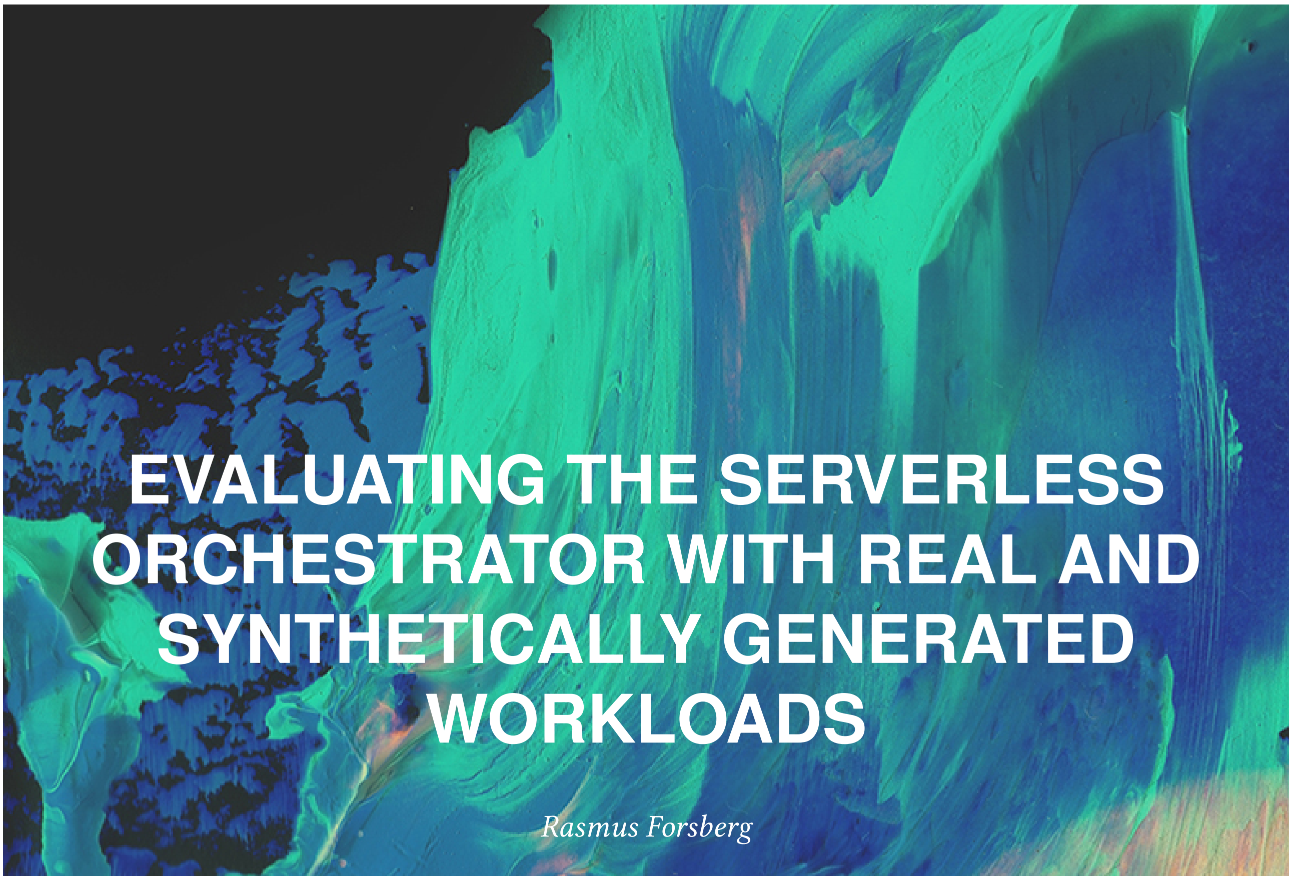

EVALUATING THE SERVERLESS ORCHESTRATOR WITH REAL AND SYNTHETICALLY GENERATED WORKLOADS

Rasmus Forsberg

Bachelor Thesis, 15 hp/credits
Computing Science

2026

Abstract

Serverless computing uses complex orchestration to manage resource scaling and initialization. Evaluating these orchestrators requires advanced benchmarking. A way of testing these orchestrators is to rely on ”naive” synthetic methodologies that use uniform or Gaussian distributions. This thesis investigates the extent to which synthetic benchmarks distort perceived system performance compared to trace driven testing based on real world data. Using an OpenFaaS test bed with a modified Locust load generator we conducted a comparative analysis between standard synthetic benchmarks and a replay of the Azure Functions production trace. We implemented a ”universal function” to simulate a wide array of workload profiles mimicking the Zipfian popularity and Log Normal execution distributions found in the production data. The results show that standard uniform function selection creates an artificially ”warm” environment masking the penalties of cold starts. The synthetic model also fails in generating outliers responsible for head of line blocking and resource contention. It can be concluded that naive synthetic benchmarking leads to an overestimation of system throughput and stability. To achieve performance evaluations that covers all cases future research should adopt trace driven methodologies that preserve the statistical irregularities of real world traffic.

Contents

1 Introduction

Serverless computing often referred to as Function as a Service (FaaS for short) has changed how applications are built and deployed in the cloud. This particular model has seen widespread use because it allows developers to focus solely on writing code while the cloud provider takes responsibility for managing the underlying infrastructure like setting up servers and scaling them to meet demand [7]. This change seems like it would simplify software development and reduce costs but it introduces its own challenges for the systems that manage these resources.

The substantial benefit of serverless computing for the user is the "pay as you go" billing model. Unlike traditional cloud servers where users pay for reserved capacity serverless users only need to pay for the time their code is actually running [2]. To make this sustainable for the cloud providers they attempt to release the resources as soon as they are not in use. This efficiency however results in a performance penalty known as a cold start.

When traffic to the service increases, the orchestrator must scale out resources to match the demand. However, this process is not instantaneous due to physical limitations on container initialization [9]. Combined with the time it takes for the system to react with its auto scaling, this creates a mismatch between the arrival of new traffic and how many resources are available [2]. The difference in resource availability this creates may be harmful to the service because the traffic Faas services often receive come in bursts and has an unpredictable nature [6]. If a spike in traffic occurs faster than the system can physically react user requests pile up which can lead to high latency or connections dropping. To ensure these systems are robust researchers must test them against realistic conditions. A way to test these systems are to use "naive" synthetic benchmarking strategies. These test send traffic in uniform patterns where all functions are invoked equally. This thesis argues that such synthetic generation testing is flawed for testing a serverless environment by having traffic follow a uniform pattern. This way of testing keeps the function containers artificiality warm or active in memory. If true this would mask the performance penalty of a cold start and fail to trigger specific theoretical weaknesses in the orchestrator.

- **The Cold Start Penalty:** Real workloads follow a zipfian popularity distribution where most functions are rarely invoked [6]. Uniform synthetic tests fail to take in to account the many unused functions leading to an underestimation of the delay of a cold start.
- **Reaction Time Failure:** Uniform synthetic traffic smooths out traffic over time and thus fail to account for the instantaneous bursts of traffic that can happen in production.

The objective of this thesis is to visualize the distortion in naive synthetic benchmarking and visualize its inability to perform an effective evaluation. Defined here as one that reveals the penalties of a cold start and the ratios of cold starts. The research question is: *To what extent do naive synthetic benchmarking methodologies distort the perceived performance of serverless orchestrators compared to trace driven production simulations?* To answer this we designed a comparative experiment using the locust load testing tool[1]. Locust was used to perform two roles which was generating specific workload patterns both synthetic and trace replays based on azure function data [5] and recording the performance metrics during the tests.

[1]Locust: An open source load testing tool. Available at `https://locust.io/`

2 Related Work

The research in serverless computing we analyzed has largely focused on three distinct areas: characterizing production workloads, developing new scheduling algorithms, and building benchmarking frameworks. This chapter surveys these areas to distinguish the specific contribution of this thesis which is demonstrating how standard synthetic benchmarking methodologies distort the perceived performance of serverless orchestrators compared to trace driven simulations.

2.1 Serverless Workload Characterization

To evaluate a system effectively one must first understand the workload it is expected to handle. A work in this domain can be viewed in [6] which analyzes the Azure Functions workloads. The findings when analyzing Azures workloads from production reveal two properties that challenge standard testing methods. First it is revealed that function popularity follows a steep zipfian distribution where 81% of functions receive at most one request per minute. Second, they identify that the patterns of received traffic follow a highly irregular and busty pattern with arrival rates often varying by orders of magnitude within short time periods

While the research in [6] provides a data model for serverless research the work targeted analysis of patterns rather than validating system behavior. Our thesis uses the statistical findings particularly the high ratio of unpopular functions and the log normal distribution of execution times as a basis to evaluate the accuracy of synthetic benchmarking tools.

2.2 Scheduling and Auto-scaling Optimizations

Another work [2] analyzes the mechanics of the container scaler identifying an instability known as "Hysteresis" where the system changes rapidly between adding and removing containers. While the work makes it clear that mechanisms to dampen the effect can stabilize the system it introduces an unavoidable delay in the reaction of said system. Article [4] introduces Hermond a scheduler designed to mitigate "Head of Line" blocking. They show that because execution times can vary wildly short requests often get stuck behind long ones in first in first out queues making it necessary to use scaling logic like processor sharing. Furthermore the article [3] proposes Fifer arguing that standard CPU based scaling logic is not efficient enough. They demonstrate that "bin-packing" requests based in queue depth can greatly improve efficiency.

In addition to scheduling researchers have proposed mechanism to reduce the cost of initialization of containers. This work [7] introduces Rewind a snapshotting technique used as a workaround for initialization overheads to reduce the latency of a cold start . In a similar manner this other work [9] analyzes the benefits of pre initialized snapshots in Firecracker mircroVMs to reduce the "actuation lag" of booting the kernel of a new container.

Our work does not propose a new scheduler or snapshotting mechanism. Instead, we use the theoretical weaknesses identified by these authors specifically the reaction delays found in [2] and the queuing effects found by [4] to explain why simple synthetic benchmarks fail to stress the system adequately compared to trace replay.

2.3 Benchmarking and Experimental Methodologies

To test these platforms, researchers rely on benchmarking tools and frameworks.

Researchers often use trace replay as their method of testing. The paper [5] develops FaaS-Rail as a tool to replay these production traces from large platforms to recreate the statistical properties of real world traffic. In a similar manner paper [9] develops the vHive framework to enable full stack experimentation using production like workloads. These works establish that trace replay is superior to random generation for capturing the bursty nature of serverless traffic.

Iterating upon these frameworks [8] creates a methodology for conducting experiments without access to source code and large scale production clusters they term in-vitro research. They validate the use of ”Synthetic Functions” which are workloads that emulate resource consumption of real applications using busy loops or sleep commands without executing business logic. They also demonstrate that random sampling of production traces in attempts to replicate a production environment with limited resources results in large amounts of statistical error. Instead of random sampling they advocate for the use of ”Workload Summaries” which are scaled down versions of production traces that preserve the invocation and resource usage of the original production trace. This research supports our decision to use a sleep function to simulate different function behaviors validating the idea that performance evaluation is possible even in small scale environments if the distribution of the workloads mimic the original production trace.

To be able to evaluate the system its necessary to observe the performance of it. In [1] FaaSLoad is developed, a tool capable of separating the initialization of the platform and the function execution time. We adopt FaaSload’s approach to isolate the causes of latency during cold starts during our experiment.

Finally for load generation we use an open source tool called Locust. Locust is commonly used for generating synthetic loads where the users are constant or ramping, it typically defaults to a uniform distribution when selecting tasks. This thesis modifies it to support production traces in its workload generation which allows a direct comparison between synthetic benchmarks and trace replays.

3 Background

This chapter introduces concepts and terms required to evaluate serverless orchestration. It covers the physical constraints of resource initialization, the mathematical models used, and the control theory logic used in the process of scaling.

3.1 The Serverless Abstraction

Cloud computing has gone from providing processing power in the form of raw infrastructure to whole environments where code can be executed without additional configuration. In the infrastructure as a service (IaaS) model users rent virtual machines (VMs) and have their own responsibility of operating system maintenance and planning for capacity. The serverless computing model or function as a service (FaaS) model flips this on its head as described in [7] the cloud provider assumes full responsibility for providing enough resources scaling those resources to match demand and balancing the demands with the resources available. The user only needs to provide code often in the form of functions which can also include event triggers such as HTTP requests or database updates. The provider is then responsible for executing the provided code on demand which has made a shift in the landscape with providers transitioning to a "pay as you go" billing model where as described in [2] users only pay for the execution time of their provided code instead of paying for idle infrastructure.

To provide this service FaaS platforms use a multi layered software stack. The standard hierarchy found in platforms like OpenWhisk and Knative is described in [1]

1. **The Orchestrator:** The central decision making component. It receives incoming traffic and calculates the capacity required of the system then it sends requests to the relevant worker nodes to handle the processing request.
2. **The Cluster Manager:** A system that oversees and maintains the pool of physical or virtual worker nodes.
3. **The Container Engine:** A software running on workers responsible for creating, managing and running the containers the execution environments are running on.
4. **Sandboxes:** The actual environment were execution of the code happens usually they are containers or microVMs that enforce security and resource limits.

3.2 The physical constraints of Scaling: Latency and Actuation

The scaling of a platform does not happen instantly it is restricted by the physical time required to create new computing resources. When a request arrives for a function with no instance available the platform must preform a cold start. This process involves three distinct phases [9].

1. **VMM/Kernel Boot:** Starting the virtual machine monitor and booting the guest kernel.

2. **Runtime Initialization:** Starting the language environment.

3. **User Code Init:** Loading dependencies and running initialization scripts.

Even in optimized environments the process of a cold start causes significant actuation lag or dead time. During this period the system cannot handle new requests because the resources to handle said requests are in the process of being created [9].

3.2.1 Warm Starts

To combat this dead time platforms use warm starts. After the execution of code the orchestrator keeps the container used for execution active in memory for a keep alive window [6]. Reusing an existing container skips the initialization phases described above reducing the latency from hundreds of milliseconds to single digit milliseconds.

3.3 Theoretical Scaling Logic

3.3.1 Reactive Feedback Loops

In trying to maintain the balance orchestrators use reactive feedback loops. As described in [2] the system tracks a metic like CPU usage or pending requests and makes a decision to scale either up or down depending on the value of the metric and the set thresholds the orchestrator operates on. However these feedback loops get unstable if they react too quickly and scales up and down rapidly in a fixed range which is called Hysteresis. To prevent this orchestrators use dampening mechanisms such as requiring a specific number of violations of the currently required resources or needing a certain time window to pass before executing the scaling command. The dampening ensures stability in the platform but it also means that the orchestrator potentially lags behind in rapid shifts of demand.

3.3.2 Head of Line Blocking

The orchestrator faces another problem which is that different code requests have differing execution times. As highlighted in [4] the problem known as Head of Line blocking which is if a short request is scheduled behind a long request in the queue the shorter request gets delayed a disproportionate amount of time. This makes first in first out queuing unfeasible in FaaS environments and forces orchestrators to scale out aggressively in order to keep queues short.

3.4 Workload Characteristics

The potential challenges discussed above increases by how traffic works in production. Analysis of the Azure Function workloads finds it to be very diverse without many patterns or trends [6].

3.4.1 Function Popularity

The number of function invocations typically follow a zipfian distribution where a very small percentage of hot functions receive the majority of traffic and are kept warm by the orchestrator. On the other hand the majority of functions are used very infrequently often less then once per minute [6]. This limited number of invocations makes it inefficient to keep the containers warm which results in the vast majority of functions suffer from cold starts.

3.4.2 Execution Duration and Burstiness

The traffic patterns are often bursty and unpredictable meaning the arrival rate changes very rapidly. Additionally execution times are often extremely short, often less then 1 second [6]. This makes for a difficult scenario where the Actuation Lag of booting a container is often longer than the time needed to execute the user's code.

4 Methodology

4.1 Experimental design

The main goal of this experiment is to compare the standard synthetic way of testing serverless platforms with the replay of a production trace and evaluate if there is any distortion between the synthetic benchmarks and the trace replay.

4.1.1 Experimental Testbed and Architecture

The experiments were performed on a local Kubernetes cluster hosted on docker desktop. The serverless platform used in the test was OpenFaas which was deployed using the standard openfaas-fn namespace. The system architecture consisted of three components that can be viewed in Figure 1:

- The Load Generator: Implemented using Locust the user class was customized to support normal synthetic generation and the trace replay.
- The Gateway: The OpenFaas Gateway was used as the point where all invocation requests were received.
- The Function Executor: Kubernetes pods that were running the function received from the workload.

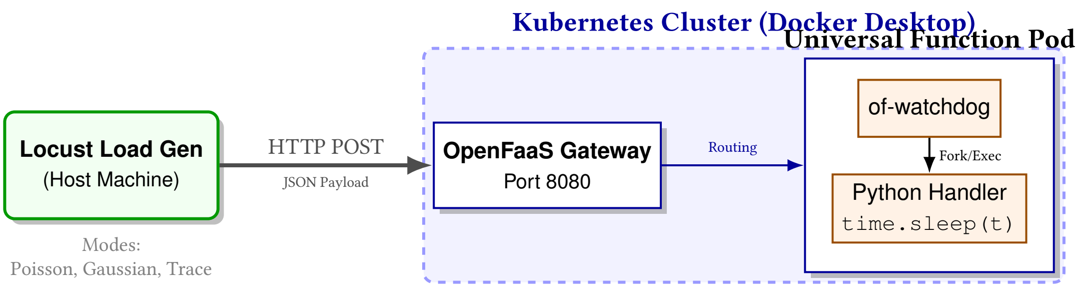

Figure 1: The architecture of the experimental setup. Locust generates load which is routed through the OpenFaas Gateway to a single pod running the sleep function.

4.1.2 Workload Design: The "Universal" Function

Simulating a real production environment usually requires deploying many containers and functions which requires a lot of processing power we did not have access to. In order to get around this a synthetic function was employed.

A single OpenFaas sleep function based on the python3-http template was deployed. Instead of performing some kind of computation the function accepts a JSON payload that describes a desired execution duration. When the function is activated it sleeps for the requested duration before returning a 200 OK response which all can be viewed in Figure 2.

With this approach a single deployed function can mimic the execution times of an infinite number of distinct function ranging for micro tasks taking 10ms to heavy data processing taking 10 seconds or more by changing the input parameter.

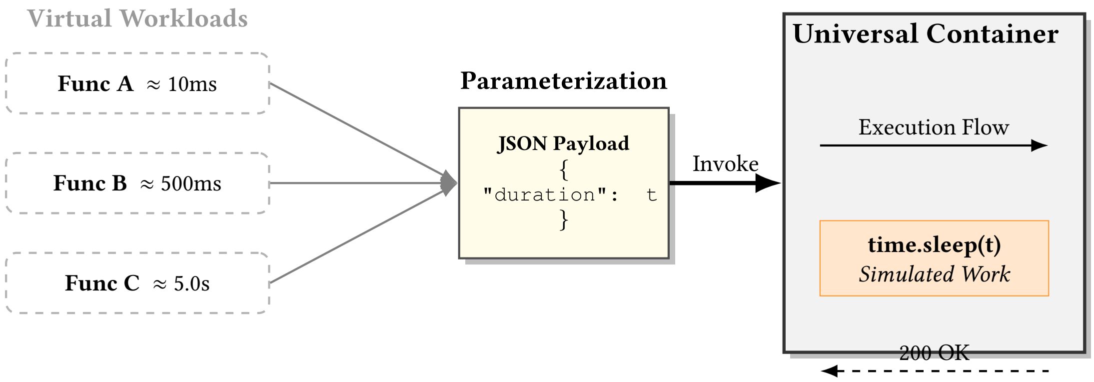

Figure 2: Showcases how the virtual workloads are mapped to a single function.

4.2 Synthetic Trace Generation Strategy

To create a repayable workload that simulates a production environment without requiring the whole multi terabyte dataset a workload generator was implemented. The generator creates a synthetic trace based on the probability distributions reported in the analysis of the Azure Functions Public Dataset [6]. The workload generation setup can be viewed in Figure 3

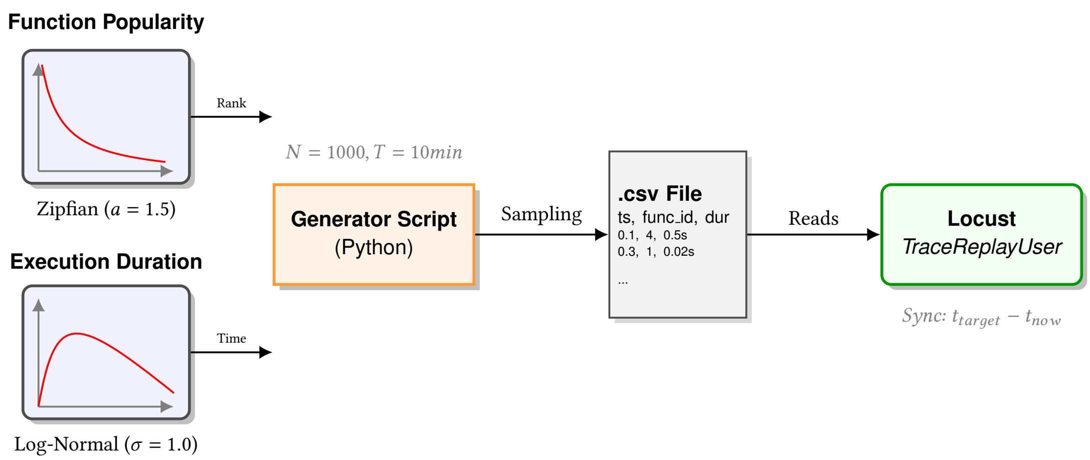

Figure 3: The Synthetic Trace Generation Pipeline. Statistical parameters taken from Azure production data (Zipfian Popularity and Log-Normal Duration) are used to create a CSV artifact. This artifact is then replayed by the Locust TraceReplayUser.

4.2.1 Function Popularity

As mentioned previously the dataset analysis reveals that serverless traffic is characterized by extreme skewness where a small number of hot functions receive the majority of invocations. To model this we used a Zipfian (Power Law) distribution:

$$P(k) \propto \frac{1}{k^a} \tag{4.1}$$

We selected a shape parameter of $a = 1.5$ to replicate the Azure trace characteristic where the top 1% of functions receive a large amount of traffic.

4.2.2 Execution Duration

The Azure trace indicates that while 50% of functions execute in less than 1 second but the distribution has a large span up to multiple minutes of execution time [6]. To capture this the execution duration was modeled using a log normal distribution:

$$f(x; \mu, \sigma) = \frac{1}{x\sigma\sqrt{2\pi}} \exp\left(-\frac{(\ln(x) - \mu)^2}{2\sigma^2}\right), \quad x > 0 \tag{4.2}$$

where x denotes the execution time in seconds $\mu = -0.7$ and $\sigma = 1.0$. These parameters were used to mimic the execution times in the dataset with the median of a log-normal distribution is e^{μ}. With $\mu = -0.7$, the median execution time is $e^{-0.7} \approx 0.49s$. This aligns with the finding in [6] that the median execution time across the entire dataset is approximately 500ms. The $\sigma = 1.0$ parameter ensures that the whole span of execution durations are covered with occasional execution times of 5-10 seconds.

4.2.3 Arrival Pattern

To generate the timestamps for the arrival pattern N random values were sampled from a uniform distribution over the experiment duration T and then sorted them: $t_1 < t_2 < \ldots < t_n$. This was done to simulate a Poisson Process.

4.3 Experimental scenarios

Three experiments were conducted to compare standard naive practices against a trace driven simulation. In all of the experiments Locust acted as the client logging timestamp, function ID and the latency for post experiment analysis.

4.3.1 Experiment A: Baseline (Poisson/Uniform)

- **Arrival Rate:** Poisson process (random inter-arrival times).
- **Function Selection:** Uniform distribution randomly selecting function IDs 0–49.
- **Goal:** Represents the standard ”naive” benchmarking approach.

4.3.2 Experiment B: Variable Load (Gaussian)

- **Arrival Rate:** Follows a Gaussian ”Bell Curve” Load Shape peaking at the 5-minute mark.
- **Function Selection:** Uniform distribution randomly selecting function IDs 0–49.
- **Goal:** Represents a synthetic test attempting to simulate a busy hour without using real trace data.

4.3.3 Experiment C: Trace Replay

- **Arrival Rate:** Deterministic replay of timestamps from the synthetic Azure trace.

- **Function Selection:** Deterministic replay using the Zipfian skew ($a = 1.5$).
- **Goal:** Validates the system against the irregularities of production traffic.

5 Results

This chapter details the data collected from the three experimental scenarios: Poisson baseline, Gaussian uniform load and the Trace driven replay. Collection of the data was performed using the Locust setup described in the Methodology. Around 3000 requests were recorded for the synthetic scenarios and 1000 requests for the trace replay.

5.1 Load profiles

Figure 4 (A) plots the arrival rate of requests per minute over the 10-minute experiment duration. The Poisson scenario indicated by a blue line produces a near constant arrival rate with slight variations around the mean caused by the randomness of the generation. The Gaussian scenario indicated by a green line produces a distinct bell curve with traffic ramping from near zero to a peak of intensity at the 5 minute mark before decreasing. The Azure Trace indicated by the red line differs a bit from the mathematical models being more jittery but there is not a massive difference visible in the arrival rate between the Poisson and trace.

5.2 Function Popularity Distribution

Figure 4 (B) displays the invocation frequency of the 50 simulated functions. The Y-axis is plotted on a logarithmic scale to show the magnitude of differences between ranked functions.

In both the Poisson and Gaussian scenarios the data shows a horizontal linear trend. This points to that the chance of invocation for a particular function is approximately equal ($\approx 2\%$) for every function ID in the set regardless of rank.

The Azure Trace on the other hand displays a non linear curve consistent with the power law distribution. The most popular function receives a high percentage of the total traffic. The amount of traffic received drops sharply for functions ranked 1-10 and almost flattens for functions ranked 30-50 which receive the least amount of invocations.

5.3 Execution Duration Variance

The Cumulative Distribution Functions (CDFs) for execution duration are shown in Figure 4 (C and D).

The synthetic scenarios (Blue/Green) show steep S-curves centered closely around the mean of 0.5 seconds. The inter quartile range for the synthetic scenarios is narrow indicating that a large number of requests completed within a tight time window fo 0.3s to 0.7s.

The Azure trace (Red) results in a much shallower slope. The curve indicates a wider spread of execution times with a lot of short running tasks (< 100ms) and some long running tasks (> 2.0s) which are absent in the synthetic data.

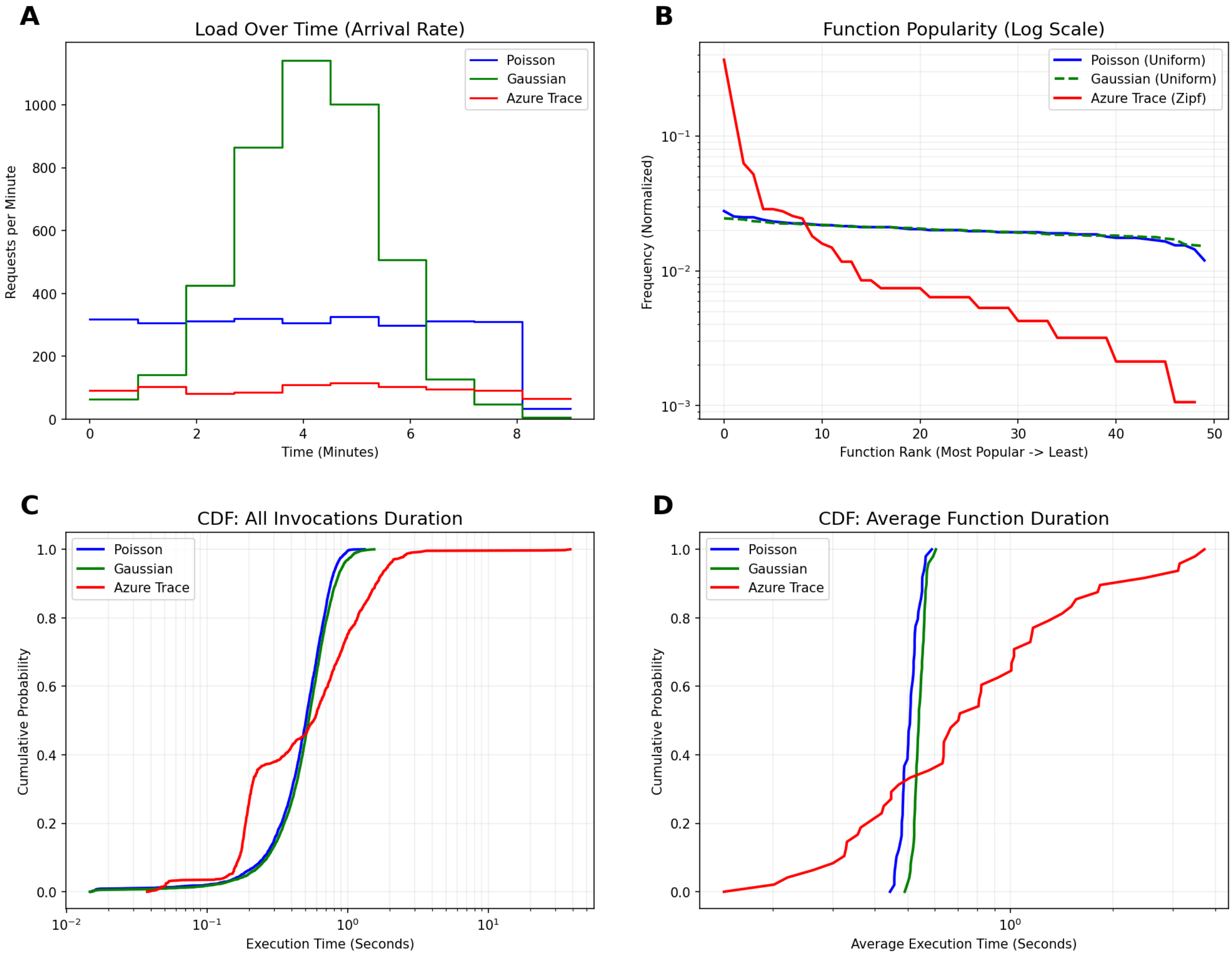

Figure 4: Comparative analysis of experimental scenarios. **(A)** Temporal load profiles displays the difference of smooth synthetic gradients with irregular trace fluctuations. **(B)** Function popularity (log-scale) illustrating the uniform distribution of synthetic models versus the Zipfian skew of the trace. **(C-D)** Execution duration CDFs demonstrating the homogeneous latency of synthetic benchmarks compared to the high variance and outliers of the trace driven workload.

6 Analysis of Results

This chapter interprets the statistical patterns observed in the analysis of the results, focusing on the direct comparisons between the synthetic models and the trace replay.

6.1 Predictability of Load Patterns

The results in Section 5.1 shows a difference in predictability of the models. The synthetic models follow a smooth gradient where the rate of change between time windows is constant which analytically represents a deterministic workload. In contrast the Azure trace shows fluctuations between time windows. While the gaussian model successfully mimics the volume of a peak load, it fails to replicate the same volatility. The data indicates that synthetic generation filters out the jitter which is a known attribute of production traffic resulting in a workload that is more artificially stable than its production counterpart.

6.2 Function Warmth and Cache Residency

The popularity distribution discussed in Section 5.2 implies a big difference in keeping functions active in the cache. The Poisson and Gaussian scenarios uses uniform selection logic which guarantees that invocations of functions are distributed evenly across all function IDs. Given the total request volume results in an invocation frequency that is statistically likely to keep all 50 functions warm (active in memory) through out the experiment. On the other hand the Zipfian skew of the trace scenario focuses the majority of the invocations on the top 10% of the functions. The remaining 90% of functions experience long periods of inactivity. This would imply that during the trace simulation the platform's resources are used to repeatedly execute a small set of functions while the majority of functions remain in an idle state.

6.3 Workload Homogeneity vs. Heterogeneity

The difference in execution duration revealed in Section 5.3 shows that standard testing largely results in a similar workload and execution time. The tight clustering of results implies that the service time for any given request is to a large part predictable. In the Trace scenario the execution times vary from milliseconds to multiple seconds. This means that the time to process each individual request can vary greatly.

7 Discussion

This chapter discusses the results displayed in the previous chapter analyzing the implications of the observed statistical discrepancies on serverless system benchmarking.

7.1 Implications of Load Dynamics on Auto scaling

The synthetic models follow a smooth gradient. Auto scaling algorithms that rely on historical trends to scale resources would probably preform well against such a workload because the rate of change is constant and predictable (see Section 5.1). In contrast, the Azure trace shows fluctuations between time windows as shown in Figure 4 (A). With this small scale experiment it would be bold to claim that the auto scaler would be overloaded but the lack of a smooth trend may pose more of a challenge to the dampeners and volatility on a larger scale may trigger Hysteresis loops where the system oscillates inefficiently [2]. That being said we can conclude that synthetic models are more predictable but it cannot be concluded from the presented data alone that a trace replay would result in instability or hysteresis in the auto scaler without measuring on a per second basis and measuring the auto scaler activity in tandem.

7.2 The Limitation of Uniform Testing

In the Poisson/Gaussian scenarios the uniform distribution ensures that all functions are invoked frequently enough to most likely remain warm as cached in memory (illustrated in Figure 4 B). In contrast, the skew of the Trace scenario forces the majority of functions to be rarely invoked. In a production environment the unused functions would be released from the cache which would cause a cold start when it is invoked again. Based on the data in Section 5.2, it would be reasonable to conclude that benchmarks which rely on uniform function selection underestimate the latency of the system by keeping all functions hot.

7.3 Execution Variance and Head of Line Blocking

The difference in execution duration has some implications for scheduling efficiency. The clustering of synthetic results visible in Figure 4 (C) suggests a largely similar workload and execution time. In the Trace scenario the execution times vary from milliseconds to multiple seconds (Figure 4 D). As described in [4] the varying execution times could lead to head of line blocking where a smaller request gets stuck behind a slower one in a shared queue. By failing to include these outliers identified in Section 5.3 in the naive synthetic models such a test would fail to stress test the schedules ability to handle head of line blocking and as a result potentially increase the perceived throughput of the system.

7.4 Study Limitations

While this thesis validates the need for trace driven workloads the experiments were constrained by available resources. The experiments were conducted on a single node local cluster with a scaled down trace of 1000 requests. While the statistically representative distributions were preserved the absolute volume of traffic was on enough to trigger massive failure or large contention for resources. The data aggregation was limited to a 1 minute window (see x-axis in Figure 4 A). This smoothed out erratic bursts which are a characteristic of production traffic. While some trends can be observed the current data lacks the resolution required to draw robust conclusions regarding the sub second reactions of the autoscaler.

8 Conclusion

The rapid adoption of serverless computing has made it necessary to develop robust ways of testing the system. As FaaS platforms abstract away infrastructure the burden of orchestrator scheduling, scaling and caching falls on the provider in its entirety. For this reason accurate benchmarking of the orchestrators is critical. This thesis sets out with the goal to evaluate the validity of standard synthetic benchmarking by comparing them against the replay of production traces.

8.1 Summary of Findings

The research question of this thesis is: *To what extent do naive synthetic benchmarking methodologies distort the perceived performance of serverless orchestrators compared to trace driven production simulations?*

Based on the comparative experiments performed using the locust setup and OpenFaas platform it can be concluded that "naive" synthetic benchmarks using uniform popularity and Gaussian execution times distort the testing environment to some degree. They present an artificially stable and optimistic view of the system state in three key areas.

1. Underestimation of Cold Start Pressure: The biggest distortion observed is caused by the uniform function selection. Our results (Figure 4 B) show that the used approach splits traffic equally between functions keeping them warm in memory. In contrast the trace Driven simulation shows that the majority of functions are rarely invoked. Because of this the synthetic benchmarks fail to cause the release of the held cache storage and the resulting cold start penalties that are found in real world situations.

2. Oversimplification of Load Dynamics: The synthetic Gaussian models produce a balanced workload where most tasks complete in a small time window (0.3s-0.7s) as shown in Figure 4 (C). The Trace workload on the other hand shows a Log Normal distribution which included both small sub millisecond tasks and longer tasks taking multiple seconds (Figure 4 D). By the outliers not being represented in the synthetic benchmarks fail to create an environment where head of line blocking scenarios that stress the scheduler can be tested.

3. Masking of Scheduling Contention: While the Gaussian model successfully simulates a busy hour (Figure 4 A) it remains predictable. The production trace on the other hand while it is not really visible in the provided results prior research implies that real traffic is characterized by erratic fluctuations. Which would indicate that predictive autoscalers would perform better against something predictable as opposed to something non predictable like real world traffic.

8.2 Implications

The implications of the research is that naive synthetic benchmarking is insufficient for evaluating serverless infrastructure. Researchers or engineers testing a new autoscaler or snapshotting mechanism using the naive methods Poisson/uniform tests will possibly overestimate their system's throughput and stability. To achieve full test coverage benchmarking tools should adopt the trace replay approach that can test more unique edge cases common in real production traffic.

8.3 Future Work

Future research could build upon this methodology by scaling the ”universal function” approach (described in Section 4.1.2) to a cluster with multiple nodes or executing the trace replays against different orchestrator configurations. It would give more data regarding how much performance degrades under realistic workload conditions. In addition a more detailed collection of metrics would allow to draw more robust conclusions especially if the orchestrators themselves are monitored more closely how they react to different workloads.

References

[1] Mathieu Bacou. FaaSLoad: Fine-grained Performance and Resource Measurement for Function-as-a-Service. In *28th International Conference on Principles of Distributed Systems (OPODIS 2024)*, volume 28 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:21, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[2] Jonatan Enes, Roberto R. Expósito, and Juan Touriño. Real-time resource scaling platform for big data workloads on serverless environments. *Future Generation Computer Systems*, 105:361–379, 2020.

[3] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan Chidambaram, Mahmut T. Kandemir, and Chita R. Das. Fifer: Tackling underutilization in the serverless era. In *Proceedings of the 21st International Middleware Conference (Middleware '20)*, pages 280–295, Delft, Netherlands, 2020. Association for Computing Machinery.

[4] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. Hermod: Principled and practical scheduling for serverless functions. In *Proceedings of the 13th Symposium on Cloud Computing (SoCC '22)*, pages 289–305, San Francisco, California, 2022. Association for Computing Machinery.

[5] Christos Katsakioris, Chloe Alverti, Konstantinos Nikas, Dimitrios Siakavaras, Stratos Psomadakis, and Nectarios Koziris. Faasrail: Employing real workloads to generate representative load for serverless research. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '24)*, New York, NY, USA, 2024. Association for Computing Machinery.

[6] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218, Boston, MA, USA, July 2020. USENIX Association.

[7] Jaehyun Song, Bumsuk Kim, Minwoo Kwak, Byoungyoung Lee, Euiseong Seo, and Jinkyu Jeong. A secure, fast, and resource-efficient serverless platform with function REWIND. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 597–613, Santa Clara, CA, 2024. USENIX Association.

[8] Dmitrii Ustiugov, Dohyun Park, Lazar Cvetković, Mihajlo Djokic, Hongyu Hè, Boris Grot, and Ana Klimovic. Enabling in-vitro serverless systems research. In *Proceedings of the 4th Workshop on Resource Disaggregation and Serverless*, WORDS '23, page 1–7, New York, NY, USA, 2023. Association for Computing Machinery.

[9] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*, pages 559–572, Virtual, USA, 2021. Association for Computing Machinery.