CNASIM: A Customizable Simulation Library for Cloud-Native Application Performance Modeling

Zengyi Wang[†*], Paul Daniëlse[*], and Zhiming Zhao[*]
zengyi.wang@pega.com, {p.e.g.danielse, z.zhao}@uva.nl

University of Amsterdam[*], Pegasystems[†]

Abstract. With the widespread adoption of cloud-native technologies, Cloud-Native Applications (CNAs) have become the mainstream approach for building scalable and maintainable software systems. However, due to their structural diversity, dynamic runtime behaviors, and complex infrastructure dependencies, existing simulators often fall short in modeling and simulating such systems effectively. To address these limitations, this paper introduces CNASIM, a modular and extensible performance simulation library for cloud-native applications. The library supports flexible modeling of various CNA architectures and behaviors, offering customizable components for services, instances, communication mechanisms, and resource management strategies. It also provides configurable interfaces, component-level metric collection, and extension mechanisms for incorporating user-defined logic. Experimental results demonstrate that our approach achieves good performance in terms of modeling flexibility, simulation accuracy, and applicability to complex scenarios, making it a practical aid for system designers and researchers to evaluate and optimize cloud-native applications efficiently.

Keywords: Cloud Computing · Microservice Architectures · Discrete Event Simulation.

1 Introduction

Cloud-Native Applications (CNAs) are software systems designed following a microservices architecture, where individual services can be independently developed, deployed, and scaled elastically. These applications leverage containerization and automated operations to enable dynamic scaling and efficient resource utilization in cloud environments. To better understand, predict, and optimize the behavior of such systems, simulating cloud-native applications has become an effective approach. Simulators allow developers and researchers to explore the performance implications, reducing experimentation costs and minimizing risks.

However, the complexity and diversity of CNAs make it extremely challenging to develop a general-purpose simulator. CNAs involve a wide range of dynamic behaviors, such as inter-service communication patterns, automatic scaling, and scheduling strategies. Additionally, CNAs vary greatly in terms of architectural styles, underlying infrastructure, and application types, making it difficult to

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29909-3_16

capture a wide range of scenarios with a single unified simulation model. Although several microservice and cloud-native simulators already exist, they fall short in addressing these challenges. Many are solely designed for microservices, but lack support for key features of cloud-native systems, such as fault injection and automatic scaling. Furthermore, most simulators do not offer extensibility, making it difficult to adapt simulated application to better match structurally and functionally diverse real-world applications. As a result, existing simulators often struggle to keep up with the fast-evolving nature of cloud-native systems.

To address these challenges, we propose CNASIM, a customizable simulation library specifically designed for performance modeling. Its modular design enables users to flexibly define and compose various service types, instance management, communication mechanisms, and resource scheduling strategies. Additionally, it provides easily extensible interfaces and highly configurable components, allowing users to implement personalized modeling logic. Specifically, the contributions of this work are:

1. The development of an open-source cloud-native application performance simulation library. Which can be found on GitHub[1].
2. The validation of the accuracy of performance prediction and the capability to model complex cloud-native architectures, demonstrating its usefulness in the design, optimization, and research of such systems.
3. Practical modeling examples of cloud-native applications, covering typical service architectures and asynchronous communication scenarios.

The remainder of this paper is structured as follows. Section 2 introduces the state of the art in cloud-native system modeling and simulation. Section 3 presents the proposed solution, including the core design principles and the implementation details. Section 4 evaluates the proposed solution through a series of experiments designed to assess its accuracy and applicability. Section 5 discusses the results, highlights the strengths and limitations of CNASIM. Finally, we conclude with Section 6 and propose potential future work.

2 Related Work

Overall, current simulators have demonstrated basic capabilities in modeling cloud-native applications. Most simulators support synchronous communication and dynamic workload generation [1–4], and some incorporate simplified models for resource constraints and network faults [5,6]. A few have also begun to explore dynamic behaviors unique to cloud-native environments, such as elastic scaling and runtime scheduling [1, 2, 4, 7].

However, there are several remaining gaps, particularly in modeling service heterogeneity and communication patterns. Simulators assume a unified resource consumption model, where request processing time is computed based on simulated CPU usage or directly defined in the requests as the CPU load [1, 2]. This

[1] `https://github.com/wzy1935/cna-sim`

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29909-3_16

Table 1. Feature comparison of cloud-native application simulators. For the scheduling policies we use the labels Workload, Infrastructure and Resources (W/I/R). We use the labels BW for bandwidth, Cong. for congestion and Loss for packet loss.

Feature	PerfSim	Courageux et al.	EvolutionSim	MiSim	μqSim	Turin et al.	Cloud-NativeSim	CNASIM
Service types	No	No	No	No	Yes	No	No	Yes
Service dependencies	Yes	Yes	Yes	Yes	Yes	Partial	Yes	Yes
Communication patterns	Sync	Sync	Sync	Sync	Sync	Sync	Sync	Sync/Async
Workload generation	Static	Dynamic	Dynamic	Dynamic	Dynamic	Static	Dynamic	Dynamic
Network simulation	Delay, Cong., BW	Delay	Delay, BW	Delay	Delay, BW	No	Delay	Delay, Loss
Resource modeling	Infra	App	Infra	App	Infra	Infra	Infra	App
Elastic scaling	No	Partial	Yes	Yes	No	Yes	Yes	Yes
Scheduling policies	W/I/R	No	W/I/R	W	No	I/R	W/I/R	W
Metric export	Yes	No	Yes	Yes	Yes	No	Yes	Yes
Extensibility	Partial	No	Sched.	Resilience	No	Partial	Scale/Sched.	Yes

approach makes it difficult to represent specialized service types, such as machine learning tasks that require GPU resources, or large distributed eco-systems where a external service's performance cannot be described using standard resource models. Furthermore, the need for communication pattern modeling is essential as the performance of many services rest from interleaving client requests through asynchronous communication. Many simulators based on microservice architecture assume purely synchronous communication. This makes it impossible to simulate message queues or event streams.

Another important aspect is the modeling of resource consumption and the impact of resource usage on performance. We identify two main paradigms; infrastructure-level modeling [2,5–8], which adopts a bottom-up approach, starting from modeling data centers, virtual machines, and network topology, and then layering services, requests, and workloads on top. And application-level modeling [1, 3] which adopts a top-down approach and models resource consumption through the virtual resources taken by the application components which requires no modeling of the hardware but measurements of component resource usage. Our design makes use of application-level modeling as it better aligns with our objective to model application structure and behavior instead of details of the underlying infrastructure. For example, public cloud users often lease physical infrastructure provided by cloud vendors and are gain more from predictions concerning the performance impact of architectural choices.

We have restricted our simulator to cloud applications. Simulators like YAFS [9] and IFogSim [10] model different computing scenarios (fog and edge computing) to provide insights into resource utilization and application quality of service. They include features for failure scenarios and more complex simulator of the network that connects the different simulated topologies. However, these simulators fall short in providing insights into application performance under various application scenarios and adopt a bottom up approach.

CNASIM was designed with these factors in mind. It additionally includes a flexible extensibility interface that supports the simulation of cloud-native applications with diverse architectural structures and runtime behaviors.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29909-3_16

3 Design and Implementation

CNASIM is a Python-based library. Unlike traditional simulators, it is not a standalone software application, which makes it easier to extend and integrate. Figure 1 shows the overall structure of the CNASIM library. The three modules on the left are responsible for the main simulation logic. The data collector and configuration API are used to help users configure custom data collection and define configuration files, respectively.

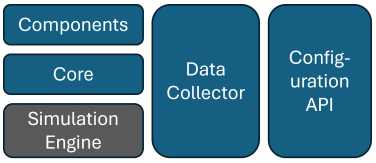

Fig. 1. CNASIM module architecture

3.1 Overview

As a basis for our simulator the SimPy library provides a process-based discrete-event simulation engine and framework. On top of this, we built the CNASim modules which integrate the basic interface for components, implements the messaging between components and simulates the various cloud-native application components, such as services, instances, gateways, etc.

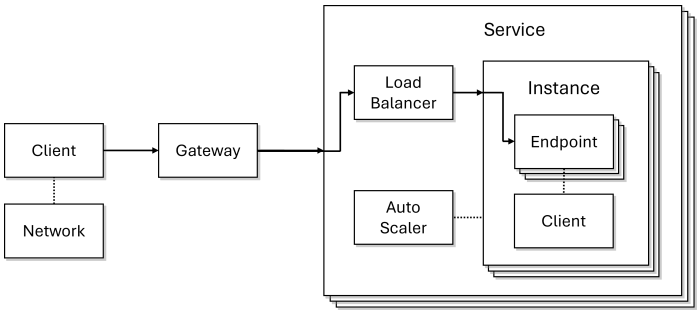

Fig. 2. CNASIM execution model

Figure 2 illustrates how core components within CNASIM interact along the request path. All requests are initiated by the client and sent to the gateway to locate the target component through the network. Once a request reaches the target, for example, the chosen service, it is passed to the load balancer,

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29909-3_16

which forwards it to one of the available instances. The instance then uses the endpoint name to dispatch the request to the appropriate endpoint, where the request is executed. An endpoint can also access instance-level resources via exposed interfaces, for example, to simulate computation tasks or to send out requests using the client bound interface.

3.2 Message Passing

The message passing mechanism is based on an asynchronous `Promise` mechanism combined with SimPy's co-routine framework. Each request is defined by a request context, which combines states a promise of a response and contains the request status, request body, response body, and timestamps of state changes. A request context can be held simultaneously by multiple components. The sender issues a request by sending the information to the receiver and can choose to wait for the request to complete, thereby blocking the current co-routine making communication synchronous again. This is released when the receiver answers the requests successfully, at which point the sender receives the response data as a return value. In the asynchronous case the sender may choose not to wait for the request completion and continue executing subsequent code. In this case, the receiver completes the request without the sender receiving a response, thus implementing one-way asynchronous communication. When the receiving component activates a delay modeling network latency is applied. The `transmit` method takes the source, target, and the request itself as parameters, which can dynamically determine delays based on request size and network topology to allow modeling complex and multi cloud deployments and cloud-edge application where network differences can have a large impact on performance [**?**].

3.3 Instance Computation Model

An instance models a single computing unit, such as a bare-metal server, virtual machine, container, or serverless instance. By default, we implemented the simulator components to model an instance as a web server hosted in a container, using a thread pool model. Each instance maintains a request queue (RQ), a thread pool (TP) consisting of several thread resources, and a set of endpoints (EP). When an instance receives a request it is put on the request queue. The instance's main loop continuously checks the queue to handle incoming requests. For each request, it attempts to spawn a new process to handle it, simulating the behavior of assigning a thread from the thread pool. This process is describe in detail in Algorithm 1.

The behavior of an endpoint can be define by the user. Receiving a request will start the simulation a computational or data task within the instance and the appropriate resources within the instance will be affected. The network components perform a similar structure when asked to simulate a large data transfer. Alogrithm 2 describes the details of this method. cnt_{at} is the active threads count, n_{CPU} is the CPU quota, and λ_{wu} is the warm-up time factor. The task

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29909-3_16

Algorithm 1 Main and Sub Process in an Instance

```
function MAIN_PROCESS( )
    while STATUS == ACTIVE do
        r ← await RQ.get()
        await TP.get()
        sub_process(r)
    end while
end function

function SUB_PROCESS(r)
    await EP.recv_request(r)
    TP.put()
end function
```

consumes CPU resources, and it is defined by a distribution D of the ideal computation time, that is, the computation time when with ideal resource quota and no inference. The default distribution is log-normal distribution to capture the long-tail characteristics of latency. The actual computation time is influenced by three factors: ideal time distribution, CPU contention, and cold start. CPU contention is determined by the CPU quota and the number of active threads. The cold start is controlled by the warm-up factor, which is computed as follows in Equation 1, where λ_{wu}^{init} is the warm-up factor init, d_{wu} is the warm-up time, and t_a is the duration since the instance became active.

$$\lambda_{wu}(t_a) = \begin{cases} \lambda_{wu}^{init} + (1 - \lambda_{wu}^{init}) \cdot \dfrac{t_a}{d_{wu}}, & 0 \le t_a < d_{wu} \\ 1, & t_a \ge d_{wu} \end{cases} \tag{1}$$

An instance has four states which represent its life-cycle. *STARTING* indicates that the instance is in the process of initialization and is not yet ready to accept requests. after a delay of d_{su} it transitions to *ACTIVE*. Here the instance has fully started and is ready to serve. *TERMINATING* indicates that the instance is shutting down; it continues processing ongoing requests but no longer accepts new ones, and after a shutdown delay of d_{sd} it transitions to *TERMINATED* and all resources are released.

Algorithm 2 Compute Process in an Instance

```
function COMPUTE(D)
    cnt_at ← cnt_at + 1
    t ← D.sample() × max(1, cnt_at / n_CPU) × λ_wu
    await timeout(t)
    cnt_at ← cnt_at − 1
end function
```

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29909-3_16

The CPU usage amount is calculated based on the integral of active threads in the ACTIVE or TERMINATING state over time. The theoretical formula is shown in Equation 2. This formula sums the number of active threads during the last one second, limited by the CPU quota, reflecting the total CPU resources consumed. Let $cnt_{at}(t)$ denote the total number of such threads at time t, and let cpu^{i}_{at} be the expected CPU footprint per thread of type i.

$$U_{\text{cpu}}(t) = \int_{t-1}^{t} \min(cnt_{at}(t)\, cpu^{i}_{at}, n_{CPU})\, dt \tag{2}$$

4 Evaluation

In this section, we consider two types of validation. The first part evaluates the accuracy of performance simulation by comparing it against a real-world web application. The second part examines the modeling applicability using a structurally complex cloud-native application as a reference case. All experiments are conducted on a workstation with an AMD Ryzen 7 5800H processor running at 3.20 GHz.

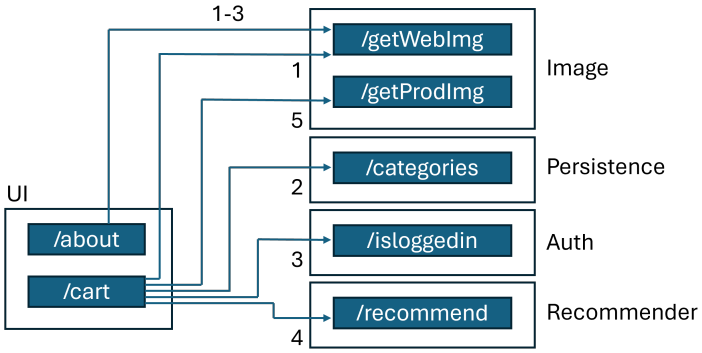

Fig. 3. Dependency graph of `/about` and `/cart`, The numbers on the arrows indicate the order of invocation

4.1 Accuracy Validation

We select TeaStore [11] as the reference system for validating the accuracy of our performance simulation. TeaStore is a microservice benchmark system that simulates a realistic online store, widely used in microservice-related research for performance testing, modeling validation, and resource scheduling experiments. It consists of six microservices: UI, Image, Persistence, Auth, Recommender, and Registry. WebUI serves as the user entry point and communicates with other

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29909-3_16

services via HTTP requests. In our experiment, we focus on two UI APIs, `/about` and `/cart`, which involve calls to all services except Registry. We analyzed the source code to obtain the service dependency graph, as shown in Figure 3.

We configured the simulator based on the structure and performance characteristics of the real application. Using performance testing on the real application, we measure the expected computation time for each interface. Based on our analysis of service dependencies and expected performance behavior, we construct the corresponding endpoints and configure them in the simulator as shown in Figure 3. To measure the expected computation time, we applied a relatively small workload to each exposed API individually on a real system, thereby ensuring that service performance measurements included minimal interference, and measure the responce time and resource usage. The computation time was then calculated as the invocation time minus the waiting time incurred by calls to other downstream endpoints.

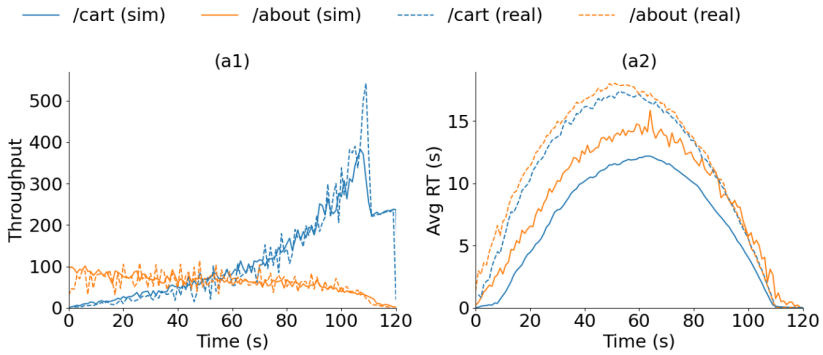

Fig. 4. Throughput and average response time results in performance testing

We designed two different workload patterns for the validation experiments, each scenario lasting 120 seconds. We employed dynamic RPS (Requests Per Second) and tested both APIs together to verify whether the simulator is capable of handling dynamic input and interference between multiple endpoints. In Scenario A, the RPS of the `/cart` endpoint increases linearly from 0 to 240, while that of `/about` decreases linearly from 120 to 0. In Scenario B, the pattern of `/cart` remains unchanged, but the RPS of `/about` is reversed.

The experimental results are shown in Figure 4 and 5. figures a1, a2 and b1, b2 represent the throughput and average response time for Scenarios A and B, respectively. The simulation results are shown as solid lines, while the real results are represented by dashed lines. From the experimental results, we can observe interference between the two endpoints. In Scenario A, during the first 60 seconds, `/about` requests leads to a request backlog. After, the server has available CPU resources to handle the queued tasks and finishes processing the backlog at 110 seconds. In Scenario B, the request rates for both endpoints

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29909-3_16

increase simultaneously and reach capacity at 70 seconds. After that point, the continuously accumulating queue causes a rise in response time.

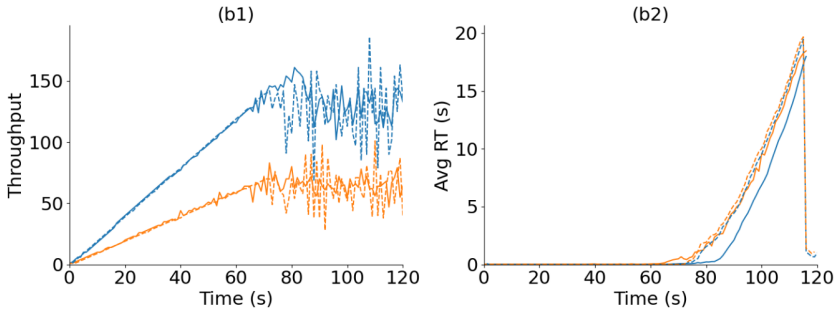

Fig. 5. Throughput and average response time results in performance testing

Based on our observations, the simulation results demonstrate a high degree of consistency with the real measurements in both trend and magnitude. The simulator accurately captures key transition points in system behavior, such as the turning points at 60 and 110 seconds in Scenario A and at 70 seconds in Scenario B. In terms of numerical fidelity, the predicted throughput and average response time closely match the actual values across most of the timeline. The primary discrepancies lie in the variance of throughput and the deviation in response time. The higher variance observed in real throughput is likely due to runtime overheads such as request contention and thread context switching, which are not explicitly modeled in our simulation. As for the response time, the relative deviation may stem from minor configuration differences between the simulated and real environments. Since the response times for both APIs are often near zero, even small absolute differences can appear significant after proportional scaling. Overall, these differences are explainable and do not undermine the validity of the simulator in capturing system dynamics.

Table 2. MAE and MAPE of response time and throughput

Scenario	**Avg Resp. Time (ms)**		**Throughput**	
	MAE	**MAPE**	**MAE**	**MAPE**
Gradual	736.9	0.483	20.1	0.042
Stepwise	517.3	0.662	23.1	0.044
Opposing	4001.9	0.581	13.3	0.216
Parallel	763.0	0.927	3.81	0.038

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29909-3_16

Table 2 reports the MAE and MAPE of average response time and throughput across different scenarios, averaged over five simulation runs. During the calculation, we excluded the first and last 3 data points and applied a moving average with a window size of 10 to smooth the data, thereby reducing errors caused by fluctuations. Notably, the MAPE values for response time are significantly higher, not solely due to simulation inaccuracies but also due to amplification effects inherent in the metric itself. For example, in Scenario Parallel, the response time MAPE reaches 0.927. This is primarily because the real system's response time started increasing approximately 5 seconds before the simulator, while the simulator's response remained near zero. This brief period resulted in extremely large relative errors that were up to several hundred times, which substantially increased the overall MAPE. In such cases, MAE provides a more intuitive reflection of the difference between the simulation and the real system.

In summary, CNASIM demonstrates a reasonable level of accuracy in simulation and is able to effectively capture the trends in performance metric changes. However, in more complex settings, such as tests involving multiple endpoints, its value predictions become less accurate.

4.2 Applicability Testing

To evaluate CNASIM's ability to model cloud-native applications with specialized architectures we selected the European Environmental Research Infrastructures Knowledge Base (ENVRI KB) [12]. ENVRI KB is a community knowledge base designed to capture and manage information about environmental and earth science research. This application that uses asynchronous communication as a case study and validated the modeling capability by comparing its performance under different architectural configurations. Our experiments focus on ENVRI KB's indexer service, which is a batch job initiated by administrators to crawl relevant information from the web and write it into an Elastic-search index. The reason for this choice is that, unlike traditional web applications, the indexer is a batch-job type task with an event-driven architecture that uses a message broker for asynchronous communication instead of synchronous HTTP requests.

The indexer workflow is as follows: administrators submit lists of datasets respective URIs to a queue, from which the indexer fetches information, processes it, and writes the results into the document store. The indexer can be divided into three components: *retrieval*, *processing*, and *ingestion*. Retrieval fetches web information based on task URLs, processing filters and transforms the data into document format, and ingestion writes the results into Elastic-search.

The experiment evaluates the difference between two architectures, shown in Figure 6. The first is a monolithic structure, where retrieval, processing, and ingestion are handled sequentially within a single worker instance before fetching the next task. The second is a decoupled workflow with three separate workers connected asynchronously. Each dashed arrow denotes a queue that is simulates using a message broker service. In an actual system, we would select Kafka as the message broker and KEDA as the scheduler to scales instances proportionally to

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29909-3_16

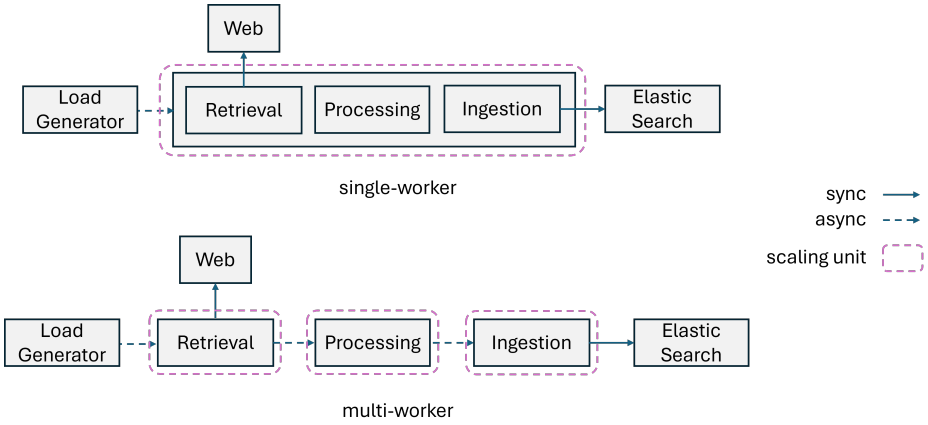

Fig. 6. Architectural setups of the indexer service

the user load. When splitting the task into three workers, we allocated CPU resources based on task types. In the single-worker setup, the indexer was assigned a full CPU core. In the multi-worker configuration, the retrieval, processing, and ingestion workers were allocated 0.1, 0.8, and 0.1 CPU cores, respectively.

The modeling process is as follows: the use of a message broker is not predefined in our simulator, so we extended CNASIM's interfaces to implement custom components. For this use case, we defined three components: `Broker`, `BrokerScaler`, and `ExternalServer`. Next we extended the `SyncServer` definition to support communication between workers and the broker. The `ExternalServer` models external services (e.g., dataset URI access and the Elastic-search service) whose performance is assumed to be stable. Additionally, the computation times for each step within workers were estimated from measurements of the original monolithic application. Subsequently, we simulate the impact of these two architectures on total task processing time and resource usage.

Averaged over five runs, the monolithic structure completed tasks in approximately 1059.94 seconds, while the multi-worker setup took about 1097.88 seconds, with the multi-worker being slightly slower. Figure 7 shows the number of instances and CPU requests over time for both architectures. The monolithic worker and the retrieval worker exhibited similar behaviors, scaling quickly to maximum capacity initially and then declining, whereas the other two workers maintained roughly six instances. The CPU request graph reveals that the monolithic application's CPU requests were about three times those of the multi-worker setup. indicate that the multi-worker architecture has an advantage in CPU allocation. The higher number of retrieval workers highlights that retrieval is the processing bottleneck. Since retrieval is I/O-intensive, it requires minimal CPU allocation, and the CPU-intensive processing task has relatively fewer in-

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29909-3_16

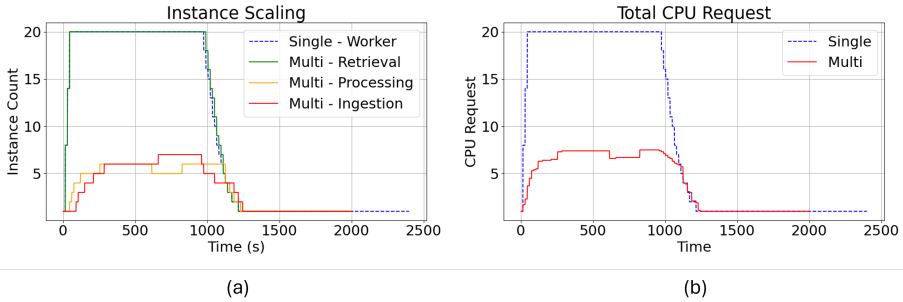

Fig. 7. Resource usage and task completion times for different architectures

stances. Consequently, the total CPU requests in the multi-worker architecture are lower, significantly improving resource efficiency.

This demonstrates that CNASIM exhibits strong affinity for modeling cloud-native applications with specialized architectures. Despite the absence of predefined support for message brokers, CNASIM's extensible interface enabled us to flexibly implement necessary components, such as those simulating key behaviors of message-broker's asynchronous communication and event-driven architectures. Moreover, the experimental results show that CNASIM can realistically capture the performance impact of different architectural designs, including task processing time and resource usage patterns, highlighting its modeling capability and applicability in complex application scenarios.

5 Discussion

In our work, we particularly emphasize concepts such as scalability and usability to address the challenges posed by the structural and typological diversity of cloud-native applications. We believe that designing a model that covers all cloud-native application use cases is unrealistic. Therefore, for specific modeling needs, we hand over control to users, allowing them to implement the functionalities they require. Conceptually, the cloud-native simulation library could include a list of collected user defined components empowering community driven development. Its highly modular components and interfaces allow users to define various needed functions by themselves. We have demonstrated this through our study of the ENVRI KB, showing that CNASIM can support applications with diverse structures, something that other simulators currently cannot do.

In the design and implementation of CNASIM, we strive to ensure ease of use. On one hand, we provide standard data collection and configuration file interfaces, through which users can compose the out-of-the-box components. When designing these interfaces, we also added support for custom components, making them more general. Of course, custom component functionality may still pose challenges for users, especially when extending existing classes, as understand-

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29909-3_16

ing the original implementation is often required. This paper does not include experiments verifying the usability of our library, which remains to be evaluated.

In the evaluation, we validated simulation accuracy. The results met expectations but leave room for improvement especially when modeling processes with many functionalities. This is because the implemented standard model is simple by design. For instance, our resource modeling only considers CPU usage and memory usage; it does not simulate network bandwidth, or interference among containers, threads, and requests. We did not test or verify the impact of these factors during model design, which contributes to the accuracy limitations.

Finally, simulation efficiency is also a limitation of this work. We did not design experiments specifically targeting efficiency, but from our testing experience, the simulation speed scales proportionally with the number of requests. In our TeaStore tests, simulation speed was roughly four times faster than the real application. However, when requests are generated frequently, the simulation speed can be slower than the real application.

6 Conclusion

This paper addresses the challenges of poor extensibility and applicability in existing CNA simulators by proposing and implementing a performance simulation library named CNASIM. Built on a modular and component-based design, CNASIM allows users to flexibly compose and extend components to simulate structurally and behaviorally diverse CNAs. We conducted in-depth discussions on modeling, implementation, and validation, proposed a clear modeling methodology, and validated the usability and accuracy of the simulator through two case studies. Experimental results demonstrate that CNASIM can support performance prediction and evaluation of complex CNA architectures while maintaining relatively high simulation efficiency. Although there is still room for improvement in terms of usability and simulation accuracy, this work provides a solid foundation for future research in simulator development and system optimization.

6.1 Future Work

To more accurately reflect real-world performance, future work will focus on enhancing simulation fidelity by incorporating additional resource dimensions, such as network bandwidth and modeling potential interference between components. Moreover, expanding the library with more component types and load balancers will enable the simulator to support a broader range of cloud-native architectures and runtime scenarios.

Acknowledgments.

This work has been partially funded by the EU Horizon-Europe ENVRI-HUB next (101131141), EVERSE (101129744), and BLUECLOUD 2026 (101094227) projects, and by the Dutch Research Council, LTER-LIFE project.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29909-3_16

References

1. S. Frank, L. Wagner, A. Hakamian, M. Straesser, and A. van Hoorn, "Misim: A simulator for resilience assessment of microservice-based architectures," in *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, 2022, pp. 1014–1025.
2. T. Wang, X. He, H. Shi, and Z. Wang, " EvolutionSim: An Extensible Simulation Toolkit for Microservice System Evolution ," in *2023 IEEE International Conference on Web Services (ICWS)*. Los Alamitos, CA, USA: IEEE Computer Society, Jul. 2023, pp. 43–49. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICWS60048.2023.00018
3. C. Courageux-Sudan, A.-C. Orgerie, and M. Quinson, "Automated performance prediction of microservice applications using simulation," in *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2021, pp. 1–8.
4. J. Wu, M. Xu, Y. He, K. Ye, and C. Xu, "Cloudnativesim: A toolkit for modeling and simulation of cloud-native applications," *Software: Practice and Experience*, vol. 55, no. 7, pp. 1185–1208, 2025. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.3417
5. M. G. Khan, J. Taheri, A. Al-Dulaimy, and A. Kassler, "Perfsim: A performance simulator for cloud native microservice chains," *IEEE Transactions on Cloud Computing*, vol. 11, no. 2, pp. 1395–1413, 2023.
6. Y. Zhang, Y. Gan, and C. Delimitrou, "μqsim: Enabling accurate and scalable simulation for interactive microservices," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 212–222.
7. G. Turin, A. Borgarelli, S. Donetti, F. Damiani, E. B. Johnsen, and S. L. Tapia Tarifa, "Predicting resource consumption of kubernetes container systems using resource models," *Journal of Systems and Software*, vol. 203, p. 111750, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121223001450
8. R. N. Calheiros, R. Ranjan, C. A. F. D. Rose, and R. Buyya, "Cloudsim: A novel framework for modeling and simulation of cloud computing infrastructures and services," 2009. [Online]. Available: https://arxiv.org/abs/0903.2525
9. 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.
10. H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, and R. Buyya, "ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments," *Software: Practice and Experience*, vol. 47, no. 9, pp. 1275–1296, 2017. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2509
11. J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research," in *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, ser. MASCOTS '18, September 2018.
12. X. Liao, D. Goldfarb, B. Magagna, M. Stocker, P. Thijsse, D. Schaap, and Z. Zhao, "ENVRI knowledge base: A community knowledge base for research, innovation and society," in *EGU General Assembly Conference Abstracts*, ser. EGU General Assembly Conference Abstracts, May 2020, p. 20708.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29909-3_16