Supporting Architecture-Level Resilience Analysis with an Integrated Chaos and Load Experimentation Framework

Sandro Speth
School of Computation, Information, and Technology
Technical University of Munich, Munich, Germany
sandro.speth@tum.de

Elias Müller, Niklas Meißner,
Niklas Krieger, Steffen Becker
Institute of Software Engineering
University of Stuttgart, Stuttgart, Germany
{firstname.lastname}@iste.uni-stuttgart.de

***Abstract*—Chaos Engineering (CE) and Software Performance Engineering (SPE) are increasingly important for validating the resilience and performance of cloud-native microservice systems. Yet, existing CE tools remain limited by environment-specific assumptions, narrow fault models, or substantial manual integration with load testing, resulting in high overhead, a high required level of technical expertise, and low reproducibility. To address these shortcomings, we present CERES, a system-independent and infrastructure-agnostic framework that unifies configurable chaos injection, structured workload generation, and automated observability in a coordinated experimentation workflow. While the tool is independent of any specific system under test, we complement it with MiSArch, a realistic microservice reference architecture that enables reproducible evaluations and supports research on architectural resilience. We evaluate the tool through a usability study with researchers and industry engineers, as well as through empirical experiments using MiSArch that explore performance limits under varying load and failure scenarios and allow the derivation of initial service-level objective boundaries. The results demonstrate that the tool substantially reduces the effort required to design, execute, and analyze combined CE and SPE experiments. Researchers and DevOps engineers benefit from a reusable, accessible, and system-independent experimentation framework that lowers the barrier to systematic and reproducible resilience assessments.**

***Index Terms*—Software Performance Engineering, Chaos Engineering, Load Testing, Reference Architecture**

I. INTRODUCTION

Cloud-native microservice systems require architecture-level evaluation under realistic load and controlled faults to assess resilience and performance. Accordingly, Software Performance Engineering (SPE) and Chaos Engineering (CE) are central methods in modern quality evaluation [1], [2]. However, current practice remains fragmented: many CE tools are infrastructure-specific, fault coverage is often limited, and coordinated load/fault/observability workflows usually require manual integration [3], [4]. The resulting overhead and tooling complexity hinder reproducible research and increase expertise barriers.

This motivates a system-independent framework that unifies workload modeling, fault injection, and synchronized execution with integrated observability. This results in our first research question:

RQ1 (Framework Design): How can a generalized, system-independent experimentation framework be architected to support customizable failure injection, workload modeling, observability, and synchronized execution across diverse deployment environments without requiring a large technical expertise in various CE and SPE tools?

To address this challenge, we present CERES (Chaos Engineering & Resilience Evaluation System)[1], an infrastructure-agnostic framework for coordinated CE+SPE experiments. Its design is grounded in structured requirements elicitation with practitioners.

A second challenge is the lack of realistic and modern reference architectures for reproducible resilience/performance studies. Popular benchmarks (e.g., Sock Shop[2], Online Boutique[3]) are useful but often limited in realism, heterogeneity, or communication diversity. We therefore introduce *MiSArch*[4], a heterogeneous 16-loosely-coupled-service reference architecture with synchronous and asynchronous communication.

For *MiSArch*, key operational boundaries under load/fault combinations are still unclear, especially SLO ranges under representative infrastructure conditions. This results in our second research question:

RQ2 (Reference Architecture Evaluation): How does the *MiSArch* reference architecture behave under combined load and failure scenarios, and what SLO boundaries can be empirically derived using the proposed framework?

We evaluate CERES through a usability study with researchers/practitioners and through controlled experiments on *MiSArch*. Results indicate reduced orchestration effort for CE+SPE studies and support derivation of initial *MiSArch* SLO baselines.

Our contributions are: (1) CERES as an infrastructure-agnostic CE+SPE experimentation framework, (2) *MiSArch* as a heterogeneous reference architecture for architecture-

[1] https://misarch.github.io/docs/docs/user-manuals/how-to-use-ceres
[2] https://github.com/microservices-demo/microservices-demo
[3] https://github.com/GoogleCloudPlatform/microservices-demo
[4] https://github.com/MiSArch

level studies, and (3) an empirical evaluation of usability and technical effectiveness including SLO derivation.

The remainder is structured as follows: Section II summarizes background and related work; Section III presents CERES; Section IV introduces *MiSArch*; Section V reports evaluation results; and Section VI concludes.

II. BACKGROUND AND RELATED WORK

Architecture-level resilience/performance evaluation in microservices builds on SPE, CE, and reference architectures. We summarize each area and identify remaining gaps.

A. Software Performance Engineering and Load Testing

SPE combines model-based and measurement-based methods for achieving performance quality goals [2], [5]. In practice, load generators such as Gatling[5], k6[6], JMeter[7], and Locust[8] enable controlled workload execution, with Gatling being particularly suited for reproducible scripted studies.

Research extends these tools with workload derivation and automation (e.g., BenchFlow [6], trace-based synthesis [7], domain-story-driven scenario generation [8]). However, these approaches generally do not provide first-class orchestration with chaos fault injection, so coordinated CE+SPE setups remain largely manual.

B. Chaos Engineering Tools and Methodologies

CE complements SPE by injecting controlled faults into running systems to validate resilience under disruption [3]. Early tools (e.g., Netflix's Chaos Monkey[9] and the wider Simian army) focused on production settings and supported only narrow failure models (e.g., instance termination, latency injection), thus they offered limited fault models and strong platform dependencies. Newer tools broaden capabilities but still tend to be platform-scoped (e.g., Chaos Mesh[10] for Kubernetes, Pumba for Docker [4], Toxiproxy for network perturbations[11]).

Chaos Toolkit[12] provides strong extensibility via provider plugins, but coordinated scheduling with structured load generation and unified observability still requires additional orchestration. Thus, reproducible architecture-level CE+SPE studies remain hard to set up end to end.

In summary, existing CE tooling faces three main limitations: (1) restricted and heterogeneous fault models, (2) strong coupling to specific infrastructures, and (3) no unified orchestration for coordinating CE with SPE. These gaps present challenges for reproducible, architecture-level experimentation.

[5]https://gatling.io
[6]https://k6.io
[7]https://jmeter.apache.org
[8]https://locust.io
[9]https://github.com/netflix/chaosmonkey
[10]https://chaos-mesh.org/docs/
[11]https://github.com/Shopify/toxiproxy
[12]https://chaostoolkit.org

C. Microservice Reference Architectures

Microservice reference architectures provide realistic yet controlled environments for evaluating architectural decisions and quality attributes [9]. Widely used systems, such as Teastore [10], Sock Shop, Online Boutique, and Train Ticket, support basic experimentation but suffer from limitations, including outdated technologies, homogeneous technology stacks, synchronous-only communication, and insufficient observability support [9]. These characteristics constrain their realism and reduce comparability across studies. Furthermore, external tooling and orchestration remain necessary for realistic resilience evaluations. Moreover, reference architectures typically do not include predefined SLOs derived from systematic experimentation, leaving researchers without established performance baselines. To address some of these deficiencies, advanced reference architectures utilizing heterogeneous technology stacks and enhanced observability are necessary.

D. Synthesis

Across SPE, CE, and microservice reference architectures, three key gaps emerge: (1) the absence of infrastructure-agnostic and configurable failure injection, (2) the lack of coordinated orchestration that unifies load generation, failure injection, and observability, and (3) the limited realism and configurability of existing reference architectures for architecture-level experimentation. These gaps motivate CERES, which integrates state-of-the-art load generation, observability, and CE tooling into a unified, system-independent experimentation framework.

III. CERES

In this section, we describe the requirements and concept for an integrated chaos and load experiment framework. We implemented a prototype of this framework for our evaluation.

A. Requirements Engineering

To address RQ1 and identify the capabilities that a CE and SPE experimentation framework must provide, we conducted a structured requirements engineering activity prior to designing CERES. We used a lightweight Volère-inspired process [11] and elicited domain knowledge through semi-structured interviews with practitioners in microservice development, operations, and performance engineering.

Domain knowledge was elicited through semi-structured interviews with practitioners experienced in microservice development, cloud operations, and performance engineering. The interviews focused on participants' existing practices for conducting load and resilience tests, the challenges encountered when combining CE and SPE techniques, and the limitations of current tooling with respect to expressiveness, reproducibility, and integration effort.

The most relevant requirements that emerged from this process concern the need for a unified experiment model that encapsulates workload definitions, chaos scenarios, and observability configuration in a single, coherent specification; reliable orchestration mechanisms that guarantee the correct

temporal coordination of workload execution and fault injection; and infrastructure independence to ensure that experiments can be executed on different microservice systems without redesign. Practitioners further emphasized the importance of extensibility, enabling the integration of additional failure models and workload generators, as well as reproducibility, requiring deterministic experiment execution and automated data collection. Finally, the requirements highlighted the need for consolidated experiment reporting, allowing users to analyze latency distributions, resource behavior, and failure propagation without having to assemble results from multiple tools. We provide a full transcript of the requirements [12].

The resulting set of requirements informed the architectural design of CERES and shaped the composite experiment model, orchestration semantics, and extensibility mechanisms. By grounding the design in practitioner experience and by employing a structured elicitation method, the requirements engineering process ensured that CERES addresses the fundamental gaps identified in existing CE and SPE practices.

B. Chaos and Performance Experimentation

CERES is organized around the nine-step workflow in Figure 1. The central artifact is a versioned experiment descriptor that unifies workload, failure scenarios, goals, and observability settings. This descriptor enables scriptable and reproducible CE+SPE runs while keeping the framework independent of specific tooling.

Execution starts when a user creates and launches an experiment (1). The executor parses the descriptor and dispatches workload and work definitions to the load component and failure definitions to the chaos component (2, 3). Because the executor operates on a technology-independent schema, providers can be exchanged or extended without changing the experiment model.

Conceptually, the experiment descriptor represents one coordinated hypothesis test over the system under test: a defined workload profile, a defined fault profile, explicit goals, and a synchronized observation window. Operationally, descriptors are versioned configuration artifacts grouped by test identifier and version, which allows controlled iteration without changing framework internals. This distinction between conceptual experiment definition and concrete executable artifacts proved useful for reproducibility and traceability across repeated runs.

Load and chaos providers perform the core actions (4, 5). For load generation, CERES uses an open workload model (e.g., via Gatling adapters) and explicitly separates *work* (request sequence) from *load* (arrival dynamics). This supports both realistic traces and stress-oriented profiles (Figure 2).

In parallel with workload execution, the chaos executor injects the defined failure modes using an infrastructure-agnostic CE tool integrated via the chaos component. Failures can target different architectural layers, including container environments, sidecar proxies, or application-level endpoints, and support types such as container crashes, network delays, and resource exhaustion. The framework allows both deterministic and non-deterministic failure specifications, as required by the elicited requirements. CERES' executor ensures that failures are injected in accordance with the timing and sequencing constraints expressed in the experiment descriptor and that any configured warm-up or steady-state measurement phases are completed before chaos is applied. This coordination preserves controlled experimental conditions and enables multi-phase scenarios in which workload intensity and failure patterns can evolve over time.

During execution, observability continuously collects platform and application telemetry (6) for the system under test, aligned with the experiment timeline for causal analysis of workload/failure interactions. Optionally, CERES performs dedicated steady-state runs to infer baselines and derive threshold-oriented goals.

In our design, this temporal alignment is essential: faults should not be interpreted without workload context, and workload effects should not be interpreted without concurrent infrastructure and application telemetry. Therefore, all data streams are captured against the same experiment timeline and persisted with experiment metadata. This supports later reconstruction of phase transitions and fault effects and enables direct comparison between baseline and faulted runs of the same experiment definition.

After completion, user-centric load metrics (throughput, latency, error rates) are merged with platform telemetry (7, 8) and exposed through one dashboard (9). Versioned executions are persisted with configuration metadata to support reproducible comparisons. Overall, this workflow realizes RQ1 by reducing cross-tool coordination effort while preserving analytical depth.

C. Analysis Result User Interface

A central requirement of CERES is to make complex experiment configurations and results accessible without requiring deep expertise in the underlying CE and SPE tools. To achieve this, the framework provides two complementary user-facing interfaces: a web-based frontend for configuring and managing experiments, and automatically generated dashboards for inspecting the results of experiment executions. Together, these interfaces form the primary interaction surface for researchers and complete the end-to-end workflow depicted in Figure 1.

The frontend exposes the experiment abstraction discussed above and offers dedicated views for work, load, failure, and goal configurations. Researchers can create new experiments using preconfigured templates corresponding to realistic or quality-attribute-oriented load patterns (Figure 2) and can adjust work scenarios to match the domain-specific behavior of their system under test (SUT). Work and load are deliberately separated in the UI, helping users reason about system behavior (what the SUT should do) and stress intensity (how much load the SUT should experience). Failure scenarios for different failure providers can be edited through simplified forms or directly through raw configuration editors (e.g., JSON), allowing both novice and expert users to interact with the same underlying artifacts. Optional warm-up and steady-state measurement phases can also be configured through

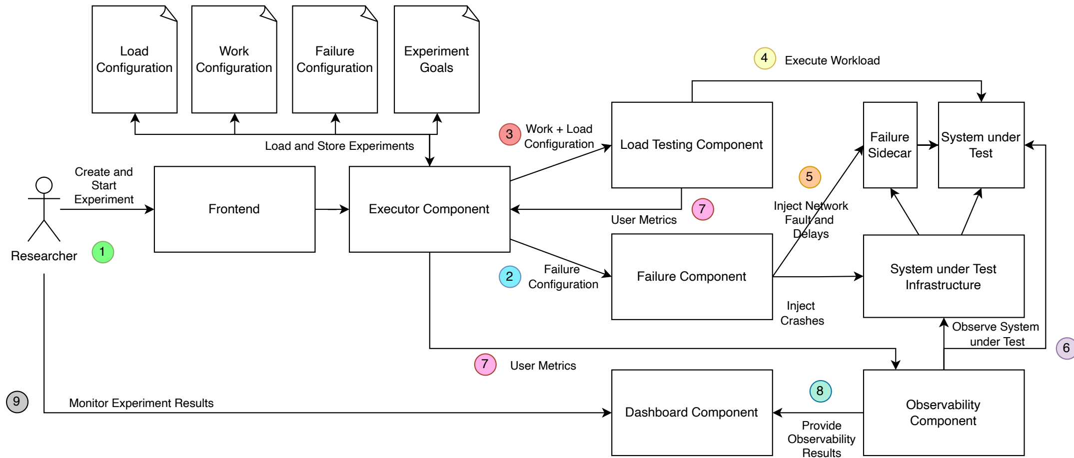

Fig. 1. Overview of the integrated performance and chaos engineering with CERES.

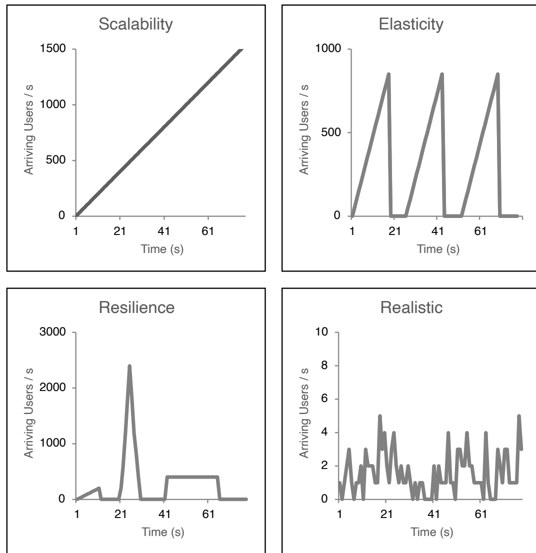

Fig. 2. Experiment Load Patterns.

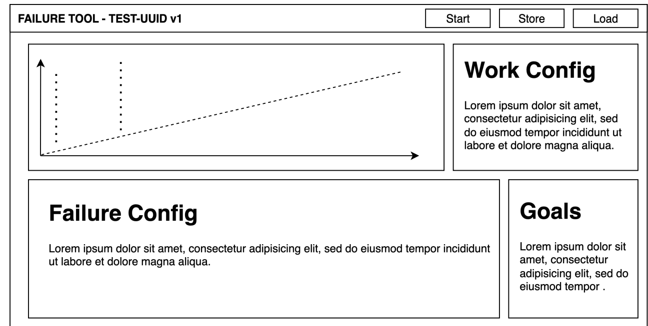

Fig. 3. Mock-Up of the Frontend.

dedicated dialog panels. The frontend is stateless, with all modifications stored through the *Experiment Executor* and versioned separately for each experiment. A mock-up of this interface is shown in Figure 3.

A key element of the frontend is the experiment graph, which visualizes the approximated number of arriving users and request rates derived from the configured load and work scenarios. Vertical markers highlight the execution points of failures from different providers, enabling users to foresee the interaction between workload phases and injected disruptions. The graph updates dynamically when configuration parameters change, providing an immediate and intuitive representation of the expected execution profile prior to running an experiment.

Once an experiment execution completes, CERES automatically generates a parameterized dashboard for the corresponding Test-UUID and version. The dashboard consolidates user-centric metrics collected by the load testing tool, stored in a time-series database such as InfluxDB[13] after parsing, along with system-level telemetry retrieved from a monitoring stack like Prometheus[14]. In our evaluation, Prometheus ingests OpenTelemetry-compliant application metrics, sidecar-level metrics, and container-level resource measurements from cAdvisor, but the framework does not mandate a specific SUT or technology stack. However, when using other signals, the researchers need to manually adapt dashboards to their purposes. By organizing these metrics along the experiment timeline and annotating workload and failure events, the dashboard enables causal reasoning about performance degradation and resilience behavior.

The dashboard is structured into sections that reflect different analytical perspectives: a summary of overall request throughput, response times, and active users; a view that highlights request-level errors; detailed per-request metrics such as percentiles and success ratios; and service-level metrics visualizing CPU, memory, and internal latencies for individual services of the SUT. Metrics related to experiment goals are visually flagged when thresholds are violated. To support

[13]https://docs.influxdata.com/influxdb/v2/
[14]https://prometheus.io

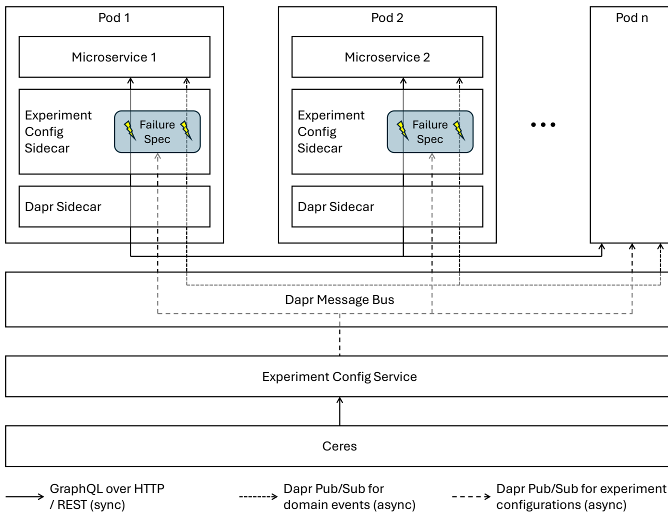

Fig. 4. Overview of *MiSArch* experiment configuration support and integration with CERES.

architectural evaluation, the dashboard also enables side-by-side comparison of executions by replaying metric queries with shifted time windows, allowing researchers to contrast different experiment versions or repeated runs under varying conditions.

By integrating sophisticated configuration capabilities, dynamic visual guidance during experiment design, and comprehensive post-execution dashboards, the analysis user interface significantly reduces the operational complexity of running CE and SPE experiments. It also provides the basis for the empirical investigations required for RQ2, enabling reproducible interpretation of SUT behavior under combined workload and failure scenarios, independent of any specific reference architecture.

IV. MISARCH

MiSArch is a heterogeneous, polyglot microservice reference architecture designed to support research on performance, resilience, and architectural evolution in distributed systems. Its primary purpose is to provide a realistic and reproducible environment for experimentation while remaining modular enough to isolate the architectural phenomena of interest. The system comprises 16 microservices, a GraphQL gateway, and a frontend. These services are intentionally implemented in multiple languages – Node.js (2), Kotlin (8), and Rust (6) – to reflect the technological diversity typical of contemporary microservice systems.

A GraphQL gateway built with GraphQL Mesh exposes a unified API and federates individual services using Apollo Federation 2. The architecture systematically applies the DataLoader pattern to prevent the n+1 query problem and to ensure predictable performance across federated boundaries. Authentication and authorization are handled by Keycloak, while a standalone frontend interacts with both the gateway and the identity provider.

MiSArch supports both Docker-based and Kubernetes-based deployments. For Kubernetes, the infrastructure is provisioned declaratively using Terraform, enabling reproducible cluster setups suitable for controlled experimentation. Observability is embedded throughout the architecture. Metrics are collected via Prometheus, traces are exported through Zipkin, and both are processed by a centralized OpenTelemetry Collector, ensuring consistent telemetry across languages, frameworks, and runtime environments.

All inter-service communication is routed through the Distributed Application Runtime (Dapr).[15] Synchronous interactions use GraphQL over HTTP via Dapr's service invocation API, predominantly between the gateway and domain services. Asynchronous communication is realized through Dapr Pub/Sub, backed by RabbitMQ, enabling loosely coupled event-driven workflows for entity lifecycle changes. The persistence layer is intentionally heterogeneous: PostgreSQL is used by Keycloak and several domain services, MongoDB supports other domain-oriented components, and Redis provides state storage for Dapr components.

To support systematic experimentation, *MiSArch* provides dedicated mechanisms for controlled failure injection and workload perturbation. Each microservice is accompanied by an experiment-configuration sidecar through which all service traffic is routed. In Kubernetes deployments, these sidecars run as separate containers within the same pod. They enable targeted degradation of service behavior by imposing artificial CPU load, memory exhaustion, delays, and probabilistic faults on both synchronous service invocations and asynchronous event traffic. Because all communication flows through Dapr and then through the sidecar layer, perturbations can be introduced consistently across protocol boundaries.

Experiment definitions are maintained by a dedicated configuration service exposing a REST API for creating, updating, and distributing failure configurations. A lightweight frontend further supports the management of experiment scenarios. Additionally, *MiSArch* includes a simulation service capable of generating realistic external signals, such as shipment or payment events, to mimic real-world asynchronous workflows and drive the system into representative states before or during experiments.

CERES integrates with *MiSArch* by invoking the experiment configuration service as part of its experiment model. Through this integration, failure definitions can be expressed and executed using the same abstraction that governs workload specification, chaos injection, and observability in CERES. This allows *MiSArch* to serve as a comprehensive evaluation platform for RQ2, while CERES itself remains system-independent and capable of orchestrating experiments on other systems under test through the Chaos Toolkit integration.

[15]https://dapr.io/

V. EVALUATION

This section evaluates CERES from two perspectives: usability (RQ1) and technical effectiveness for coordinated CE+SPE experimentation and SLO derivation on *MiSArch* (RQ2). We provide our replication package on a public archive.[16]

A. Usability Evaluation

We evaluated perceived usefulness and usability of CERES with participants experienced in microservices and CE/SPE practice.

1) Design: The study followed a task-based protocol combined with think-aloud observations. Ten participants took part: six from academia and four from industry, all with prior experience in SPE, CE, or microservice engineering. CERES was deployed in a Kubernetes environment and evaluated with *MiSArch* as system under test.

After a short onboarding, participants performed a guided but non-trivial experiment task. They had to modify an existing experiment by adapting work/load settings, defining a fault scenario, executing the run, and interpreting the generated dashboard output. We intentionally used a modification task instead of a pure replay task so that participants had to reason about both configuration semantics and execution behavior.

Participants could use integrated help pages during execution. In several sessions, one author observed interaction behavior to document misunderstandings, navigation problems, and cognitive load indicators.

Data collection combined: (1) closed Likert-scale items on functionality and usability, (2) SUS [13], and (3) open feedback on strengths, confusing aspects, and missing features. This mixed design allowed us to relate quantitative scores to concrete interaction problems.

2) Results: All participants completed the task, indicating that CERES is operationally suitable for coordinated CE+SPE workflows. Likert responses in Section V-A1 were positive for the core activities: experiment creation (median 4.0), combining load and chaos (median 4.0, mean 4.4), execution (median 4.0), and understanding load/failure interplay (median 4.0, mean 4.4). Participants also reported that they could configure the requested scenario (median 5.0), suggesting that the experiment descriptor is expressive enough for realistic study tasks.

Lower ratings were observed for feedback and guidance during task execution (median 3.0) and confidence for use in real projects (median 3.5). These results indicate that the conceptual workflow is understandable, but UI guidance and validation can be strengthened.

SUS scores ranged from 57.5 to 75.0, with a mean of 68.5. Following Bangor et al. [14], this lies at the lower boundary of "Good" usability: acceptable, but with clear room for improvement.

Qualitative comments aligned with these findings. Participants highlighted experiment visualization, versioning, and dual graphical/code editing as strong points. Reported issues included fragmented navigation, insufficient validation and syntax support, naming ambiguities, and dashboard interpretation challenges (e.g., metric semantics and relative time understanding).

Taken together, the results suggest that the main conceptual model is understandable and useful, while remaining friction points are concentrated in interaction design. In other words, the limiting factor is less the underlying experiment abstraction and more how clearly the UI communicates it during configuration and interpretation.

3) Discussion: The usability findings provide evidence for RQ1: users with limited onboarding were able to complete an end-to-end CE+SPE experiment cycle using one framework. Positive ratings for configuration and execution suggest that the unified descriptor and synchronized workflow are comprehensible in practice.

At the same time, lower guidance-related ratings indicate that practical adoption depends on further UI refinement. Most issues concern interaction design rather than conceptual model validity. Consequently, improvements in contextual guidance, validation, and dashboard explainability are likely to increase confidence without requiring changes to the architectural core of CERES.

4) Threats to Validity: Internal validity: participants received brief onboarding; we mitigated this via think-aloud observation and realistic tasks instead of speed-focused measures.

Construct validity: tasks covered core CE+SPE activities, but not all possible workflows; subjective interpretation of qualitative feedback remains a potential bias source.

External validity: the sample size was small and domain-specific, and evaluation occurred in a controlled setup; broader industrial studies are needed.

Conclusion validity: we report descriptive patterns without inferential statistics; triangulation across observations and questionnaire responses improves credibility.

B. Effectiveness Evaluation

This subsection addresses RQ2 by evaluating technical effectiveness: reproducible orchestration of CE+SPE experiments and derivation of SLO boundaries for *MiSArch*.

1) Design and Experimental Setup: We ran empirical experiments on *MiSArch* deployed on Kubernetes. Each service used one pod and horizontal scaling was disabled to keep a fixed-capacity baseline.

For each scenario, we established steady state without faults, then increased workload intensity using an open workload model (Gatling). Fault scenarios were injected through Chaos Toolkit integration in CERES, including container termination, CPU/memory exhaustion, and network perturbations (latency, packet loss), also in multi-phase combinations.

This setup was chosen to maximize internal control for architecture-level analysis. By fixing scaling behavior, observed degradations can be attributed more directly to workload-fault interaction rather than autoscaling dynamics.

[16] https://anonymous.4open.science/r/ceres-replication-package-2D4E/README.md

TABLE I
SUMMARY OF THE USABILITY QUESTIONNAIRE RESPONSES. STRONGLY DISAGREE (1) - STRONGLY AGREE (5)

Question	Median/Mean/Mode	Std	Min/Max/Range	25%	50%	75%
Q1: I found the process of creating an experiment intuitive.	4 / 3.9 / 4	0.57	3 / 5 / 2	4	4	4
Q2: The software made it easy to combine load and chaos testing.	4 / 4.4 / 4	0.52	4 / 5 / 1	4	4	5
Q3: Executing the experiment was straightforward.	4 / 4.3 / 4	0.67	3 / 5 / 2	4	4	5
Q4: The software provided sufficient feedback and guidance during the task.	3 / 3.2 / 3	0.63	2 / 4 / 2	3	3	3.75
Q5: I understood the relationship between load testing and chaos injection.	4 / 4.4 / 4	0.52	4 / 5 / 1	4	4	5
Q6: I was able to configure the experiment the way the task requested.	5 / 4.5 / 5	0.85	3 / 5 / 2	4.25	5	5
Q7: I think the tool combines all necessary features for creating load and chaos experiments.	4 / 4.0 / 5	1.05	2 / 5 / 3	3.25	4	5
Q8: I would feel confident using this software in a real project.	3.5 / 3.6 / 3	0.70	3 / 5 / 2	3	3.5	4

At the same time, scenario design remained realistic by combining resource and communication disturbances in staged timelines, which approximates operational stress conditions more closely than isolated single-fault tests.

CERES synchronized workload timing, fault timing, and telemetry capture (Prometheus/OpenTelemetry). Configurations were executed repeatedly to assess reproducibility.

The scenario set included both isolated and compound disturbances. Isolated scenarios establish interpretable baseline effects for each fault type; compound scenarios capture interaction effects that are common in production incidents. This combination was necessary to evaluate whether CERES can express and execute experiments that are both analytically tractable and practically realistic.

The multi-phase scenarios were selected to stress interactions rather than isolated fault effects, for example by injecting network degradation after sustained load growth or combining resource pressure with communication disturbances. This design reflects realistic operational stress patterns in distributed systems and provides stronger evidence for architecture-level behavior under compound conditions.

2) Results: Across repeated runs, CERES maintained stable temporal alignment of workload phases and fault events; metric variance was limited to expected platform noise. This indicates reproducible orchestration behavior under the evaluated conditions.

The unified experiment model expressed both simple and multi-stage scenarios. Resulting traces and metrics supported bottleneck localization, propagation analysis, and recovery observation.

Combined scenarios exposed effects not visible in isolated tests. In particular, CPU exhaustion plus network latency produced stronger tail-latency degradation than either fault alone. Asynchronous paths were more robust to short network perturbations than synchronous request–response paths, which showed sharper latency and error increases.

These observations are relevant for architecture evaluation because they reveal where local degradations become global service-quality issues. They also show that CE and SPE should not be treated as independent analyses when deriving resilience-oriented performance expectations.

Overall, the results indicate that CERES does not only automate experiment execution, but also supports meaningful interpretation at architecture level: event timing, workload phases, and telemetry can be analyzed in one coherent timeline. This is critical for identifying causal hypotheses and for comparing repeated runs or experiment variants.

3) Derived MiSArch SLOs: We used these experiments to derive initial SLO boundaries for *MiSArch* as reproducible research baselines (not production guarantees).

Without injected faults, load ramp-up revealed a stable region and a degradation region. *MiSArch* sustained about nine arriving users per second while keeping error rate below 1%. In this region, read-heavy endpoints showed stricter latency limits (e.g., around 150 ms at p95), while write-heavy endpoints tolerated higher but bounded p95 latencies (on the order of seconds). Across endpoints, observed latencies ranged from roughly 100 ms (p75) to about 4000 ms (p99) under high but acceptable load.

Under injected faults, moderate disturbances (e.g., brief restart/latency events) generally preserved these ranges with increased tails. More severe compound failures (especially resource exhaustion plus network degradation) pushed some endpoints beyond acceptable latency/error thresholds. Thus, CERES supports both baseline SLO derivation and robustness assessment of those SLOs under adverse conditions.

Because experiments are defined in CERES and published in the replication package, these SLO profiles can serve as reusable benchmarks for comparative studies.

Importantly, these SLOs are intended as experimentally grounded reference values for research comparability. They provide a baseline for future work that evaluates architectural changes in *MiSArch*, alternative deployment decisions, or different CE/SPE orchestration tooling under similar conditions.

4) Discussion: Overall, the effectiveness study indicates that CERES fulfills its intended role: it orchestrates realistic CE+SPE experiments reproducibly and provides data suitable for architecture-level analysis and SLO derivation. The derived boundaries should be interpreted as setup-specific reference values, since evaluation used one non-scaling *MiSArch* deployment and a bounded scenario set.

Within these limits, the study still provides a reproducible baseline for subsequent comparisons across architecture revisions, deployment configurations, and future tooling extensions. This is particularly valuable for cumulative research, where comparable experiment definitions and timelines are required to attribute observed changes to architectural decisions.

5) Threats to Validity: Internal validity: platform scheduling noise may affect fine-grained timings; repeated runs and fixed replica/resource settings reduce this risk.

Construct validity: we used reproducible timing, sequencing, and telemetry consistency as orchestration indicators; other constructs may complement these measures.

External validity: findings come from one non-autoscaling deployment and a limited scenario set; broader systems and scaling regimes are needed for stronger generalization.

Conclusion validity: we report consistent descriptive trends without inferential statistics; claims should be interpreted accordingly.

VI. Conclusion and Future Work

This paper presented CERES, a system-independent and infrastructure-agnostic framework for conducting coordinated Chaos Engineering (CE) and Software Performance Engineering (SPE) experiments, and *MiSArch*, a heterogeneous microservice reference architecture for evaluation. Addressing RQ1, we designed a unified experiment abstraction that combines workload modeling, configurable failure injection, observability, and synchronized execution. The architecture of CERES enables the deterministic orchestration of load and chaos scenarios, consolidating experiment results into a single analysis interface. This reduces the effort and tool-specific expertise typically required for CE and SPE workflows.

To address RQ2, we applied CERES to *MiSArch* and derived empirically grounded SLO boundaries for a representative deployment. The experiments demonstrated that CERES reliably coordinates load and faults across repeated runs, exposing resilience characteristics that cannot be observed through load testing or fault injection alone. The resulting SLO profiles for *MiSArch* provide a reusable benchmark for future architecture-level studies and tool comparisons.

The usability evaluation with researchers and practitioners indicates that CERES is practical and learnable: participants were able to configure and execute realistic experiments with limited introduction, rated the integration of load and chaos testing positively, and appreciated the unified experiment descriptor and dashboards. At the same time, the SUS results and qualitative feedback highlight potential areas for improvement regarding guidance, UI coherence, and configuration support.

Future work includes refining the user interface and help mechanisms, introducing higher-level scenario templates for complex experiments, and extending the set of supported failure models and providers. Further evaluations in elastic, autoscaling environments and longitudinal industrial case studies would strengthen external validity, while additional experiments on alternative systems and configurations would advance *MiSArch* as a broadly useful benchmark for resilience and performance research.

Beyond these next steps, we see two concrete implications for the community. For researchers, the combination of a unified experiment model and a heterogeneous reference architecture enables more cumulative studies: experiment definitions, perturbation schedules, and evaluation criteria can be reused across architecture variants, making comparisons less dependent on undocumented integration scripts. For practitioners, the same approach supports earlier resilience-oriented performance evaluation during architecture design, not only in late-stage operations. By testing workload-fault interactions with explicit goals and synchronized telemetry, teams can identify brittle service interactions and revise architecture decisions before incidents materialize in production.

Acknowledgment (AI Disclosure)

The authors used ChatGPT to create a fluent version of an initial manual written text. Grammarly was used to further improve spelling, grammar, and readability. All technical ideas, analyses, results, and conclusions in this paper were conceived, developed, and verified solely by the authors. The authors take full responsibility for the content of the final manuscript.

References

[1] C. Bennett and A. Tseitlin. (2012, July) Chaos monkeys released into the wild. [Online]. Available: http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html

[2] M. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," in *Future of Software Engineering (FOSE '07)*, 2007, pp. 171–187.

[3] A. Basiri, N. Behnam, R. De Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.

[4] H. Jernberg, P. Runeson, and E. Engström, "Getting started with chaos engineering-design of an implementation framework in practice," in *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020, pp. 1–10.

[5] C. U. Smith, "Software performance engineering," in *IFIP International Symposium on Computer Performance Modeling, Measurement and Evaluation*. Springer, 1993, pp. 509–536.

[6] A. Avritzer, V. Ferme, A. Janes, B. Russo, H. Schulz, and A. van Hoorn, "A quantitative approach for the assessment of microservice architecture deployment alternatives by automated performance testing," in *European Conference on Software Architecture*. Springer, 2018, pp. 159–174.

[7] H. Schulz, "Automated generation of tailored load tests for continuous software engineering," Ph.D. dissertation, University of Stuttgart, 2021.

[8] H. Möllers, "Domain-driven resilience testing of business-critical application systems," Master's thesis, University of Hamburg, 2024.

[9] S. Speth, S. Stieß, and S. Becker, "A saga pattern microservice reference architecture for an elastic slo violation analysis," in *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, 2022, pp. 116–119.

[10] 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 *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, Sep. 2018, pp. 223–236.

[11] J. Robertson and S. Robertson, *Volere Requirements Specification Template*, Atlantic Systems Guild, August 2007. [Online]. Available: https://www.volere.org/templates/volere-requirements-specification-template/

[12] S. Anonymous, "Customizable Chaos Experiment Framework for the MiSArch Microservice Reference Architecture," Master's thesis, University of Anonymous, 2025.

[13] J. Brooke *et al.*, "Sus: A 'quick and dirty' usability scale," *Usability Evaluation in Industry*, vol. 189, no. 194, pp. 4–7, 1996.

[14] A. Bangor, P. Kortum, and J. Miller, "Determining what individual sus scores mean: Adding an adjective rating scale," *Journal of usability studies*, vol. 4, no. 3, pp. 114–123, 2009.