Benchmarking Function Hook Latency in Cloud-Native Environments

Mario Kahlhofer[1], Patrick Kern[1], Sören Henning[1,2], Stefan Rass[2]
{mario.kahlhofer, patrick.kern}@dynatrace.com, {soeren.henning, stefan.rass}@jku.at
[1]Dynatrace Research, [2]Johannes Kepler University Linz

Abstract

Researchers and engineers are increasingly adopting cloud-native technologies for application development and performance evaluation. While this has improved the reproducibility of benchmarks in the cloud, the complexity of cloud-native environments makes it difficult to run benchmarks reliably. Cloud-native applications are often instrumented or altered at runtime, by dynamically patching or hooking them, which introduces a significant performance overhead. Our work discusses the benchmarking-related pitfalls of the dominant cloud-native technology, Kubernetes, and how they affect performance measurements of dynamically patched or hooked applications. We present recommendations to mitigate these risks and demonstrate how an improper experimental setup can negatively impact latency measurements.

1 Introduction

Cloud-native technologies aim to build loosely coupled, resilient, observable, and secure systems [3]. Observability and security are typically achieved by dynamically instrumenting or altering already built applications with *function hooks* [2]. These are small pieces of code added to an application's functions. In particular, security tools need to dynamically modify, redirect, or block specific execution patterns, which often results in significant performance penalties [7].

Careful benchmarking is required to measure the performance impact of such changes. Besides *empirical standards* for software benchmarking [9] and *methodological principles* for performance evaluation in cloud computing [6], we address benchmarking-related pitfalls of cloud-native environments with:

1. Recommendations on how to measure the latency of function hooks in cloud-native environments.
2. A demonstration of an improper experimental setup that makes hypothesis testing harder.

2 Cloud-Native Benchmark Suite

Cloud environments are frequently used to build complete benchmark suites, as they provide a well-reproducible environment [8]. A typical benchmark suite (Figure 1) consists of a *system under test (SUT)*, e.g., the patched application, a *load generator* sending requests to that application, and a *monitoring tool* measuring performance metrics. Latency is often measured directly by the load generator.

In Kubernetes, workloads are organized into *pods* of one or more *containers* which share storage and networking resources. Physical or virtual machines that run these pods are called *nodes*.

Recommendation 1 When measuring latency, ensure that the load generator and the SUT are in separate containers within the same pod. Otherwise, additional network hops may distort the measurements.

Recommendation 2 If components of the benchmark suite need to be in separate pods, ensure that both pods are deployed on the same physical node, e.g., by specifying *node restrictions* in Kubernetes.

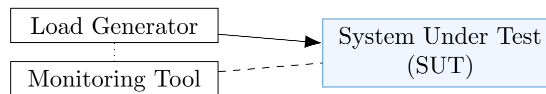

Figure 1: Typical components of a benchmark suite

Recommendation 3 Weigh the benefits of a *service mesh* against its additional network overhead. Service meshes wrap each application behind a reverse proxy and make it easier to monitor and control inbound and outbound network traffic [10].

Recommendation 4 Generally avoid benchmarking in *multi-tenancy clusters*, i.e., clusters that are shared across teams, either physically or virtually.

3 Function Hook Granularity

We distinguish four layers [2, 12] where function hooks or patches can be injected (Figure 2):

- **Application-level** hooks use methods implemented by the application's developers, e.g., a plugin system. Since such systems are not widely available, this layer cannot be used for general-purpose hooks on already built applications.

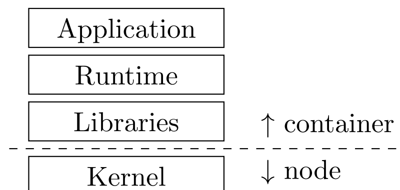

Figure 2: Typical layers of a software application

- **Runtime-level** hooks use native capabilities of language runtimes to modify applications, e.g., the JVM Tool Interface (JVM TI), the .NET Profiling API, or Node.js module preloading.
- **Library-level** hooks override symbols in shared libraries, e.g., by the "LD_PRELOAD trick" [2].[1]
- **Kernel-level** hooks use native capabilities of the operating system to modify application behavior, e.g., kernel modules or eBPF programs.

Recommendation 5 The monitoring tool should be placed as close as possible to the layer where the hook is injected. Testing farther away pollutes measurements with noise from other layers (Section 4.2).

To achieve optimal results, hooking and monitoring should be done at the "same layer", i.e., by embedding monitoring functionality into the hook itself, e.g., by recording timestamps before and after the hook is executed, directly in the hook's code.

Moving the monitoring tool further away from the hook is justifiable when one wants to more accurately represent real-world behavior instead.

Recommendation 6 Describe if the benchmark measures the specific hooking overhead in isolation (*micro benchmark*), or rather represents a real-world application with a hook injected into it (*macro benchmark*) [1]. Benchmarking both cases and discussing the differences is recommended.

4 Demonstration

We first build a Java application that simply responds with "Hello World" to any HTTP request. We then implement a *library-level LD_PRELOAD hook* that blocks all requests that contain specific keywords (Listing 1). The hook changes the `read`[2] system call by overriding the corresponding symbol in the C standard library.[3] We then measure our application's network performance with and without the hook.

```
ssize_t read(int fd, void *buf, size_t count) {
  read_t read_ptr = (read_t)dlsym(RTLD_NEXT, "read");
  ssize_t bytes_read = read_ptr(fd, buf, count);
  if (is_http_socket(fd)) {
    if (contains_keyword(buf, count)) {
      // trace or block call
    }
  }
  return bytes_read;
}
```

Listing 1: A hook on the `read` symbol of glibc

Low-level function hooks carry the risk that the hooked function is used by high-level functionality for a purpose other than the one originally intended. For example, the `read` call that we override here is also used to read regular files, not just network packets.

[1] https://man7.org/linux/man-pages/man8/ld.so.8.html

[2] https://man7.org/linux/man-pages/man2/read.2.html

[3] The full source code can be found at https://github.com/dynatrace-research/function-hook-latency-benchmarking

Recommendation 7 Therefore, describe how the hooked function is typically used by applications and ensure that the benchmarks reflect their proper use, e.g., with *synthetic micro benchmarks* [1], but also real-world behavior. Suitable cloud-native, real-world reference applications are TeaStore [4], DeathStarBench [5], or Unguard [11] for security use cases.

4.1 Experimental Setup

Our experiment consists of two containers: Locust (a performance testing tool) as the load generator with embedded monitoring, and the SUT. With containers, we not only represent cloud-native paradigms, but also isolate concerns between the *benchmark owner* and the *SUT owner*. We compare four conditions:

1. **In Docker** (a popular container runtime): Both containers run on a single server, communicating through the host network.
2. **In Kind** (a tool for running Kubernetes using Docker containers): Both containers run inside a single pod on a local, single-node Kind cluster.

The remaining two conditions use the AWS EKS service (a popular enterprise-grade cloud provider).

3. **In EKS pod**: Both containers run inside a single pod in a managed, single-node AWS EKS cluster.
4. **Across EKS nodes**: Both containers run in separate pods, each pod on a different node, in a managed AWS EKS cluster with two nodes.

Docker and Kind are running on a 24-core (Intel Xeon E5-2680 v3) Ubuntu 22.04 server with 64 GB memory. EKS nodes are t3.medium EC2 instances (2 vCPUs, Intel Xeon Platinum 8000, 4 GB memory).

Recommendation 8 Ensure that the servers do not hit any resource limits during the experiment to avoid performance degradations due to resource contention.

We measure the round-trip time (RTT) of $50,000$ HTTP request-response interchanges between Locust and the SUT. We empirically observed that the RTT definitely stabilizes under all four conditions after $\sim 4,000$ warm-up requests (Figure 3).

4.2 Hypothesis Testing and Results

Figure 4 shows the RTT distribution per condition, with and without the hook, after warm-up requests.

Our function hook must introduce a performance overhead: To test the null hypothesis that the mean RTT is the same with and without the hook, we use an independent two-sample t-test, assuming equal but unknown variances and equal sample sizes.[4] Let $\bar{x}_1$ and $\bar{x}_2$ be the sample means, n be the sample size per condition, and s_p the *pooled standard deviation*[5], then the test statistic is given by: $t = (\bar{x}_1 - \bar{x}_2) \,/\, (s_p \sqrt{2/n})$.

[4] We use the `stats.ttest_ind` test from the SciPy package.

[5] With sample variances s_1^2 and s_2^2 and equal sample sizes, the pooled variance is defined by $s_p^2 = (s_1^2 + s_2^2) \,/\, 2$.

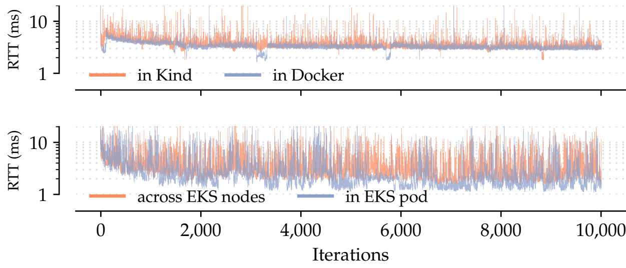

Figure 3: Logarithmic lag plot on measured RTT

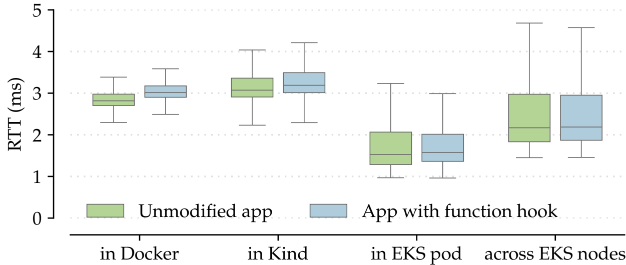

Figure 4: Tukey boxplots on 50,000 RTT measurements per condition, without any warm-up requests

With $n = 46,000$ samples left per condition after removing warm-up requests, the hooking overhead is significant ($\alpha = 0.05$) in three of four conditions (all $p < 0.0001$). The *across EKS nodes* condition is not significant ($p = 0.2713$) since we lose a lot of statistical power in EKS due to the higher RTT variance.

As expected and in line with related work [6], measurements taken in EKS generally exhibit a much higher variance than measurements on our own server, due to diverse factors that can hardly be controlled for. We expected the *Docker* condition, being the most minimal setup, to show the lowest variance ($s_p = 0.50$), and the *Kind* condition, which just adds a few Kubernetes components, to show the second-lowest variance ($s_p = 0.84$). We also expected that the network latency *across EKS nodes* shows the highest variance ($s_p = 1.95$; 3.9 times higher than in *Docker*). Packets in that condition traverse the origin container, pod, and node, some intermediate network that interconnects nodes, until they reach the target node, pod, and container again. In a multi-cluster environment, a communication path with that many hops is common. Kubernetes *services*, which are widely used abstractions on top of pods, would add even more hops to that route. Perhaps surprising is that the variance of the *EKS pod* condition was still relatively high ($s_p = 1.64$). Keeping network communication within the same pod decreased the variance, but it seems that the background noise of our EKS cluster is still relatively high and affecting inter-pod traffic.

Recommendation 9 As shown, measurements in cloud-native environments tend to have a higher variance than in local environments [6]. To regain statistical power, the sample size must be increased.

Recommendation 10 Conducting experiments in differently configured environments is a general principle [6, P2] that is especially relevant for cloud-native environments. Different cloud providers, service meshes, or network setups help increase diversity.

5 Conclusion

This work provides 10 practical recommendations for researchers and engineers who benchmark function hook latency in cloud-native environments, but want to reduce the measurement bias introduced by these environments. We have shown that function hook latency measurements can be easily contaminated by noise, without doing anything obviously wrong. We hope to raise awareness while providing practical guidance for similar latency-based benchmarks, as some of our recommendations are also broadly applicable.

References

[1] J. Waller and W. Hasselbring. *Performance Benchmarking of Application Monitoring Frameworks*. KCSS 2014/5. 2014.

[2] J. Lopez et al. “A Survey on Function and System Call Hooking Approaches”. In: *HaSS* 1.2 (2017).

[3] CNCF. *Cloud Native Definition*. 2018.

[4] J. von Kistowski et al. “TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research”. In: MASCOTS ’18. 2018.

[5] Y. Gan et al. “An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems”. In: ASPLOS ’19. 2019.

[6] A. V. Papadopoulos et al. “Methodological Principles for Reproducible Performance Evaluation in Cloud Computing”. In: *TSE* 47.8 (2019).

[7] W. Viktorsson, C. Klein, and J. Tordsson. “Security-Performance Trade-offs of Kubernetes Container Runtimes”. In: MASCOTS ’20. 2020.

[8] S. Henning, B. Wetzel, and W. Hasselbring. “Reproducible Benchmarking of Cloud-Native Applications with the Kubernetes Operator Pattern”. In: SSP ’21. 2021.

[9] P. Ralph et al. *Empirical Standards for Software Engineering Research*. 2021. preprint.

[10] S. Henning, B. Wetzel, and W. Hasselbring. “Cloud-Native Scalability Benchmarking with Theodolite Applied to the TeaStore Benchmark”. In: SSP ’22. 2022.

[11] Dynatrace LLC. *Unguard: An Insecure Cloud-Native Microservice-Based Application*. 2023.

[12] C. Islam, V. Prokhorenko, and M. A. Babar. “Runtime Software Patching: Taxonomy, Survey and Future Directions”. In: *JSS* 200 (2023).