

A Configurable Microservice Testbed for Runtime Experiments

Maria C. Borges

`mb@ise.tu-berlin.de`

Information Systems Engineering,
Technische Universität Berlin

Sebastian Werner

`werner.sebastian@tum.de`

Information Systems Engineering,
Technische Universität München

Abstract

We present a configurable microservice testbed for evaluating alternative deployment technologies in realistic cloud-native environments. The testbed extends the OnlineBoutique microservice application to support selectively deploying individual services as either Docker containers or WebAssembly modules. It enables different deployment variants while keeping the application logic and workloads unchanged, providing a common and reproducible basis for systematic experiments on alternative microservice runtimes.

1 Introduction

Cloud-native applications are typically built as collections of loosely coupled microservices that can be developed, deployed, and scaled independently. The properties of these applications have motivated extensive research, including studies of performance, resource efficiency [8] and reliability [4]. To support such research, open-source demo applications are commonly used as testbeds. These applications provide set of services with distinct technology stacks and reflecting common characteristics of cloud-native applications.

Traditionally, microservice applications have relied on containers as a runtime technology [3]. However, the runtime landscape is evolving. WebAssembly (Wasm) has emerged as an alternative deployment technology, offering lightweight runtimes, small binaries, and fast startup. While Wasm has historically been limited in its compatibility, recent developments have substantially improved its integration with existing microservice technology stacks [10]. Initial studies comparing Wasm and containers have reported mixed results, with Wasm still trailing containers in several aspects [5, 7]. At the same time, recent work demonstrates that Wasm can provide benefits for selected microservice workloads [9, 10]. These developments motivate further empirical research into alternative microservice runtimes.

Such comparisons require testbeds that allow runtime technologies to be compared under controlled and reproducible conditions. Existing microservice testbeds, however, typically provide their own mechanisms for configuring and deploying the application,

making it difficult to switch deployment technologies or evaluate alternative runtimes. This makes comparisons between runtimes difficult to reproduce.

In this work, we present a configurable microservice testbed designed to support systematic research on microservice runtime and deployment technologies.

2 Requirements

WebAssembly (Wasm) is increasingly being explored as a runtime for server-side applications. WebAssembly System Interface (WASI)¹ and other recent ecosystem developments have addressed key compatibility gaps, making WebAssembly increasingly viable for cloud-native applications. This motivated us to investigate the feasibility of deploying a microservice application using Wasm.

To investigate alternative runtime technologies systematically, we require an experiment testbed that allows for different runtimes to be compared under controlled and reproducible conditions. We therefore designed our testbed according to criteria commonly demanded of a good benchmark [1, 2], ensuring that the system is relevant, repeatable, fair, portable, and understandable.

- R1 Relevant: The testbed should use a representative microservice benchmark application and preserve its functionality, service structure, and interfaces across runtime implementations.
- R2 Repeatable: Builds, deployments and workloads must be automated as much as possible, such that its trivial to deploy the system anywhere again. Services should use controlled sources of randomness where applicable to ensure consistent behavior across runs.
- R3 Fair: Runtime comparisons should use equivalent application implementations and experimental conditions. Changes required to support a different runtime should be applied consistently across runtimes.
- R4 Portable: The testbed should support deployment across different environments while integrating alternative runtimes into established cloud-native infrastructure, e.g. Kubernetes.

¹<https://wasi.dev>

R5 Understandable: To ensure that services and modifications remain understandable, we need to separate plumbing code and boilerplate as best as possible from the core logic. Runtime-specific modifications should be documented to keep the resulting application implementations understandable for other researchers.

3 Testbed

From these requirements, we made the following design choices:

Application. Rather than developing an application specifically for Wasm, we chose to port an existing application. We build our testbed on Google’s *OnlineBoutique*, a commonly used microservice benchmark application [6]. This avoids tailoring the application to the capabilities of the runtime (R3) and allows us to examine which parts of a typical microservice stack can already be supported.

Online Boutique is a polyglot microservice application consisting of eleven services communicating primarily over gRPC. The microservices are implemented in different programming languages and rely on diverse libraries and dependencies, potentially exposing limitations of the still-maturing Wasm ecosystem. They also use different communication mechanisms, including HTTP and gRPC (R1).

The original application already provides Docker-based deployments and Kubernetes integration, making it a suitable basis for investigating alternative runtime technologies. Its existing deployment infrastructure also provides a starting point for automated and portable experiments (R2, R4).

Dual-target Services. To support comparisons between container and Wasm runtimes, we modified the services such that each can be built for both deployment targets.

We initially explored porting services in multiple languages. While WebAssembly supports compilation from many languages, WASI support remains uneven; for example, we encountered challenges using basic HTTP functionality in Go. Rust currently offers more mature WASI support, so we chose to re-implement the services in Rust. All ported services use the same core logic and adhere to the same gRPC interfaces as the original implementation (R1). Each ported service can be built from a single source tree as both a native binary packaged as a container image and a WebAssembly module packaged as an OCI artifact (R3). As WASI support matures across other languages, the testbed can be extended with additional implementations (R1).

Table 1 provides an overview of all services in the testbed (R5). For the initial implementation, we selected three services for dual deployment: *shippingservice*, *recommendationservice*, and *cartservice*. The services represent different

Table 1: Services of the testbed. Struck-out languages denote the implementation of the upstream demo that we replaced. Only services marked *dual* are built for both runtimes.

Service	Language	Runtime variants
frontend	Go	container
productcatalogservice	Go	container
checkoutservice	Go	container
currencyservice	Node.js	container
paymentservice	Node.js	container
emailservice	Python	container
adservice	Java	container
redis (cart store)	—	container
shippingservice	Go Rust	dual
recommendationservice	Python Rust	dual
cartservice	C# Rust	dual

interaction patterns commonly found in microservice applications (R1). *Shippingservice* implements a simple request-response interaction, allowing direct comparison of the two runtimes without additional service dependencies. *Recommendationservice* extends this pattern by calling downstream services, allowing us to evaluate communication between runtimes as well as potential downstream effects. Finally, *Cartservice* interfaces with a database/cache over TCP, representing yet another common microservice interaction pattern and allowing us to evaluate runtime support for external dependencies.

Shared Orchestration. Both container and Wasm services are deployed through the same orchestration layer, Kubernetes, with Wasm services executed through *containerd* using the *runwasi*² shim. This allows different runtime configurations to be evaluated for typical cloud-native deployment environments (R1, R4).

The use of WebAssembly with Kubernetes introduces some runtime-specific quirks. In the Wasm deployment, service communication may be handled by the runtime rather than directly by the service, depending on the runtime. Container deployments require their own gRPC server code to provide equivalent communication behavior. To ensure fairness, we use the same libraries to implement this server and compile it directly into the binary also used for Wasm deployments where necessary. This keeps the application logic equivalent across runtimes while isolating runtime-specific modifications from the core service logic (R3, R5).

Experiment Automation. Online Boutique already provides a strong basis for automation, with automated builds and Kubernetes deployment configurations. We extended these mechanisms to accommodate the additional Wasm artifacts and runtime configurations (R2).

²<https://github.com/containerd/runwasi>

The mix of runtimes deployed in a given experiment is expressed declaratively as a composition of Kustomize³ components. This allows a service to be switched from a container to a Wasm runtime through a single configuration change, without modifying the application or the deployment logic. The same mechanism can be used to construct deployments containing only containers, only Wasm services, or a combination of both (R2, R4).

The complete testbed is open-source⁴.

4 Demonstration

To illustrate the use of the testbed, we report one representative experiment comparing the container and Wasm variants. The complete OnlineBoutique application was deployed on a bare-metal machine equipped with an Intel Xeon E3-1275v6 processor and 64 GB of RAM, with `shippingservice`, `recommendationservice`, and `cartservice` running as Wasm services. We used the workload configuration provided with the testbed, comprising 1,000 total users with an arrival rate of two users per second.

Table 2 reports the mean performance metrics for both configurations. The Wasm configuration resulted in 3% higher latency and 1% lower throughput, while the error rate remained effectively unchanged.

These results represent a single illustrative experiment; a broader evaluation across runtime configurations and workloads is part of our ongoing work.

5 Conclusion

We presented a configurable microservice testbed for controlled experiments on alternative runtime and deployment technologies. Based on Online Boutique, the testbed supports service-level selection between container and WebAssembly deployments, including mixed-runtime configurations, while providing a common application structure, workload, and orchestration environment.

The initial experiment demonstrates how the testbed can be used to compare alternative deployment configurations without changing the intended application logic. The results are intended as an initial demonstration rather than a comprehensive assessment. Further experiments are needed to assess how well the results generalize to other microservice environments.

For this, we plan to extend the testbed with additional applications and dual-target services to cover more of the microservice technology stack and a wider range of service interaction patterns. We also plan to explore a variety of Wasm runtimes and configurations.

Wasm is rapidly evolving and shows promise as an alternative runtime for cloud-native applications, but

Table 2: Container vs. Wasm performance (mean $\pm$ 95% CI)

	Container	Wasm	Δ
Latency [ms]	887.3 $\pm$ 8.6	917.1 $\pm$ 8.8	+3.35%
Throughput [rps]	57.83 $\pm$ 0.15	57.48 $\pm$ 0.25	-0.61%
Errors [%]	0.0188 $\pm$ 0.006	0.0181 $\pm$ 0.006	-0.0007 pp

its ecosystem is still maturing. Limited WASI support across languages and incomplete support for libraries commonly used in microservices currently still constrain its use for microservices. At the same time, these gaps offer exciting opportunities for research. As support matures, the testbed can be extended to explore new service implementations, interactions, and capabilities, including areas such as observability that are currently difficult to cover with Wasm. Ultimately, we see the testbed as a foundation for exploring what becomes possible as Wasm evolves.

References

- [1] K. Huppler. “The Art of Building a Good Benchmark”. In: *Performance Evaluation and Benchmarking*. Springer, 2009, pp. 18–30.
- [2] D. Bermbach, E. Wittern, and S. Tai. *Cloud Service Benchmarking*. Springer, 2017.
- [3] D. Gannon, R. Barga, and N. Sundaresan. “Cloud-Native Applications”. In: *IEEE Cloud Computing* 4.5 (2017), pp. 16–21.
- [4] M. C. Borges et al. “Informed and Assessable Observability Design Decisions in Cloud-Native Microservice Applications”. In: *2024 IEEE 21st International Conference on Software Architecture (ICSA)*. 2024, pp. 69–78.
- [5] S. Kakati and M. Brorsson. “The Small Container: Lightweight Compute for Multi-architecture Cloud-Edge Systems”. In: *SN Computer Science* 6.8 (2025).
- [6] J. von Kistowski et al. “Microservice Benchmark Applications”. In: *Systems Benchmarking: For Scientists and Engineers*. Springer, 2025, pp. 305–321.
- [7] M. Liu et al. “WebAssembly for Container Runtime: Are We There Yet?” In: *ACM Transactions Software Engineering Methodology* 34.6 (2025).
- [8] S. Werner et al. “A Comprehensive Experimentation Framework for Energy-Efficient Design of Cloud-Native Applications”. In: *2025 IEEE 22nd International Conference on Software Architecture (ICSA)*. 2025, pp. 176–186.
- [9] J. Bauer, S. Werner, and M. C. Borges. “Kirin: Cloud-native WebAssembly Service Orchestration”. In: *2026 IEEE International Conference on Cloud Engineering (IC2E)*. 2026.
- [10] M. Ziółkowski, I. Fajjari, and H. Tarasiuk. “WebAssembly as a Lightweight Path to Sustainable and High-Performance Cloud-Native Computing”. In: *2026 IEEE 23rd Consumer Communications & Networking Conference (CCNC)*. 2026, pp. 1–6.

³<https://github.com/kubernetes-sigs/kustomize>

⁴<https://github.com/blablalorges/rust-in-pieces>
<https://github.com/ISE-TU-Berlin/piecing-it-together>