A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

By

Madhavi Jaival

December 2022
The thesis of Madhavi Jaival is approved:

___________________________________  ________________________
Adam Kaplan, PhD                      Date

___________________________________  ________________________
Mahdi Ebrahimi, PhD                   Date

___________________________________  ________________________
Katya Mkrtchyan, PhD, Chair           Date

California State University, Northridge
Acknowledgments

I would like to express my gratitude and appreciation to Dr. Adam Kaplan and Dr. Katya Mkrtchyan for their continuous support and guidance for my thesis and for their knowledge in the field that they patiently shared with me throughout and helped me in strengthening my background for working on the thesis. This work would not have been possible without their encouragement and imaginative ideas. In addition, I would like to give a word of thanks to my committee member, Dr. Mahdi Ebrahimi, for being so generous with their time and support during this thesis.

I also appreciate the CSUN IT team's help with AWS account creation and their willingness to assist me in resolving any difficulties relating to AWS services.

Finally, a huge thanks to my parents for all the times they have assisted me over my academic career. I was able to get through the challenges, pain, and difficulty when things became tough because of their support. I am grateful to my spouse, Mr. Sanjay Jaival, for his unwavering support and constant encouragement during the research and writing of this thesis. Without his support, I wouldn't be able to hustle well.
# Table of Contents

Copyright .......................................................................................................................... ii
Sigature Page ................................................................................................................ ii
Acknowledgments .......................................................................................................... iv
Table of Contents ........................................................................................................ v
List of Figures ................................................................................................................ vii
List of Tables ................................................................................................................ ix
Abstract ........................................................................................................................ x

1. Introduction ................................................................................................................... 1
2. Background ................................................................................................................... 4
3. Related Works ............................................................................................................. 7
4. Methodology ............................................................................................................... 10
   “HelloWorld” function ............................................................................................... 11
   “S3Trigger” Function ............................................................................................... 12
   DynamoDB .................................................................................................................. 12
   “Insert” Function ...................................................................................................... 13
   “Select” Function ..................................................................................................... 13
   API gateway .............................................................................................................. 13
   Tools ............................................................................................................................ 14

5. Results ......................................................................................................................... 15
   Initial Test of “HelloWorld” function ........................................................................ 15
   HelloWorld function test under concurrent requests ............................................ 16
   HelloWorld function test under maximum throughput and concurrency level ........ 17
   Latency Injection for HelloWorld function ............................................................. 18
   Initial Test of “S3Trigger” function .......................................................................... 21
   S3Trigger function test under concurrent request ................................................. 22
   S3Trigger function test under maximum throughput and concurrency level .......... 24
   Latency Injection for S3Trigger function .................................................................. 26
   Initial test of DynamoDB “Insert” function .............................................................. 28
Insert function test under maximum throughput and concurrency level ........................................... 30
Latency Injection for DynamoDB Insert function ............................................................................ 31
Initial test of DynamoDB “Select” function ..................................................................................... 34
Select function test under maximum throughput and concurrency level ..................................... 34
Latency Injection for DynamoDB Select function .......................................................................... 35
Early abort functions for HelloWorld, Select, Insert and S3Trigger ........................................... 37
6. Conclusion ........................................................................................................................................ 40
   Reference ......................................................................................................................................... 41
Figure 25 "Insert" function over 10000 requests ................................................................. 30
Figure 26 Maximum throughput for each concurrency ......................................................... 31
Figure 27 %of requests failed/succussed for each concurrency ............................................ 31
Figure 28 Average Request Latency with Probabilistically Injected Delays At 1000 Concurrency Level (Insert Func.) ................................................................................................................. 32
Figure 29 Requests failed under different probabilities for “Insert” function ..................... 32
Figure 30 Average response time for 25% probability with different concurrency (Insert func.) 33
Figure 31 Scan data from DynamoDb test Postman result ....................................................... 34
Figure 32 Maximum throughput for each concurrency ........................................................... 35
Figure 33 %of requests failed/succussed for each concurrency ............................................. 35
Figure 34 Average Request Latency with Probabilistically Injected Delays At 1000 Concurrency Level (Select Func.) ................................................................................................................................. 36
Figure 35 Requests failed under different probabilities for “Select” function ..................... 36
Figure 36 Average response time for 25% probability with different concurrency (Select func.) 37
Figure 37 Average Latency with Probabilistic Failure for early-abort functions .................. 38
Figure 38 Number of requests which early aborted (415) for all early abort functions .......... 39
List of Tables

Table 1 A Comparison of four serverless computing providers ........................................... 5
Abstract

Chaos Engineering in the Cloud

By

Madhavi Jaival

Master of Science in Computer Science

The growth of serverless computing has made it one of the most popular cloud computing execution models among enterprises. Since the conception and introduction of Serverless computing, a number of cloud service providers have developed and released their own Serverless computing platforms. For example, Microsoft Azure Functions, Amazon AWS Lambda, IBM Cloud Functions, and Google Cloud Functions. Due to its cost-effectiveness, and limited scope of administration, Serverless Computing has fast become a favorite cloud computing execution model. Meanwhile, with the rise of distributed cloud architectures and microservices in last decade, many development teams have adopted the principles of Chaos Engineering to assess and plan for potential application failure resulting from random failures or delays. In prior literature, serverless developers measured and reported cold-start penalties and transaction latency, whereas Chaos Engineers have studied security and resiliency in cloud infrastructure and hosted applications. In this thesis, we combine these approaches to measure the performance of a set of serverless cloud functions which implement common server-side file and database operations. We study each function's performance response under a set of controlled chaos experiments, wherein we emulate various client load conditions, as well as inject random delays into the function execution. We find that under heavy 1000-client load, even the longest-latency operations can provide as much as 36.5% improvement to response time by failing early.
1. Introduction

Modern applications need to support an increasingly large and geographically widespread user base, while requiring increasing amounts of compute power, data storage, as well as increased stability and security. To meet this trajectory, cloud computing has quickly become a standard hosting model. The widespread adoption of the cloud computing paradigm has generated a profound change in the way software is designed and hosted.

The term "cloud" refers to servers that are accessible over the internet, and our databases and software are hosted on these servers. Cloud server infrastructure is placed in data centers throughout the world. If we use cloud computing, we don't need to install any software or physical servers on our computers. It is a primary benefit of cloud computing. Users may access the same files and applications from any device since processing and storage take place on servers in a data center rather than locally on users' devices [24].

There are three primary cloud service models: each cloud model provides unique benefits that may fulfill the needs of different requirements. a) SaaS (Software as a Service), this model provides easy access to web applications hosted in the cloud. We do not need to install and maintain apps because the vendor owns the full system stack, which we may access through a web browser. b) IaaS (Infrastructure as a Service) is the virtual availability of cloud-based computing resources. A provider of IaaS in the cloud can offer the complete spectrum of computer infrastructure, including storage, servers, and network equipment, along with maintenance and support. c) PaaS (Platform as a Service) is a cloud-based environment where we may build, test, and organize apps for our projects/businesses. A PaaS implementation improves the software development process also it provides a virtual runtime environment that facilitates the development and testing of applications. All resources provided in the form of servers, storage, and networking can be managed by the business or a platform provider [16].

In 2014, Amazon popularized the Serverless Computing paradigm, with the introduction of the AWS Lambda. Two years later, Google Cloud Functions and Microsoft Azure Functions were released as similar offerings on other popular cloud platforms. In Serverless Computing frameworks, the server itself is hidden from the developer. Developers do not provision nor manage server environments. Rather, applications are written as a set of serverless cloud functions
which are hosted on and executed within small runtime containers. These functions are called by event handlers, and they are designed to be stateless. Thus, their containers can take advantage of the auto-scaling and provisioning infrastructure, without any effort on the part of the developer. The pricing of Serverless computing is based on function invocation and function uptime. This provides a cost-efficient service, as developers are only billed for the execution time of these containers, without any need to pay for unused system resources nor idle server time [10].

Serverless architecture is often seen as the subsequent stage in the evolution of Platform-as-a-Service. In a serverless architecture, applications depend on a third-party service known as Backend as a Service (BaaS) which also known as Function as a Service (FaaS). Using this service paradigm, a cloud provider provides backend functions such as data storage, allowing developers to concentrate on building front-end code. The cloud provider (AWS, Azure, or Google Cloud) is responsible for running a piece of code by dynamically assigning resources. And just charge for the resources used to execute the code. Code is often executed inside stateless containers that triggered by several events, such as HTTP requests, database events, queuing services, monitoring alerts, file uploads, and scheduled events [25].

There are several benefits to employing serverless computing, including the fact that developers do not need to maintain the server, and the code only executes when the serverless application requires backend functionalities, and the code will automatically scale up as required. In contrast to conventional server design, developers must anticipate how much capacity they will need before beginning work and then acquire the capacity, whether they wind up utilizing it or not. Serverless architecture is a novel approach of developing and delivering applications that enables developers to concentrate on the application's code. This strategy decreases time, operating expenses, and system complexity. While serverless eliminates the requirement to set up and configure real servers and virtual machines by using third-party vendors [25].

In the serverless architecture (FaaS) Application code is written by developers as a collection of distinct functions. When it is triggered by an event, such as an HTTP request, each function will carry out a distinct operation. Following the typical testing phases, developers deploy their functions and triggers to a cloud provider account. When a function is invoked, the cloud provider either performs the function on an operating server or, if there is no running server, will create a new server to execute the function. This execution process is hidden from developers.
Even though serverless architecture has existed for more than a decade, AWS Lambda was the first popular FaaS platform to be released in 2014 [26]. The majority of developers still use AWS Lambda to build serverless applications. Google and Microsoft, however, have their own FaaS offerings, notably Google Cloud Functions (GCF) and Azure Functions [19].
2. Background

Server maintenance is completely hidden from developers thanks to a new application deployment paradigm, **Serverless computing**, as we discussed in the introduction. Application runtime settings are partially visible to users. This allows developers to focus on developing their functions, which are small applications.

There are two primary Serverless computing use cases. The first is the hosting of a stateless application, whereas the second is a functional programming model application. Common examples of stateless systems utilizing functional programming paradigms include publishing application programming interfaces. Due to the benefits of the stateless paradigm, developers may easily deploy an application on numerous servers to scale it. A Serverless computing application or project often consists of several functionalities. These functions are often stateless, brief, and independent. A function is often defined by a short scripting language-based piece of code and is dedicated to executing a single specific task [19].

In Serverless Computing frameworks, the server itself is hidden from the developer. Developers do not provision nor manage server environments. Rather, applications are written as a set of serverless cloud functions which are hosted on and executed within small runtime containers. These functions are called by event handlers, and they are designed to be stateless. Thus, their containers can take advantage of the auto-scaling and provisioning infrastructure, without any effort on the part of the developer. The pricing of Serverless computing is based on function invocation and function uptime. This provides a cost-efficient service, as developers are only billed for the execution time of these containers, without any need to pay for unused system resources nor idle server time [10].

Amazon developed the virtualization technology Firecracker, which is a virtual machine monitor (VMM) built in Rust. It is the engine that powers all Lambda functions for a user. Firecracker generates and maintains a huge number of Linux Kernel-based Virtual Machines (KVMs), which are microVMs that are faster and more secure than conventional VMs. Firecracker has a REST API for creating, deleting, and managing virtual machines. When a user creates a new lambda function and uploads the code, the Firecracker REST-API is used to establish a microVM with the CPU and memory settings of the function.
AWS maintains basic images with language/runtime-specific bootstrap code. This is the actual code that launches user’s handler, sends it the request, and returns the result to the caller. This code is also used to monitor different metrics that are subsequently used to determine a user’s bill. After Firecracker has constructed a new microVM with a language-specific runtime for the user, the user's code is placed in its /var/runtime/bin subdirectory. This is where the bootstrap code sits as well. Essentially, user's function can execute and receive requests. AWS will eventually shut down the VM to save resources on their end. This is another API call to the Firecracker library. Incoming requests, such as those received through API Gateway, cause Firecracker to restart the VM so that it can process the request [12].

In the study by Wang et al., they generated 40 measuring functions with the same memory size and executed each with concurrent requests to determine the instance placement differences amongst three providers, including AWS lambda. After doing this experiment, they determined that AWS seemed to have the best concurrency support among the three providers. Using AWS lambda, the maximum amount of Memory that could be assigned to all containers on a single VM was determined to be 3,328 MB. And AWS lambda was considering instances placement as a bin packing problem and attempting to install a new container on a current VM in order to optimize VM memory consumption rates [19].

<table>
<thead>
<tr>
<th>AWS</th>
<th>Azure</th>
<th>Google</th>
<th>IBM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language support</td>
<td>C#, Go, Java, Node.js, PowerShell, Python, Ruby</td>
<td>C#, F#, Java, Node.js, PowerShell, Python, TypeScript</td>
<td>Go, Java, Node.js, Python, Swift</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Standard: 1000 per region (soft limit)</td>
<td>There is no stated concurrent limit</td>
<td>Standard: 1000</td>
</tr>
<tr>
<td></td>
<td>Reserve: varies</td>
<td>There is no stated concurrent limit</td>
<td></td>
</tr>
<tr>
<td>Cold Start</td>
<td>Average: &lt; 1 sec</td>
<td>Average: &gt;5 sec</td>
<td>Average: 0.5 – 2 sec</td>
</tr>
<tr>
<td>Maximum Timeout</td>
<td>15 Minutes</td>
<td>10 Minutes</td>
<td>9 Minutes</td>
</tr>
<tr>
<td></td>
<td>128 MB – 14000 MB (Premium and Dedicated Plans)</td>
<td></td>
<td>128 MB – 2048 MB</td>
</tr>
</tbody>
</table>

Table 1 A Comparison of four serverless computing providers

As noted, before, there are several reasons why AWS Lambda was used for this study. Table 1 compares four major providers of serverless computing. First, we can observe that
additional languages are supported by AWS Lambda. With IBM, Azure, and Google cloud services, the scripting and runtime languages available to developers are restricted and sometimes vendor specific. AWS Lambda natively supports languages such as Java, C#, Python, and JavaScript, among others.

The concurrency, which is the number of requests that may execute concurrently. AWS delivers this concurrency at a very high level, and it is also configurable, while IBM also gives good concurrency, but it is not configurable. The concurrent execution management of Azure and Google cloud functions is a bit ambiguous. Next, we may evaluate function's memory. If the memory is set too low, the function will take a long time to run, and there is a chance of timeout; nevertheless, if the memory is set too high, the user will be charged for wasted resources. All four of these cloud service providers provide varying maximum Memory options. The highest amount of memory that IBM, Azure, and Google cloud services provide is lower than that offered by AWS.
3. Related Works

Modern software-based services are implemented as distributed systems with complex behavior and failure modes. As a developer, it is essential to understand and enhance the resilience of the system. Therefore, in 2010, Netflix introduced Chaos Engineering. Chaos Engineering is the discipline of experimenting on a software system in production in order to build confidence in the system’s capability to withstand turbulent and unexpected conditions [1]. Chaos Engineering is not merely an approach to resistance testing, but also a means to characterize the steady state behavior of the system and monitor its availability and performance. With Chaos Engineering, developers could gain deep insight into the resilience of their application, including what conditions make the system stall or stumble. This is achieved by embracing and measuring failure, rather than avoiding it.

In 2011, the Simian Army was born [18]. Chaos Monkey was the first tool, invented by Netflix, whose purpose was to test the resilience of an infrastructure as its computers become unavailable. Chaos Monkey emulates a pseudo-random slaughter of AWS virtual machines. Since 2011, there have been more than ten “monkeys” and other “animals” added to the Simian Army, some of which are antagonistic to application resources, whereas others are neutral or benign. Netflix pioneered the techniques of Chaos Engineering to achieve application resiliency on cloud infrastructure, based on their experience with the Simian Army. The purpose of Chaos Engineering is not to hurt the application in production, but rather to build better understanding and confidence in a system’s resilience through controlled sensitivity experiments.

In the H. Jernberg’s published research, they designed framework to guide an implementation process of Chaos Engineering in a company and then maintaining the practice as a continuous testing strategy. In this framework ‘discovery’ activity generates a backlog of Chaos Experiments that are both possible and relevant to the system under test. As a part of the 'Implementation' activity, only one Chaos Experiment is set up and executed. Succeeding that comes the to ‘sophistication' activity, which aims to improve the validity and safety of a Chaos
Engineering implementation. Finally, the 'expansion' activity systematically expands the set of Chaos experiments that can be performed with current tools [7].

To tune application performance, serverless developers monitor cold-start effects and average end-to-end latency, as well as CPU utilization, function instance lifetime, and maximum idle time before shutting down. Wang et al studied the characteristics of serverless platforms’ architectures by evaluating data collected from over 50,000 measurement functions run under varied situations to test performance, and efficiency of resource management [19]. They created a Lambda "measurement function" which collects invocation timing and function instance runtime information, and then executes specified subroutines (e.g., measuring local disk I/O throughput, network throughput) based on received messages. With this, they tested scalability and instance placement, cold-start effects and VM provisioning, instance lifetime, idle instance recycling, and inconsistent function usage.

The maximum integration timeout provided to an HTTP API in API Gateway is 30 seconds by default. This timeout period cannot be increased. As request bandwidth scales up, response latency can vary wildly, directly impacting user experience [17]. Request timeout is a key challenge in Serverless Computing, as timeout has a massive influence on response latency, thus impacting the user in a visible way. Thus, we begin our exploration of serverless function performance by measuring the sensitivity of these functions to scaled request bandwidth.

Kitzes and Kaplan established benchmarks that allowed them to directly exercise, measure, and compare the performance of Node.js and Apache/PHP in typical client-server interactions [9]. They found that for the common operation of delivering large static files, there was no discernible difference in speed between the Apache and Node.js engines. Node.js was found to outperform Apache in cases where a webserver hosted a single application and performed as well as Apache in cases in which a server hosted multiple applications which were each being simultaneously accessed by thousands of client requests. Kitzes and Kaplan's benchmark suite was comprised of a set of common web service operations such as database access, large file fetch, and static string response.

H. Lee’s study compared IaaS and FaaS based on experiments with big data and deep learning applications. This study evaluated the concurrent invocations on serverless computing platforms like Amazon Lambda, Microsoft Azure Functions, Google Cloud Functions, and IBM
Cloud Functions (Apache Open Whisk). They indicate that a task which is small enough to operate on a function instance with a memory restriction of 1.5GB to 3GB and an execution time constraint of 5 to 10 minutes is a good candidate for a serverless cloud function. Lee et al demonstrated this using a benchmark suite of representative web-service operations. This was comprised of basic HTTP triggers, database requests, and S3 object requests [10], which were serverless analogs of the web server benchmarks used by Kitzes and Kaplan [9]. These same operations provided the basis for the benchmark suite of Lambda functions used in the present work.

Cui has documented the difficulty in applying the principles of Chaos Engineering to Serverless Computing platform [2]. This arises from the lack of configuration management options in serverless frameworks, which reduces the amount of chaos that can be generated by a developer. Meanwhile, developers have the increased burden of hardening security at the perimeter of each function as opposed to hardening once at the perimeter of an entire service which encapsulates a set of functions.

Samdan has performed chaos experiments in stateless environments and thus demonstrated that latency is the most chaos-sensitive serverless metric [14]. Samdan points out that in Serverless Computing, if an expected response is late, this is often a signal that the service is broken. Zhu [21] studied the impact of bandwidth-demand and delay-injection on a mobile game server application, implemented as a set of cloud functions interacting with databases and object stores. Zhu injected both constant and random latencies into each function and scaled the client load to generate both chaos and bursty traffic.

Zhu measured the impact of cold-start effects and provisioned concurrency on this application's response. In this thesis, we expand-upon and extend prior research work [21][8] by injecting random delays into our lambda functions at random intervals to characterize their chaos sensitivity. We also scale client load to compare against the effects of random delays. Moreover, we also measure the effects of random cloud function failure, which emulates the effect of a Chaos Monkey [18] in a serverless setting.
4. Methodology

We build and deployed functions on AWS Lambda. All the functions in AWS lambda are written in JavaScript using Node.js 16.x. AWS Lambda supports a variety of programming languages, including Java, C#, Python, and JavaScript, all of which execute on the Node.js runtime. Because of the necessity for cold start performance, we picked this language since Java and C# has unusually slow cold start times in serverless applications. The blog posted by Igor Skoldin finds, for an I/O-intensive job with concurrent execution, NodeJS offers the benefits without the complications of multithreading. This helps to illustrate NodeJS’s popularity for serverless systems[15]. For serverless functions, this might be a deal breaker. The functions are deployed independently on Serverless platforms, which offer instant per-request elasticity. This flexibility comes at the expense of the cold-start problem, which happens when previously implemented (warmed) Lambda instances make the next function call faster than a newly uninitialized (cold) instance. As a result, the latency of the function response varies. The blog posted by N. Malishev finds that, Node.js improved cold start timing by 74.6%, making it the most effective programming language of all [11].

Figure 1 shows the organization of the cloud infrastructure used in this study. Our Amazon AWS Virtual Private Cloud (VPC) is hosted in the us-west-2 region. Our Division of Information Technology division, which manages the AWS resources for our institution, was able to deploy a Bastion Host running an SSH server with Multi-factor Authentication. This Bastion Host is within our AWS VPC, but also accessible on its public subnet using a public IP address. We use the Bastion Host to limit VPC server exposure. We log into this server to connect to our EC2 instance, which is hosted on a private subnet. The EC2 instance behaves as a client machine which sends HTTP requests to our web services by executing the benchmarking tool hey [4]. Hey is an HTTP load generator based on ApacheBench (ab) [27]. Hey allows a user to configure total number of requests to run and total number of concurrent clients.

After all requests are launched and responses are received, the tool provides aggregate response time (total time taken for request), the completed requests per second, and response time histogram and distribution. In this study, we launch 10000 total HTTP requests for each server configuration, with concurrent clients scaled from 50 to 1000. The HTTP request headers and
bodies are directed to AWS API Gateway, which invokes an AWS Lambda function and passes the HTTP request to the Lambda as a JSON object. The Lambda can then interact with other AWS services to process the request. In this study, our Lambdas make use of the AWS services DynamoDB and Simple Storage Service (S3). DynamoDB is a scalable NoSQL database which allows eventually consistent read-access. S3 is an object storage service which stores objects in containers known as S3 buckets.

In this study, we measure the performance of a set of lambda functions which are representative of common web-service operations [10][9][21]. These operations are comprised of a basic HTTP trigger, DynamoDB row-insert and row-select, and an S3 object fetch, as follows.

“HelloWorld” function

The first function is to evaluate the functionality of our internal HTTP client without making use of any URLs or websites hosted on third-party servers, such as Google or YouTube. As a result of this, we require our own internal API, which consists of a "body" section that contains a message and an API handler. Other libraries and AWS config settings are also necessary, such as X-Ray segments. A developer can design a Lambda application by using example code and configuration settings provided by AWS Lambda. A number of blueprints are provided by AWS Lambda, and these blueprints can be found on the AWS Lambda website. In
order to test the API Gateway, we created a "HelloWorld" function that characterizes the baseline response of the web service, with no compute-intensive nor I/O-bound functionality. This is similar to the "no-op" application employed by Kitzes and Kaplan [9]. This function will send the "GET" request to the function that is part of our internal API, and it would return the body message of that request, which only has two terms: the status code, and a simple string: "Hello from Lambda!"

“S3Trigger” Function

We developed a function that is associated with the Amazon Simple Storage Service (S3) to test the service of huge static files to retrieve picture from storage. The 3.2 MB JPEG image file that we save into storage and measure the throughput of picture retrieval [9]. To retrieve the image from the S3 bucket, we are using the "get object" method in S3. Functions may send an attachment since the "Content-Type" is "application/jpg" and the "Content-Disposition" is "attachment". We may have the choice to save the image every time we use the API rather than having to download it with each request. The "body" section of this function is where we use the "b64encode" technique to convert the image we into a binary-encoded format in base64 format, so that it may be returned as an ASCII string. This scenario represents the common transfer of profile pictures and other photographs, for instance on a social media platform [9].

DynamoDB

Instead of using the most popular database on AWS, we picked DynamoDB. The reason for that, in contrast to other transactional databases, such as Oracle, MSSQL, or PostgreSQL, AWS DynamoDB is schemeless. This means that it does not need conformation to a specific schema of data types, tables, etc., which is a significant advantage over these other databases. It also provides the important benefits such as consistently high performance and millisecond latency [5]. AWS DynamoDB, in contrast to other NoSQL databases, offers data types such as key-value pair and document data structures such as JSON, XML, and HTML. AWS DynamoDB also automatically increases throughput capacity to match workload expectations and splits and re-partitions our data as the size of your table grows.

In this study, we have set up two tests using DynamoDB by developing two lambda functions - one for a "POST" request and another for a "GET" request. We named both functions respectively “Insert" and the other is "Select". All the data, we use in this study is from ODI (One Day International) CRICKET batting player’s high score data in JSON file format. We generated
a single JSON file with 50 lines of data from all the best players for different years. Additionally, we created a DynamoDB table named "Cricket" and inserted "Id" as the partition key and "Runs" as the sort key.

“Insert” Function

The first test is inserting items into a DynamoDB table using a "POST" request. We next implement a function that inserts data into our DynamoDB table through a "PutItem" API call. We add data to the table from the JSON file. The method chooses one player at random and retrieves its "playerId," "PlayerName," and "Runs" then adds these variables to the DynamoDB table. Since we may have a high volume of traffic in a single test, we use a randomizing strategy to improve the write loads. We use the Math.random() function which generate a random number from 1 to 50, which we then use to search for a player's ID in our JSON file. If it finds a match, the function then inserts the player's data to our DynamoDB as attributes "PlayerId," "PlayerName," and "Runs"; otherwise, it simply overwrites the existing data.

“Select” Function

In the second test, the same table is used as in the "Insert" function. So far, we have inserted all the data into our cricket table using the "insert" method. We use "scan" API calls to read and retrieve all player items in the cricket table.

API gateway

To deploy an HTTP endpoint on Lambda for this test we used the API Gateway, which was developed in Node.js and solely available on AWS, Amazon API Gateway is a software as a service (SaaS) solution. The AWS ecosystem's backbone is delivered by Amazon API Gateway. A REST API in API gateway is a set of resources and methods that are integrated with HTTP endpoints on the backend, Lambda functions, or other AWS services [22]. API Gateway enables programmers to set up HTTP endpoints on Lambda. For converting AWS functionality to HTTP requests, API Gateway is a collection of tools and methods. Processing access data or returning data from an application using the HTTP "GET," "POST," "PUT," and "DELETE" protocols is the main use case for API Gateway. Lambda functions may also be called over API Gateway. When we access the API using an endpoint, it also offers several additional choices, such as letting API Gateway enable a Lambda function to perform the whole HTTP request and mapping all API resource's methods to a single Lambda function.
Tools

The first tool we used is the serverless-artillery, this is a contemporary, robust, and user-friendly performance testing toolkit. It may be used to launch scalable applications that remain performant and resilient under high load. This is a new AWS-based load-testing system. This tool may be used to tailor the number of requests per second and overall duration for various degrees of traffic load [6].

The second tool we employed was "hey," which was originally known as boom and was inspired by Tarek Ziade's “boom” tool, this hey tool for AWS was released by Jaana Dogan [20]. Based on the ApacheBench (ab) tool, hey is an HTTP load generator [27]. The primary purpose of “hey” in this study is that a user may choose the number of total requests, concurrency level. After receiving all responses, the tool executes the queries and provides precise figures and information.

The last tool we used is Postman, a graphical user interface API testing tool, to test our APIs. This application's simplicity of use allows us to save time when conducting testing. Postman is an API client for developing, testing, sharing, and documenting APIs. It makes the request to the server and receives the response back from the server for backend testing. The same thing can be achieved through API templates such as Swagger too. In both Swagger and Postman, we are not required to develop a framework to receive the response from the service [23]. This is the primary reason why Postman is widely used by developers and automation engineers to verify that the service is up and running alongside the latest version of an API that is being deployed into the regions. This application can make various types of HTTP requests (GET, POST, PUT, PATCH) to the web server and receives the response. We don't need to do anything additional or set up a framework to send and receive requests with Postman.
5. Results

We build and deploy all the functions of the AWS lambda via console and used AWS CLI to run the output of the test. We evaluate the performance and behavior of each function based on initial cold-start latency, median response time, maximum throughput, and maximum latency. Figure 1. shows that we used the way to test the cold-start latency of the lambda function using a logging statement outside the handler of our function. When the Instance starts running then the first line will execute but after that, the instance stays live then first line won’t be executed anymore and the code of the cold start in the handler will run the second code “console.log”.

Initial Test of “HelloWorld” function

We start with the most fundamental test, which is initial measurement of the function's cold-start and warm-start latencies. The result that has been given by the AWS X-Ray can be seen in Figure 2. X-Ray offers information regarding the length of time spent on each step for the duration of one function. The duration of all the steps to complete our first test was 365 milliseconds in total. In this instance, there is an additional phase that is referred to as function initialization that takes 206 milliseconds of time before our function is really executed. This is part of the cold start latency which refers to the amount of time it takes the servers hosting the function to load all its dependencies.

<table>
<thead>
<tr>
<th>Name</th>
<th>Res.</th>
<th>Duration</th>
<th>Status</th>
<th>0.0ms</th>
<th>50ms</th>
<th>100ms</th>
<th>150ms</th>
<th>200ms</th>
<th>250ms</th>
<th>300ms</th>
<th>350ms</th>
<th>400ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>HelloWorldLambda</td>
<td>200</td>
<td>365 ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2 “HelloWorld” cold start function X-Ray traces

Function initialization also includes the amount of time spent running code outside of the function. After the successful completion of the function’s initialization stage, actual execution will begin, and the entire process takes 16.6 milliseconds. We may determine that the overall duration time is 365 milliseconds by looking at the top bar or using AWS CloudWatch.
When we begin the second measurement, our function will begin executing immediately after sending the request. Because the function has recently become "warm," there is no cold start time and initialization time for the function. Figure 3 demonstrates that our second measurement requires 81.0 milliseconds in total.

![Figure 3 “HelloWorld” warm start function X-Ray traces](image)

**HelloWorld function test under concurrent requests**

In this test, we want to see how well our function works under varying conditions regarding the number of requests per unit of time. In this test, we use the "hey" tool. At the beginning, we configured the 10000 total HTTP requests, and the concurrency has been set 500 VU per second. This is a simple test to determine how well HelloWorld function can cope with the baseline conditions.

![Figure 4 “HelloWorld” function over 10000 requests](image)

The output of the "HelloWorld" function is displayed in Figure 4, this is the value that is returned by the "hey" in the AWS CLI window. It includes the number of scenarios that have been
started and finished, as well as the mean response per second, the detailed response time in seconds, and the response times of the percentile.

We got all 10000 requests response with the status code 200; they have all been successfully completed. We have noticed the consistency of the mean response per second, which is 4,235 VU requests per second that means this is the steady state of the function. Due to cold start latency to the longest of the test's response time 1.12 seconds. The average response time is 0.11 seconds. In the meantime, there are two response times that must be noticed, 95% of all responses are served and finished within 0.64 seconds, while 99% of all responses are served and finished within 1.00 seconds.

**HelloWorld function test under maximum throughput and concurrency level**

The purpose of this test is to determine how well our function handles failure under maximum throughput and concurrency requests. AWS Lambda provides a configuration known as provisioned concurrency. Provisioned concurrency initializes the specified number of execution environments, allowing them to prepare for a high volume of concurrent function calls. Lambda assigns an instance to process the event when our function is invoked. In addition, once the function has completed its execution, it can automatically handle another request depending on the amount of concurrency that the developer configures. AWS Lambda offers 1,000 unreserved account concurrencies, including 100 for functions that don't have a concurrency limit; by default, these concurrencies are in the same account and region.

In this experiment, we create request traffic with the "hey" tool to study the impact of scaling client load from 50 to 1000 concurrent requests, sending 10000 total HTTP requests to HelloWorld function. This will help us to provide the maximum throughput that our function can handle under the 1,000-concurrency level. We re-use the “hey” tool collect data to construct a histogram shown in Figure 5, the left y-axis and corresponding labeled bars represents the average number of requests per seconds received from the HelloWorld function at a given concurrency level. The right y-axis and corresponding line represent the average end-to-end latency experienced by each request, reported in seconds.

We note that HelloWorld experiences the largest reported response bandwidth of 5523 responses per second when experiencing 200 simultaneous requests per second. However, the corresponding average request latency of 0.03 seconds is not the lowest average request latency
reported, which is 0.02 seconds. Moreover, HelloWorld experiences the largest average request latency at a concurrency level of 1000 simultaneous clients, but the corresponding response bandwidth of 3568 responses per second is not the smallest response bandwidth reported (which is 2162 responses per second at a concurrency level of 50). That means the HelloWorld function tends to have a much better response (thousands of responses per second, with requests experiencing an average latency of less than 0.25 second) than the DynamoDB and S3 Trigger function, which fetches files from AWS S3 Bucket.

![Figure 5 Maximum Request throughput and Avg. response time](image)

![Figure 6 Percentage of requests failed/succeeded under different concurrency](image)

Figure 6 is representing the failure and success rate as a request demand increased, the x-axis represents the value of concurrent requests for 10,000VU, and y-axis represents the percentage of request failed. The blue color represents the number of requests succeeded; orange color represents the number of requests failed. For example, here we got 0.01% of request failed under 1,000 concurrency level, due to Lambda function timeout which resulted with HTTP code 500, and under remaining other concurrency level requests are 100% succeeded with HTTP status code 200.

**Latency Injection for HelloWorld function**

The next step is to generate some basic "chaos" by injecting latency into each of our functions. As noted by Cui in his study [2], Two methods exist for injecting latency. The first method is to add fixed delays to operations on each execution, while the second method is to configure how frequently and how much delay need to add to functions. To generate chaotic delay within HelloWorld function, we insert an additional random delay between 100 ms and 1 sec, between 100 ms and 2.5 sec, or between 100 ms and 5 sec. This delay is injected into the execution
of HelloWorld function under study. The delay is added probabilistically, such that delays are injected into 5%, 10%, 15%, 25%, or 35% of HelloWorld function executions. We perform these experiments at a concurrency level of 1000 simultaneous client requests, for 10000 total requests.

In Figure 7, we compare these chaotic delays with the situation in which no latency is injected (labeled "no add. latency"). With a 5% injection probability, adding 1 second, 2.5 seconds, or 5 seconds of latency results in a slightly longer average response time than not adding any latency. Similar to the 10% probability of injection, but in this case all the added latency has the same average response time for the execution of 1000 client requests. For 15%, the 1 second latency injection demonstrates a greater average response than the other two latency injections. With a probability injection of 25%, the average response time for 1 second, 2.5 seconds, and 5 seconds increases as the probability of injection increases. For 35% probability, 5sec latency injection has a greater average response time than 1sec and 2sec latency injections. We notice that regardless of how much maximum latency is added to function execution or how frequently, the average request latency tends to saturate at 2-3x the original request latency with no added delays.

![Figure 7 Average Request Latency with Probabilistically Injected Delays At 1000 Concurrency Level (HelloWorld Func.)](image)

We construct a histogram of above experiment for failures as shown in Figure 8. This graph depicts the number of failed requests (out of a total of 10,000) with probabilities of 5%, 10%, 15%, 25%, and 35%. The x-axis represents the injection latency, which ranges from 0 seconds (no added latency) to 100 milliseconds to 1 second, 2.5 seconds, and 5 seconds. If there is no latency added function, the success rate is 100%, but if there is a latency added function, such as 100 ms to 1
second, the success rate decreases slightly. Similarly, the failure rate increases at all levels of probability injection addition.

![HelloWorld Chart for success & failed Requests](image1)

**Figure 8** Requests failed under different probabilities for “HelloWorld” function

Figure 9 depicts the average response time for all concurrency levels when the constant probability delay is added to the function. To influence our latency configuration, we then added three maximum latency values. The goal is to determine the resilience of HelloWorld function under maximum concurrency over a constant 25% chance of injecting the maximum latency into the HelloWorld function.

![Average response time for 25% probability with different concurrency (HelloWorld func.)](image2)

**Figure 9** Average response time for 25% probability with different concurrency (HelloWorld func.)
We set the number of HTTP client requests to 10000, the concurrency levels to 50, 100, 200, 500, and 1000, and the latency to 1 seconds, 2.5 seconds, and 5 seconds. The no added latency (blue bar) function has a lower average response time than the other three latency injected functions for 50 concurrent client requests. Similarly, we can see that the 100 and 200 concurrent requests have the same result as the previous one, except 2.5sec latency (grey bar) injection has a longer average response time than the 5sec latency injection. For 500 and 1000 concurrent requests, the response time was reduced by half for all three latency injections in the graph, while the no latency added average response time increased and remained the same for 1000 concurrent requests at the 1sec latency added function.

Initial Test of “S3Trigger” function

After creating the S3 bucket and adding an image file to it, we create a function to retrieve a 3.2 MB image file from the S3 bucket. We are ready to begin testing with the S3 bucket service. We test the function that can accept the parameters bucket name and key, which is our image file, and then it retrieves the file using the Get Object API.

![Figure 10 Function output in postman](image)

The output of the API test using Postman shown in Figure 10. Instead of JSON, we returned a base64 string representation of the picture in lambda (buffer.toString('base64')), forcing API Gateway to encode the string to binary, and add a specified Content-Type (so we don't have to utilize their restricted binary support, which requires us to send a special Accept header). Postman also provides the service to save the response generated by the API so we can save response to our local file system using “Save Response” button.
The Figure 11. is the X-Ray tracing of our image fetching lambda “S3Trigger” function. We can see that the initialization for this function is longer than the previous function. This initial test totally cost 3.3 seconds to complete all the steps. Before the function executes, there is an additional step called function initialization which is denoted in output is Init duration which is 416 milliseconds in this case and it’s a cold start of our function. After the initialization step is done, our function starts executing, then it takes total time 2.7 seconds to complete the task of our function.

When we started working on our second measurement, S3Trigger function begins to execute immediately after the function makes the request. Because the function has recently become "warm" there is no cold start time and no function startup time. As a result, our second measurement took 1822.02 milliseconds, as shown in Figure 12.

S3Trigger function test under concurrent request

We want to test “S3Trigger” function in this experiment using a variety of request rates and durations. In this test, we employ the "Serverless- artillery" tool. We set the "time" to 180, which equals a 3-minute duration, then configure the “target” to our lambda function gateway endpoint, and the "arrivalRate" to 10 virtual users, which represents ten requests per second; hence,
there are 1800 requests in 3 minutes. This is a simple test to examine how our function handles this initial test.

Figure 13 shows the s3test.yml file in this file “config” section defines the main target section. Additionally, it may be used to load and set up plugins and unique JS code. Next is the “phase” section which helps to define how artillery calls the virtual users’ requests in a specified time. And the last section is “scenario” which is the definition of one or more scenarios for the virtual users (VUs) that Artillery will develop may be found in the scenarios section. Each scenario consists of a set of actions that corresponds to a typical series of requests or messages provided by an application user. It consists of “flow” attribute a variety of actions taken by a virtual user. Such as an HTTP-based application, to run GET and POST requests; for a Socket.IO test.

```
# Thank you for trying serverless-artillery!
# This default script is intended to get you started quickly.
# There is a lot more that Artillery can do.
# You can find great documentation of the possibilities at:
# https://artillery.io/docs/
config:
  # this hostname will be used as a prefix for each URI in the flow unless a complete URI is specified
  target: "https://xkcrv3v96.execute-api.us-west-2.amazonaws.com/Test"
phases:
  -
    duration: 100
    arrivalRate: 10
scenarios:
  -
    flow:
    -
      get:
        url: "/lambdaimagetest1?key=Test2.jpg"
```

Figure 13 S3test.yml file for serverless artillery test

The output of the "S3Trigger" function is shown in Figure 14, and this result is returned by the "serverless-artillery" tool in the AWS CLI window. It included the number of scenarios initiated and performed, the request rate, response time in milliseconds, and percentile response time. All 1800 requests are completed without failing a single request with a status code of 200. The requested rate per second is 10 virtual users, indicating that the number of requests issued is consistent and stable.
Because of the delay introduced by the cold start, the maximum response time is 3768 milliseconds. The median response time is 1353.1 milliseconds. In the meantime, there are two response times that need to be taken into consideration: p95 and p99. This indicates that 95% of all requests are served and completed within 1720 milliseconds of this test, and 99% of all requests are served and completed within 3328 milliseconds of this test.

The purpose of our second basic test is to examine how our functions handle failure under conditions of high concurrent load and throughput. In this experiment, we used the "hey" tool to create request traffic. To test if our functions could manage this load, we set the Total number of HTTP requests to 10,000 and 500 simultaneous clients requests. The result is shown in Figure 15.

Figure 14 “S3Trigger” function over 1800 requests

Figure 15 “S3trigger” function over 10000 requests
Since the S3Trigger function fetch the request from S3 bucket, the response time is typically 16.07 seconds longer than the HelloWorld functions. Compared to the average response time of our HelloWorld function, which is 0.11 seconds, there is a huge gap. According to the hey tool's "S3Trigger" lambda function output, all 10,000 requests were successfully completed with the HTTP status code 200. The overall time of the execution is 338.52 seconds, with the slowest being 20 seconds, the quickest being 1.15 seconds, and the average duration is 19.17 seconds. The request rate is 29.53 requests per second.

Then we raised the number concurrency level to 1000 to see if the lambda could manage this much concurrency. We chose this option since AWS Lambda offers 1000 unreserved account concurrency. Figure 16. illustrates the result of the "S3Trigger" lambda function, indicating that our 80% request was successfully completed with the HTTP status code 200. On the other hand, the 1 request with the HTTP code 500 and the other 20% of requests received problems with the message "too many open files". This error indicates that our program attempted to open a network socket but had already reached the maximum number of open files allowed by the operating system. On Linux, "max open file limit" is set per process or user by default. There are numerous ways to configure this limit, including the ulimit command. Ulimit is used to display or limit the amount of system resources that individual users can access [3]. We increased the open file descriptor value by 50,000 using the ulimit command.

![Figure 16 “hey” output before adding ulimit](image1.png)

![Figure 17 “hey” output after adding ulimit](image2.png)
Then, we quickly reran the test, and this time, all our requests executed successfully without “too many open files” error message. In Figure 17. shows the result of the “S3Trigger” function after adding the ulimit all our request successfully returned with the status code 200 with only 5 client timeout and 1 server timeout error. Which increased the success rate of S3Trigger function by around 99.94%.

After receiving a successful response from the S3Trigger function, we proceed with the concurrent client requests test, increasing the number of concurrent client requests from 50 to 1000 and sending a total of 10000 HTTP requests to the S3Trigger function. Figure 18 depicts the outcome of this experiment. For each level of concurrency (simultaneous clients). The average number of responses per second received from the S3Trigger function at a given concurrency level is represented on the chart by the left y-axis and corresponding labeled bars. The right y-axis and corresponding line represent the average end-to-end latency experienced by each request. We should mention the S3 Trigger function, which retrieves files from an AWS S3 bucket. S3Trigger has tens of responses per second, with average latency ranging from 1.5 seconds to 20 seconds.

![Figure 18 Maximum Request throughput and Avg. response time](image)

**Latency Injection for S3Trigger function**

We use the same configuration code within the S3Trigger function as we used in HelloWorld function to add chaos to the "S3Trigger" function. The random latency then is injected into our function, and the test has been run with 10000 HTTP client requests and 1000 concurrency
(simultaneous client) requests. Figure 19 depicts a bar graph of the average response time of each latency injection with randomly injected delays of 5%, 10%, 15%, 25%, or 35%. We are comparing chaotic delays to no latency injection (labeled "no add. Latency"). We notice that the S3Trigger function has an average request latency of about 20 seconds, regardless of how much latency is added or how frequently. This suggests that S3Trigger (and, indeed, the AWS S3 service itself) may have a latency-response that is more sensitive to client load (as illustrated in Fig. 19) than chaotic system delays (as shown in Fig. 20).

![Figure 19 Average Request Latency with Probabilistically Injected Delays At 1000 Concurrency Level (S3Trigger Func.)](image1)

![Figure 20 Requests failed under different probabilities for “S3Trigger” function](image2)

We also note the rare cases in which adding chaotic delay results in the same average request latency as adding no delay. In Figure 20, we can see that for all probability, latency of 1 sec (orange bar), 2.5 sec (gray bar), and 5 sec (yellow bar) are all the same as no additional delay (blue bar), the injected latency clearly influences the failure rate in any configuration, with no additional delay (0 sec) having a higher success rate than the other three groups (100ms to 1sec, 100 ms to 2.5 sec, and 100 ms to 5 sec). That means when we inject latency, we are getting better average response times for all latency injection, but this results in 15% - 18% request failure.
As shown in Figure 21, we test the S3Trigger function under a constant 25% probability and random delay between 100ms and 1 sec, 100 ms and 2.5 sec, and 100 ms and 5 sec. We observe that the average response time for concurrency level 50 increases as the random delay increases from 0 (no added latency) to 5 seconds. As concurrency level increases from 200 to 1000, the average response time also increases. At concurrency level 1000, the average response time is 20 sec which is same for all added delay and no added latency (blue bar). We conclude that the S3Trigger function, which retrieves a file from the AWS S3 service, may have a response latency that is more responsive to client load.

Initial test of DynamoDB “Insert” function

Our previous test carried out a function against S3Bucket as an external service. In the following function test, we execute against a second AWS external service, DynamoDB. Once the
identify access management service and table configuration have been successfully configured and deployed, DynamoDB service testing can begin. We test a function that employs the "PutItem" API call, which inserts a record into our DynamoDB table. This function is referred to as "Insert".

In the beginning, we conduct a simple test to collect an initial trace from AWS X-Ray. Our function only inserts one item per request into the table (invocation). After executing our function ten times, the resulting data are displayed in Figure 22. We use the "Id" of the player as the partition key for this table.

![Figure 23 DynamoDB “Insert” function traced by X-Ray](image)

The AWS X-Ray result is shown in Figure 23. In this case, the “Insert” function total response time is 1.4 seconds. Again, we execute the tests six times, and X-Ray reports and initialization time for insert function is 423 millisecond and an invocation response time is 874 milliseconds. This time also includes the writing data to the DynamoDB.

![Figure 24 POST API call test using Postman tool](image)
Then, using the same configuration as in the previous steps, we configure the API Gateway endpoint for our function. REST API was chosen as the HTTP endpoint API type in previous tests. We use the same REST API in DynamoDB tests. Within the AWS API Gateway service, we then create a POST endpoint for this API. We get a new URL for our POST function when this succeeds, that we use to run the following tests. Figure 24. shows the results of testing the accuracy of our function with the "Postman" tool. The "BODY" window displays the data that the function inserted into the table, validating the correct functionality of the API.

Insert function test under maximum throughput and concurrency level

After running the initial test, we used the same "hey" tool to generate request traffic as we did with our HelloWorld and S3trigger functions to determine the initial average response time. Additionally, we added 10,000 VU with 500 concurrent virtual users. The "hey" output illustrated in Figure 25. We got all 10000 requests response with the status code 200; they have all been successfully completed.

![Figure 25 "Insert" function over 10000 requests](image)

We have noticed the consistency of the mean response per second, which is 204 VU requests per second that means this is the steady state of the function. Due to cold start latency to the longest of the test's response time 4.29 seconds. The average response time is 0.94 seconds. In the meantime, there are two response times that must be noticed, 95% of all responses are served and finished within 3.26 seconds, while 99% of all responses are served and finished within 3.38 seconds.
Next, we test the Insert function with 10,000 virtual user requests at varying concurrency levels from 50 to 1000, collect the resulting data, and create the bar graph shown in Figure 26. The left y-axis represents the maximum throughput of requests per second, the right y-axis and corresponding line represents the average end-to-end latency experienced by each request in seconds, while the x-axis represents the concurrency of requests used for testing. As a result, the maximum throughput level of the "Insert" function can be estimated to be approximately 480 requests per second with the default setting. Also Insert function requests experiencing an average latency of 0.1 second to over 1 second, depending on the load. This means DynamoDB tends to have far better response than the S3Trigger function which fetches files from an AWS S3 bucket. Figure 27 depicts the failure and success rates as the request volume increases. The x-axis represents the number of concurrent requests assigned, and y-axis represents the number of requests failed or success. Resultantly, nearly 4% of 1,000 concurrent requests failed.

Latency Injection for DynamoDB Insert function

We start our latency test on DynamoDB Insert function by using the same scenario as prior work. First, we configure total 10000 HTTP requests, with 1000 concurrency level and adding latency delay as 100 ms to 1 sec, 2.5 sec, or 5 sec respectively. In Figure 28, we observe that the average request latency more than doubles (from nearly 1 sec to over 2 sec) if a random latency of up to 1 sec is added to just 5% of Lambda executions. We note that adding latency of up to 2.5 seconds or up to 5 seconds does not increase average request latency much further (as much as 2.3 seconds in the worst case). This result is counter-intuitive, as it may appear that we are adding increasing amounts of latency to the function execution. However, the amount of latency is
randomly chosen to be from a minimum of 100 ms to a maximum of 1, 2.5, or 5 sec. Using a maximum random delay of 1 second could, in practice, generate delays that average 0.95 seconds in practice, whereas using a maximum random delay of 2.5 seconds could generate delays that average 0.4 seconds.

For the preceding tests, we created a histogram of failed and successful requests for a total of 10000 HTTP requests with random probability latency injections, as shown in Figure 29. We note that 0 sec (no add. latency bar group) has higher success rates than latency injections. That is, in any configuration, the injected latency has a clear impact on the failure rate.
Figures 28 and 29 show that, the successful request rate for the group of 100 ms to 2.5 sec latency injection increases as the average response time decreases; for example, the average request time for 35% probability for <=2.5 sec is around 0.8 sec, and the success rate is 90%. Similarly for 100 ms to 5 sec as the average response time increases the failure rate of requests also gets increased; In Figure 29, For 5% of probability and latency of 5 sec has around 1.3 sec average response time which is lower than other probability. Figure 29 shows that the failure rate for the 5% probability for latency of 100 ms to 5 sec is lower than for the other probabilities.

![Insert function Latency Injection Average res time for 25% probability](image.png)

Figure 30 Average response time for 25% probability with different concurrency (Insert func.)

Once again, we test Insert function under different concurrency level from 50 to 1000 (simultaneous concurrent request) with fixed probability of 25% and random delay injection from 1 sec to 5 sec, shown in Figure 30. The delay <= 5 sec average request time for the 50 concurrent requests is greater than no add. Latency (blue bar) and two additional random delays (1 second and 2.5 seconds). In contrast, for concurrency 200, latency injection <= 5 sec (yellow bar) results in the same average response time as no add. Latency (blue bar) which is approximately 0.27 seconds. We note that as we increase the concurrency level from 500 to 1000, the average response time for no add. Latency is better than all added random delay latency.
Initial test of DynamoDB “Select” function

To test further, we design a second function to execute DynamoDB API calls. In this function case, we retrieve the maximum Runs associated with a given player's name and five other attributes from the entire set of stored cricket player data. We expect that this search will take significantly longer to complete than the "HelloWorld" and "Insert" functions that were previously measured.

![Figure 31 Scan data from DynamoDb test Postman result](image)

We use the "Scan" method to build this "Select" function, and we test its functionality on a small "cricket" table, this is the same table we used for the "Insert" function to insert items. Figure 31. shows the output from Postman, that the Select function successfully return all the data from this table which we added using our previous Insert function.

Select function test under maximum throughput and concurrency level

After running the initial test, we used the same "hey" tool to generate request traffic as we did with our previous functions to determine the initial average response time. Additionally, we added 10000 VU with 50 to 1000 concurrent virtual users to Select function. The result of this experiment is presented in the Figure 32, for each concurrency level. On the graph left y-axis and corresponding labeled bars represent the average number of responses per second received from
the Lambda at a given concurrency level. The right y-axis and corresponding line represent the average end-to-end latency experienced by each request, reported in seconds.

As we note in our previous all functions a steep increase in request latency as the number of simultaneous clients increases from 200 to 500. But, in Select function we see monotonically increasing latency as the concurrency level increases. In the case of Select at a concurrency level of 1000 simultaneous requests, 5% of these requests fail due to request timeout, which may impact the reported average in the hey tool shown in Figure 33. This is because the reported average latency is only taken over successful requests.

**Latency Injection for DynamoDB Select function**

Next, we proceed to execute “chaos” experiments on Select function. To study the latency sensitivity of the DynamoDB Select function, we create the same configuration as in the previous functions, with 1000 concurrent client requests for a total of 10,000 requests. The probabilities of adding delay to the Select function are 5%, 10%, 15%, 25%, or 35%. In Figure 34, we contrast these chaotic delays with the situation in which no latency is added ("no add. delay"). For Select function, if a random latency of up to 1 sec is applied to just 5% of Lambda executions, the average request latency more than doubles (from roughly 1 sec to over 2 sec). We observe that adding up to 2 or 5 seconds of latency does not significantly increase the average request latency (as much as 2.3 seconds in the worst case). This outcome is counter-intuitive, as it may appear that we are adding increasing levels of latency to the function execution. However, the amount of latency is randomly chosen to be from a minimum of 100 ms to a maximum of 1, 2.5, or 5 sec. Using a
maximum random delay of 1 second may result in delays averaging 0.95 seconds but using a maximum random delay of 2.5 seconds could result in delays averaging 0.4 seconds. We see that the average request latency tends to saturate about two to three times the initial request latency, regardless of how frequently or how much maximum latency is added to function execution.

In some instances, injecting a chaotic delay result in a lower average request latency than adding no delay. For instance, adding random delays between 100 ms and 5 sec (yellow bar) with probabilities of 5%, 10%, or 35% is better (request latency is lower) than adding no additional delay (blue bar) for each of these probabilities. Figure 35 shows that, the Select Lambda fails in around 5% of all invocations. These failures may have a net positive impact on the request latency of successful requests. Herein lies the opportunity in chaos: function failure may be better for overall application performance than chaotic delay-injection, as applications can re-launch failed requests quickly.

The last test we perform with Select function is to scale the concurrent client request from 50 to 1000 with constant 25% of probability and random latency injections between 100 ms and 1sec, 2.5 sec, and 5 sec. For concurrency levels 50 and 100, the random delay between 100 ms and 2.5 sec has a lower (request latency is high) request response rate than the >= 5 sec latency injection, as shown in Figure 36. Also for concurrency levels 200 and 500, random delay >=2.5

Figure 34 Average Request Latency with Probabilistically Injected Delays At 1000 Concurrency Level (Select Func.)  
Figure 35 Requests failed under different probabilities for “Select” function
has a lower average response rate (0.36 sec) than the other two latency injections, and it is better (lower average time) than the no-addition latency for 500 concurrent clients. We observe that as client load increases, the trend of request latency does not necessarily correspond to the trend of response bandwidth (number of responses per second). As one example of this behaviour, the average response time of all the random latency injection are better (lower response time) for concurrency levels of 1000 simultaneous clients. However, for concurrency levels of 200 to 500 average response rate is lower which is average 3.2 seconds.

**Early abort functions for HelloWorld, Select, Insert and S3Trigger**

We explore the potential for chaotic function failure to improve overall application performance, by optimistically returning an error (HTTP status code 415) at the start of the Lambda function with a given probability. We induce failure with a probability of 5%, 10%, 15%, 25%, and 35%, to determine the impact of this early failure on average request latency. In each such experiment, we launch a total of 10000 requests to the Lambda function. These results are reported in Figure 37. We notice that across the functions, the average request latency tends to decrease as the probability of failure increases. This effect is particularly pronounced at the extremes, in the case of 5% failure rate versus 35% failure rate, at concurrency levels at-or-exceeding 500 simultaneous client requests. For every function, the greatest opportunity to exploit probabilistic failure occurs under the heaviest load, as 1000 simultaneous clients are making requests. This is true across functions, and is notable in the case of S3Trigger, which typically experiences an
average 20 second request latency. In the case of concurrency levels at and above 200, the S3Trigger request latency drops from the usual 20 seconds to 16.6, 14, and 12.7 seconds at 200, 500, and 1000 simultaneous requests, respectively as the failure rate increases from 5% to 35%.

![Figure 37 Average Latency with Probabilistic Failure for early-abort functions](image)

For the previous experiment on the early abort functions, we draw the bar graph for number of requests which returning the 415 HTTP (early abort) requests for each probability is shown in Figure 38. This graph depicts 500 HTTP for server errors, 200 HTTP for successful requests, and 415 HTTP for early abort requests. It is clear from this graph that as the probability increases from 5% to 35%, the occurrence of 415 (early abort error) also increases. Also, we observe that as the rate of 415 (early abort) rises, the rate of success falls by the same rate. This is a highly ideal situation, in which a function fails very quickly, and can be re-launched and hence succeed with little overall penalty to the application. If the function was to fail later, the average request latency might reflect the additional delay until eventual failure.
Although we are looking for opportunity in the most ideal circumstance, this opportunity has practical application. We anticipate that if we could design cloud functions which abort early as soon as any signs of eventual failure are detected, or eventual failure is predicted for all functions in this study, their performance and resilience would also improve. For example, S3Trigger function under heavy load, early failure can reduce the request latency from an average 20 seconds to 12.7 seconds, a 36.5% performance improvement.

Figure 38 Number of requests which early aborted (415) for all early abort functions
6. Conclusion

In this study, we constructed four functions: two for the DynamoDB interface, one for retrieving a file from an AWS S3 bucket, one which simply returns a string as a result. The DynamoDB functions outperform and are more resilient than the S3 function, which performs the worst of all functions and has the highest average response rate and lowest request throughput. The average cold-start and warm-start delay of each Lambda function was then measured. According to the results of our function, the difference between cold-start and warm-start latencies is large. We measured the average response of each function to determine the basic performance of our functions at various concurrency levels ranging from 50 to 1000.

Then we characterize the performance of serverless cloud operations under conditions of high client load, chaotic delays, and chaotic failure. We found a set of common operations that describe normal web-service activity. We have implemented these as serverless cloud functions in AWS Lambda and subjected these functions to a number of scenarios in which 10000 client requests are launched against them. These experiments vary the client concurrency level as well as the probability of injecting random delays or failures into the executing functions.

We find that the number of responses received per second does not trend with the average request latency as client load increases. We further demonstrate that the injection of chaotic delays tends to have a bounded impact on average request latency, regardless of how much delay is injected as well as its probability. Finally, we explore the impact of probabilistic early failure in the system and determine that for our longest-running function (S3Trigger) under heavy load, early failure can reduce the request latency from an average 20 seconds to 12.7 seconds, a 36.5% performance improvement.
Reference


