Performance Characterization of Serverless Computing

Theodor Amariucai

4th Year Project Report
Artificial Intelligence and Computer Science
School of Informatics
University of Edinburgh

2021

Abstract

Serverless computing has seen rapid adoption because of its instant scalability, flexible billing model, and economies of scale. In serverless, developers structure their applications as a collection of functions, sporadically invoked by various events like clicks. The high variability in function image sizes, invocation inter-arrival times, and request burst sizes motivates vendors to scrutinize their infrastructure to ensure a seamless user experience across all of their services. To monitor serverless platform performance, identify pitfalls, and compare different providers, the public attention has turned to benchmarking, whereby measurement functions are deployed to the cloud to gather insights regarding response latencies, transfer speeds, and vendor policies.

This work introduces our open-source framework for serverless performance evaluation, intending to enable researchers and developers to benchmark multiple cloud platforms. Using this framework, we conduct one of the biggest serverless measurements to date, launching over 320,000 function instances to characterize system performance in AWS Lambda and vHive - the Edinburgh Architecture and Systems (EASE) virtual machine orchestrator system.

We find that server response times for cold function instances are, on average, ten times slower and more unpredictable than for warm function instances. When requests arrive in bursts of over 100, we show that while warm instances incur a penalty, AWS optimizes cold instances by fetching resources in batches. We achieve this by incorporating tail latency into our studies, a metric that is often overlooked in the literature. Our analysis further reveals that AWS allocates resources aggressively to avoid queueing and that invocations do not share function instances even when receiving up to 500 concurrent requests. We also discover that inter-function transfer latencies are much higher in AWS than in vHive and that AWS caps network bandwidth for low-memory configurations and throttles it for larger payloads regardless of allocated memory.

Based on these insights, we stress the importance of regular platform performance monitoring for improved customer satisfaction. Finally, to boost performance and economic efficiency from a user's perspective, we suggest that serverless applications are designed with a trade-off in mind between the function image size and the transfer chain length.

Acknowledgements

I would first like to thank Dmitrii Ustiugov for his commitment to the success of my project. Without his expertise and judicious guidance, this endeavor would never have been as elaborate and comprehensive as you find it today.

I would also like to thank my supervisor, Boris Grot, for all his much-valued feedback and support offered throughout my journey.

Table of Contents

Chapter 1

Introduction

Historically, in the traditional computing paradigm, application developers would buy or lease dedicated machines, typically in data centers, to operate their systems. Those machines required both significant initial capital expenditure and high ongoing operational costs. Moreover, as developers coped with peak computational loads in systems with varying demand, they had to plan in advance, provision, and pay for underutilized machines during periods of average load [5].

More recently, improvements in Internet connectivity and the rise of warehouse-scale computing systems have enabled web services to develop at an unprecedented rate [17]. An opportunity then presented itself to companies as they became able to outsource increasingly more elements of their technology stack: servers, storage, networking, virtualization (Infrastructure as a Service); runtime, middleware, operating system (Platform as a Service); language-level dependencies, data (Function as a Service); or even complete applications (Software as a Service).

As the cloud paradigm delegates increasingly more responsibilities to the provider, it allows companies to allocate fewer human resources towards infrastructural operations and more towards product development, quality assurance, and customer service. Because of this, cloud computing has become an essential part of every modern business [13].

This chapter introduces *serverless computing* and its positioning in the cloud ecosystem and describes how this novel paradigm brings value to the businesses that provide it and their clients. We further explain how critically assessing serverless infrastructures leads to an enhanced user experience and improved customer satisfaction. Finally, we outline the aims of this project by explaining (1) the research questions explored, which address the lack of quantitative serverless characterization data in the market, and (2) the steps that we took in designing our serverless benchmarking framework.

1.1 Serverless Computing

The oxymoron known as serverless computing is a recent branch of the cloud computing industry defined by two distinct collaborating components: Function as a Service

(FaaS) and Backend as a Service (BaaS).

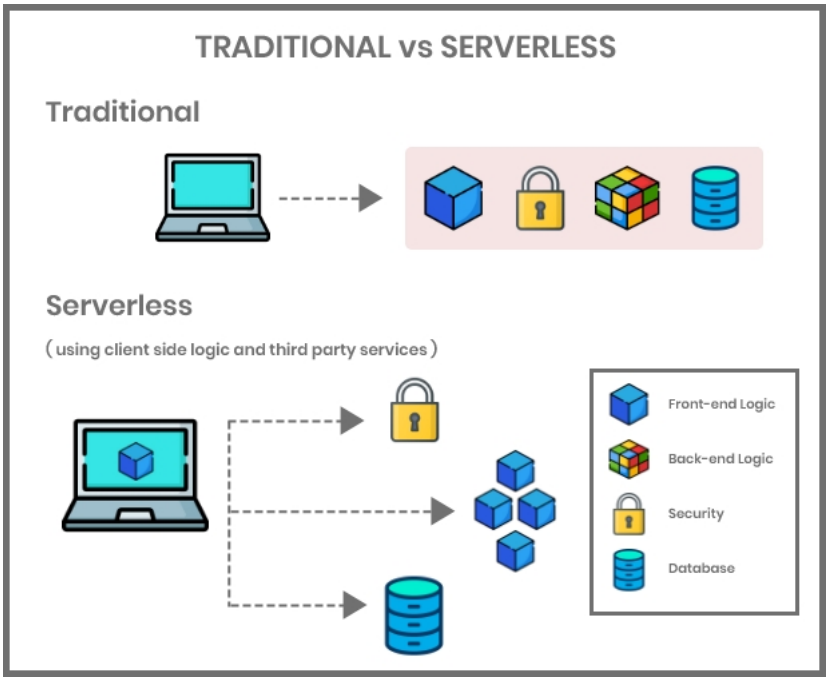

Figure 1.1: *Serverless changes the traditional computing paradigm* [31]

Function as a Service, the first component of serverless, is a way of executing modular pieces of code on demand. It allows developers to only focus on the function written and abstract away most of the usual time-consuming system administration operations, e.g., virtual machine management. With serverless offerings today, the developer specifies the cloud function memory size, execution time limit, language runtime, but not any other resource needs [25]. Finally, the provider proportionally allocates CPU resources (cores and computing time) to the amount of memory configured.

The lightweight nature of FaaS is then revealed: short run time, small memory footprint, straightforward configuration, and a stateless computation. Indeed, by itself, a FaaS solution will most likely be used as an event trigger, a generic webhook, or a scheduled maintenance operation. With such confined capabilities, use cases for this technology might seem very limited at first: how can complex applications be developed with such simple, inflexible functions? To overcome those limitations, FaaS is often paired with an ecosystem of BaaS offerings [25].

Backend as a Service, the second component of serverless, is a cloud service model in which developers outsource most of the behind-the-scenes aspects of a web or mobile application: hosting, authentication, database management, remote updating, push notifications. For example, a serverless application on AWS might use Lambda (FaaS) with S3 (object storage), DynamoDB (key-value database), and API Gateway (endpoint exposure). Step Functions can then glue these components together, centrally organizing multiple Lambda functions in a state machine language. This language can form a sequence of execution steps that allow the application to handle significantly more complex business logic.

We can now see serverless computing as an attempt at unifying the short-lived computations of FaaS with the robust networking, long-term storage, and security of BaaS. This combination brings a few advantages for consumers, most notably [4]:

1. **Faster delivery to market**. By eliminating operational overhead, teams can release quickly, get feedback, and iterate.
2. **Lower costs**. With a pay-for-value billing model, clients do not pay for over-provisioning, and their resource utilization is optimized on their behalf. For example, when comparing raw serverless costs to *serverful* alternatives[1], cloud providers report that customers see cost savings of up to 10x depending on the type of application[2] [39].
3. **Easier development**. Serverless applications have built-in service integrations, so developers can focus on building the application instead of configuring it.
4. **Seamless scalability**. With technologies that automatically scale from zero to peak demands, companies can adapt to customer needs.

It is, therefore, necessary for competitive businesses to embrace and excel at serverless.

Alongside serverless consumers, providers also benefit from this new paradigm:

1. **Economies of scale**. Enormous data centers yield energy savings and reduce operational costs for vendors by accommodating large customer pools.
2. **Amplified optimizations**. Security and performance adjustments apply to the entire infrastructure and all the clients who actively rely on it. For example, for a single security update to the microVM internal logic, improvements propagate through the entire infrastructure and are shared among hundreds of companies.
3. **Customer retention**. By offering all necessary backend services for an application, the potential to attract and keep customers increases, especially since each vendor offers unique features and workflows [1]. Many customer retention strategies involving innovative services thus become available.

The success story behind serverless computing stems from the computing research community's long-term efforts worldwide, which has received increasingly higher amounts of funding [13].

Despite its advantages and already widespread adoption, serverless today is mostly suited towards best-effort and latency non-critical services. Moreover, there is an underwhelming amount of quantitative data in the market that can accurately describe serverless systems on demand in a meaningful, structured way. As we transition into the next computing era, this is bound to change: there will be an increased demand for assessing and monitoring the quality of provider offerings as by 2026, over two-thirds of all enterprises across the globe will be entirely run in the cloud [13].

[1] For example, Amazon Elastic Compute Cloud (EC2).

[2] For example, web/mobile/IoT, streaming, or batch computations.

1.2 Motivation

Tail latency (Section 2.1.1) is the small percentage of responses from cloud systems that take the longest compared to the bulk of the responses. As soon as services scale up, tail latency inspection detects when responses slow down and quantifies the phenomenon. For example, the percentage of slow requests can surge from 1% to 63% as parallelism increases from 1 to 100 servers (Section 2.1.1). Companies aim to satisfy 99% of customers, not 37%, which is why we scrutinize tail latency in our studies.

Designing an efficient, scalable, and user-friendly serverless platform that exhibits low tail latencies remains challenging for system software designers [42]. Firstly, clients expect satisfying response times even when the system performs under stress conditions during peak demands. Secondly, new difficulties arise as these innovative but intricate serverless systems are developed:

- Functions have a small memory footprint. They now have to be tightly packed onto servers for economic reasons, leading to performance hiccups [42].
- Functions are short-lived. A new instance is created for each incoming request if there are no idle ones to reuse, leading to startup latency.
- Functions are stateless. They have to be efficiently chained together to accommodate complex business logic, leading to data transmission overhead.

To monitor serverless platform performance, the public attention has turned to benchmarking, whereby measurement functions are deployed to the cloud to gather insights regarding vendor policies and average response times. In time, the degradation of such metrics can lead to a poor customer experience and low client satisfaction.

Many extensive attempts have already been made to characterize serverless systems (Section 3). However, experiments conducted were generally either lacking tail latency analysis, not broad enough, not configurable enough, no longer maintained, or had a different focus: statistical soundness [27], user billing and expenses [23, 14]. Moreover, tail latency plays a critical role in online services' responsiveness yet is missing from many benchmarking endeavors in the literature (Table 3.1).

Another reason for developing a new serverless evaluation framework, apart from addressing deficiencies in the literature, is to compare multiple providers based on an established set of criteria. Because of the current lack of quantitative data in the market that can accurately describe some specific aspects of performance and workload adaptability in serverless infrastructures (Section 1.3), it is currently hard for users to decide on adopting any particular serverless provider [36].

Next, we present this project's goals, explaining how we address the shortage of tail latency studies in the literature and describing the unique contributions that we bring to the serverless profiling community in the form of our specialized framework.

1.3 Project Aims

A *cold* start (Section 2.1.2) occurs when the code has not been executed in a long time (5–25 minutes) and has to be loaded from the disk. To boost response times, serverless infrastructures keep function instances in memory for prolonged durations, allowing them to reply to requests more quickly (*warm* start). As serverless workloads are often bursty (e.g., popularity peaks and declines, breaking news emerge), deployments are not over-provisioned but auto-scaled on demand (e.g., new instances are spawned); hence cold starts are the first candidate for high tail latencies in serverless systems.

The high variability in function image sizes, data transfer payloads, invocation inter-arrival times, and request burst sizes is another reason cloud providers experience high tail latencies. As vendors often guarantee exceptional performance levels, they must detect when performance drops below acceptable levels and quickly identify and fix the problem [34] to ensure a seamless user experience across their services.

We next investigate four research directions that cause undesirably high tail latencies by slowing down the auto-scaling process in serverless computing: bursty server behavior, function image fetch delay, active requests queuing, and inter-function transfer speeds. As those dimensions have been understudied in the literature (Table 3.1), this project brings new valuable contributions to the community's characterization efforts.

1.3.1 Sources of Tail Latency in Serverless

1. **Bursty Behavior**. Under peak demand, the server experiences an unusually high number of simultaneous incoming requests. Such large bursts could overwhelm the receiving end's physical resources (e.g., the load balancer) and cause response delays or even loss. We show that when requests arrive in bursts of over 100, while warm instances incur a penalty, AWS optimizes cold instances by fetching resources in batches. We achieve this by incorporating tail latency into our studies; a metric often overlooked in the literature.

2. **Image Fetch Delay**. In a cold start, the binaries have to be downloaded, containerized, booted, and primed to be run (Section 2.1.2). Depending on the binaries' size, this process can add significant delay to the system's response time (20–60 times worse performance) [30]. Indeed, our studies reveal that image size plays a direct role in determining the cold start latency. We also find that server response times for cold function instances are, on average, ten times slower and more unpredictable than for warm instances.

3. **Request Queuing**. We seek to find whether providers queue incoming requests for efficiency purposes and, if so, to discover the number of requests that trigger this scenario. For example, for short service times, 500 concurrent requests might be processed more efficiently by spinning up 250 instances and delivering two requests to each, rather than cold-starting 500 instances. We discover that AWS allocates resources aggressively to avoid queuing and that invocations do not share function instances even when receiving up to 500 concurrent requests.

4. **Inter-function Transfer Speeds**. Serverless computations have to be efficiently

chained together to accommodate for more complex business logic. This chain of communication takes the form of inter-function data transfers (inline or via storage) and has to be efficient to prevent bottlenecks. Despite this, data transmission in real-world serverless systems uses storage, which is radically slower and more expensive than point-to-point networking [24]. Our analysis reveals that inter-function transfer latencies are much higher in AWS than in vHive and that AWS caps network bandwidth for low-memory configurations.

Next, we explain why the existing profiling tools cannot satisfy our research goals by identifying their main limitations and introduce our benchmarking framework that thoroughly evaluates each of the four research dimensions above.

1.3.2 Benchmarking Framework

After reviewing the serverless profiling literature (Section 3), we discovered that most experiments conducted were generally either lacking tail latency analysis, not broad enough, not configurable enough, no longer maintained, or had a different focus: statistical soundness [27], user billing and expenses [23, 14].

To reach our research goals, we contribute to the community's serverless profiling efforts by creating an open-source, parametrized benchmarking framework from scratch (further detailed in Section 4.1). In its design, we consider all the essential requirements necessary for describing tail latency across multiple serverless platforms:

- **Flexibility and preciseness**. We use the Golang programming language, which provides remarkably convenient features for our purposes, e.g., simple multi-threading and concurrency using Goroutines and WaitGroups.
- **Configurability**. We wish to model warm, cold, and bursty invocations, as well as data transfers (inline or via storage). Therefore, the tool must vary the workload characteristics (e.g., image size, service time) and alter the traffic shape (e.g., inter-arrival times, burst sizes) according to the specifications.
- **Speed**. Even when modeling cold starts, the client must perform the benchmarking in an acceptable amount of time. Furthermore, function deployments should be automated to a high degree for an optimized turnaround speed.
- **Reproducibility**. All experiments have to be easily repeatable, and the results between two runs must be similar, unless because of the instability of the system under test.
- **Robustness**. The framework has to avoid technical pitfalls, e.g., client-side contention caused by responses queuing at the network interface controller (NIC). It also has to adopt a solid statistical approach, e.g., take sufficient samples.
- **Independence from any provider**. Multiple high-quality serverless providers must be supported. Our tool currently covers AWS Lambda and vHive, the open-source Edinburgh Architecture and Systems (EASE) VM orchestrator system.

Images are an essential tool for analyzing and describing tail latency in real-world systems. For this reason, we used the benchmarking client's output to create visualizations

that help disseminate the enclosed information and lead to a better understanding of the data. For this purpose, our custom-made plotting scripts can read latency records from disk and generate summarizing graphs, as featured in Section 5.

We ensure our evaluation suite's maintainability through a continuous integration pipeline that regularly verifies the correctness of all framework aspects. It includes syntax checks (source code is analyzed to flag programming errors, bugs, stylistic errors, and suspicious constructs), unit tests, integration tests, and build checks for the measurement function and the benchmarking client.

Apart from the checks mentioned above, we are interested in validating the data obtained in our experiments. "Performance measurements often go wrong, reporting surface-level results that are more marketing than science" [28]. To avoid this, we take four precautionary steps (further detailed in Section 4.2) toward higher-quality performance measurements [34]: we allow enough time, we measure one level deeper, we consider additional detail, and we minimize statistical variability.

1.4 Report Structure

This project is situated at the intersection between serverless computing and performance benchmarking. The report is structured as follows:

- Chapter 2 offers the technical background necessary for understanding our work. It begins with an overview of server performance and how it leads to customer satisfaction and continues with two phenomena linked to lower overall serverless system performance: tail latency and cold starts. In the last section, we investigate the common challenges present in a typical FaaS architecture.
- Chapter 3 is entirely dedicated to placing our work in the broad context of serverless computing benchmarking. This topic has recently been receiving increasingly more attention in the academic community, which is why we take this opportunity to shed some light on our contributions. We present the challenges we address and what differentiates our tool from others.
- Chapter 4 focuses on the methodology employed in our experimentation. We begin with the inner workings of our tool: the benchmarking procedure and client parameters (e.g., inter-arrival time, burst size), coordinator steps (flowchart), and, finally, the measurement function and its workload parameters (e.g., service time, transfer size). We then present the hardware used and explain the four steps we took toward higher-quality performance measurements [34].
- Chapter 5 covers the five main experiments that we performed: we assessed CPU slowdown (auxiliary), described the bursty server behavior, observed the image fetch delay, found whether active requests are queued in AWS, and measured the transfer speeds (inline and via storage). We then present and interpret the data.
- Chapter 6 reviews our main achievements and contributions, presents five directions for further work and concludes this report.

Chapter 2

Background

We begin this chapter by explaining why low overall tail latency is one of the most desirable characteristics when assessing serverless systems' performance.

To understand the tail latency phenomenon (Section 2.1.1), we define it, visually describe it (Figure 2.1), and assess the impact that it has on businesses. We then look at cold starts (Section 2.1.2), one of the principal causes behind high tail latencies, and explain why they often lead to decreased cloud system performance.

Finally, we analyze common challenges present in a typical FaaS architecture to comprehend its inner workings and better direct our performance characterization efforts (Section 2.2).

2.1 Server Performance

Systems that respond to user actions quickly (within 100ms) feel more fluid and natural to users than those that take longer [12]. Since user satisfaction directly translates into revenue, current industry leaders in cloud infrastructure will find reliability in the form of speed and fault tolerance guarantees highly desirable.

Indeed, latency requirements apply to most production workloads in the serverless ecosystem. Among the applications analyzed by the Standard Performance Evaluation Corporation research group (SPEC RG), 32% of them have latency requirements for all functionality, 28% have partial requirements, and 2% have real-time requirements [20]. Those requirements regard both the average and tail response latencies.

Nevertheless, technology providers have not traditionally pursued tail latency optimization to the same extent because of its more subtle influence that significantly impacts customer experience only as the applications scale up. This trend, however, is about to change as tail-tolerant operations, an area still under research and exploration [17], have attracted the attention of increasingly more businesses in recent times.

2.1.1 Tail Latency

Tail latency in the serverless scenario is the small percentage of response times from a system that, out of all the responses serviced, take the longest compared to the bulk of its response times. They are at the tail end of a system's response time spectrum and are often expressed as the 99th percentile response times [38].

We can observe the concept of tail latency visually through the empirical cumulative distribution function (CDF) in Figure 2.1. Each hollow, black point in the image represents a response latency sampled from AWS Lambda during a run of our benchmarking tool. The right-hand side of the plot, resembling a "tail" and situated to the right of the red dotted line, displays much higher response latencies (74ms+) than the average expected value (47ms). To stress this contrast, we highlight the average response latency with a black dotted line.

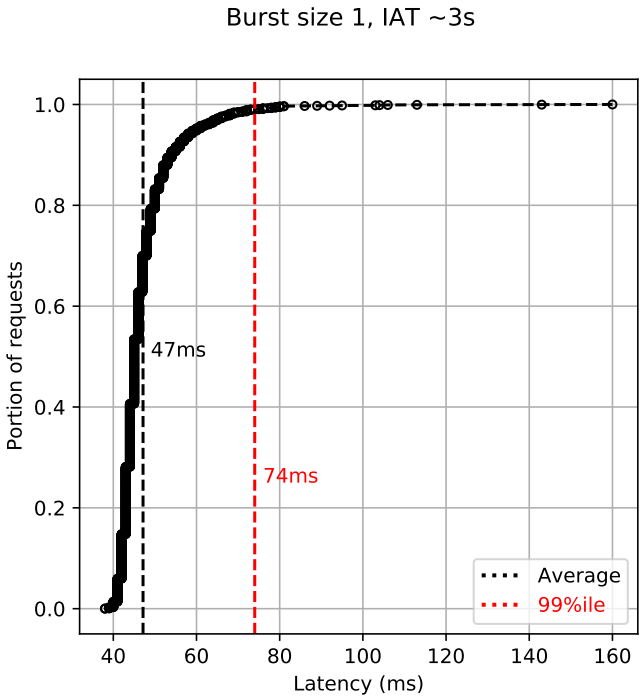

Figure 2.1: *Empirical CDF of 3000 requests issued every 3 seconds*

To assess the impact that tail latency has on businesses, let us once again consider the example in Figure 2.1. The server typically responds within 47ms but with a 99th-percentile latency of almost double the time (more than 74ms). In other words, if user requests are handled on just one such server, 1 user request in 100 will be slow (take double the time). We notice that service-level latency in this scenario is affected by a very modest (1%) fraction of latency outliers [10].

At first glance, we can easily underestimate the importance of measuring tail latency in serverless systems. Compared to the impact that average latency has on user experience, tail latency often fades away in significance. After all, it represents only 1% of the responses. However, this is a severe mistake, and enterprises often pay the price for overlooking this crucial measure when their services scale up.

If a user collects responses from 100 servers in parallel, 63% of requests will be slow. Even for services with only 1 in 10,000 requests experiencing high latencies at the single-server level, 2,000 servers responding to requests in parallel will see almost 1 in 5 responses take almost twice as much time as the rest [10]. We can see how the impact of tail latency on performance is amplified as soon as the services scale up.

Companies aim to satisfy 99% of customers, not 37%. Cloud services set their standards similarly high, aspiring towards 99% satisfactory response times. Nevertheless, there are no hard guarantees today: serverless is mostly suited towards best-effort and latency non-critical services, and tail latency plays a direct role in this.

While eliminating all latency variability sources in large-scale systems is impractical, tail-tolerant software techniques attempt to form a predictable whole out of less predictable parts [17]. To track their progress and impact, serverless benchmarking tools can be used to plot latency distributions as in Figure 2.1 and to monitor changes.

Next, we look at cold starts - the principal cause behind high tail latencies and, consequently, behind undesirable response times in serverless systems.

2.1.2 Cold Starts

A *cold* start happens when a function takes longer than usual to execute [29]. This mostly occurs when the invocation is scheduled right after deployment or when the function has not been used in a while, e.g., after 10 minutes without invocations. However, when the function is already in memory and can immediately serve requests, we regard it as *warm*.

The cold start impact on FaaS performance is still a key obstacle of serverless adoption today, especially for latency-sensitive workloads [37]. When a cold start occurs, the function code must be downloaded, containerized, booted, and primed to be run. An example of this process can be seen in Figure 2.2: multiple steps and components

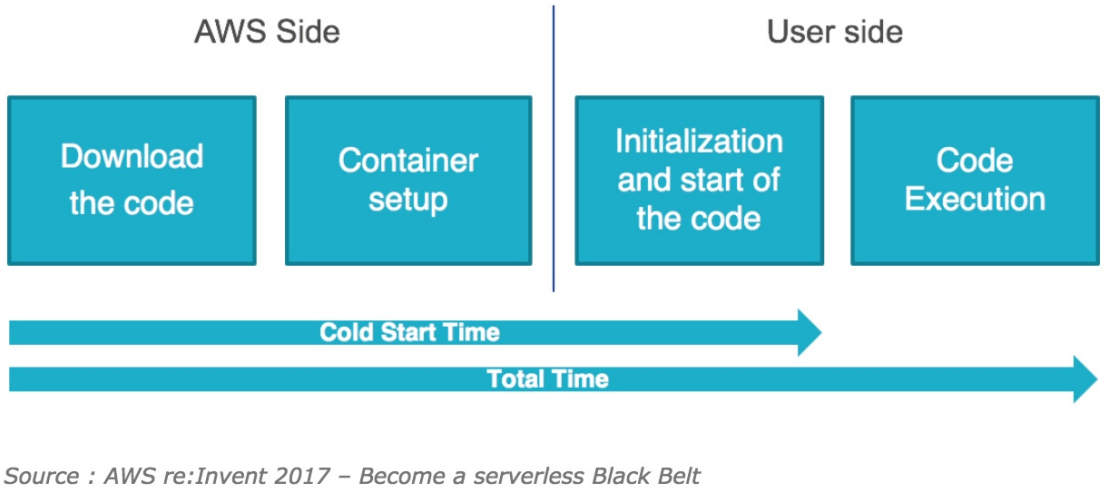

Figure 2.2: *Cold start overhead in a serverless system [33]*

which ensure correct operation are involved, but they also slow the system down considerably.

However, promising solutions in this space include reducing overhead through better engineering, profiling function performance more accurately, or introducing more sophisticated scheduling policies [37]. Serverless providers already take different approaches with the latter: on AWS and IBM, it usually takes around 10 minutes of no activity for the computing instance to be recycled and re-inserted into the pool of available ones, and up to 20 minutes on Azure. On Google, this time varies from 10 minutes up to 10 hours [29].

When given a chance, customers often replace non-responsive applications with faster alternatives. This is why the technology industry has been so heavily focused on cold starts: warm function instances dramatically improve average response times.

The direct connection between cold starts and tail latency can also be recognized: even when invoking a function at short intervals to keep the instances warm, at least the first invocation, which involves downloading and priming code, will be cold. In turn, this manifests in the form of a tail on the application's latency distribution (Figure 2.1).

In the next chapter, we cover some common serverless architecture challenges that might aggravate tail latency and lead to poor system performance when not addressed.

2.2 Serverless Architecture Challenges

High tail latency in a service's individual components can increase the variance in the entire response time distribution. Moreover, component-level variability in cloud systems is drastically amplified by scale. Some of the causes behind this phenomenon include [17]:

- Machine resource sharing (CPU cores and caches, memory, network bandwidth)
- Global resource sharing (network switches and shared file systems)
- Maintenance activities (e.g., periodic garbage collection in some languages)
- Multiple layers of queuing in intermediate servers and network switches

The over-provisioning of resources and careful real-time software engineering can be used at all levels and in all components to reduce the base causes of variability in cloud infrastructures [17].

Serverless systems today, however, face a set of additional challenges at the *Function Management Layer* and *Workflow Composition Layer* [37]. We next explore three particular examples that will guide both the design of our performance evaluation framework and our experimental analysis in Section 5: startup time minimization, scalability, and function composition.

2.2.1 Startup Time Minimization

Although cloud functions have a much lower startup latency than traditional VM-based instances [25], the delays incurred when starting new instances can still be high for some applications. As previously mentioned in the context of cold starts (Section 2.1.2), we identify four elements that make up most of a serverless function's startup time [25]:

1. Scheduling and provisioning resources
2. Downloading the application software environment (e.g., source code, operating system as part of the container setup)
3. Performing application-specific startup tasks (e.g., loading and initializing the libraries and data structures)
4. Executing the function code

The intermediate steps above can dwarf the others. While it usually takes less than one second to start a cloud function, it can take tens of seconds to load all application libraries [11]. Downloading the function source code (or image) often also incurs a relatively significant delay, as the process relies on two critical factors: storage performance and network bandwidth.

The location where the image is pulled from plays a role in determining the reply's speed: the sooner the code is available, the sooner the container can start. The function registry (*Function Management Layer*) serves as a local or remote repository for serverless functions and keeps the function binaries in a store. The location of this store, in turn, influences the startup time of cold function instances [37]. This also hints at the negative influence that larger binaries can have on system performance: the more code has to be fetched, the slower the responses will be.

To quantify the image fetch delay of an arbitrary serverless function, the image size could be artificially altered and the change in response latency measured (Section 5.3).

2.2.2 Scalability

"Elastic, automatic scaling in response to changes in demand is a main advertised benefit of the serverless model" [40]. Therefore, it comes as no surprise that most of the use cases in serverless computing indeed experience bursty workload patterns [19].

A bursty workload follows a pattern that involves sudden and unexpected load spikes or a significant amount of sustained noise and variation in intensity. This behavior is mainly unpredictable for any scenario that involves a set of human users, as user behavior can seldom be scheduled or reliably controlled [19].

For this reason, engineers that have been struggling with bursty workloads and face constant performance issues are more likely to migrate their applications to the cloud to ensure application consistency and to benefit from scalability guarantees [19].

Nonetheless, how is the cloud able to provide those guarantees and ensure seamless application scalability? One of the mechanisms involved uses function routers at the

Function Management Layer. Their job is to route incoming requests or events to the correct function instance. If no function instance is available, the function router queues the events to await the termination of currently-running instances [37]. The cloud provider can also set up policies that eliminate queuing times: the function router would only wait for the booting of new function instances, which is often faster.

This queuing behavior can have negative consequences on an application's response times: the more requests are queued as they wait for data to be processed by the server, the slower the response times.

Therefore, serverless performance evaluation suites should understand how bursty workloads affect scalability and application performance (Section 5.2). Moreover, they should identify request queuing when it occurs and quantify it accordingly (Section 5.4).

2.2.3 Function Composition

The ability for a serverless application to grow in complexity relies on function composition and coordination. Platforms today, however, have no knowledge of the data dependencies between serverless functions, let alone the volume of data they might exchange [25]. This ignorance can lead to a sub-optimal placement at the *Workflow Composition Layer*, causing inefficient communication patterns.

One such communication pattern uses a producer-consumer scenario to share states between functions. In this case, consumers need to know as soon as the data is available from the producers [25]. This process is often negatively impacted as the transfer between cloud functions is usually done through slow object storage systems.

Object storage services such as AWS S3, Azure Blob Storage, and Google Cloud Storage are highly scalable and provide inexpensive long-term object storage, but exhibit high access costs and high access latencies [25]. According to recent tests, these services take at least ten milliseconds to read or write small objects [6].

While capacity demands vary, all functions rely on slow storage to maintain and transfer the application state during its lifetime. Once the application finishes, the state can be discarded. This suggests the need for developing new, fast, ephemeral, and durable serverless storage to ease some of the issues posed by current storage solutions [25].

To validate the implementation of fast ephemeral storage and other intermediate performance adjustments made on serverless infrastructures, a baseline should be established and regularly consulted - much like a snapshot of the system in time.

Serverless performance evaluation suites should therefore characterize inter-function transfer speeds and monitor changes over time (Section 5.5).

Chapter 3

Literature Review

Serverless computing has seen a plethora of academic interest in recent years, and for a good reason. With serverless computing, developers fully delegate server management and are billed entirely based on usage with millisecond precision. Matters of availability, resource allocation, and fault-tolerance become the cloud provider's responsibility, and developers can solely focus on the application logic.

In a survey by the SPEC research group, the main drivers of serverless adoption were found to be the reduced operational cost (33%), reduced operational effort (24%), scalability (24%), and, lastly, the performance gains (13%) [19]. However, the serverless computing ecosystem is broad and diverse: use cases include machine learning pipelines, video processing workflows, map-reduce workloads, HTML rendering.

To devise a relevant benchmarking framework, we need to identify, at an abstract level, the composition of this novel ecosystem: use cases, characteristics, structure, and organization of applications.

3.1 Serverless Ecosystem

In one of their studies, SPEC RG compile and dissect 89 serverless use cases from four unique sources: 32 from open-source projects, 23 from white literature, 28 from grey literature, and 6 from the area of scientific computing [19]. This study is highly relevant to us as we attempt to guide our benchmarking tool's design by real-world insights: out of all applications analyzed, at least 55% of them are already in production.

Next, we look at two sets of serverless function characteristics determined and formalized by SPEC RG: *application* characteristics and *workload* characteristics.

3.1.1 Application Characteristics

Application characteristics describe the serverless application's structure and properties and focus on characteristics such as "How many functions does the application consist of?" and "Which managed cloud services does the application use?" [19].

The first observation is that the currently dominating platform for serverless applications on the market is AWS Lambda, with 80% adoption [19]. This fact motivated our decision to make AWS Lambda the first serverless vendor supported by the framework.

The most popular function chaining patterns are also of interest because of the potential impact of inter-function transmission rates on overall application performance. The granularity of serverless functions is still controversial: opinions range from wrapping each programming function as a serverless function to full microservices as a serverless function. On this matter, we find that 68% of the applications have at least two different inter-communicating functions in their composition [19] and this motivates our experiments in Section 5.5.

Serverless applications depend on a wide variety of cloud services, with the three most used ones being cloud storage (used by 61% of the applications), cloud database (47%), and cloud API gateway (18%) [19]. Given the ephemeral nature of FaaS functions, and consistent with other survey results [28], it is unsurprising that persistency services are the most popular external services [19].

In the following subsection, guided by the literature on the subject, we look at common workload parameters in serverless functions.

3.1.2 Workload Characteristics

Workload characteristics aim to describe the traffic patterns of a serverless function, e.g., "How long do functions usually run for?" "Is the workload bursty?" and "What is the data volume per request?" [19].

Most of the serverless functions surveyed (67%) are short-running, with running times in the order of milliseconds or seconds [19]. Therefore, our measurements' target precision is in the order of milliseconds, as anything more granular adds little to no introspective benefit. Most serverless providers bill as a multiple of 100 milliseconds, with AWS being a recent exception: starting December 2020, they round up duration to the nearest millisecond with no minimum execution time [21].

While the network and storage device load among most of the surveyed use cases is low (44% of them exhibit less than 1MB), we notice that a significant amount of them (37%) displace between 1MB and upwards of 1GB of data [19]. This is why, in our evaluation framework experiments, we cater for a wide spectrum of data volume loads: code sizes of up to 230MB for image fetch delay (Section 5.3) and payloads of up to 1GB for network transfer bandwidth and latency (Section 5.5).

In any scenario that involves a set of human users, we consider the workload pattern to be bursty, as user behavior can seldom be scheduled or reliably controlled [19]. With this in mind, the SPEC research group found that 81% of the analyzed use cases exhibit bursty workloads. For this reason, we look at burstiness in serverless platforms from multiple angles: median latency, tail latency, and overall variability in conjunction with the function image size (Sections 5.2 and 5.3).

3.2 Benchmarking Tools

Studies show that serverless computing today might be far from ideal [24]: it is constrained (limited function lifetimes, no specialized hardware), it processes data inefficiently (I/O bottlenecks due to low network bandwidth and slow storage), and wrong policies can work against it (e.g., inadequate request queuing). Many serverless application developers have conducted their own experiments to measure cold start latency [15], function instance lifetime [16], maximum idle time before shut down [41], and CPU usage [7]. Unfortunately, their experiments were ad-hoc, and the results may be misleading because of the lack of control over contention by other instances [40].

Another issue is that of breadth. For example, the cold start latency study in [15] assesses function images of up to 15MB, while our study goes up to 230MB (Section 5.3) and finds contradictory results. Similarly, *ServerlessBench* evaluates the image fetch delay for code sizes of up to only 70MB [42]. Inter-function transmission latency analysis in *ServerlessBench* also lacks in depth: payloads of up to a maximum of 50KB are considered as opposed to 1GB in our studies (Section 5.5). Such missed opportunities can lead to insightful findings.

One notable paper featuring similar characterization goals to ours is *Peeking behind the curtains of serverless platforms* [40]. They conduct experiments exploring cold-start latency, CPU slowdown, and network throughput. We look at similar metrics but focus on other areas as well: tail latency as observed through CDFs, variability in the image sizes and transfer payloads, and bursty server behavior. Moreover, some of our experiments are identical in scope yet report different results, e.g., CPU slowdown in AWS seems to have increased since 2018 (Section 5.1). The most probable cause for this is updated vendor policies. Other studies in the paper appear to be outdated as well. For instance, the security study regarding VM tenant isolation is no longer applicable because VM co-residency of cross-tenant function instances was excluded by design: in AWS, only one function instance is now assigned to a microVM [3].

Another issue that permeates many serverless performance characterization endeavors is rigid code used solely for research purposes and no longer maintained afterward. Some examples include *FunctionBench* (5 pre-configured file sizes to measure S3 throughput) [26] and *ServerlessBench* (11 pre-configured, highly specific tests) [42]. In contrast, in our studies, we run all of our experiments with the same internal logic for enhanced maintainability and configurability; the only elements that change are the JSON configurations (more details in Appendix A).

Our methodology (and many others) focuses on platform-level or end-to-end issues and the reverse engineering of commercial services' behavior. Another approach is taken by *FaaSprofiler* [36]: by benchmarking the *open-source* Apache OpenWhisk FaaS platform, a higher degree of introspection is possible: branch mispredictions per kilo-instruction, inter-function interference (Figure 3.1). Nevertheless, some of the studies are remotely similar to ours, e.g., container slowdown, function cold starts.

We pay close attention to statistical accuracy (Section 4.2). Other serverless evaluation frameworks, however, make statistical soundness one of their key priorities. Such an example is *LANCET*, a microsecond-scale tool designed to measure open-loop tail

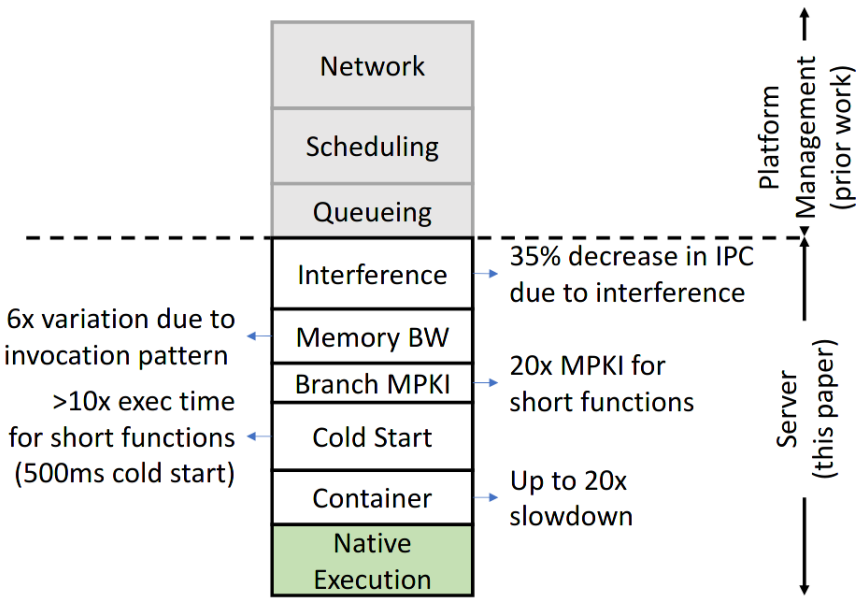

Figure 3.1: *Server-level overhead of FaaS applications (FaaSProfiler) [36]*

latency [27]. It leverages self-correcting statistical testing techniques such as Pearson auto-correlation or the Anderson Darling Test and offers high precision using hardware time-stamping at the NIC. They also automatically detect the optimal number of samples to collect for statistically significant results.

Other serverless performance evaluation tools seek to provide economic insights for the end-user, which is out of our project's scope. For example, *ServerlessBench* [42] emphasizes how many resources should be provisioned for optimal savings, plots billing statistics, and analyzes costs for three real-world applications involving Alexa skills, image processing, and data analysis [42]. Another example is *SeBS* [14], where some of the studies involve assessing the fairness and efficiency of pricing models and finding how expensive offloading computations to FaaS platforms can be.

Our project is a micro-benchmark framework that evaluates single general aspects of FaaS platforms. A more specialized, application-driven approach is taken by *BeFaaS*: just like *ServerlessBench*, they focus on evaluating realistic use cases of FaaS applications. They include two built-in benchmarks (e-commerce and an IoT application) and the option of importing your own application (not verified) [23]. Another example is *SeBS*, where practical applications include web microservices, multimedia processing pipelines, compression utilities, graph computations, and image recognition engines.

Other frameworks place their focus on sophisticated visuals[1] and the sheer number of different regions and programming languages analyzed. Such an example is *FaaSDom*, a platform profiling across seven languages and four providers spread over 39 regions [29]. However, they outsource their throughput/latency experiment infrastructure to *wrk2* [2], which arguably compromises on precision: we measure in the order of milliseconds, whereas *wrk2* uses requests/seconds.

[1]For example, Graphana, the multi-platform, open-source, interactive visualization web application.

Benchmarking Tool	Tail Latency	Bursty Behavior	Image Fetch	Function Transfer
This Work	✓	500 reqs.	230MB	1GB
ServerlessBench [42]	✓	✗	72.6MB	50KB
LANCET [27]	✓	✗	✗	✗
SeBS [14]	✗	✗	✗	6MB
FaaSprofiler [36]	✗	100 reqs.	✗	✗
FaaSDom [29]	✗	✗	✗	✗
Peeking [40]	✗	✗	✗	✗
BeFaaS [23]	✗	✗	✗	✗
FunctionBench [26]	✗	✗	✗	✗

Table 3.1: Our micro-benchmarking contributions to the serverless ecosystem

We conclude our literature review by noting the lack of tail latency studies in most examined papers. Furthermore, experiments conducted are generally either not broad enough, not configurable enough, no longer maintained, or have a different focus: statistical soundness [27], user billing and expenses [23, 14].

Table 3.1 summarizes the differences between our solution and other serverless benchmarking tools. In order of priority, the criteria are as follows:

1. A tail latency study analyzes the 99th percentile server latencies through cumulative distribution functions or other visual or tabular means.
2. A bursty behavior study (Section 5.2) explores how bursty workloads affect overall application latency. We do not consider the cases where concurrent requests are used to analyze instance microVM placement (as in [40]), or where the unit of measurement is too imprecise (e.g., requests/second, as in [29]).
3. An image fetch delay study (Section 5.3) analyzes whether cold start performance is affected by the function source code size (image size). Any measurements where the image size is fixed (as in [40, 36]) are not considered.
4. A function transfer study (Section 5.5) evaluates data transfer speeds (either inline or via storage) between instances in a producer-consumer scenario. Any measurements where the payload size is fixed or highly specific (as in [23]) are not considered.

Chapter 4

Methodology

We propose a serverless evaluation framework written in Golang that can benchmark multiple providers and offer performance insights along four research dimensions: bursty server behavior, function image fetch delay, request queuing, and inter-function transfer speeds. To achieve this, we model the incoming traffic accordingly: we vary the incoming burst size, function image size, and transfer payload size (Figure 4.1).

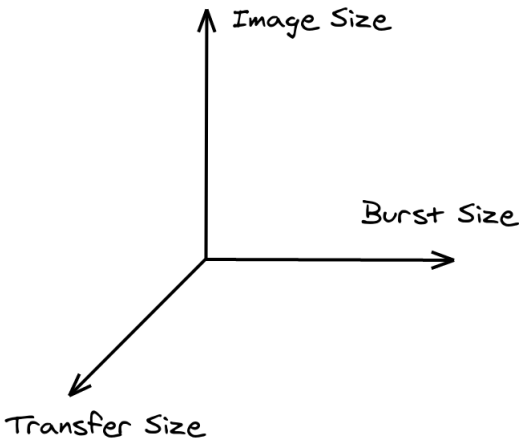

Figure 4.1: *Potential tail latency variability vectors in serverless computing*

We take the viewpoint of a serverless user to characterize serverless platforms' performance and resource management efficiency. Vantage points are first set up in the same cloud provider region to manage and invoke functions from one account via official SDKs. We execute experiments under various settings by adjusting function configurations and workloads, and we interpret the key factors that impact measurement results. From this perspective, our methodology is similar to that of [40].

We begin this chapter by explaining the inner workings of our tool: the benchmarking procedure and client parameters (e.g., inter-arrival time, burst size), coordinator steps (flowchart), the measurement function and its workload parameters (e.g., service time, transfer size). We then present the hardware used and explain the four steps that we took toward higher-quality performance measurements [34].

4.1 Serverless Evaluation Framework

To begin with, we provide an overview of our benchmarking solution and define the main terms used throughout the rest of our studies:

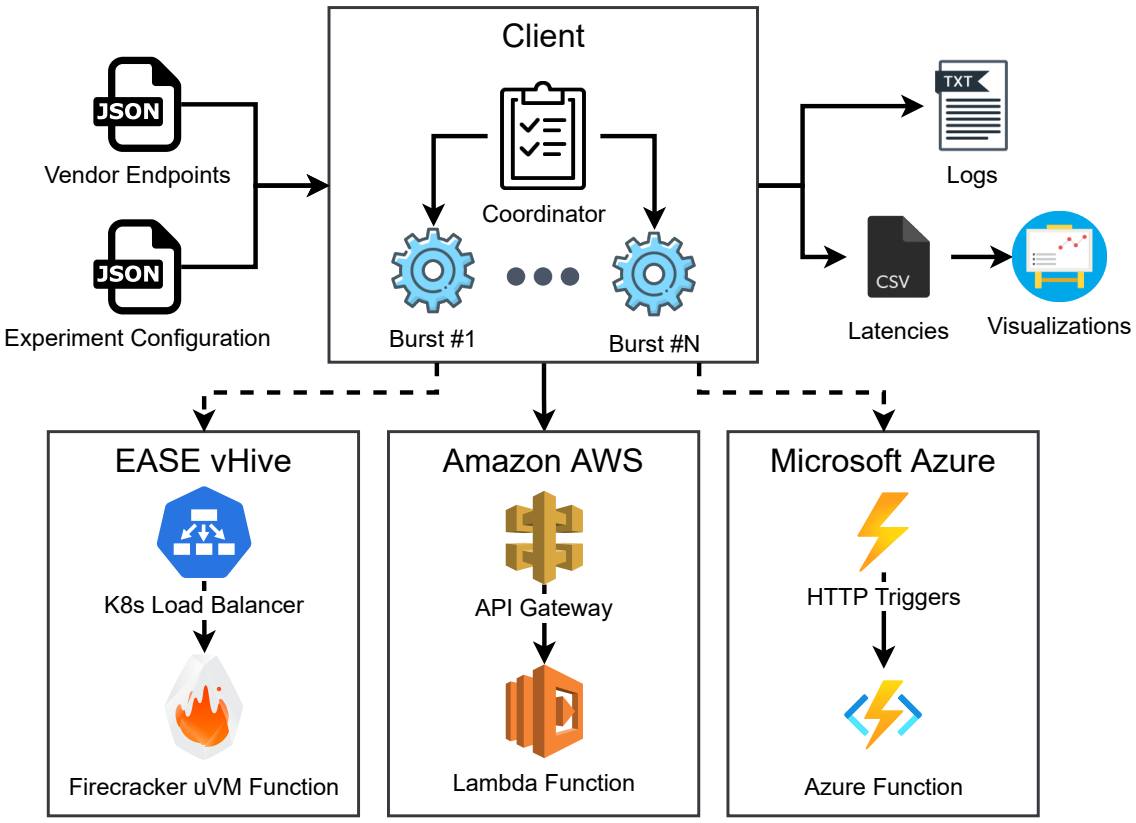

Figure 4.2: *Benchmarking client overview*

- The *coordinator* orchestrates the entire benchmarking procedure (Figure 4.4).
- The *experiment configuration* (Figure 4.2) is an input JSON file used to specify and customize the experiments (example included in Appendix A).
- An *endpoint* is a URL used for locating the function instance over the Internet. As seen in Figure 4.2, this URL most often points to resources such as AWS API Gateway[1], Azure HTTP Triggers, vHive[2] Kubernetes Load Balancer, or similar.
- The *vendor endpoints* (Figure 4.2) input JSON file is only used for providers such as vHive that do not currently support automated function management (e.g., function listing, deployment, repurposing, or removal via SDKs or APIs).
- The *inter-arrival time* (IAT) is the time interval that the client waits for in-between sending two bursts *to the same endpoint*. To add some variability and simulate a more realistic scenario, we sample this from a shifted exponential distribution. For example, if we set the IAT to 10 minutes (modeling cold starts for most vendors), generated values can be, e.g., 10m12s, 10m27s, 11m.

[1]This uses the HTTP protocol, e.g., *https://0js3qm31w5.execute-api.us-west-1.amazonaws.com/*
[2]This uses the gRPC protocol, e.g., *producer.default.192.168.1.240.xip.io:80*

- Multiple endpoints can be used simultaneously by the same experiment to speed up the benchmarking. The JSON configuration field *parallelism* defines this number: the higher it is, the more endpoints will be allocated, and the more bursts will be sent in short succession (speeding up the process for large IATs).
- The *latencies* CSV files (Figure 4.2) are the main output of the evaluation framework. They are used in our custom Python plotting utility suite to produce insightful visualizations (as seen throughout Section 5).
- The *logs* text file (Figure 4.2) is the final output of the benchmarking client. Log records are useful for optimizing code and debugging problematic behavior.

We integrate all necessary server-side functionality into a single function that we call a *measurement function*. This approach is similar to that taken in [40] and other serverless performance evaluation frameworks. A measurement function can perform up to three tasks, depending on the use case:

1. It always collects function instance runtime information[3].
2. If applicable, the function will simulate work by incrementing a variable in a busy-spin loop. This can be as simple as "*for i := 0; i < incrementLimit; i++{}*".
3. If applicable, the function records invocation timing (Figure 4.3). This is particularly useful for our data transfer studies where we complement client-measured round-trip time with internal function timestamps for validation purposes [34].

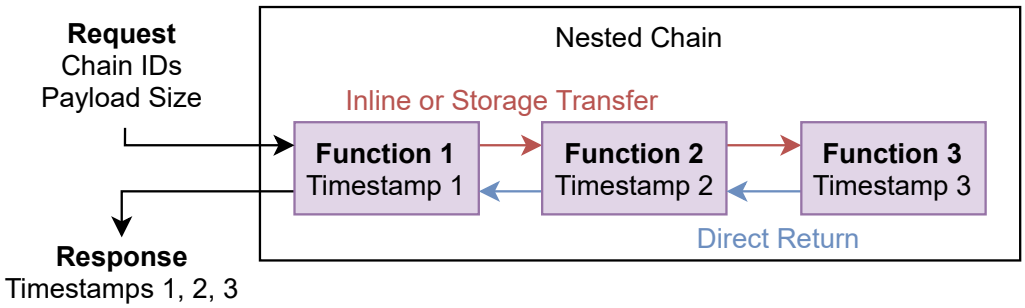

Figure 4.3: *Transfer measurement for a chain of serverless functions*

AWS Lambda supports two separate deployment packages for measurement functions: zip files or container images. Both come with their advantages and disadvantages: zip file packaging is the older and more mature method, while container image packaging is the newer option (December 2020) [35]. The latter increases function image size capacity from 250MB to a maximum of 10GB but still has some downsides. For example, the online console in AWS does not report the image size for container-packaged functions, making it impossible for the tool's automated deployment mechanism to work in experiments that vary the image size. For this reason, we still use zip packaging in those scenarios and limit our image fetch delay experiments to sizes of up to 230MB (Section 5.5).

[3]For example, an internal request ID (e.g., *0db48eae-fa67-452a-895a-2e04649aaccf*) can later be used to search the provider's logs for debugging purposes.

Finally, we look at the procedural steps adopted by the framework (Figure 4.4):

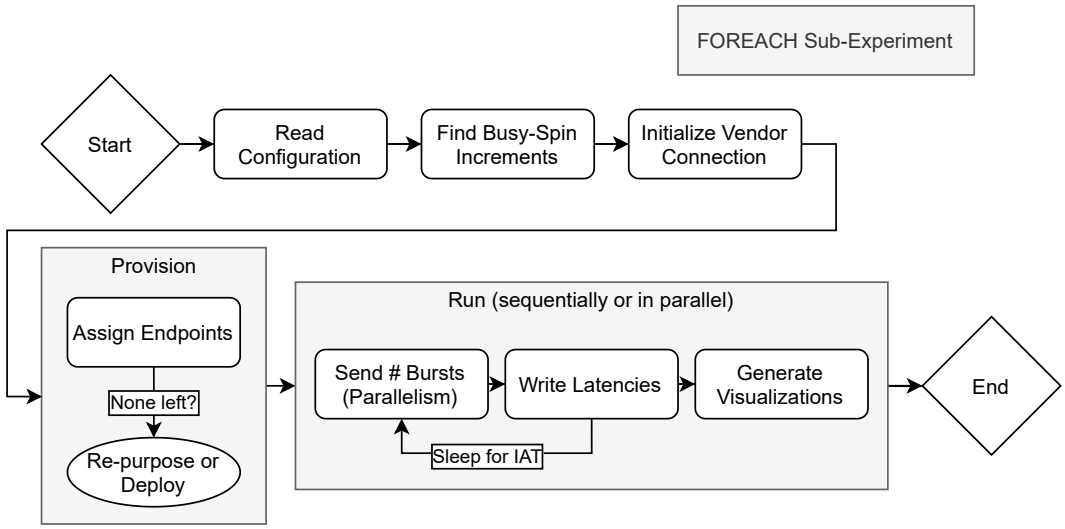

Figure 4.4: *Coordinator flowchart*

1. The JSON configuration file is read and parsed, and any default field values are assigned. If the configuration file is missing, the program throws a fatal error.
2. Experiment service times (e.g., 10 seconds) are translated on the client machine into numbers representing busy-spin increment limits (e.g., 10,000,000). In turn, those are used by the measurement function on the server machine to keep the processor busy-spinning (Section 4.2).
3. A connection with the serverless vendor is established. This is abstracted away behind a common interface having only four functions: *ListAPIs*, *DeployFunction*, *RemoveFunction*, and *UpdateFunction*. Used exclusively throughout the codebase, this interface offers seamless integration functionality with any provider.
4. In the provisioning phase, existing endpoints are first queried either using official provider APIs or from a local file. The corresponding serverless functions are then updated, deployed or removed to match the specified configuration file[4].
5. The last step runs all the experiments either sequentially or in parallel: bursts are successively sent to each available endpoint, followed by a sleep duration specified by the IAT. The process is repeated until all responses have been recorded to disk (Figure 4.2). Finally, statistics and visualizations are generated.

In the next section, we identify desirable characteristics in a robust and performant benchmarking client implementation.

[4] *Parallelism*, *package type*, *function memory* and *function image size* are the configuration fields that are considered when provisioning for an experiment.

4.2 Benchmarking Principles

To better understand the latencies that we measure, we need to define a breakdown of their underlying composition and how the comprising elements interact together. The following equation, tailored to the serverless scenario, forms our initial assumption:

$$measuredLatency = infraConst + slowdown \times serviceTime + queuing + coldStart \tag{4.1}$$

- *infraConst* is a constant that covers the minimum latency between the client and the server in a situation in which the services are unloaded. In AWS Lambda, on average, it can be as low as 47ms (Figure 5.2).
- *slowdown* (Section 5.1) is the factor of CPU sharing/allocation provided by the vendor, and depends on the requested function memory.
- *serviceTime* defines the useful work performed by the measurement function as timed on the client-side. Note that due to CPU slowdown, service time is often amplified on the server-side for low-memory functions (Section 5.1).
- *queuing* (Section 5.4) is a hypothetical delay associated with the necessity for requests to be queued on the server side due to constrained physical resources. Alternatively, this can sometimes be employed for efficiency purposes.
- *coldStart* (Section 2.1.2) is the time required for the serverless infrastructure to initialize a fresh new instance by pulling the function back into main memory.

Consistently monitoring the measured server latency plays a crucial role in serverless systems: it ensures that standard requirements are regularly met, leading to increased user satisfaction and revenue gains.

Making reliable performance measurements, however, takes time. Furthermore, as those measurements tend to be run repeatedly, having a robust infrastructure becomes even more valuable [34] in speeding up the benchmarking process. To this end, and also to improve the maintainability of the codebase, we developed a continuous integration pipeline that regularly verifies the correctness of all framework aspects:

- Syntax (source code is analyzed to flag programming errors, bugs, stylistic errors, and suspicious constructs)
- Successful build of client and measurement function
- Unit tests (external connections, endpoint provisioning logic, utilities)
- Integration tests (3 for AWS - zip packaging, image packaging, and inline data transfers; 1 for vHive - inline data transfers)

Apart from the checks mentioned above, we are interested in validating the data obtained in our experiments. ”Performance measurements often go wrong, reporting surface-level results that are more marketing than science” [34]. To avoid this, we take four steps toward higher-quality performance measurements [34]:

1. **We allow enough time.** Since we measure non-trivial systems, we started (November 2020) at least five months before the submission deadline.
2. **We measure one level deeper**. When quantifying the data transfer speeds in Section 5.5, we break down the latency by using internal function timestamps alongside the overall round-trip time. We then determine how much time is spent in the network and how much time is spent in the actual transfer and check whether they seem consistent with each other [34].
3. **We consider more detail**. Instead of just looking at average values, we graph the entire distributions and analyze the shapes and tail latencies to examine if they provide valuable additional information.
4. **We minimize statistical variability**. To avoid unexpected response times, e.g., caused by network congestion, we place the client and server in close proximity. When benchmarking the AWS servers in northern California, we deploy our tool to the closest CloudLab [18] node in Utah. When benchmarking vHive in our data transfer experiments, we deploy it alongside our tool to the same physical node. The CloudLab machine has a 10-core x86_64 CPU at 2.4GHz and 64GB DDR4 memory in both scenarios. However, as often happens in serverless, AWS Lambda provides no hardware specifications.

In addition to minimizing variability, we take a prudent overall approach when interpreting results to avoid outliers and exceptional deviations. We gather at least 3000 samples for most experiments and make two assumptions [27]: (1) system environment remains identical and stable during the entire experiment and (2) the latency samples are independent and identically distributed.

One challenge we faced in this area was contention at the NIC on the client-side, which deformed the latency distributions and rendered our results untrustworthy at best. To fix this, we changed the program logic to sent bursts in sequence rather than concurrently. As a result, NIC contention was reduced, benchmarking speeds remained quick for long IATs, and the distributions were corrected.

Finally, we avoid issuing sleep-related commands to prevent the de-scheduling of measurement functions on the server-side when simulating work. If sleep-related commands were employed, the system could free resources and perform better than in applications with realistic workloads, to the point where the *service time* parameter is completely ignored. As this is unacceptable, we check for the absence of compiler loop optimizations. Go's scheduler is not preemptive in this scenario [8], which means that busy-spinning will keep the processor busy when simulating work.

Chapter 5

Experimental Analysis

”A good performance evaluation provides a deep understanding of a system’s behavior, quantifying not only the overall behavior but also its internal mechanisms and policies. It explains why a system behaves the way it does, what limits that behavior, and what problems must be addressed to improve the system.” [34]

From a research perspective, our experiments are concerned with three distinct serverless architecture challenges (introduced in Section 2.2): startup time minimization, scalability, and function composition. We investigate four corresponding research directions that cause undesirably high tail latencies by slowing down the serverless computing auto-scaling process: bursty server behavior, function image fetch delay, active requests queuing, and inter-function transfer speeds. In turn, those dimensions are mainly affected by the function image size (source code size on disk), the incoming burst size (number of simultaneously serviced requests), and the transfer payload size (size of the message transmitted between two function instances).

We propose an initial auxiliary experiment that analyzes CPU slowdown under various memory and service time configurations to calibrate expectations regarding service time differences between the client-side and the server-side.

5.1 CPU Slowdown

In the serverless computing paradigm, the user only specifies the amount of function memory, and the cloud provider later allocates CPU resources proportionally to that. For example, AWS allocates almost four times as much computing time for functions when transitioning from 128MB to 480MB of RAM (Figure 5.1).

The allocated memory also determines the number of cores assigned to a function. For applications that can leverage a higher degree of parallelism, this translates to significant performance gains. Nevertheless, a 10GB function is not always 5 times faster than, e.g., a 2GB function (Figure 5.1). Not only is it improbable that the physical machine processing the request has enough CPUs available for use, but most of the commercially written code is not designed to use such a high degree of parallelism [7].

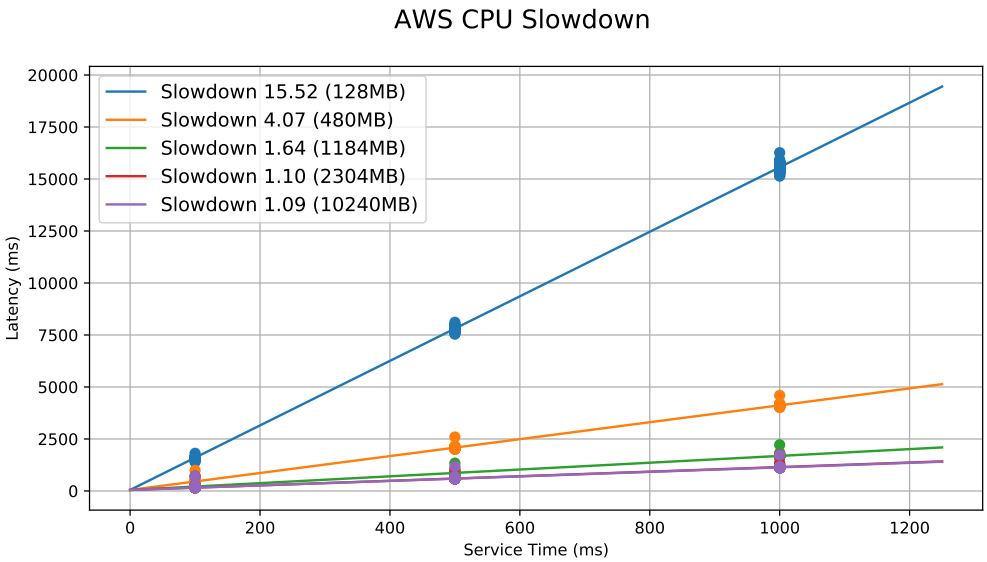

Figure 5.1: *CPU slowdown in AWS Lambda for five different memory configurations*

To define the CPU slowdown in AWS based on function memory, we vary the allocated memory from 128MB to a maximum of 10GB. We then linearly increase service time as measured on the CloudLab machine (*100ms*, *500ms*, *1000ms*), establishing if response latency is predictable and whether it shares a functional relationship with the application service time for different memory configurations.

Figure 5.1 shows that lower memory functions are slowed down as much as 15 times compared to when running natively on a full thread inside the client machine. The functional relation between response latency and service time is linear regardless of allocated memory, and an entire thread seems to be given to the instance only beyond the 2GB threshold. Our results are similar to those from other studies, e.g., *FaasProfiler* reports up to 20x slowdown in Apache OpenWhisk (Figure 3.1) [36].

Let us now consider CPU utilization as the inverse of CPU slowdown: the less slowdown a function experiences, the more CPU it must be using (in other words, the bigger its processor time-share). Compared to an identical study from *Peeking behind the curtains of serverless platforms* [40], we notice that CPU utilization has decreased in AWS since 2018 for functions in the lower memory range: a full CPU thread is now given at around 2GB instead of 1.5GB (Table 5.1).

Serverless systems that are less sensitive to allocated memory from a latency viewpoint can be more attractive to customers: the code will run quicker for the same memory and price ranges. Furthermore, should the developer ever need to allocate less memory

Memory	128MB	480MB	832MB	1184MB	1536MB	2304MB
CPU (2018) [40]	7%	28%	50%	71%	92%	N/A
CPU (2021)	6%	24%	42%	60%	77%	92%

Table 5.1: Change in AWS Lambda CPU utilization rates from 2018 to 2021

to a function (e.g., after algorithm optimization), they would benefit the most from contracting providers that offer the best slowdown-to-memory ratio.

5.2 Bursty Behavior

”Elastic, automatic scaling in response to changes in demand is the main advertised benefit of the serverless model” [40].

To understand how bursty workloads affect scalability and thus application performance, we devised experiments that place the serverless system under stress. We start with an analysis of single requests (burst size *1*), after which we increase the request burst sizes to *100*, *300*, and *500*. We also explore whether idle resource reclamation plays a role in shaping the latency distributions; in other words, whether cold or warm function instances behave differently when exposed to the same traffic pattern.

All experiments in this sub-chapter were performed on AWS Lambda using measurement functions of the same memory (2048MB) and programming language (Golang) configuration.

5.2.1 Single Requests

We created 150 measurement functions and invoked each with a single request at a time. All measurement functions returned immediately as the service time was set to 0 seconds. We paused for 10 minutes between invocations (cold starts) and repeated the process. After 20 rounds of measurements for each function (3000 samples in total), we concluded the experiment. We then repeated the entire procedure with an inter-arrival time of 3 seconds to simulate warm starts.

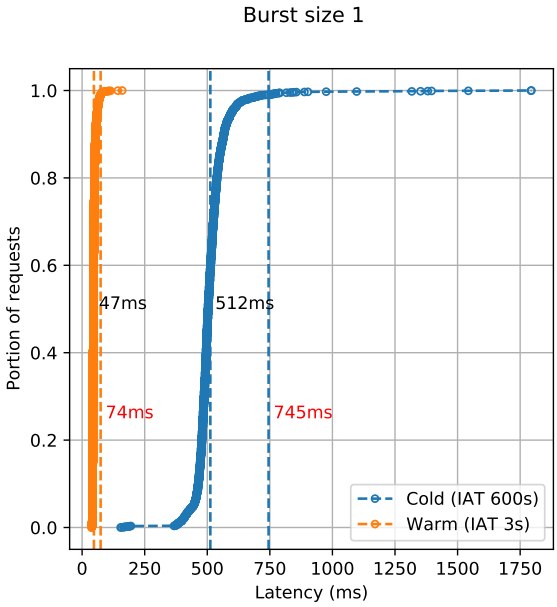

Figure 5.2: *Average and tail latencies in warm and cold single requests*

Although not surprising, the results we obtained (Figure 5.2) experimentally validate some of the background knowledge presented in Section 2:

- AWS has two types of latencies for a minimal Golang measurement function: warm (approx. 50ms) and cold (approx. 500ms).
- Cold function instances have a significantly longer tail (spanning 1050ms, from 745ms to 1795ms) than warm requests (spanning 86ms, from 74ms to 160ms).
- Cold function instances have a much larger standard deviation (75ms) than warm requests (7ms).

As expected, responses are significantly faster when function instances are warm (up to 10 times quicker). Furthermore, cold instance responses are more unpredictable: they have a longer and sparser tail and a standard deviation that is more than ten times larger. One potential explanation for this phenomenon could be system component variability (Section 2.1.2): as more individual components are involved in a cold start (function code has to be downloaded, containerized, booted, and primed), the entire system behavior might inherit their compounded unpredictability.

5.2.2 Bursty Requests

We kept the conditions identical to the above experiments but only used a single measurement function that we invoked with 500 concurrent requests at a time. After six rounds of measurements and 3000 samples gathered for each of the two inter-arrival times (cold - 10 minutes; warm - 3 seconds), we concluded our experiments.

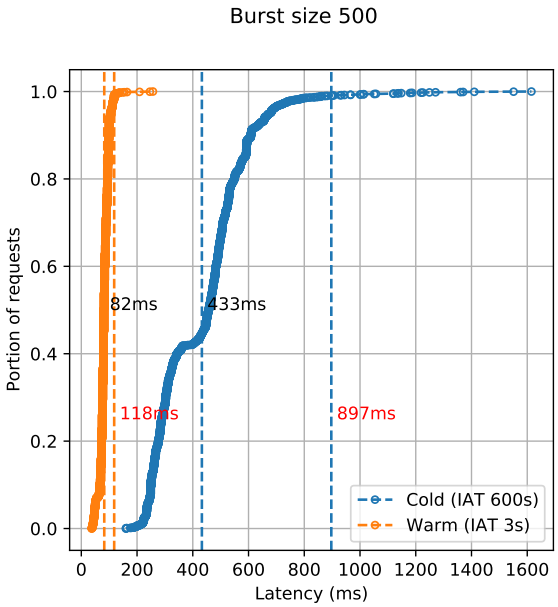

Figure 5.3: *Average and tail latencies in warm and cold bursty requests*

The results we obtained (Figure 5.3) offer some insights into how well the serverless system behaves under stress conditions and whether idle resources reclamation plays a role in shaping the tail latency:

- Compared to the previous experiments, warm instances are 43% slower on average (82ms vs. 47ms), and cold instances are 16% faster on average (433ms vs. 513ms).
- Warm instances exhibit a longer tail than before (62% longer), while cold instances a shorter one (32% shorter). Both tails are more evenly distributed, whereas before, they were denser towards the lower end.
- Both cold (159ms vs. 74ms) and warm (16ms vs. 7ms) instances have more than doubled their standard deviation compared to the single-request experiments.

We conclude that bursty requests have an overall degrading effect on system performance: unpredictability increases (standard deviations are significantly higher), and tail latency is more evenly distributed (indicating an increased number of exceptionally high latencies).

Nevertheless, we still notice a positive impact on the average and tail latencies of cold function instances. One interpretation of this behavior could be related to the fact that the system is now aware of the extensive amount of provisioning it has to do. As the requests arrive within such a short time interval, the downloading and priming of resources can be batched. This contrasts with the single-request case in which the system had to perform those operations from the beginning for every single request.

Finally, we run two more experiments with intermediary burst sizes (*100* and *300* simultaneous requests) to discover any missed study opportunities and to confirm the general trend (Figure 5.4). By measuring from four different angles, we provide a deeper and more accurate understanding of the system under review [34].

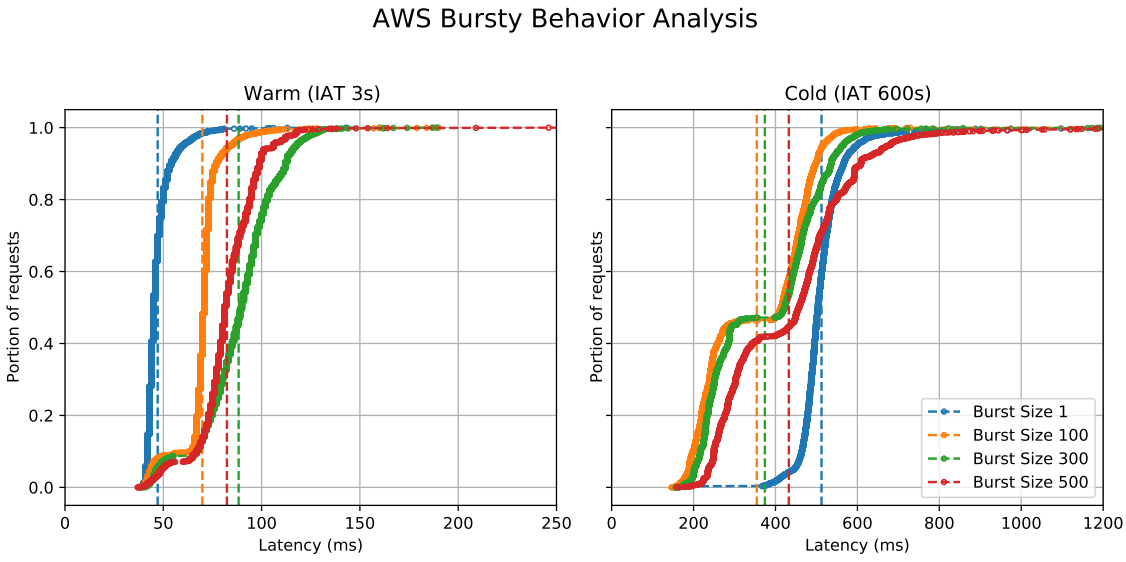

Figure 5.4: *Latency distributions with averages across various burst sizes*

As burst size increases, we notice how warm function instances become slower on average (distributions shift to the right), and their response times are more unpredictable than before (increased standard deviation as observed in the flattening of the distributions).

Once again, we find that the opposite effect occurs for cold function instances: single requests (burst size 1) are by far the slowest. As the burst size increases, distributions are shifted to the left, towards lower latencies. However, beyond the threshold of at least 100 requests per burst, performance starts to degrade again: there seems to be a limit to the performance optimization gains obtainable through batching.

5.3 Image Fetch Delay

The cold start impact on FaaS performance is still a key obstacle of serverless adoption today, especially for latency-sensitive workloads [37]. When a cold start occurs, the function code must be downloaded, containerized, booted, and primed to be run. In this sub-chapter, we explore whether the function source code size influences cold start performance.

To experimentally deduce this, we deploy 615 distinct functions of the same memory (2048MB) and language (Golang) configuration and equally split them across 5 different image sizes: *9MB*, *60MB*, *120MB*, *180MB*, *230MB*. We achieve this by automating the deployment process: a new file is generated, filled with arbitrary bytes of the specified size, and bundled together with the binary in the deployment package for each group of 123 functions (one group for each image size).

We also once again vary the request burst sizes (*1*, *100*, *300*, *500*). In doing so, we seek to find whether larger bursts impact the image fetch delay in any meaningful way. We always invoke each function more than 10 minutes apart (IAT) to only collect cold start latencies.

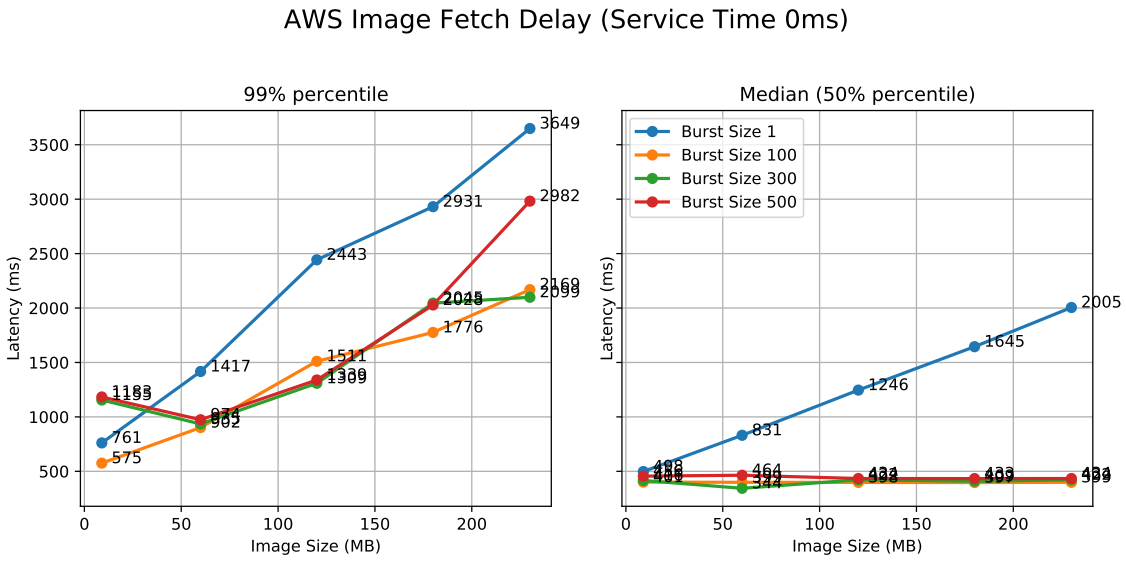

Figure 5.5: *Tail and median latency across various burst and image sizes*

Some of the results we obtained (Figure 5.5) are not surprising given the insights from Section 5.2. Others, however, experimentally validate our intuitive thoughts (Section 2.1.2) on the relation between cold starts and function image size:

- As image size increases, cold starts become slower (image pull delay). For single requests, as we vary the image size from 9MB to 230MB, tail latency (99th percentile) suffers the most with an increase of almost 380% (from 761ms to 3649ms) compared to 305% (from 498ms to 2005ms) for the median latency.
- For single requests, median latencies linearly increase by approx. 400ms for every extra 60MB in image size. We can then approximate the image fetch bandwidth to around 150 MB/s for functions with 2048MB allocated memory.
- Larger bursts (*100*, *300*, *500* requests) maintain the median latencies constant at around 400ms regardless of the function image size. They are also constantly faster than single requests, regardless of percentile (50th or 99th). Those two observations reinforce the takeaways from Section 5.2.

We conclude that image size is a key source of tail latency. This is in accordance with the results of *ServerlessBench*: "Functions with larger code sizes suffer from longer startup latency due to larger data transmission and package import overhead. Serverless application developers should optimize the function codes to import the minimal needed packages and pack only necessary dependencies." [42]

Finally, we look at the same experiments in more detail. Instead of just plotting the two percentiles (50th and 99th), we graph the entire distributions for burst sizes *1* and *100* and analyze the obtained shapes and tail latencies. We seek to find whether they provide any valuable additional information and gain a deeper and more accurate understanding of the system under review [34].

Figure 5.6 shows the CDFs for single requests (burst size *1*) and for bursty requests (burst size *100*) corresponding to the same data as in Figure 5.5. In both scenarios, we notice a predictable pattern in the latency distributions as the image size increases.

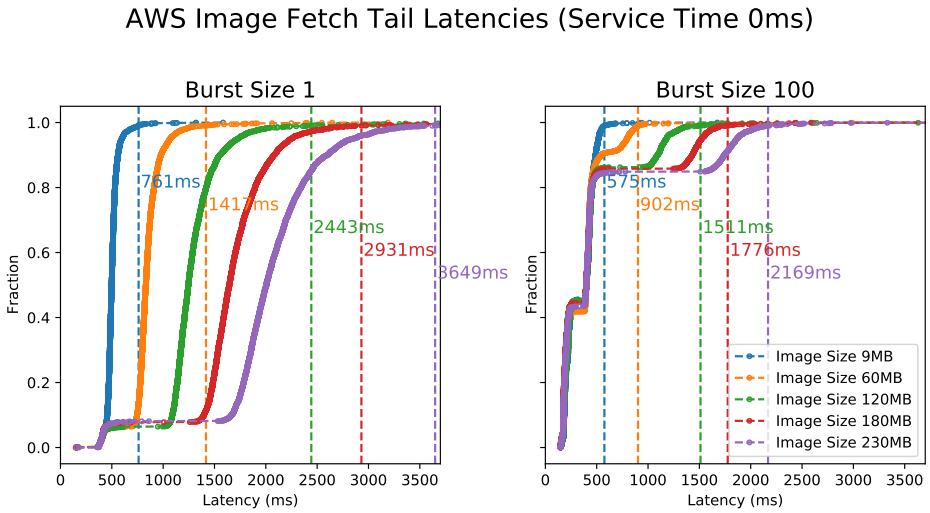

Figure 5.6: *Latency distributions with tail across various burst and image sizes*

For single requests, larger image sizes lead to a rise in response time unpredictability (higher standard deviations and flatter distributions) and to increasingly higher tail latencies (from 761ms to 3469ms). Larger bursts, however, exhibit different behavior.

Tail latency still increases monotonically with image size, but the first half of the distributions remains unchanged. This agrees with the data presented in Figure 5.5: median latencies are unaltered regardless of the function image size.

This analysis seems to reveal why single requests are regularly slower than bursty requests in the context of cold starts. As hypothesized in section 5.2, the downloading and priming of resources is batched: more than half of the requests benefit from resources freshly-fetched for the remaining requests.

In contrast, in the single-request scenario, the system cannot batch and re-use resources and must repeat the same operations from the beginning for each new request. This can also be observed in the bursty scenario by looking at the second half of the distributions: the same flattening effect occurs as when the burst size is 1 (Figure 5.6).

5.4 Request Queuing

The design of the supporting components (e.g., load balancer, message queue) in serverless platforms can influence the overall scalability performance of the entire infrastructure [42].

For example, 500 concurrent requests might be processed faster and with fewer resources by spinning up 250 instances and delivering two requests to each, rather than cold-starting 500 instances. This can happen when the system is aware of the typical instance runtime (e.g., 50ms) and when this runtime is lower than a cold start (e.g., 450ms). Nevertheless, request queuing can add a layer of complexity to the system that might not be worth considering given the usual workload diversity and the long-term technical maintenance costs associated with these commercial platforms.

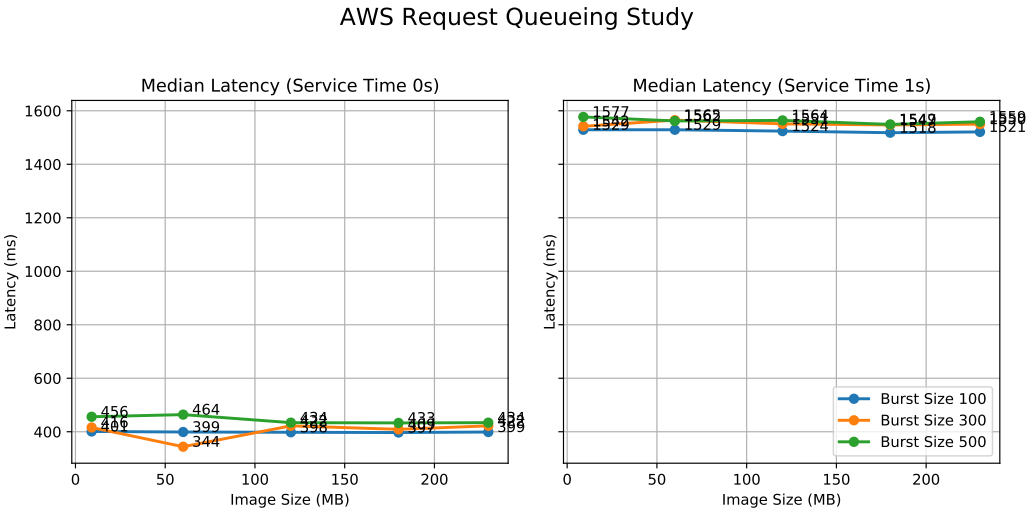

Figure 5.7: *Median latency across two different function service times*

In this study, we seek to find whether AWS queues incoming requests for efficiency purposes and, if so, to discover the number of requests that triggers this scenario. For this purpose, we run the same experiments as in Section 5.3 but with a service time

of 1 second. The reason behind this is to test whether any extra delays are incurred as the much-slower instances (1 second service time), when part of a larger burst (100+ requests), wait for each other to terminate.

We plot the median latencies for both service times in Figure 5.7 (each of the 30 data points is a median over 3000 samples). We then interpret the results:

- The latency shift remains unchanged at 1.1 seconds (from an average of 400ms to an average of 1500ms), regardless of the request burst size or the function image size.

Let us recall from the CPU studies in Section 5.1 that slowdown is approximately 1.1 for a function with around 2048MB allocated memory (Figure 5.1). If the system employed an assertive queuing policy for bursts of less than 500 concurrent requests, we would notice considerably higher latencies (because the service time is 1 second).

We conclude that AWS allocates resources aggressively to avoid queuing and that cold invocations do not share function instances. However, this observation can be inaccurate in the following scenarios:

1. Request forwarding is optimized to record the typical service times for instances and consider them in its decision-making process. In this case, a cold start is faster than waiting, and we would not detect queuing even if it was present.
2. Only a small percentage of requests are queued, in which case the median latency does not capture the phenomenon.

5.5 Inter-function Transfer Speeds

Serverless computations have to be efficiently chained together to accommodate for more complex business logic. This communication chain takes the form of inter-function data transfers (inline or via storage) and has to be efficient to prevent bottlenecks. Despite this, data transmission in real-world serverless systems uses storage, which is radically slower and more expensive than point-to-point networking [24].

To understand how inter-function transfer size variability affects communication performance in AWS and vHive, we devised experiments that vary the transmission payloads between 1KB and 4MB in inline transfers and between 60MB and 1GB in storage transfers.

For validation purposes, we measure one level deeper by breaking down the latency using internal function timestamps alongside the overall round-trip time (as specified in Section 4.2). We then determine how much time is spent in the network and how much time is spent in the actual transfer and check whether they seem consistent with each other [34].

5.5.1 Inline Transfers

We created 35 measurement functions in AWS with 128MB memory and equally split them across 7 different transfer payload groups of 5 functions each (*1KB*, *10KB*,

100KB, *1MB*, *2MB*, *3MB*, *4MB*). We then invoked each function with a single request at a time and a minimal IAT to keep the instances warm. All measurement functions returned immediately after the transfer as the service time was set to 0 seconds. After 600 rounds of measurements for each function (3000 samples per group), we repeated the process with allocated memories of 1.5GB and 10GB.

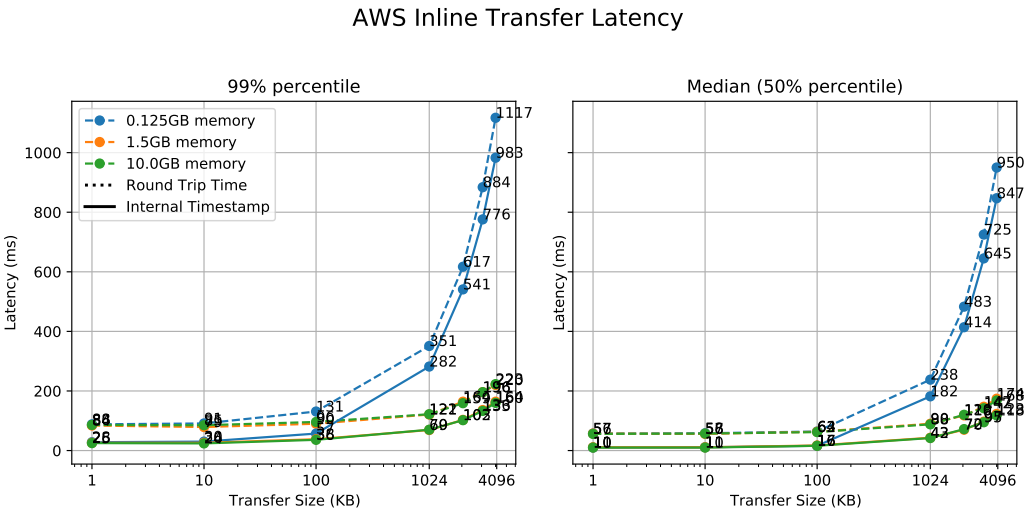

Figure 5.8: *Inline transfer latencies in AWS Lambda across seven payload sizes*

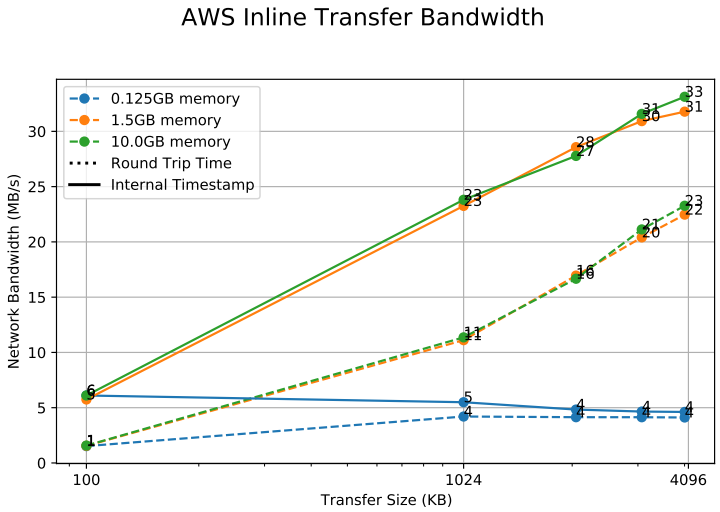

Figure 5.9: *Inline transfer bandwidth in AWS Lambda across five payload sizes*

The results we obtained (Figure 5.8) offer some insights into the limitations of low-memory serverless functions and their behavior as the payload size increases:

- Low-memory (128MB) function latencies are identical to those of higher-memory functions for smaller payloads. However, beyond the 1MB threshold, the gap widens considerably: tail latency is 302% higher in low-memory functions for 1MB transfers and 502% higher for 4MB transfers. Similarly, median latency is 333% higher for 1MB transfers and 561% higher for 4MB transfers.

- There seem to be no dissimilarities in communication performance between 1.5GB functions and 10GB functions: they share almost the same tail and median latencies regardless of transfer payload size.

We conclude that AWS limits the transfer bandwidth based on the amount of allocated function memory. Indeed, after plotting the bandwidth for payload sizes of over 100KB (Figure 5.9) based on the median latencies in Figure 5.8, we notice how the 128MB instances are limited to around 4MB/s, while 1.5GB+ instances are seemingly less restricted and reach transfer speeds up to eight times faster (30+MB/s) depending on the payload size.

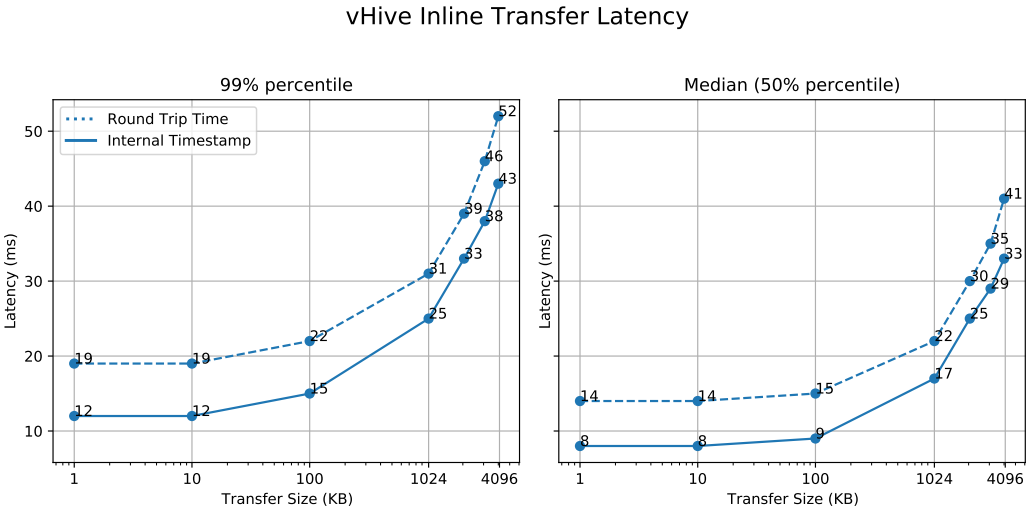

Figure 5.10: *Inline transfer latencies in vHive across seven payload sizes*

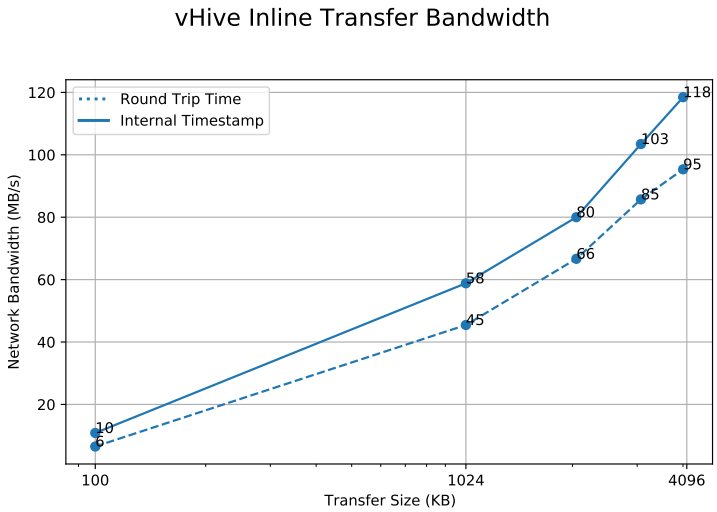

Figure 5.11: *Inline transfer bandwidth in vHive across five payload sizes*

We then performed the same experiments in vHive (Figures 5.10, 5.11), where the allocated memory for function instances was set to 256MB. We reveal some surprising insights when comparing the two platforms:

- The transfer bandwidth in vHive is consistently above that of AWS, even for function instances with maximal memory: speeds are 66% higher for 100KB transfers, 152% higher for 1MB transfers, and 257% higher for 4MB transfers.

One of the reasons behind vHive's superior function communication performance might be explained by the fact that both the benchmarking framework and the platform itself were deployed on the same CloudLab node [18], eliminating the infrastructure constant in Equation 4.1.

We next look at inter-function transfers with significantly larger payloads that exceed the maximum inline transfer limit and can only be executed through storage systems.

5.5.2 Storage Transfers

We created 89 measurement functions in AWS with 10GB memory and split them across 7 different transfer payload groups (*60MB*, *125MB*, *250MB*, *375MB*, *500MB*, *750MB*, *1GB*). We then invoked each function with a single request at a time and a minimal IAT to keep the instances warm. All measurement functions returned immediately after the transfer as the service time was set to 0 seconds. After 1000 samples gathered per group, we concluded the experiments.

The results we obtained (Figures 5.12, 5.13) offer some insights into the rate-limiting behavior of serverless platforms:

- The median transfer bandwidth is non-monotonic. It steadily increases as the payloads grow and peaks at 137MB/s for payloads of 250MB. Following that, for larger payload sizes of up to 1GB, bandwidth sharply drops by 41%.

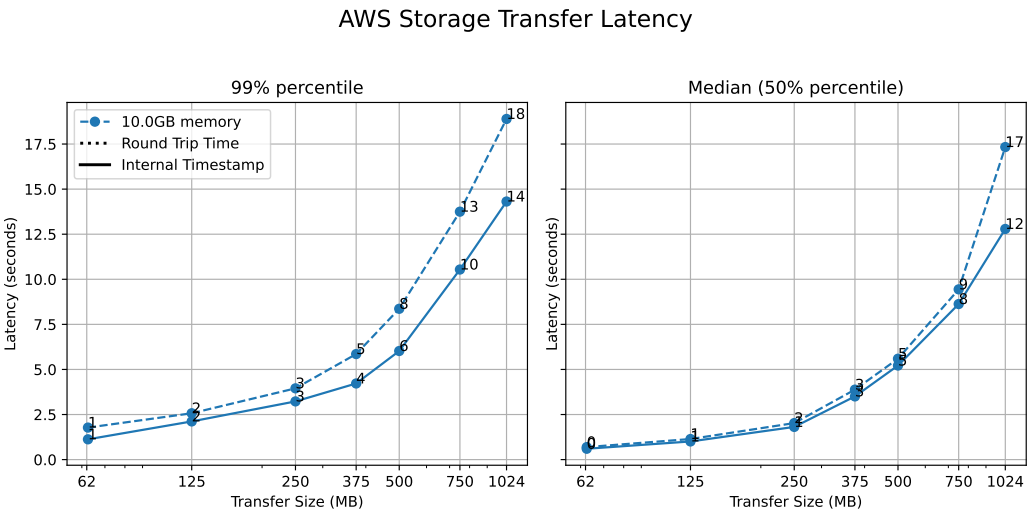

Figure 5.12: *S3 transfer latencies in AWS Lambda across seven payload sizes*

In its architecture, AWS Lambda uses Firecracker [3], a virtualization technology that leverages KVM to launch lightweight micro-virtual machines (microVMs) in non-virtualized environments in a fraction of a second. Firecracker implements bandwidth throttling, the effects of which we observe in Figure 5.13. Based on its open-source

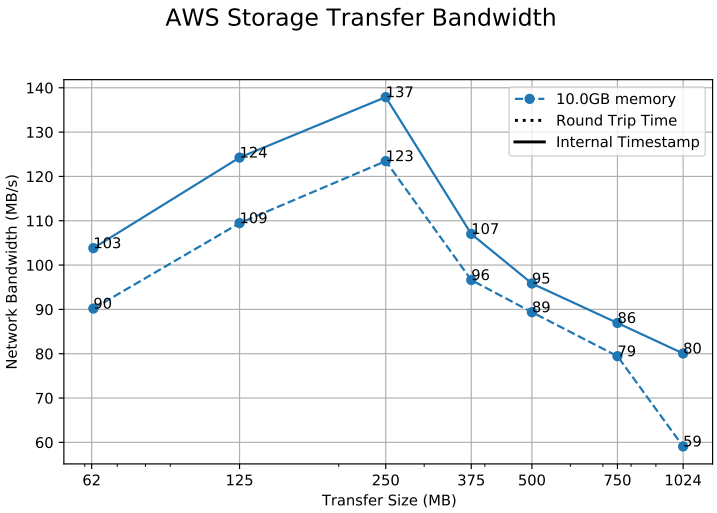

Figure 5.13: *S3 transfer bandwidth in AWS Lambda across seven payload sizes*

code [22], we find that the implementation involves replenishable buckets with support for bytes/second mode of operation.

Furthermore, there are two I/O budgets: ”Initial-burst” and ”Steady”. The former is an initial extra credit that does not replenish and can be used for an opening burst of data. When transfer sizes surpass the 250MB threshold, the two budgets are quickly consumed, and performance suffers as the ”Steady” bucket has to recharge constantly.

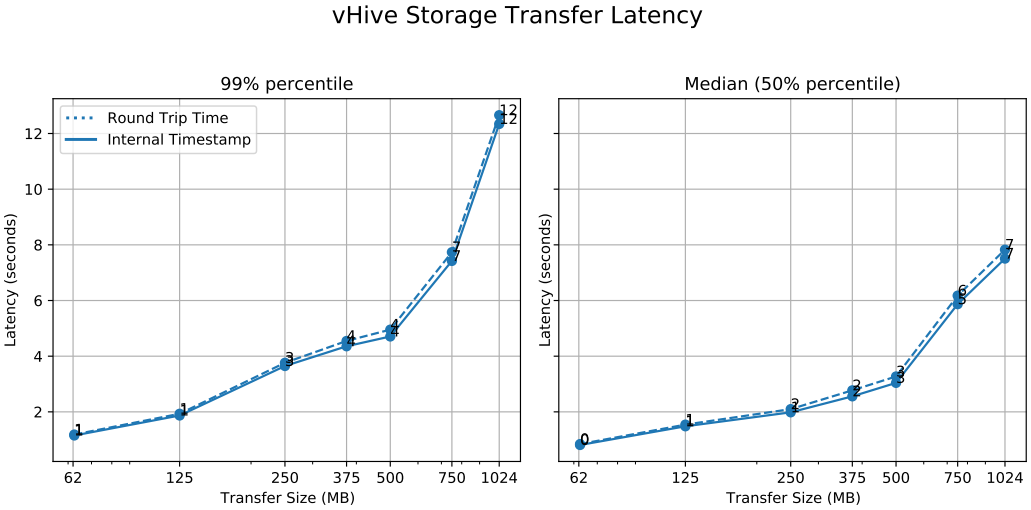

Figure 5.14: *Minio [32] transfer latencies in vHive across seven payload sizes*

We then performed the same experiments in vHive (Figures 5.14, 5.15), where the allocated memory for function instances was set to 6192MB. Once again, we reveal surprising insights when comparing the two platforms:

- The transfer bandwidth in vHive is above that of AWS Lambda for larger payload sizes: speeds are 6% higher for 375MB transfers, 35% higher for 500MB transfers, and 41% higher for 1GB transfers.

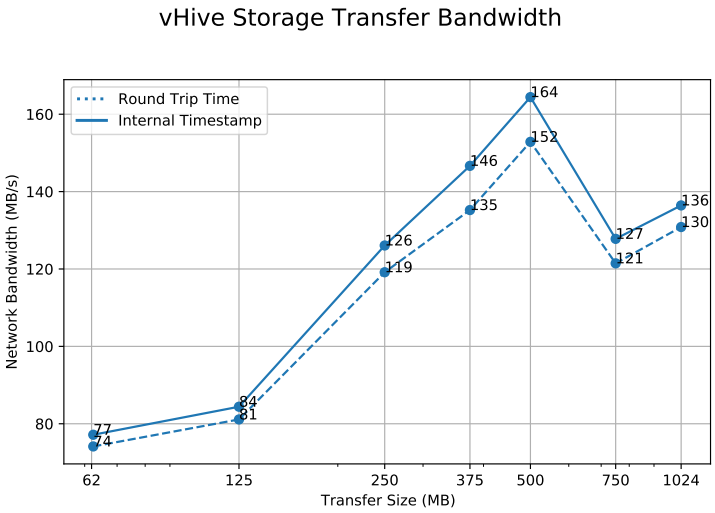

Figure 5.15: *Minio [32] transfer bandwidth in vHive across seven payload sizes*

- However, AWS Lambda exhibits superior communication performance for smaller payloads: bandwidth is 25% higher for 60MB transfers and 32% higher for 125MB transfers.

In its architecture, vHive also uses Firecracker [3] to take advantage of the same security and workload isolation [9] benefits featured in AWS Lambda. This could explain why we observe similar bandwidth throttling effects as in Figure 5.13, except now, this occurs after the 500MB threshold rather than the 250MB threshold.

5.6 Discussion

Results so far suggest that the function image size, the transfer payload size, and the request burst size can all significantly influence tail, average and median latencies in serverless systems and, consequently, their overall performance.

We find that server response times for cold function instances are, on average, ten times slower and more unpredictable than for warm function instances. When requests arrive in bursts of over 100, we show that while warm instances incur a penalty, AWS optimizes cold instances by fetching resources in batches. We achieve this by incorporating tail latency into our studies, a metric that is often overlooked in the literature. Our analysis further reveals that AWS allocates resources aggressively to avoid queueing and that invocations do not share function instances even when receiving up to 500 concurrent requests. We also discover that inter-function transfer latencies are much higher in AWS than in vHive and that AWS caps network bandwidth for low-memory configurations and throttles it for larger payloads regardless of allocated memory.

Based on these insights, we stress the importance of regular platform performance monitoring for improved customer satisfaction. Finally, to boost the economic efficiency from a user's perspective, we suggest that serverless applications are designed with a trade-off in mind between the function image size and the transfer chain length.

Chapter 6

Conclusion

Serverless computing has seen rapid adoption because of its instant scalability, flexible billing model, and economies of scale. In serverless, developers structure their applications as a collection of functions, sporadically invoked by various events like clicks. The high variability in function image sizes, invocation inter-arrival times, and request burst sizes motivates vendors to scrutinize their infrastructure to ensure a seamless user experience across all of their services. To monitor serverless platform performance, identify pitfalls, and compare different providers, the public attention has turned to benchmarking, whereby measurement functions are deployed to the cloud to gather insights regarding response latencies, transfer speeds, and vendor policies.

Many extensive attempts have already been made to characterize serverless systems (Section 3). However, experiments conducted were generally either not broad enough, not configurable enough, no longer maintained, or had a different focus: statistical soundness [27], user billing and expenses [23, 14]. Moreover, tail latency plays a critical role in online services' responsiveness yet is missing from many benchmarking endeavors in the literature (Table 3.1). For this reason, we systematically scrutinize tail latency in our studies, incorporating it into almost every analysis performed.

6.1 Achievements

This work introduces our open-source framework for serverless performance evaluation, intending to enable researchers and developers to benchmark multiple cloud platforms. Using this framework, we conduct one of the biggest serverless measurements to date, launching over 320,000 function instances to characterize system performance in AWS Lambda and vHive - the Edinburgh Architecture and Systems (EASE) virtual machine orchestrator system.

Throughout our research, we investigated four dimensions that affect serverless system performance and which have been understudied or entirely overlooked in the community: image fetch delay, bursty behavior, request queuing, and inter-function transfer speeds. The last experiments (Section 5.5) were carried out across both serverless platforms, leading to the qualitative and quantitative comparison between them.

Our research reveals that server response times for cold function instances are, on average, ten times slower and more unpredictable than for warm function instances. When requests arrive in bursts of over 100, we show that while warm instances incur a penalty, AWS optimizes cold instances by fetching resources in batches. We achieve this by incorporating tail latency into our studies, a metric that is often overlooked in the literature. Our analysis further reveals that AWS allocates resources aggressively to avoid queueing and that invocations do not share function instances even when receiving up to 500 concurrent requests. We also discover that inter-function transfer latencies are much higher in AWS than in vHive and that AWS caps network bandwidth for low-memory configurations and throttles it for larger payloads regardless of allocated memory.

We conclude that regularly monitoring serverless platforms' performance is key to maintaining an exceptional client experience while keeping serverless computing up to standards in terms of speed and resiliency. To boost performance and economic efficiency from a user's perspective, we suggest that serverless applications are designed with a trade-off in mind between the function image size and the transfer chain length.

6.2 Future Work

We envision extending our framework along the following directions:

- Supporting more serverless providers, such as Microsoft Azure Functions and Google Cloud Functions, would allow for more comparisons to be constructed along the four existing research dimensions at minimal extra operational cost.
- Including more measurement function runtimes would provide more relevant insights for users and vendors alike. For instance, JavaScript and Python are the most used programming languages for cloud functions, each used by 32% of the cases as found in a SPEC RG survey [19].
- Incorporating realistic workloads (e.g., HTML rendering, machine learning inference, video processing) would enable developers to gain a more holistic understanding of their specialized application's behavior in the cloud.
- Extending the continuous integration pipeline with additional tests would increase the robustness of the framework. For instance, vHive inline transfers are already automated, but the vHive storage transfer tests require additional configuration on the GitHub runner when setting up *Minio* [32].
- Performing additional experiments on vHive would foster a deeper understanding of the open-source platform, leading to an increased awareness in the serverless community. For example, similar to the AWS Lambda studies, its bursty behavior or image fetch delays could be assessed and quantified.

Bibliography

[1] Why use serverless computing? Accessed 21 February 2020.

[2] giltene/wrk2, March 2012.

[3] firecracker-microvm/firecracker, October 2018.

[4] Build and run applications without thinking about servers, 2019.

[5] Gojko Adzic and Robert Chatley. Serverless computing: Economic and architectural impact. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 884–889, New York, NY, USA, 2017. Association for Computing Machinery.

[6] Sachin Agarwal. Public cloud object-store performance is very unequal across aws s3, google cloud storage, and azure blob storage, June 2018.

[7] Mustafa Akin. How does proportional cpu allocation work with aws lambda?, January 2018.

[8] Go Authors. Source file src/runtime/preempt.go, 2019.

[9] Jeff Barr. Blogs, November 2018.

[10] L.A. Barroso, U. Hölzle, and P. Ranganathan. *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2018.

[11] Eric A. Brewer. Kubernetes and the path to cloud native. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, page 167, New York, NY, USA, 2015. Association for Computing Machinery.

[12] Stuart Card, George Robertson, and Jock Mackinlay. The information visualizer, an information workspace. pages 181–186, 01 1991.

[13] Sanjay Chaudhary, Gaurav Somani, and Rajkumar Buyya. *Research advances in cloud computing*. Springer, 2017.

[14] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. Sebs: A serverless benchmark suite for function-as-a-service computing, 2020.

[15] Yan Cui. How does language, memory and package size affect cold starts of aws lambda?, June 2017.

[16] Yan Cui. How long does aws lambda keep your idle functions around before a cold start?, June 2017.

[17] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.

[18] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[19] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. A review of serverless use cases and their characteristics, 2021.

[20] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. Serverless applications: Why, when, and how? *IEEE Software*, 38(1):32–39, January 2021.

[21] Sylvia Engdahl. Blogs, December 2020.

[22] Firecracker-Microvm. firecracker-microvm/firecracker, 2018.

[23] Martin Grambow, Tobias Pfandzelter, Luk Burchard, Carsten Schubert, Max Zhao, and David Bermbach. Befaas: An application-centric benchmarking framework for faas platforms, 2021.

[24] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back, 2018.

[25] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley, February 2019.

[26] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. pages 502–504, 07 2019.

[27] Marios Kogias, Stephen Mallon, and Edouard Bugnion. Lancet: A self-correcting latency measuring tool. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 881–896, Renton, WA, July 2019. USENIX Association.

[28] Philipp Leitner, Erik Wittern, Josef Spillner, and Waldemar Hummer. A mixed-method empirical study of function-as-a-service software development in industrial practice. *Journal of Systems and Software*, 149:340–359, 2019.

[29] Pascal Maissen, Pascal Felber, Peter Kropf, and Valerio Schiavoni. Faasdom.

Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems, July 2020.

[30] Nathan Malishev. Aws lambda cold start language comparisons, 2019 edition, September 2019.

[31] Shah Meena. *What is Serverless Computing?* View Labs, January 2020.

[32] MinIO. High performance, kubernetes native object storage.

[33] Chris Munns. Aws re:invent 2017: Getting started with serverless architectures (cmp211), November 2017.

[34] John Ousterhout. Always measure one level deeper. *Commun. ACM*, 61(7):74–83, June 2018.

[35] Danilo Poccia. New for aws lambda – container image support, December 2020.

[36] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. Architectural implications of function-as-a-service computing. 10 2019.

[37] Erwin van Eyk, Alexandru Iosup, Johannes Grohmann, Simon Eismann, André Bauer, Laurens Versluis, Lucian Toader, Norbert Schmitt, Nikolas Herbst, and Cristina L. Abad. The spec-rg reference architecture for faas: From microservices and containers to serverless platforms. 23(6):7–18, November 2019.

[38] Henry He Virtual Instruments. Storage: How 'tail latency' impacts customer-facing applications, August 2019.

[39] Timothy A. Wagner. Debunking serverless myths. page 30, 07 2018.

[40] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, July 2018. USENIX Association.

[41] Frederik Willaert. Aws lambda container lifetime and config refresh, April 2016.

[42] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 30–44, New York, NY, USA, 2020. Association for Computing Machinery.

Appendix A

Tool Configuration

```
{
  "Sequential": false,
  "Provider": "aws",
  "SubExperiments": [
    {
      "Title": "transfer",
      "Bursts": 3000,
      "BurstSizes": [1],
      "PayloadLengthBytes": 0,
      "IATSeconds": 600,
      "IATType": "stochastic",
      "PackageType": "Zip",
      "Visualization": "all",
      "DesiredServiceTimes": ["0ms"],
      "Parallelism": 5,
      "FunctionMemoryMB": 128,
      "FunctionImageSizeMB": 50.8,
      "DataTransferChainLength": 2,
      "StorageTransfer": true
    },
    {
      "Title": "burstiness",
      "Bursts": 30,
      "BurstSizes": [100],
      "DesiredServiceTimes": ["0ms"]
    }
  ]
}
```

This JSON configuration example could be used as input for our benchmarking tool.

It specifies that the sub-experiments should run in parallel (Sequential is *false*) and that the provider used is AWS. There are two sub-experiments included: the first one

specifies all possible configurable fields, while the second one uses all possible default values. Both sub-experiments collect 3000 samples, but in different ways: the first one sends one request at a time to 5 endpoints in parallel and gathers, using storage, data transfer latencies from a chain of 2 functions. The second one uses only one endpoint (by default), models warm starts (default IAT is *0s*), and, most probably, tests burstiness (as the burst size is *100*).

For a full explanation of each parameter used, the project wiki page is available at:

- *https://github.com/ease-lab/vhive-bench/wiki/Customize-Experiments*