network} is a configuration parameter specified declaratively. This distinction allows researchers to isolate the impact of network topology from resource contention.

Figure 2 illustrates the component interaction throughout the request lifecycle. Network delays (δ) translate into asynchronous timeouts during transmission phases, while service (T_{service}) and queuing (T_{queue}) times depend on the internal execution of the Runtime Model. The diagram confirms the symmetry of the communication path and the isolation of latency sources.

b) Hierarchical Latency Propagation: The operational lifecycle of a request is modeled through a two-tier hierarchy of duplex channels, representing distinct boundaries:

1) *Client–Load Balancer layer:* Bidirectional communication governed by the round-trip delay wi_to_lb.
2) *Load Balancer–Container layer:* Intra-datacenter communication governed by lb_to_vm_default.

To illustrate how these delays aggregate, Table II presents a breakdown of a single request's lifecycle. Each transition appends a timestamped entry to the request's timeline attribute (see Listing 2), creating an auditable trace that adheres to FAIR principles [16].

c) Declarative Configuration and Overrides: Delays are specified in the scenario YAML, enabling version-controlled experimentation. The framework supports granular overrides to simulate heterogeneous environments, such as multi-region deployments (Listing 3).

TABLE II. REQUEST LATENCY BREAKDOWN ACROSS CHANNEL HIERARCHY.

Event	Comp.	Time (ms)	δt	Delay Source
Req. generation	WI	0.0	—	—
LB rx request	LB	10.0	+10.0	`wi_to_lb` (fwd)
C rx request	C	15.0	+5.0	`lb_to_vm` (fwd)
RM complete	RM	25.0	+10.0	Service
LB rx response	LB	35.0	+5.0	`lb_to_vm` (bwd)
Client rx response	WI	40.0	+10.0	`wi_to_lb` (bwd)

Listing 2: RequestTimestamp structure for FAIR traceability

```
@dataclass
class RequestTimestamp:
    time: float           # Simulated time
    component_id: str     # e.g., "Container_c1"
    event: str            # e.g., "receive_from_lb"
    metadata: dict        # queue_length, vm_id, etc.
```

d) Advanced Channel Semantics: The duplex channel architecture incorporates two features essential for cloud orchestration:

Implicit Response Routing: By encapsulating the return path within the channel state, the architecture avoids complex correlation ID management. When the Load Balancer dispatches a request, the response inherently propagates backward through the same topological path. This simplifies component logic, as containers remain agnostic of the upstream topology.

Half-Close Semantics: The channels implement half-close semantics to support graceful shutdown. A component can close its output direction (signaling no further requests will be sent) while keeping the input direction open to process pending messages. This is the foundational mechanism for the 'drain' termination policy detailed in Section V.

D. Simulation Configuration

The simulation runtime is configured declaratively via a YAML file 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

Listing 3: Network configuration with heterogeneous overrides

```
network_delays:
  wi_to_lb: 10.0          # Default Client-LB latency
  lb_to_vm_default: 5.0   # Default intra-datacenter latency
  vm_container_overhead: 2.0  # Container overhead

# Overrides for specific topology scenarios
vm_network_configs:
  - name: "vm_remote_az"
    network_delay: 50.0  # Higher latency for cross-AZ VMs
```

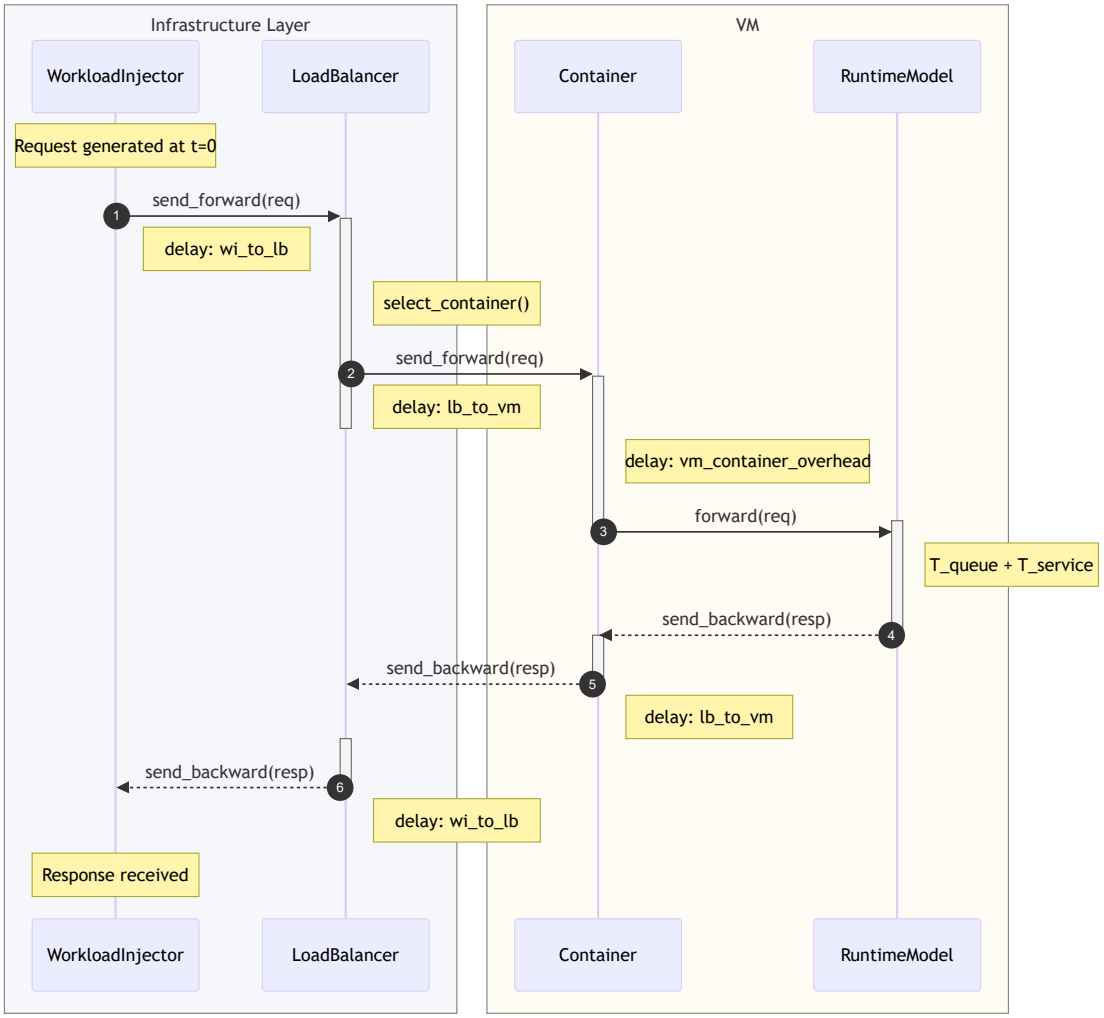


Figure 2. Sequence diagram of the request lifecycle showing asymmetric network delays, container overhead, and internal runtime states ($T_{\text{queue}}, T_{\text{service}}$).

at runtime by compatible plugins. An full example of such a configuration file is provided in Appendix A. This enables integration with external tools, such as cost optimizers, whose solutions can be imported through the appropriate plugin.

The framework 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 the primitives defined in Section III-C. This design allows for flexible composability and simplifies the development and integration of experiment-specific logic without entangling it with the simulation kernel.

1) Reproducibility through Declarative Specification: Experimental reproducibility is achieved by capturing the complete simulation specification as a version-controlled YAML file. Formally, each simulation run is parameterized by a configuration tuple:

$$\Theta = \langle \theta_{\text{kernel}}, \theta_{\text{plugins}}, \theta_{\text{inputs}}, \theta_{\text{net}} \rangle \tag{3}$$

where:

- θ_{kernel} specifies simulation duration and stop conditions.
- θ_{plugins} declares the mapping of functional roles to specific plugin implementations.
- θ_{inputs} specifies workload traces and infrastructure specifications.
- θ_{net} specifies the network latency parameters defined in Section III-C2.

By versioning Θ as a single artifact, experimental variations can be expressed as controlled perturbations $\Delta\Theta$, such as sweeping a latency parameter or switching a load-balancing policy, rather than as implicit code modifications. This design makes experiment-defining assumptions explicit and auditable, adhering to the FAIR principles [16].

IV. Plugin System and Extensibility

A central design principle of Nuberu is decoupled extensibility: users should be able to alter simulation behavior without modifying the core source code. The architecture leverages the `pluggy` library [18] for dynamic discovery and Python Protocols [19] for static interface enforcement.

A. Discovery and Validation Mechanism

The framework uses the `pluggy` library [18] 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 is generated, 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 framework 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.

Table III summarizes the primary protocols defined in the framework. Each protocol enforces a specific set of methods that ensure the component complies with the simulation kernel interfaces.

C. Plugin Taxonomy: Core vs. Custom

The framework unifies the extension mechanism using a strict *Factory Pattern* for all plugin types. Regardless of their category, all plugins expose a factory hook that the kernel invokes to instantiate the component. This design ensures that runtime dependencies (such as the `Environment` or `EventBus`) are injected only when the simulation is fully initialized, while allowing configuration parameters to be captured earlier from the YAML definition.

Listing 4: Example of a Core Plugin implementation using the Factory Pattern.

```
from nuberu.plugins import PluginBase, hookimpl

class RuntimeModelSimple:
    # Implementation details omitted for brevity
    ...

class RuntimeModelPlugin(PluginBase):
  plugin_name = "runtime_model_simple"

  @hookimpl
  def get_runtime_model_factory(self, conf: dict):
    # Capture config from YAML
    q_size = conf.get("queue_size", 1000)

    # Factory receives runtime dependencies
    def factory(env, container, channel):
      return RuntimeModelSimple(env, container, channel,
                                q_size, self.logger)
    return factory
```

While the instantiation pattern is shared, plugins are classified into two categories based on the nature of the object they return.

1) Core Plugins: These plugins provide essential implementations for system modules such as *Load Balancers*, *Allocators*, *Runtime Models* or *Workload Injectors*. The factory returns a functional component responsible for specific simulation decisions (e.g., routing a request).

Listing 4 presents a simplified implementation of a Runtime Model plugin. The factory captures the `queue_size` from the configuration closure and returns a callable that the kernel will later invoke with the necessary dependencies.

2) Custom Plugins: Custom plugins run concurrently with the simulation to perform auxiliary tasks, such as real-time monitoring or live visualization. Their factory returns an object that adheres to the `CustomPluginProtocol`, enforcing a specific lifecycle:

- `prepare()`: Invoked once before the simulation starts (e.g., to subscribe to events).
- `run()`: Invoked at $t = 0$ to spawn asynchronous background processes.
- `finish()`: Invoked after the simulation ends for cleanup.

D. Implementation Example

Listing 5 demonstrates a minimal Custom Plugin that monitors request completion events. Note that the `RequestMonitorPlugin` defines the factory hook, which instantiates the `RequestMonitor` runtime object. This separation allows the runtime object to safely interact with the `env` and `event_bus` injected by the factory.

E. Project Structure and Development Workflow

To support the extensibility model described above, the framework is organized as a *monorepo workspace*. This structure enforces a physical separation between the simulation

TABLE III. KEY PROTOCOL INTERFACES DEFINING THE EXTENSION POINTS OF THE FRAMEWORK.

Component Type	Protocol	Factory Method	Functional Role
Infrastructure	`InfrastructureSpecs`	get_infrastructure_factory()	Defines Apps, InstanceClasses, and ContainerClasses for the simulated environment.
Allocation	`AllocationSpecs`	get_allocations_factory()	Maps container replicas to VMs based on optimization results or heuristics.
Performance	`PerformanceSpecs`	get_rps_for_app_factory()	Computes RPS capacity for a given app on an instance class.
Load Balancer	`LoadBalancerSpecs`	get_load_balancer_factory()	Implements routing logic (Round-Robin, SWRR) to distribute requests.
Workload	`WorkloadSpecs`	get_distribution_params_for_app_factory()	Extracts traffic parameters (num_reqs, time_slot) from traces.
Distribution	`ArrivalDistributionSpecs`	get_interarrival_times_factory()	Generates arrival times based on statistical distributions (Poisson, etc.).
Runtime Model	`RuntimeModelSpecs`	get_runtime_model_factory()	Models service time, queuing logic, and execution behavior.
Cost Model	`CostModelSpecs`	get_cost_model_factory()	Calculates infrastructure costs based on VM usage.
Custom Plugin	`CustomPluginSpecs`	get_factory()	Provides auxiliary processes for monitoring, logging, or visualization.

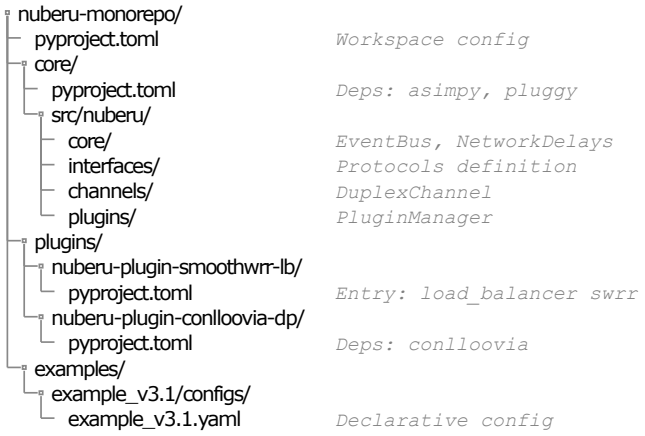


Figure 3. Directory structure of the monorepo workspace.

kernel and the extension ecosystem, mirroring the logical decoupling provided by the architecture.

The repository is structured into two primary directories, as illustrated in Figure 3:

- `core/`: Contains the simulation core built on top of the aSimPy discrete-event library, the definitions of Protocol interfaces, and the Event Bus implementation. It maintains minimal external dependencies to ensure a lightweight and stable core.
- `plugins/`: Hosts the extensions ecosystem. Each subdirectory correspond to an independent Python package with its own `pyproject.toml` configuration file.

This organization enables strict *dependency isolation*. For instance, a data-provider plugin that depends on computationally intensive libraries (e.g., `pandas` or `scikit-learn` for trace parsing and preprocessing) can declare these requirements locally, without propagating them to the core engine or to other lightweight plugins.

From a workflow perspective, the build system (based on `uv`) treats the repository as a unified workspace. This approach allows developers to install the core and a specific subset of plugins in editable mode, thereby supporting rapid prototyping and iterative development while preserving the stability of the simulation kernel.

V. EXPERIMENTAL VALIDATION: COMPARING OPTIMIZED ALLOCATIONS WITH SIMULATED BEHAVIOR

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 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.

To demonstrate how the proposed architecture supports rigorous, scenario-driven evaluation, we present a case study executed with *Nuberu*. The goal of this validation is twofold: first, to demonstrate the structural fidelity of the simulator by verifying that it produces physically consistent results under stress (e.g., network contention and queuing); and second, to show how these realistic simulations expose hidden assumptions in static optimization models, guiding their refinement.

A. Description of the scenario to simulate

The scenario is derived from the output of *Conlloovia* [20], an optimizer that allocates container replicas on VMs to minimize cost while ensuring the throughput expressed in rps (requests per seconds) of each application reaches its 95th percentile (p95). Inputs include VM instance classes, container classes, and performance matrices (see Table IV). We analyze a one-hour segment involving two applications: *app0* (stable load, p95=44 rps) and *app1* (variable load, p95=117 rps), as shown in Figure 4.

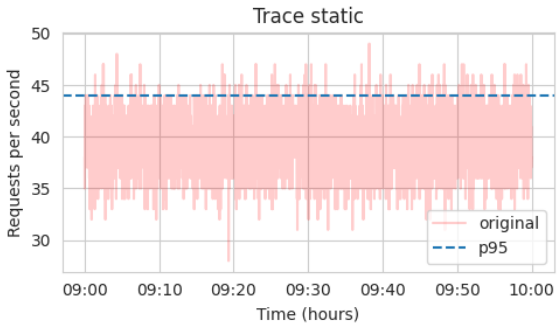
Listing 5: Implementation of a Custom Monitoring Plugin

```
from nuberu.plugins import PluginBase, hookimpl
from nuberu.core.events import EventTopic

class RequestMonitor:
    """Implementing␣CustomPluginProtocol."""
    def __init__(self, env, event_bus, logger):
        self.env = env
        self.event_bus = event_bus
        self.logger = logger
        self.queue = None

    def prepare(self) -> None:
        """Phase␣1:␣Subscribe␣to␣relevant␣topics."""
        self.queue = asimpy.Store(self.env)
        self.event_bus.subscribe(
            EventTopic.REQUEST_COMPLETED,
            self.queue)

    def run(self) -> None:
        """Phase␣2:␣Spawn␣concurrent␣collection␣loop."""
        self.env.process(self._monitor_loop())

    async def _monitor_loop(self) -> None:
        while True:
            event = await self.queue.get()
            self.logger.info(
            f"Req␣{event.payload['id']}␣done")

    def finish(self) -> None:
        """Phase␣3:␣Finalize␣artifacts."""
        self.logger.info("Sim␣finished.")

class RequestMonitorPlugin(PluginBase):
    """Plugin␣Entry␣Point."""
    plugin_name = "request_monitor"

    @hookimpl
    def get_factory(self, config: dict):
        # Factory injects dependencies
        def factory(env, event_bus):
            return RequestMonitor(
            env,
            event_bus,
            self.logger)
        return factory
```

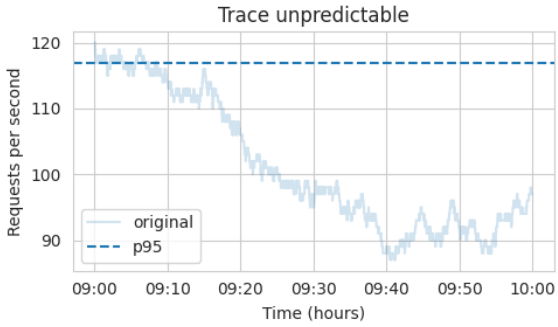
TABLE IV. PERFORMANCE IN REQUEST PER SECOND (RPS) OF EACH CONTAINER CLASS ON EVERY VM INSTANCE CLASS

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

It is important to clarify the relationship between the optimizer's constraints and the simulator's metrics. *Conlloovia*'s objective is to minimize cost subject to a throughput constraint: the allocated capacity must meet or exceed the 95th percentile of the forecasted load. While the optimizer does not explicitly model response times, latency Service Level Objectives (SLOs) are implicit in the input Performance Matrix. The maximum rps defined for each container class represent a benchmarking-derived limit, below which the container is known to satisfy its latency requirements.



(a) Workload for app0



(b) Workload for app1

Figure 4. Plot of the workloads for each application

In contrast, *Nuberu* simulates the full request lifecycle, including network hops and queuing delays, allowing us to measure the 99th percentile of Response Time (p99 RT) as an emergent output. This distinction is crucial: the experiments validate whether the optimizer's static assumption—that satisfying throughput constraints guarantees acceptable latency—holds under dynamic runtime conditions such as network contention or variability.

The deployment plan computed by *Conlloovia* is visualized in Figure 5. The solution involves 38 VMs hosting a total of 116 containers.

A critical detail in this deployment is the performance heterogeneity introduced by the cost-minimization objective. While the majority of App1 containers are of the high-performance class (*cc1app1*, dark blue, $\approx$ 1.02 rps), the optimizer allocated two instances of a lower-performance class (*cc0app1*, light blue, $\approx$ 0.50 rps). This decision is perfectly rational from a throughput perspective: these cheaper, slower containers provided exactly the marginal capacity needed to reach the target p95 load without paying for a full-sized instance. However, this creates a potential problem: while the aggregate throughput is sufficient, the presence of these slow nodes introduces a risk of head-of-line blocking if the load balancer is unaware of their limited capacity.

Using a custom plugin, *Nuberu* bootstraps the exact infrastructure proposed by Conllovia to inject traffic and collect run-

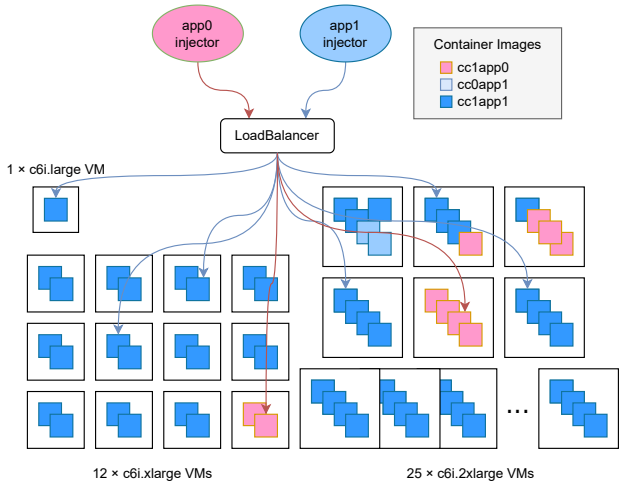


Figure 5. Optimized Deployment Plan. The solution mixes different container classes to minimize cost.

time metrics, allowing us to compare the theoretical "optimal" performance against simulated reality.

B. Experimental design

To assess the analytical power of the simulator, we expand the scenario into 32 configurations by combining five binary dimensions. This design allows us to isolate specific architectural factors:

1) **Load injection (Load)**: either the original *trace* (realistic variability) or a synthetic *Poisson* process matching the p95 throughput which is the target of the optimizer.
2) **Load Balancing (LB)**: either a naive Round-Robin (*RR*) or a Smooth Weighted Round-Robin (*SWRR*) [21], which accounts for container performance heterogeneity.
3) **Queuing model (Q)**: either *zero buffer* ($Q = 0$, requests dropped if busy) or *large queues* ($Q = 1000$).
4) **Termination policy (Term)**: Specifies the shutdown strategy, defined as either *hard* (immediate termination) or *drain* (graceful termination). Under the *drain* policy, containers continue processing until their request queues are empty, ensuring the underlying VMs remain active until all their containers are shut down.
5) **Network Latency (Lat)**: either *ideal* (0 ms) or *realistic* (20 ms per channel). In Nuberu's topology, a request traverses four network hops, as seen in Figure 2, namely: Client $\rightarrow$ LB $\rightarrow$ Container $\rightarrow$ LB $\rightarrow$ Client. Therefore, a 20 ms channel latency implies a theoretical round-trip time (RTT) floor of $4 \times 20 = 80$ ms.

All experiment definitions and results are available in the public repository [22].

C. Discussion

Table V summarizes three selected cases that illustrate the divergence between theoretical optimization models and runtime behavior. We analyze these results also to validate the structural fidelity of the simulator. The complete results for all 32 configurations are detailed in Table VI.

1) Baseline Validation: Intrinsic Time and Buffering: The first block of results (Case A) serves as a baseline validation, confirming that the simulator outcomes align with the optimizer's predictions under idealized conditions. In the absence of queues ($Q = 0$) and network delays ($Lat = 0$), the simulator isolates the raw computational performance of the allocated resources. The observed p99 response time of 0.98 s represents the *intrinsic processing time* (T_{service}) of the container mix chosen by *Conlloovia*.

However, this case with $Q = 0$ also reveals a critical limitation of static capacity planning. Despite sufficient aggregate capacity to handle the load volume, the completion rate drops to 94.8%. This discrepancy occurs because the optimizer assumes a uniform arrival rate, whereas the simulator models the actual stochastic nature of traffic (Poisson arrivals). In a zero-buffer system ($Q = 0$), this stochasticity produces momentary micro-bursts, where inter-arrival times momentarily drop below service times, causing immediate request drops due to the lack of a backlog to absorb the variance.

When adequate buffering is introduced ($Q = 1000$), the completion rate rises to 100.0%. This validates that the allocated capacity was indeed sufficient, provided that the infrastructure includes queues to smooth out arrival jitter. The cost of this stability is a slight increase in tail latency (from 0.98 s to 1.12 s) due to queuing time.

2) Network Overhead: Case B introduces the network topology constraints. Comparing the ideal baseline ($L = 0$) with the network-aware simulation ($L = 20$ ms), the p99 response time degrades from 1.45 s to 1.53 s.

The introduction of the network produces two distinct effects on the Cumulative Distribution Function (CDF):

1) **Latency Floor:** The curve shifts to the right. The minimum response time increases by exactly 80 ms, as shown Figure 6, validating the correct modeling of the 4-hop topology (4×20 ms RTT).
2) **Tail Expansion:** In some configurations (e.g., the top right four configurations for $Q = 1000$ in Table VI) the gap widens significantly at the 99th percentile. This confirms that network latency is not merely additive; it increases the "time-in-system" for each request, indirectly increasing queue occupancy and further inflating wait times.

3) The heterogeneity problem: Policy Efficiency: The most dramatic finding appears in Case C, where the p99 response time explodes to near 990 seconds under the Simple Round-Robin (RR) policy. This failure is a direct consequence of the cost-optimization performed by *Conlloovia*. As noted in Section V, the optimizer minimized costs by mixing high-performance containers with a few low-performance instances (0.5 rps) to meet the marginal demand. While mathematically optimal, this heterogeneity creates a trap for naive load balancers. The RR policy blindly directs traffic to these slow containers, creating massive Head-of-Line blocking. Figure 7 illustrates the magnitude of this collapse using a logarithmic scale. The *Time-To-Drain* (TTD) metric reveals that after the workload ends, the infrastructure must remain active for

TABLE V. **SELECTED SIMULATION CASES.** COMPARISON OF THEORETICAL BASELINE VS. REALISTIC CONSTRAINTS. *(App1 results shown)*

		Configuration			App1 Outcomes	
Case Type	**Workload**	**Buffer (Q)**	**Lat (L)**	**LB**	**% Completed**	**p99 RT (s)**
A. Ideal Baseline	Poisson	0	0 ms	SWRR	94.8%	0.98
(Validation)	Poisson	1000	0 ms	SWRR	**100.0%**	1.12
B. Network Impact	Trace	1000	0 ms	SWRR	100.0%	1.45
	Trace	1000	20 ms	SWRR	100.0%	1.53
C. Policy Saturation	Trace	1000	20 ms	RR	99.8%	**989.77**

TABLE VI. COMPLETE EXPERIMENTAL RESULTS FOR ALL 32 SIMULATION SCENARIOS (SPLIT BY QUEUE CAPACITY).

Scenarios with Zero Buffer / No Queuing ($Q = 0$)

				Completed		RT (p99)		Cost
				app0	app1	app0	app1	
Term	LB	Load	Lat					
drain	RR	poisson	0	82.6%	95.0%	0.212	0.980	$10.63
			20	82.4%	94.9%	0.293	1.061	$10.63
		trace	0	100.0%	98.5%	0.212	0.980	$10.63
			20	99.7%	98.5%	0.293	1.061	$10.63
	SWRR	poisson	0	82.6%	94.8%	0.212	0.980	$10.63
			20	82.4%	94.7%	0.293	1.061	$10.63
		trace	0	100.0%	94.4%	0.212	0.980	$10.63
			20	99.7%	94.3%	0.293	1.061	$10.63
hard	RR	poisson	0	82.6%	94.9%	0.212	0.980	$10.62
			20	82.4%	94.9%	0.293	1.061	$10.62
		trace	0	100.0%	98.5%	0.212	0.980	$10.62
			20	99.7%	98.5%	0.293	1.061	$10.62
	SWRR	poisson	0	82.6%	94.8%	0.212	0.980	$10.62
			20	82.4%	94.7%	0.293	1.061	$10.62
		trace	0	100.0%	94.3%	0.212	0.980	$10.62
			20	99.7%	94.3%	0.293	1.061	$10.62

Scenarios with Large Queues ($Q = 1000$)

				Completed		RT (p99)		Cost
				app0	app1	app0	app1	
Term	LB	Load	Lat					
drain	RR	poisson	0	100.0%	99.8%	0.355	842.870	$10.82
			20	100.0%	99.8%	0.436	852.609	$10.82
		trace	0	100.0%	99.8%	0.212	977.790	$10.82
			20	100.0%	99.8%	0.293	989.771	$10.82
	SWRR	poisson	0	100.0%	100.0%	0.355	1.118	$10.63
			20	100.0%	100.0%	0.436	1.200	$10.63
		trace	0	100.0%	100.0%	0.212	1.448	$10.63
			20	100.0%	100.0%	0.293	1.529	$10.63
hard	RR	poisson	0	100.0%	99.2%	0.355	5.702	$10.62
			20	100.0%	99.2%	0.436	5.783	$10.62
		trace	0	100.0%	99.2%	0.212	1.062	$10.62
			20	100.0%	99.2%	0.293	1.143	$10.62
	SWRR	poisson	0	100.0%	100.0%	0.355	1.118	$10.62
			20	100.0%	100.0%	0.436	1.199	$10.62
		trace	0	100.0%	100.0%	0.212	1.448	$10.62
			20	100.0%	100.0%	0.293	1.529	$10.62

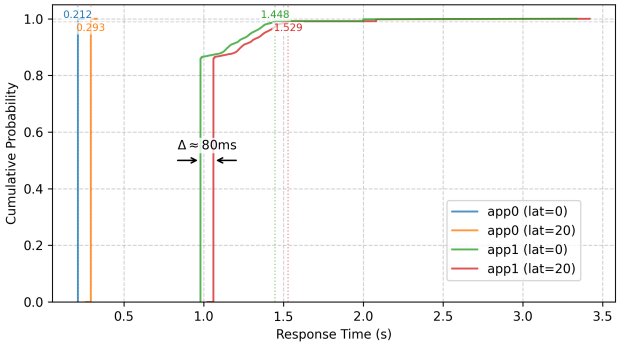


Figure 6. Impact of Network Latency on Response Time CDF ($Q = 0$). The shift highlights the 80 ms RTT floor introduced by the 4-hop topology.

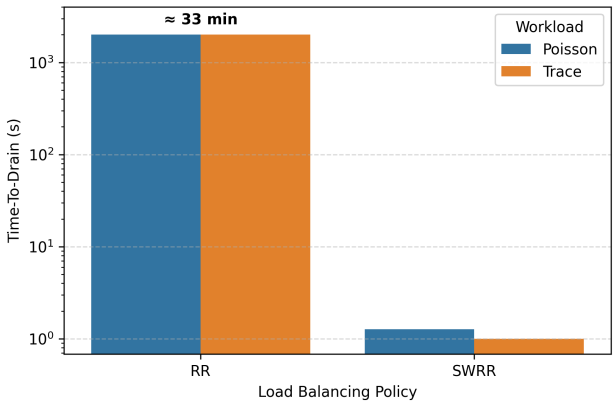


Figure 7. Impact of Load Balancing Strategy on Time-To-Drain (TTD). Naive RR saturates slow containers, forcing extended VM uptime.

≈ 2000 seconds (33 minutes) to process the backlog created by RR, compared to just ≈ 1 second for SWRR. Crucially, despite this, the operational cost remains nearly identical to the optimal case (< 2% difference). This is explained because the extra time required by draining is consumed by a single VM instance, the one containing the slow containers, while the other 37 VMs were already shut down after 1 s of draining.

This highlights a dangerous blind spot: a system can be cost-

efficient according to billing metrics while being functionally broken for the end-user.

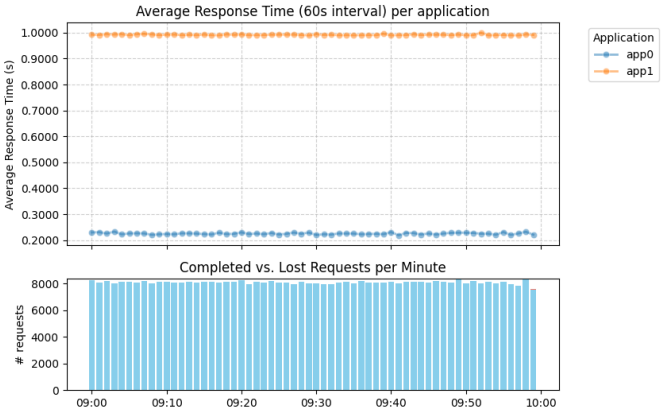
Finally, an analysis of the saturated RR runs (see Table VI) reveals extreme sensitivity to initial conditions. For instance, in the configurations with $Q = 1000$, drain policy and RR load balancer, with trace based workload, the mere addition of 20 ms of network latency causes the p99 response time to jump from ≈ 978 s to ≈ 990 s, as can be seen in the first two rows on the right-hand side of Table VI.

This 12-second discrepancy, triggered by a 0.08 s (80 ms RTT) structural perturbation, represents a non-linear amplification factor of nearly $300\times$. This confirms that the instability is not random noise, but a structural property of saturated queues: in the heavy tail of the distribution, the CDF curve becomes nearly horizontal (slope ≈ 0). In this region, microscopic shifts in probability, whether from network jitter or workload shape, translate into massive discrepancies on the time axis. This validates *Nuberu*'s ability to reproduce the non-linear sensitivity of real-world congestion.

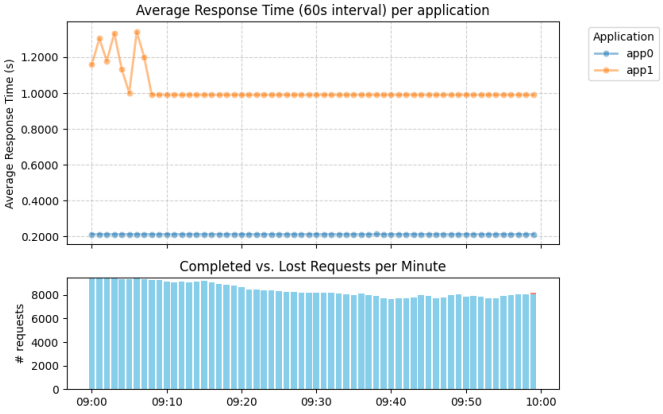
4) Temporal Dynamics and Workload Fidelity: Finally, we analyze the impact of workload modeling on system stability over time. Static optimization relies on aggregated metrics (e.g., global p95 throughput), implicitly assuming that load fluctuations are smoothed out. Figure 8 challenges this assumption by contrasting a synthetic Poisson process against a real-world trace.

In the configurations using synthetic workload (Figure 8a), the response time remains artificially flat. Since the Poisson process is stationary with a fixed λ, it fails to capture the temporal correlations and burstiness of real traffic. In contrast, the trace-driven simulation (Figure 8b) reveals the true dynamic behavior. Here, we observe a transient congestion event during the first 10 minutes for *app1* (orange line), where the response time spikes significantly above the baseline. This corresponds to a period where the specific input rate momentarily exceeded the p95 threshold used by the solver.

This figure allows us to visualize when and how long the system violates its SLOs under these conditions. While *app0* (blue line) remains stable throughout due to adequate headroom, *app1*'s degradation highlights the temporal risks that static provisioning metrics cannot predict. This confirms the simulator's value not just for computing averages, but for stress-testing the deployment against the specific shape of the workload over time.



(a) Synthetic workload with Poisson arrivals



(b) Trace based workload

Figure 8. Response time and number of requests completed for the configurations with SWRR balancing, large queues and 'hard' termination (last two rows of Table VI)

D. Performance and Scalability

The simulation framework's efficiency was evaluated using the 32 configurations presented above. On a server equipped with 32 cores, a single 1-hour simulation configuration completes in approximately 1 minute and 30 seconds. Thanks to the discrete-event independence and the modular design, the 32 configurations were executed in parallel, completing the entire experimental suite in 3 minutes and 21 seconds.

The current bottleneck identified in the prototype is the monitoring layer. The default observer plugin dumps every request lifecycle event into a JSONL file for offline debugging and metric generation (as used in Figure 8). These files can reach sizes of up to 1.3 GB per configuration, introducing significant I/O overhead. However, *Nuberu*'s architecture allows this monitor to be easily replaced by more efficient plugins (e.g., streaming to an external time-series database or performing on-the-fly aggregation), which would likely further accelerate the simulation. While the current implementation also exhibits high memory consumption due to non-optimized internal loops (e.g., $O(N \times M)$ component iterations), these are identified as targets for future engineering optimizations and do not compromise the logical validity of the results.

VI. Conclusion and Future Work

This paper has introduced a modular, extensible framework for cloud simulation 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.

Experimental results confirmed that ignoring network structural latency underestimates tail response times, validating the importance of the proposed bidirectional channel modeling. Furthermore, the evaluation highlighted that naive load balancing in cost-optimized heterogeneous clusters leads to extreme performance degradation during termination. Notably, this degradation in system behavior has minimal impact on total billing costs, suggesting that cost-driven optimization approaches may overlook service quality issues arising from queue imbalances.

From a software engineering perspective, the framework combines runtime plugin discovery, static interface contracts via Python `Protocol`s, and version-controlled configuration. Together, these elements align the framework with the FAIR principles, enabling both local reproducibility and broader community validation of alternative orchestration strategies.

Building on the preliminary performance analysis presented, future work will focus on engineering optimizations to address the identified I/O and memory bottlenecks, specifically targeting the monitoring layer efficiency. Simultaneously, we plan to expand the plugin ecosystem to support emerging paradigms such as Large Language Model (LLM) serving, spot-instance strategies, and hybrid cloud-edge deployments. These extensions will further validate the framework's ability to model complex, heterogeneous environments.

Appendix A
Example Simulator Configuration

Listing 6 shows the configuration file for an example scenario, with Poisson arrivals, Round-Robin load balancing, and custom network delays. This configuration use specific latencies: 20 ms for both client-to-balancer and balance-to-VM channels. The runtime model is configured with a bounded queue capacity of 1000 requests to model resource contention, as discussed in the experimental evaluation. Furthermore, although this baseline sets global defaults, Nuberu allows for fine-grained network modeling: the user can specify per-VM or per-application overrides (via vm_network_overrides) to simulate heterogeneous topologies.

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. Luque, J. L. Díaz, J. Entrialgo, and R. Usamentiaga, "Modular and reproducible simulator architecture for composable cloud systems," in *Proceedings of SIMUL 2025: The Seventeenth International Conference on Advances in System Modeling and Simulation*, Lisbon, Portugal: IARIA / ThinkMind Digital Library, 2025, pp. 82–87, ISBN: 978-1-68558-300-2. Available from: https://www.thinkmind.org/library/SIMUL/SIMUL_2025/simul_2025_1_180_50098.html.

[2] R. Luque and J. L. Díaz, "Nuberu: A modular cloud simulation framework," version 0.4.0, 2026, Available from: https://github.com/asi-uniovi/nuberu-releases [retrieved: February, 2026].

[3] 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.

[4] 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.

[5] N. Mansouri, R. Ghafari, and B. M. H. Zade, "Cloud Computing simulators: A comprehensive review," *Simulation Modelling Practice and Theory*, vol. 104, p. 102 144, Nov. 2020, ISSN: 1569-190X. DOI: 10.1016/j.simpat.2020.102144.

[6] 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.

[7] S. Lata and D. Singh, "Cloud simulation tools: A survey," *AIP Conference Proceedings*, vol. 2555, no. 1, p. 030 003, Oct. 2022, ISSN: 0094-243X. DOI: 10.1063/5.0109181.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] I. Lera, C. Guerrero, and C. Juiz, "YAFS: A simulator for IoT scenarios in fog computing," *IEEE Access*, vol. 7, pp. 91 745–91 758, 2019. DOI: https://doi.org/10.1109/ACCESS.2019.2927895.

[13] 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.

[14] 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.

[15] J. Massa et al., "ECLYPSE: A Python Framework for Simulation and Emulation of the Cloud-Edge Continuum," *Journal of Software: Evolution and Process*, vol. 38, no. 1, e70081, Jan. 2026. DOI: 10.1002/smr.70081.

[16] M. D. Wilkinson et al., "The FAIR guiding principles for scientific data management and stewardship," *Scientific Data*, vol. 3, p. 160 018, 2016. DOI: 10.1038/sdata.2016.18.

[17] SimPy Development Team, "Simpy: Discrete event simulation for python," Version 4.1.2, 2025, Available from: https://simpy.readthedocs.io/ [retrieved: July, 2025].

[18] pytest-dev, "Pluggy: A minimalist production ready plugin system," Version 1.5.0, 2025, Available from: https://pluggy.readthedocs.io/ [retrieved: July, 2025].

Listing 6. Example YAML simulation configuration

```yaml
config_version: "3.2"
simulation:
  seed: 12345
  stop_time: 3600
  eager_start: true
  drain_pending_requests: true

network_delays:
  wi_to_lb: 20
  lb_to_vm_default: 20

logging:
  log_to_file: true
  log_to_console: false
  level: critical
  component_levels:
    plugins.report_generator: info
  output_path: logs/
  log_file_name: "example.log"

custom_plugins:
  - plugin_name: monitor_simple
    plugin_config:
      dump_file: "example.jsonl"
  - plugin_name: report_generator
    plugin_config:
      output_format: jsonl
      output_path: "example.report"

infrastructure:
  plugin_name: conlloovia
  plugin_config:
    pickle_path: "conlloovia_data.p"

runtime_model:
  plugin_name: simple
  plugin_config:
    queue_size: 1000

performance_data:
  plugin_name: conlloovia
  plugin_config:
    pickle_path: "conlloovia_data.p"

allocation:
  plugin_name: "conlloovia"
  plugin_config:
    pickle_path: "conlloovia_data.p"

workloads:
  - app: app0
    distribution:
      plugin_name: poisson
      plugin_config:
        num_reqs: 124427
        time_slot_size: 3600
  - app: app1
    distribution:
      plugin_name: poisson
      plugin_config:
        num_reqs: 362587
        time_slot_size: 3600

load_balancer:
  plugin_name: "roundrobin"
  plugin_config: {}
```


[19] I. Levkivskyi, J. Lehtosalo, and L. Langa, "Pep 544: Protocols: Structural subtyping (static duck typing)," Python Enhancement Proposal, Python-Version: 3.8, Final, 2017, Available from: https://peps.python.org/pep-0544/.

[20] 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.

[21] M. Dounin, "Upstream: Smooth weighted round-robin balancing," Commit, nginx source tree, May 2012, Available from: https : / / github . com / phusion / nginx / commit / 27e94984486058d73157038f7950a0a36ecc6e35 [retrieved: July, 2025].

[22] R. Luque, J. L. Díaz, J. Entrialgo, and R. Usamentiaga, "Nuberu simulation results – experiments repository," 2026, Available from: https : / / github . com / asi - uniovi / nuberu - experiments - results / tree / simul2026 [retrieved: February, 2026].