MicroViSim: Simulation and Visualization of Kubernetes-Based Microservice Systems

Wei-Kai Lin[1], Shang-Pin Ma[1*], Shin-Jie Lee[2], Wen-Tin Lee [3]
Department of Computer Science and Engineering, National Taiwan Ocean University, Keelung, Taiwan[1]
Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan, Taiwan[2]
Department of Software Engineering and Management, National Kaohsiung Normal University, Kaohsiung, Taiwan[3]
11257005@mail.ntou.edu.tw, albert@ntou.edu.tw, jielee@mail.ncku.edu.tw, wtlee@nknu.edu.tw

***Abstract*—Microservice architecture has emerged as the dominant architectural pattern for modern software systems, offering scalability, modularity, and fault tolerance. However, its highly distributed and interdependent nature makes evaluating architectural designs and proposed changes increasingly complex. This paper presents MicroViSim, a lightweight tool for simulating load and visualizing service dependencies and performance metrics in Kubernetes-based microservice systems. MicroViSim enables developers and operators to pre-evaluate new or evolving architectures prior to deployment by generating service dependency graphs, performance indicators, and load simulation results from user-defined YAML configurations. These visual insights help uncover potential bottlenecks and performance risks early in the design phase, thereby reducing future maintenance costs and the likelihood of system failures. Experiments using a real-world microservice application, Bookinfo, demonstrate that MicroViSim effectively identifies architectural weaknesses and supports informed deployment decisions.**

Keywords—microservices, architecture visualization, Kubernetes, load simulation, system design evaluation

I. INTRODUCTION

Modern software systems must continuously evolve to meet changing requirements and operational demands. Traditional monolithic architectures often hinder this evolution, as their tightly coupled structures make modular growth and large-scale changes risky and costly. In contrast, microservice architectures have emerged as a dominant style by decomposing applications into smaller, loosely-coupled services that can be developed, deployed, and scaled independently [1]. This shift enables greater flexibility, resilience, and team autonomy. However, it also increases architectural complexity, as containerized microservices orchestrated by Kubernetes [2] give rise to intricate dependencies and dynamic workloads. Therefore, understanding and managing these evolving architectures requires systematic approaches for analysis and simulation.

Despite these advantages, the adoption of microservices introduces significant challenges. As systems scale, the growing number of services and their complex interdependencies make it difficult to reason about overall system behavior. Developers must account for indirect service call chains, communication latencies, and the propagation of faults across services. These issues are especially critical in early-stage architectural design, where performance and reliability cannot yet be empirically observed. Moreover, inappropriate architectural changes may introduce cascading failures or bottlenecks, substantially degrading the fault tolerance of microservice systems [3].

To mitigate these risks, simulation-based approaches have gained attention as a means of evaluating architectural decisions prior to deployment. One such direction is chaos engineering, which introduces controlled faults into live systems to assess their resilience [4, 5]. While effective, these techniques require production-like environments and carry potential risks, making them difficult to adopt during early design or planning phases. Other simulation frameworks, such as PerfSim [6] and EvolutionSim [7], leverage performance modeling and discrete event simulation to estimate the behavior of microservice systems under various conditions. However, these tools often require detailed execution traces or complex modeling processes, which can be burdensome for teams without access to rich runtime data or deep simulation expertise. Moreover, while existing simulators usually target brownfield systems by extracting runtime traces from deployed services. Greenfield projects face similar challenges, as lightweight and accessible tools for validating early-stage designs remain scarce.

To address these gaps, this work builds upon the non-intrusive microservice visualization and monitoring tool KMamiz (Kubernetes-based Microservices Analyzing and Monitoring system using Istio and Zipkin)[8], and proposes a mechanism for visual comparative analysis of microservice architecture versions and load simulation: MicroViSim (Microservice Visualization and Simulation), a lightweight, non-intrusive tool designed for simulating load and visualizing service dependencies in Kubernetes-based microservice architectures. MicroViSim supports both greenfield and brownfield scenarios: it allows developers to either import configurations and traces from existing systems or define hypothetical service architectures and request patterns using simple YAML-based configuration files. The tool generates dependency graphs, simulates traffic propagation, and estimates endpoint-level latency, error rates, and risk indicators. These visual outputs provide developers and operators with practical insights to enable early detection of bottlenecks and design flaws, before any (new) service is actually deployed.

II. RELATED WORK

Courageux-Sudan et al. [9] presented a performance modeling approach using the SimGrid platform. Their model automatically extracts simulation parameters from runtime traces and supports the evaluation of different deployment strategies, such as CPU upgrades and service placement. PerfSim [6], proposed by Khan et al., integrates system-level tracing tools (e.g., eBPF, Jaeger) with discrete event simulation

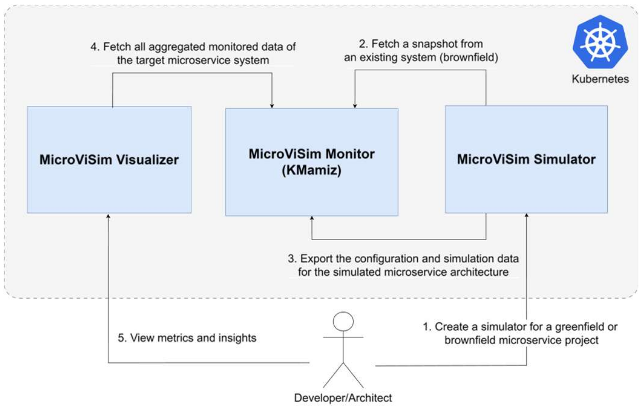
2151-0849/25/$31.00 ©2025 IEEE
DOI 10.1109/ASEW67777.2025.00038

to assess how resource allocation, scheduling policies, and network topologies impact system performance. PerfSim is tailored for cloud-native microservice chains and is suited for performance tuning and capacity planning. EvolutionSim [7] is an extensible simulator designed to evaluate the impact of system evolution strategies, such as autoscaling, load balancing, and overload control. Built on a MAPE-K feedback loop and discrete-event simulation, it models complex service interactions and dynamic user behavior. MiSim, proposed by Frank et al. [10], focuses on resilience analysis by simulating fault injection scenarios without requiring system deployment. It emphasizes runtime behavior by modeling detailed event-driven interactions and common resilience mechanisms using discrete event simulation.

While existing tools offer powerful support for resilience simulation in well-defined systems, they typically assume the presence of detailed runtime data, making them less applicable in early development stages. In contrast, MicroViSim is designed to support both greenfield development and architecture evolution, allowing teams to model and simulate new or modified microservice designs using lightweight configurations without requiring a deployed system, and view a visual comparison of two configurations.

III. MICROVISIM: APPROACH DESCRIPTIONS

A. *System Overview and Operational Concepts*



4. Fetch all aggregated monitored data of the target microservice system
2. Fetch a snapshot from an existing system (brownfield)
Kubernetes
MicroViSim Visualizer
MicroViSim Monitor (KMamiz)
MicroViSim Simulator
3. Export the configuration and simulation data for the simulated microservice architecture
5. View metrics and insights
1. Create a simulator for a greenfield or brownfield microservice project
Developer/Architect

Fig. 1. MicroViSim System Overview

The conceptual architecture of MicroViSim is illustrated in Fig. 1. The system is composed of three main components: the monitor, the visualizer, and the simulator. The monitor component provides real-time monitoring of microservices deployed on Kubernetes and introduces a new feature that allows the current system configuration to be exported for future simulation use. The visualizer provides a user interface that allows developers to view quality metrics, such as total requests, error amount, and latency for all service endpoints, and architectural insights for either a real target system or a simulated system. The simulator component focuses on simulating service structures that have not yet been deployed. By injecting a user-defined simulation configuration, it enables developers to explore architectural feasibility and estimate potential risks before deployment.

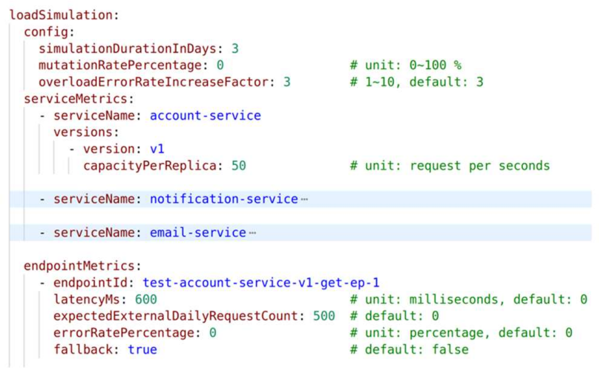
The typical MicroViSim workflow involves four steps: (1) monitoring the deployed system to collect service dependencies and metrics; (2) exporting a snapshot as a simulation configuration; (3) editing the configuration to model new or modified architectures; and (4) running simulations to generate dependency graphs, performance metrics, and reports. This YAML-driven process enables practical validation of architectural changes or prototypes without actual deployment, making it well-suited for early-stage system analysis.

B. *Data Design for Simulation Configuration*

The process of generating simulation data from the configuration file is designed as follows. Administrators may start with an initial configuration exported from the monitoring component and modify it to reflect architectural changes. Alternatively, they may create a configuration file from scratch to simulate a completely new microservice architecture (i.e., a greenfield microservice project). The simulator then generates corresponding simulation data based on the information provided in the configuration. Based on the design concepts, MicroViSim uses a YAML-based configuration file to describe the structure and behavior of the system under test, supporting flexible and repeatable simulations.

TABLE I. FIELDS IN THE LOAD SIMULATION SECTION OF THE CONFIGURATION

Field	Description
simulationDurationInDays	Duration of the simulation in days (affects output range, not real time)
mutationRatePercentage	Daily variability in request volume (as a percentage increase/decrease)
overloadErrorRate increaseFactor	Multiplier for error rate when service capacity is exceeded
serviceMetrics. capacityPerReplica	Processing capacity per replica (requests per second)
endpointMetrics. latencyMs	Average latency per endpoint in milliseconds
endpointMetrics. expectedExternalDaily RequestCount	Expected daily inbound requests from external clients
endpointMetrics. errorRatePercentage	Baseline error rate under normal conditions
endpointMetrics.fallback	Whether the endpoint executes a fallback mechanism when a dependency fails



```
loadSimulation:
  config:
    simulationDurationInDays: 3
    mutationRatePercentage: 0                # unit: 0~100 %
    overloadErrorRateIncreaseFactor: 3       # 1~10, default: 3
  serviceMetrics:
    - serviceName: account-service
      versions:
        - version: v1
          capacityPerReplica: 50             # unit: request per seconds

    - serviceName: notification-service…

    - serviceName: email-service…

  endpointMetrics:
    - endpointId: test-account-service-v1-get-ep-1
      latencyMs: 600                         # unit: milliseconds, default: 0
      expectedExternalDailyRequestCount: 500 # default: 0
      errorRatePercentage: 0                 # unit: percentage, default: 0
      fallback: true                         # default: false
```

Fig. 2. An Example of the Load Simulation Section in the Configuration

The configuration file consists of three major sections: Services Info, Endpoint Dependencies, and Load Simulation. The Services Info section defines service-level metadata, including service name, version, deployment namespace, and number of replicas. Once the service metadata is defined, each service version must specify the list of endpoints it provides. The endpoint definition includes basic path information, supported HTTP methods, and sample input/output formats. The Endpoint Dependencies section defines inter-endpoint call relationships within the system. Each entry maps an endpoint ID to a list of other endpoint IDs on which it directly depends. The Load Simulation section configures the behavior of the simulation engine. It specifies the simulated workload volume, failure rates, and endpoint-specific performance characteristics. This section enables dynamic analysis of the system under varying traffic conditions. Key fields are listed in Table I, with an example configuration shown in Fig. 2.

C. *Load Simulation Mechanism*

MicroViSim simulates request propagation across service endpoints to estimate system behavior under predefined workloads. It models service dependencies as a directed graph, where each node represents an endpoint and edges represent invocation relationships. To traverse this graph, the simulator applies a depth-first search (DFS) strategy, enabling it to simulate how traffic flows between services. During traversal, visited endpoints are marked to avoid infinite loops caused by cyclic dependencies [11].

The simulation models not only simulate request flows but also error propagation behaviors, including support for fallback mechanisms. The logic is as follows: (1) if an endpoint fails due to its own error rate, the request terminates and does not propagate to downstream dependencies; (2) if a downstream endpoint fails, and the upstream endpoint does not have a fallback mechanism, the error is propagated upstream, simulating a cascading failure; and (3) if fallback is enabled, the error is contained, allowing the upstream endpoint to continue processing without failure. Besides, to better simulate real-world-like traffic variation, the system divides each simulated day into hourly time intervals. Incoming requests are distributed randomly across time slots based on each endpoint's configuration. The simulation consists of two phases as follows.

Phase 1: Baseline Request Propagation

In this phase, request flows are simulated using the default (basic) error rate configured for each endpoint. This establishes the baseline load on each service and determines the expected number of successful and failed requests without considering overload effects.

Phase 2: Overload Estimation and Error Adjustment

In this phase, the simulator evaluates whether each service becomes overloaded based on the number of requests it receives and its processing capacity. When overload is detected, the error rate of affected endpoints is adjusted to reflect degraded performance. The following four-step process defines the overload-based error adjustment model:

Step 1: Service Capacity Calculation. The processing capacity of a service in a single time slot is defined as Eq. (1).

$$C = Rep \times CPR \tag{1}$$

- C: Total processing capacity of the service (requests per second)
- Rep: Number of replicas for the service
- CPR: Capacity per replica (i.e., how many requests one replica can process per second)

Step 2: Overload Ratio Calculation. The overload ratio quantifies the degree to which the incoming request rate exceeds the service's capacity, as in Eq. (2).

$$O = \begin{cases} \frac{R}{C} - 1, if\ R > C \\ 0, otherwise \end{cases} \tag{2}$$

- O: Overload ratio
- R: Total number of incoming requests received by the service during the time slot
- C: Service capacity, as calculated in Eq. (1)

Step 3: Overload-Induced Error Rate. When a service is overloaded, its error rate is assumed to rise exponentially according to the overload level, as in Eq. (3).

$$E_{overload} = 1 - e^{-k \cdot O} \tag{3}$$

- $E_{overload}$: Additional error rate caused by overload
- K: Growth factor for error rate increase (default value: 3)
- O: Overload ratio from Eq. (2)

Note that this model assumes that as the overload ratio increases, the error rate approaches 1 (i.e., 100%) exponentially. This is a common exponential growth assumption, which in practice can also be replaced with a user-defined model.

Step 4: Final Error Rate Adjustment. The endpoint's total error rate E is computed as the sum of its base error rate and the overload-induced rate, as in Eq. (4).

$$E = E_{basic} + (1 - E_{basic}) \times E_{overload} \tag{4}$$

- E: Final adjusted error rate under load
- E_{basic}: The baseline error rate under normal (non-overloaded) conditions
- $E_{overload}$: Overload-induced error rate from Equ. (3)

To ensure numerical stability, the implementation enforces an upper bound so that E is less than or equal to 1.0.

Based on the above process, the simulator records the number of successful and failed requests per endpoint per hour. These results are presented in the metrics dashboard for post-simulation analysis and performance evaluation.

IV. Experimental Evaluation

A. *Experimental Objective*

The objective of this experiment is to validate whether MicroViSim can help developers analyze and compare the load-handling behavior of different architectural designs before deployment. Specifically, we aim to (1) simulate workloads on different architectural designs, and (2) identify potential bottlenecks and error-prone components under load.

B. Case Study: Bookinfo System

We evaluated MicroViSim using the Bookinfo system, a well-known sample microservice application consisting of four core services: productpage, details, reviews, and ratings. To test architectural flexibility, we extended the system with two new features: book and review recommendations. Two architectural designs were simulated: (1) Design *A*: A single recommendation service handles both features, and (2) Design *B*: Two services (book-recommendation and review-recommendation) are devised separately.

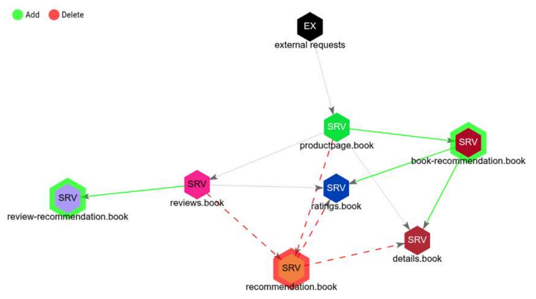
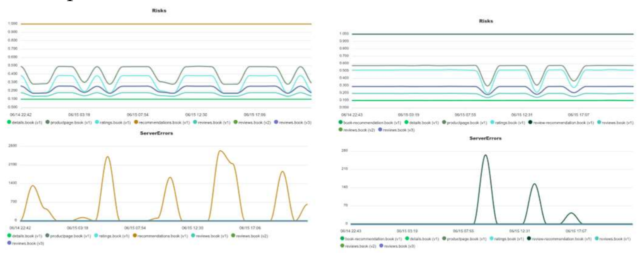


Fig. 3. Service-level dependency comparison diagram of the two simulated Bookinfo designs

The service-level dependency comparison diagram (a functionality provided by MicroViSim) of the two simulated Bookinfo designs is shown in Fig. 3. In this diagram, hexagons labeled SRV represent microservices, while the black hexagon (EX) denotes external requests. Green solid arrows (Add) indicate newly added dependencies, and red dashed arrows (Delete) mark removed ones. Both options were configured with ~10,000 simulated homepage visits per day.

C. Experimental Results and Observations



Simulation results of Design A *Simulation results of Design B*

Fig. 4. Simulation results of Bookinfo for Design A and Design B

Simulation results in MicroViSim (Fig. 4) show clear differences in system behavior under the two architectural designs. In Design A (left side), the unified recommendation service became the main bottleneck, with high overload-induced error rates and elevated risk scores. This indicates that consolidating multiple responsibilities into a single service can compromise scalability and fault isolation.

In contrast, Design B (right side) exhibited more stable performance. The separation of book-recommendation and review-recommendation allowed traffic to be distributed more evenly, reducing the likelihood of overload in individual services. Although the review-recommendation service still experienced some stress under peak load, the modular design localized its impact, enabling targeted optimizations.

These observations demonstrate MicroViSim's ability to identify architectural weaknesses and simulate system behaviors, and provide actionable insights for design decisions.

V. Conclusion

This paper presents MicroViSim, a lightweight and non-intrusive tool for simulating load and visualizing microservice architectures on Kubernetes. By supporting both greenfield and brownfield scenarios, MicroViSim allows developers and operators to assess architectural changes before deployment, helping to identify bottlenecks, evaluate fallback strategies, and make informed design decisions. Through our case study using the Bookinfo system, we demonstrate that the tool can guide architectural improvements under simulated workloads.

Future work will focus on enhancing usability and scalability. We plan to (1) incorporate dynamic traffic patterns and user-behavior models (e.g., as in [11]) to enable more realistic and load-sensitive simulation scenarios, and (2) integrate additional Kubernetes components, such as StatefulSets, Horizontal Pod Autoscaling (HPA), and the Kubernetes Gateway API, to improve fidelity.

Acknowledgment

This research was sponsored by the National Science and Technology Council (NSTC) in Taiwan under the grants 112-2221-E-019-021-MY3 and 114-2221-E-006-094-MY3.

References

[1] S. Newman, *Building microservices*. " O'Reilly Media, Inc.", 2021.
[2] Y.-T. Wang, S.-P. Ma, Y.-J. Lai, and Y.-C. Liang, "Qualitative and quantitative comparison of Spring Cloud and Kubernetes in migrating from a monolithic to a microservice architecture," *Service Oriented Computing and Applications,* vol. 17, no. 3, pp. 149-159, 2023.
[3] G. He *et al.*, "Guardian of the Resiliency: Detecting Erroneous Software Changes Before They Make Your Microservice System Less Fault-Resilient," in *2024 IEEE/ACM 32nd International Symposium on Quality of Service (IWQoS)*, 19-21 June 2024 2024, pp. 1-10.
[4] A. Basiri *et al.*, "Chaos Engineering," *IEEE Software,* vol. 33, no. 3, pp. 35-41, 2016.
[5] H. Tucker, L. Hochstein, N. Jones, A. Basiri, and C. Rosenthal, "The Business Case for Chaos Engineering," *IEEE Cloud Computing,* vol. 5, no. 3, pp. 45-54, 2018.
[6] 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, 2021.
[7] 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)*, 2023: IEEE, pp. 43-49.
[8] Y.-T. Wang, S.-P. Ma, Y.-J. Lai, and Y.-C. Liang, "Analyzing and monitoring kubernetes microservices based on distributed tracing and service mesh," in *2022 29th Asia-Pacific Software Engineering Conference (APSEC)*, 2022: IEEE, pp. 477-481.
[9] 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: IEEE, pp. 1-8.
[10] S. Frank, L. Wagner, A. Hakamian, M. Straesser, and A. v. Hoorn, "MiSim: A Simulator for Resilience Assessment of Microservice-Based Architectures," in *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, 5-9 Dec. 2022 2022, pp. 1014-1025, doi: 10.1109/QRS57517.2022.00105.
[11] M. Camilli and B. Russo, "Modeling performance of microservices systems with growth theory," *Empirical Software Engineering,* vol. 27, no. 2, p. 39, 2022.