SERVERLESS COMPUTING:
APPLICATIONS, IMPLEMENTATION, AND PERFORMANCE

A Thesis

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Master of Science in Computer Science and Engineering

by

Garrett McGrath

Paul R. Brenner, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana

April 2017

SERVERLESS COMPUTING:

APPLICATIONS, IMPLEMENTATION, AND PERFORMANCE

Abstract

by

Garrett McGrath

Following the lead of AWS Lambda, services such as Azure Functions, Google Cloud Functions, Apache OpenWhisk, Iron.io IronFunctions, and OpenLambda have emerged as a new cloud offering coined "serverless computing", where application logic is split into functions and executed in response to events. In this work, I detail real world applications utilizing these platforms, and explore how existing applications can be adapted to run in serverless environments. Additionally, I present the design of a novel performance-oriented serverless computing platform implemented in .NET, deployed in Microsoft Azure, and utilizing Windows containers as function execution environments. Metrics are proposed to evaluate the execution performance of serverless platforms and conduct tests on the prototype as well as existing commercial platforms. These measurements show the prototype achieving greater throughput than other platforms at most concurrency levels, and I examine the scaling and instance expiration trends in the implementations.

To my family,

for your constant support and encouragement

of even, and especially, my sillier ideas.

CONTENTS

FIGURES

ACKNOWLEDGMENTS

I am grateful for the support of Notre Dame's Center for Research Computing and Department of Computer Science. This work builds upon previous publications, and I would like to thank Patrick Raycroft, Stephen Ennis, and Brenden Judson for their collaboration on those works, as well as Jared Short and Trek10 for sharing their experiences with large-scale serverless applications. Additionally, my experiences at Microsoft were invaluable in the creation of this work, as was the guidance of David Messner, Raj Rangarajan, Ryan Jansen, Samuel Banina, and Mark Overholt. I also appreciate the constructive feedback Dr. Douglas Thain and Dr. Peter Bui provided during the defence of this thesis. Finally, I am very thankful to Dr. Paul R. Brenner for his mentorship and advice over my years at Notre Dame.

CHAPTER 1

INTRODUCTION

Cloud computing has revolutionized application infrastructure technology over the past decade, causing a steady shift from bare-metal and traditional virtualization methods to cloud IaaS, where massive scale allows cloud providers to improve datacenter utilization and energy-efficiency. Large-scale cloud computing has served as an impetus for infrastructure advances, with containerization and microcontainers reducing virtualized environments to the absolute minimum required to run applications. Meanwhile, the rise of microservice software development paradigms have increased the granularity of applications, while allowing different services to scale independently. These trends are driven in part by the economic desires of cloud providers to improve utilization by only allocating as many resources as needed, when they are needed, and of cloud customers to decrease costs by only paying for the resources they need, when they need them [7].

Applications have remained largely unchanged through these iterations of virtualization and software development patterns, even while their systems and organization has shifted. However, for resource allocation to continue approaching actual resource usage, the way applications are developed and defined must be changed. Serverless computing attempts to solve this challenge by decomposing applications into a set of short-lived functions, which are executed in response to events. These events can be triggered from sources external to the cloud platform but also commonly occur internally between the cloud platform's service offerings, allowing developers to easily compose applications distributed across many services or data sets within a

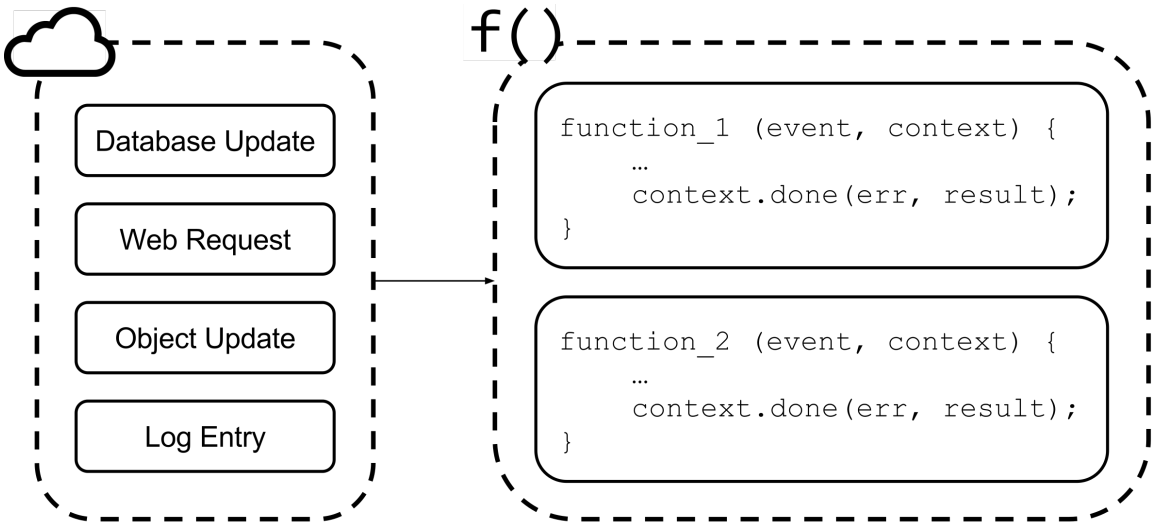

Figure 1.1. Overview of function abstraction, showing examples of event sources and basic function signature, where `event` contains the event state and `context` contains execution metadata and is used to signal function completion.

cloud. Figure 1.1 illustrates events that can trigger function execution, as well as a generic function signature. Function execution and scaling are entirely managed by the serverless platform, giving cloud providers control over resource allocation and function placement. Serverless computing fundamentally changes how applications are built and deployed, promoting functions, rather than applications, as the deployable, scalable unit of logic. Additionally, serverless platforms only bill customers for the time functions spend executing, and there are many reports of cost advantages to deploying microservices to serverless platforms rather than building out traditional applications [35, 37]. Functions are still more expensive than virtual machines per second, but the fined grained nature of invocation leads to cost savings and overall resource reductions. Others have tried to calculate the points at which serverless or virtual machine deployments become more cost effective [39].

Amazon Web Services can be credited with the creation of serverless computing

TABLE 1.1

OVERVIEW OF EXISTING SERVERLESS COMPUTING PLATFORMS

Serverless Platform	Environment	Open-Source	Deployment
AWS Lambda [1]	Linux Container	No	Commercial
Azure Functions [23]	Windows VM	Partially	Commercial
Google Cloud Functions [12]	Linux Container	No	Commercial
Apache/IBM OpenWhisk [33]	Linux Container	Yes	Both
Iron.io IronFunctions [18]	Linux Container	Yes	Deployable
OpenLambda [25]	Linux Container	Yes	Deployable

after the release of AWS Lambda in December 2014. Since then, Microsoft, Google, and IBM have released competing commercial cloud products, and multiple open-source efforts have emerged in the space. Table 1.1 shows a high-level overview of existing serverless offerings, including the type of function execution environment used, whether the platform is open-source, and whether the platform is used via a commercial offering or is privately deployable. Every serverless platform uses Linux containers as function execution environments except Azure Functions, which is an expansion of Azure's WebJobs offering, and uses Windows virtual machines as function hosts. This choice has a number of implications which are discussed in depth in the implementation chapter. The platforms offered by the largest cloud providers, Amazon, Microsoft, and Google, are closed-source, although the function execution runtime of Azure Functions is open-source. Apache OpenWhisk is fully open-source and was started by IBM, who maintains a commercial offering of the platform. Both Iron.io IronFunctions and OpenLambda are fully open-source projects without commercial offerings.

As cloud leaders continue to invest in the development of these serverless platforms

industry analysts are projecting increasing traction, with Gartner reporting that "the value of [serverless computing] has been clearly demonstrated, maps naturally to microservice software architecture, and is on a trajectory of increased growth and adoption" [20]. Forrester argues that "today's PaaS investments lead to serverless computing," viewing serverless computing as the next-generation of cloud service abstractions [14]. Given this growing momentum, it is important to consider the strengths of serverless computing and the challenges it faces in overcoming its current limitations and nascency.

This thesis studies serverless applications and the implementation and performance of serverless platforms. After an in-depth look at how serverless applications are constructed in AWS Lambda, I detail real world applications utilizing serverless platforms, discuss how serverless computing enables and/or limits these applications compared to IaaS technologies, and explore how existing applications can be adapted to run in serverless environments. These applications motivate new software design paradigms and highlight compelling use case scenarios and barriers to entry for serverless computing. Additionally, I present the design of a novel performance-oriented serverless computing platform implemented in .NET, deployed in Microsoft Azure, and utilizing Windows containers as function execution environments. Implementation challenges such as function scaling and container discovery, lifecycle, and reuse are discussed in detail. Metrics are proposed to evaluate the execution performance of serverless platforms and conduct tests on the prototype as well as AWS Lambda, Azure Functions, Google Cloud Functions, and IBM OpenWhisk. Measurements of these metrics show the prototype achieving greater throughput than other platforms at most concurrency levels, and I examine the scaling and instance expiration trends in the implementations. Finally, I analyze the limitations of the implementation, and discuss how they can be overcome using the current design.

CHAPTER 2

SERVERLESS APPLICATIONS

2.1 Introduction

This chapter focuses on real world applications initially architected or modified to leverage serverless computing in AWS Lambda. I note where the simple serverless function abstraction allows designs that scale dynamically with few dependency limitations. At the same time I document where serverless computing does not sufficiently allow for traditional design dependencies inherent in existing architectures. This leads to a discussion of how application programming paradigms which leverage serverless computing will necessarily differ from software designs which leverage more traditional infrastructure.

The application of serverless computing is an active area of development and research [21]. Other work has studied the architecture of scalable chatbots in serverless platforms [40]. There are multiple projects aimed at extending the functionality of existing serverless platforms. Lambdash [13] is a shim allowing the easy execution of shell commands in AWS Lambda containers, allowing developers to explore the Lambda runtime environment. Other efforts such as Apex [5] and Sparta [31] allow users to deploy functions to AWS Lambda in languages not supported natively, such as Go. More performance-oriented applications have emerged, such as a MapReduce-like word counter that rapidly parallelizes operations using AWS Lambda. [17].

Serverless computing has proved a good fit for IoT applications, intersecting with the edge/fog computing infrastructure conversation. There are ongoing efforts to

integrate serverless computing into a "hierarchy of datacenters" to empower the foreseen proliferation of IoT devices [9, 26]. AWS has recently joined this field with their Lambda@Edge [2] product, which allows application developers to place limited Lambda functions in edge nodes. AWS has been pursuing other expansions of serverless computing as well, including Greengrass [3], which provides a single programming model across IoT and Lambda functions. Serverless computing allows application developers to decompose large applications into small functions, allowing application components to scale individually, but this presents a new problem in the coherent management of a large array of functions. Additionally, AWS recently introduced Step Functions [4], which allows for easier organization and visualization of function interaction.

2.2 Applications in AWS Lambda

Serverless computing introduces a new paradigm for architecting cloud-based software and IT infrastructure solutions. Through the use of a simple function abstraction, software and systems engineers can reduce their software platforms into individual components (functions) with a greatly reduced set of hardware and software dependencies. When the functions are fully integrated with a robust cloud infrastructure, each function has access to a broad range of natively integrated services. Given sufficiently low overhead between the end user (client), serverless function calls, and subsequently called cloud services; modern applications can be written as series of largely independent function calls with minimal provisions in place to handle legacy infrastructure dependencies such as hardware resource limits, OS and library dependencies, and server/VM boundaries.

The following section studies these claims, but first I expand upon the concept of serverless applications and serverless architectures, using AWS Lambda as an example platform. Figure 2.1 illustrates the high-level concept underpinning AWS

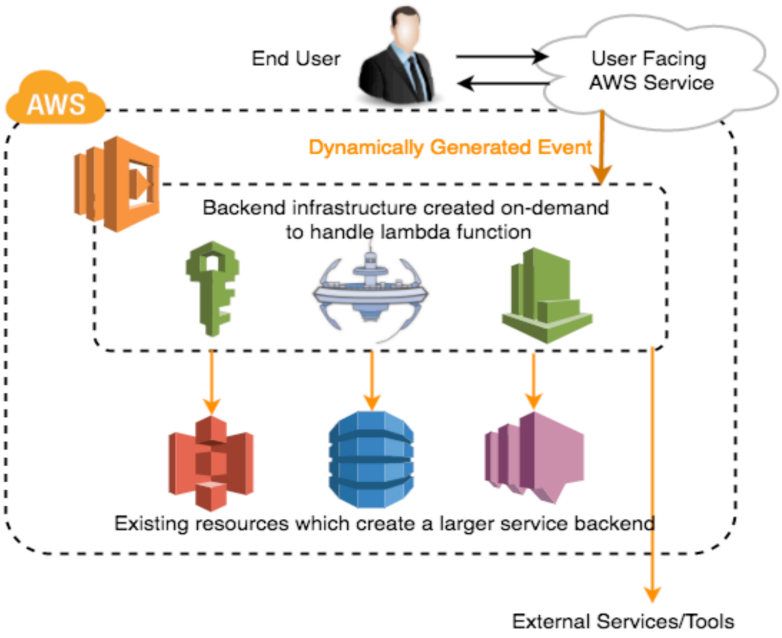

Figure 2.1. High-level application architecture in AWS Lambda

Lambda. An end user, which may be a software agent or physical user, interacts with an AWS service that is capable of invoking Lambda functions in response to the users action. This action could have involved uploading an image to S3, making an API request, querying a database or any tangible discrete action upon a Lambda-capable AWS resource. The details of the user action upon the AWS resource are recorded in a JSON object and passed as the event parameter to a lambda function. The developer only writes the function code to handle the event, as all infrastructural resources necessary to run the functions code are dynamically provisioned upon function invocation. Figure 2.1 shows the Lambda function internally utilizing IAM, ECS and CloudWatch. However, this is a thin view of the mechanics involved in dynamically creating a run-time infrastructure for the Lambda function. Once invoked, the Lambda function is free to interact with AWS services such as S3, DynamoDB, SNS (shown) or any service that is accessible through the web. AWS and other cloud providers' large portfolio of products considerably increases the utility of serverless

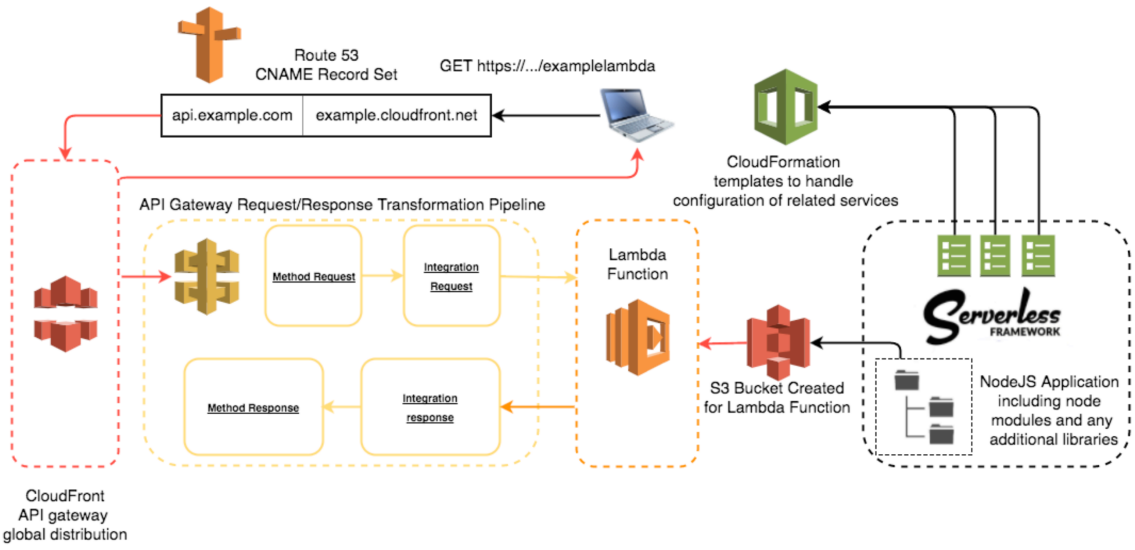

Figure 2.2. Detailed view of web application architecture in AWS Lambda

computing by allowing functions to run in response to numerous diverse events. In this sense it is not merely the solidification of lightweight virtualized environments that lead to serverless computing but also the diversification of public cloud provider product portfolios.

Figure 2.2 gives a more in-depth view of how a Lambda function may be used in a specific context; in this case handling an incoming API request. Importantly, this figure introduces the open-source Serverless Framework [30]. In basic terms, the Serverless Framework serves as an application framework and development/deployment tool for serverless computing, and currently supports AWS Lambda, Azure Functions, Google Cloud Functions, and IBM OpenWhisk.

In this example, an end user makes a API call to "api.example.com". The base of the request URL is resolved through Route53 to a global CloudFront distribution that is associated with the applications API, as defined in API Gateway. CloudFront passes the API request headers and body onto API Gateway. API Gateway applies authentication logic, modifications to the request, and any necessary format transfor-

mations before invoking a Lambda function and passing an event JSON representing the incoming request to the function. The function may then interact with a number of AWS based or external services to process and then respond to the incoming event. The Serverless Framework project, as illustrated by the directory structure in Figure 2.2, contains all code elements of the invoked Lambda function. The project is also free to provision and configure other component resources of the application through CloudFormation. Once the invoked Lambda function has handled the incoming request a response JSON is returned to API Gateway. This response proceeds through the outbound stage of API Gateways transformation pipeline before being served to the end user via the CloudFront distribution assigned to the API. Using this approach, independent functions may be written for each API method, and invoked independently.

At the enterprise level there are several benefits of serverless architectures to consider. Each Lambda function invocation is given dedicated, isolated, and independent resources at run-time, handling burst loads effortlessly. API Gateway also has considerable capacity to scale considering it use of AWS CDN, CloudFront, which is hardened to manage bursty and prolonged heavy load conditions. Continual development of Lambda functions is aided by built in versioning functionality in both API Gateway and Lambda. Finally, enterprise level applications may see considerable cost savings as AWS places small costs on API Gateway use and charges fractions of cents per Lambda function invocation.

2.3 Applications

2.3.1 Blogging in Node.js: Ghost

AWS Lambda, used in conjunction with API Gateway, is geared towards the creation of new microservices in the cloud. However, the extreme granularity of Lambda

functions and API Gateway resources makes managing the development and deployment of complex APIs untenable. The Serverless Framework aims to solve these issues, and provides organization and complex deployment of applications built on Lambda and API Gateway. Despite these improvements, Lambda, API Gateway, and the Serverless Framework by design only enable ways of writing new applications, rather than ways of deploying existing applications. Unlike advances from bare metal to virtual machines, or VMs to containers, serverless computing requires new paradigms of software construction, and is incompatible with the traditional tools and frameworks used in building web applications. This section motivates and explains an effort to bridge this gap, Lambdefy, which provides serverless applications deployment-time and run-time tools for executing traditional Node.js web applications in AWS Lambda for users who are unfamiliar with the complexities of API Gateway and Lambda.

Traditional web applications follow a familiar pattern of API construction during start-up, followed by API execution once a web server is listening for requests. Numerous frameworks exist to make this process as effortless as possible, and most involve listing method/route combinations and linking them to endpoint logic. These paradigms are problematic in a serverless context for multiple reasons. Lambdefy allows these conflicting paradigms to coexist, and is composed of two pieces that address different aspects of the serverless process. The first is a Serverless Framework plugin, which builds API Gateway and Lambda specifications at deployment-time and specifies how API Gateway processes input and output to/from Lambda. The second is a Node.js package that acts as the Lambda event handler for the web application, and translates between AWS Lambda events and web requests/responses at run-time.

Serverless functions are designed to perform specific tasks, and to collectively function as a web application, but a traditional application is not easily split into its

individual endpoints. Therefore, to generically deploy traditional web applications to Lambda, the entire application must remain together as a single Lambda function, which can easily be deployed by the Serverless Framework. API Gateway allows web requests to trigger Lambda function executions, to pass information about the web request to the triggered function, and to return a HTTP response. However, API Gateways design encourages very detailed description of an APIs resources and endpoints, which is a tedious and seemingly unnecessary requirement for any API developer, and especially for developers uninterested in low-level interactions with API Gateway. Ideally, API Gateway would extract the information from the incoming web request, pass it to the Lambda function, and return a response with a status code and headers as specified, without requiring specifics about the API beforehand. Lambdefys deployment-time logic generically achieves this goal, and is configured by Serverless settings files.

A traditional web application receives requests by listening to a port, while a serverless function receives information as function arguments. Furthermore, the arguments Lambda functions receive are structured differently than web requests, and Lambda functions must handle return values and error conditions instead of returning a HTTP response. Additionally, a traditional web application will not be able to process a Lambda invocation, because it is missing a function designated to handle the incoming events. The run-time layer of Lambdefy generates this handler given the port the application listens on, as well as information about the start-up timing of the application (synchronous vs. asynchronous) and the applications protocol (HTTP vs. HTTPS). When the Lambda function is triggered, this handling function will receive the event, generate a web request from the event information, send this request to the application, process the applications response, and return an object to API Gateway used to generate a web response to the client.

Ghost is a popular open-source blogging platform built on the Express framework

in Node.js [11]. This application was used to test the capabilities of the Lambdefy logic, and to experiment with the performance of a traditional web application hosted in AWS Lambda. Using such a heavyweight application was useful for exposing issues in Lambdefy during development, and in highlighting the fundamental differences and limitations of serverless computing in comparison to other deployment methods (ex. VMs).

2.3.2 High Performance Media Management System

Trek10, Managed Performance Architects for the Cloud; is a company of expert cloud engineers partnered with AWS to bring cloud based enterprise IT solutions. In partnership with University of Notre Dame scientists they regularly explore the most cutting edge new cloud technologies. Their engineers have recently implemented several real world solutions using serverless paradigms. In particular, Trek10 focuses on the use of AWS API Gateway and AWS Lambda backed by solution-specific supporting AWS services. This section discusses a real world business use case implemented by Trek10.

Large enterprises often consist of multiple internal entities, each operating with a degree of autonomy. Such enterprises often desire to create a media management service (MMS) to unify media asset upload, storage and distribution. Trek10s solution objective is to decrease storage and management costs, and increase and streamline the process of sharing media assets between enterprise entities. Presently, only image assets are being addressed by Trek10s solution. Future work intends to generalize the MMS solution to address video assets and other common asset formats.

A serverless approach was selected for the MMS use case due to potential scalability and cost saving improvements over traditional architectures. The MMS solution designed consists of three core features: Asset upload, asset distribution, and asset manipulation. The solution architecture is built upon Amazon API Gateway which

routes requests and responses to a back-end. The back end consists of AWS Lambda for event compute, AWS Simple Storage Service (S3) for object storage, AWS NoSQL database DynamoDB for maintaining state, and AWS CloudFront for asset distribution and low latency asset delivery.

The solution asset upload begins with a request to an API Gateway endpoint, associated with the MMS, which dispatches a presigned URL. The asset is then directly uploaded by the requester to S3 via the signed URL. This illustrates a key difference between traditional architectures and serverless architectures. In a traditional architecture, the server receives the upload and can handle any post-processing or metadata storage as a result of receiving the upload. As the uploads for MMS go directly to S3 for storage, AWS Lambda is necessary to signal some processing agent that there is work to be done. In this case, on a successful S3 upload, an event is dispatched to a Lambda function with sufficient information to enable the function to process of uploaded objects metadata. The processed metadata is then stored in DynamoDB for later querying and asset management.

Trek10s MMS solution required a method of managing the distribution of uploaded assets. Passing out an S3 object URL is a simple method of enabling asset distribution but is not always appropriate. Trek10s MMS use case outlined requirements for dynamic arbitrary resizing of images, expiration of assets links, and the grouping of assets into sets in order to enable simplified management. When requesting an image asset, an MMS consumer requests a signed CloudFront URL through a known asset set or asset id. An MMS consumer can be considered an entity within an enterprise which will ultimately serve assets to the enterprises end users. The signed URL allows the MMS to control the duration of time that the asset can be retrieved via that specific URL. The URL is then passed on, via the MMS consumer, to an end user through a browser or mobile app front-end. Through signed CloudFront URLs objects can be sourced directly from S3, thus avoiding the maintenance of any kind

of server. CloudFront enables high performance response times by caching assets at multiple CloudFront edge locations.

One of the most compelling aspects of this serverless-based approach is how a task such as image resizing can be handled. Trek10 set out to make the manipulation of entire asset sets a simple as possible. Resizing one to two image assets is a trivial task. For example, a basic solution could pull an image from S3, resize it and simply write the re-sized image back to S3. For a limited number of assets, one can run such an operation in a serial process. However, in the case of ten or more images, a quick response time may require a multi-threaded application. Scaling even further to a solution which can handle resizing requests for hundreds or potentially thousands of images typically demands the implementation of a queueing mechanism and some form of worker pool.

Using Lambda functions, Trek10s MMS solution can resize several hundred images in an embarrassingly parallel fashion. This approach results in approximately the same time cost for a single image re-sizing without the significant overheads of managing a queue/worker pool. This is enabled by a Lambda fan-out approach. Lambda fan-out takes advantage of the concept of instance based resources. When the resizing of an asset set is requested, the master request function invokes a worker lambda function for each image within the asset set. As discussed, one can consider each resizing operation as an independent isolated event with its own CPU, memory, and network resources. This serverless fan-out paradigm allows for performant results while reducing development and infrastructure cost.

2.4 Discussion and Results

2.4.1 Blogging in Node.js: Ghost

Lambdefy enables direct comparison of web application deployments to both VMs/containers and serverless computing. This comparison highlights the strengths and weaknesses of each deployment method, and can guide future development of the services themselves and the applications they host. The Ghost blogs deployment to AWS Lambda is a useful case study for analyzing how application development and structure should change to robustly perform in a serverless context.

The transience of the containers executing in Lambda provides many new challenges to application developers. Most notably, any application deployed to a serverless platform will be required to start-up frequently. This starkly contrasts more traditional environments, where applications start-up infrequently, and only as the deployment scales up with total application load. Therefore, start-up time becomes a key measure of application performance in a serverless environment, but in a traditional application this delay has little reason for optimization. In the case of Ghost, every time the application starts it must build its API routes and connect to the data layer, but it also performs many checks and even considers making database migrations before it starts its web server. Of course, Ghosts implementations seem very sane in a world of traditional deployments, but in a serverless application, this start-up time is incredibly costly. With serverless technologies disrupting cloud programming paradigms, applications that are agnostic of their deployments should begin to optimize start-up latency.

Serverless instance transience also forces completely stateless execution, both within code and within the file system. Any changes made to local state will only exist for a short time in a single execution instance, which forces separation between application logic and data layers. Additionally, due to the extreme horizontal scal-

ing capabilities of serverless computing, an applications data layer must be capable of performantly handling many concurrent and short-lived connections that may not close gracefully. For example, when experimenting with Ghost, the application used a standard MySQL image in AWS RDS as its data storage. However, because Lambda containers do not give warning on closure, connections to the RDS instance were ungracefully terminated, and quickly built up due to the long default connection timeout in the RDS instance. Soon, requests were failing due to maximum concurrent connections. In traditional deployments this problem would never exist, but in a serverless environment, the applications deployment forced changes (connection timeout value) in the data layer to accommodate the behavior of Lambda.

2.4.2 High Performance Media Management System

A core component of the serverless-based approach involves pushing workloads and operations to purpose specific services. For example, uploading assets is not readily handled by API Gateway and Lambda alone. Rather, uploading media assets is handled directly through an AWS S3 presigned URLs. AWS Lambda and API Gateway allow for the federation of application process specific services such as S3, DynamoDB and CloudFront. This separation of concerns is a primary benefit of serverless paradigms.

The solution architecture discussed was implemented in its entirety, along with additional functionality for asset lifecycle management using the Serverless Framework. As seen in Figure 2.3, the architecture has been shown to support 525 requests per second without any signs of performance degradation. Figure 2.4 compares the latency of cache hits and misses. The response latency for image resizing requests, for non-cached results, for a set of any size is on average 600-700ms from the time of request until assets are re-sized and asset URLs are dispatched. On a cache hit, 82.5% of requests complete in under 10 milliseconds. This performance is achieved

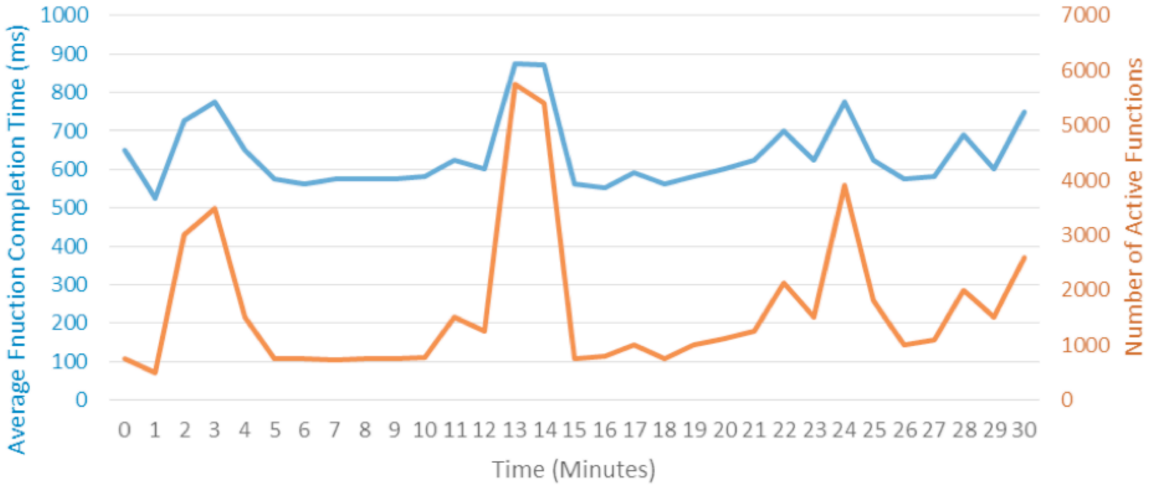

Figure 2.3. Average worker Lambda function completion time under varying load

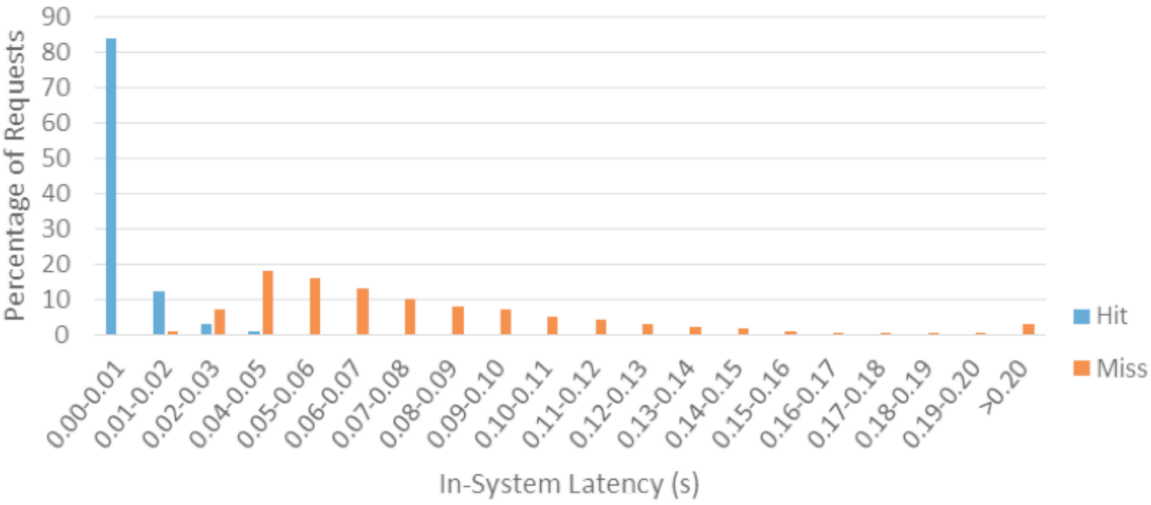

Figure 2.4. CloudFront in-system latency under 300 requests per second load

without the need to consider or manage scaling, system resource constraints, queuing, cluster management, and a multitude of further configuration and operational tasks. The adopted approach enabled the entire core system to be built by two engineers on a part time basis within two months, while such engineers attended to several unrelated business responsibilities.

These applications highlight the benefits serverless computing can have in specific problem domains, while arguing that effective serverless applications must consider different performance metrics than applications deployed to more traditional infrastructure. Moreover, the heavy reliance of these applications on other cloud services shows that effective use of serverless architectures requires a greater level of cloud competency and experience than deploying applications to IaaS offerings. The performance results of the MMS show that serverless applications can reliably and performantly parallelize under heavy load, and the ability of the fan-out pattern to rapidly scale and parallelize workloads motivates development of low-latency scaling in serverless platforms.

CHAPTER 3

SERVERLESS IMPLEMENTATION

3.1 Introduction

Serverless computing is a partial realization of an event-driven ideal, in which applications are defined by actions and the events that trigger them. This language is reminiscent of active database systems, and the event-driven literature has theorized for some time about general computing systems in which actions are processed reactively to event streams [34]. Serverless function platforms fully embrace these ideas, defining actions through simple function abstractions and building out event processing logic across their clouds. IBM strongly echoes these concepts in their OpenWhisk platform (now Apache OpenWhisk), in which functions are explicitly defined in terms of event, trigger, and action [6].

Beyond the event-driven foundation, design discussions shift toward container management and software development strategies used to leverage function-centric infrastructure. Iron.io uses Docker to store function containers in private registries, pulling and running the containers when execution is required [10]. Peer work on the OpenLambda platform presents an analysis of the scaling advantages of serverless computing, as well as a performance analysis of various container transitions [15]. Other performance analyses have studied the effect of language runtime and VPC impact on AWS Lambda start times [36], and measured the potential of AWS Lambda for embarrassingly parallel high performance scientific computing [19].

Serverless computing is quickly proliferating across many cloud providers, and is powering an increasing number of mobile and IoT applications. As its scope and

popularity expands, it is important to ensure the fundamental performance characteristics of serverless platforms are sound. This work hopes to aid in this effort by detailing the implementation of a new performance-focused serverless platform, and comparing its performance to existing offerings.

3.2 Prototype Design

I have developed a performance-oriented serverless computing platform[1] to study serverless implementation considerations and provide a baseline for existing platform comparison. The platform is implemented in .NET, deployed to Microsoft Azure, and has a small feature-set and simple design. The prototype depends upon Azure Storage for data persistence and its messaging layer. Besides Azure Storage services, the implementation consists of two components: a web service which exposes the platform's public REST API, and a worker service which manages and executes function containers. The web service discovers available workers through a messaging layer consisting of various Azure Storage queues. Function metadata is stored in Azure Storage tables, and function code is stored in Azure Storage blobs.

Figure 3.1 shows an overview of the platform's components. Azure Storage was chosen because it provides highly scalable and low-latency storage primitives through a simple API, aligning well with the goals of this implementation [8]. For the sake of brevity these storage entities will be referred to as queues, tables, and blobs, with the understanding that in the context of this paper these terms apply to the respective Azure Storage services.

[1] Available: `https://github.com/mgarrettm/serverless-prototype`

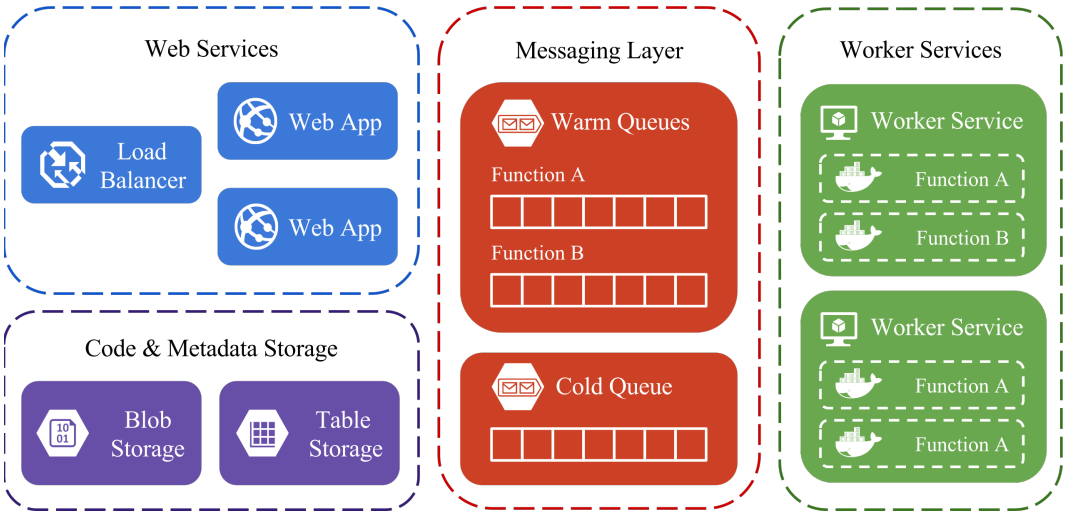

Figure 3.1: Overview of prototype components, showing the organization of the web and worker services, as well as code, metadata, and messaging entities in Azure Storage.

3.2.1 Function Metadata

A function is associated with a number of entities across the platform, including its metadata, code, running containers, and "warm queue". Function metadata is the source of truth for function existence and is defined by four fields:

1. Function Identifier - Function identifiers are randomly generated GUIDs assigned during function creation and used to uniquely identify and locate function resources.
2. Language Runtime - A function's language runtime specifies the language of the function's code. Only Node.js functions are currently supported, which was chosen because of its support across all commercial serverless computing platforms.
3. Memory Size - A function's memory size determines the maximum memory a function's container can consume. The maximum function memory size is currently set at 1 GB. The CPU cores assigned to a function's container is set proportionally to its memory size.
4. Code Blob URI - A zip archive containing a function's code is provided during function creation. This code is copied to a blob inside the platform's storage account, and the URI of that blob is placed in the function's metadata.

Function containers will be discussed in detail below, as will warm queues, which are queues indexed by function identifier which hold the available running containers

of each function.

3.2.2 Function Execution

This implementation provides a very basic function programming model and only supports manual invocations. While the processing of event sources and quality of programming constructs are important considerations in serverless offerings, our work focuses on the execution processing of such systems, for which manual execution support is sufficient.

Functions are executed by calling an "/invoke" route off of function resources on the REST API. The invocation call request bodies are provided to the functions as inputs, and the response bodies contains the function outputs. Execution begins in the web service which receives the invocation calls and subsequently retrieves function metadata from table storage. An execution request object is created containing the function metadata and inputs, and then the web service attempts to locate an available container in the worker service to process the execution request.

Interaction between the web and worker services is controlled through a shared messaging layer. Specifically, there is a global "cold queue", as well as a "warm queue" for each function in the platform. These queues hold available container messages, which consist of a URI containing the address of the worker instance and the name of the available container. Messages in the cold queue indicate a worker has unallocated memory in which it could start a container, and visible messages in a function's warm queue indicate existing function containers not currently handling execution requests.

The web service first tries to dequeue a message from a function's warm queue. If no messages are found, the web service dequeues a message from the cold queue, which will assign a new container to the function when sent to the worker service. If all workers are fully allocated with running containers, the cold queue will be empty.

Therefore, if the web service is unable to find an available container in both the warm queue and cold queue, it will return HTTP 503 Service Unavailable because there are no resources to fulfill the execution request. For this reason, the cold queue depth is a possible target for auto-scaling, as it reflects the free execution space across the platform.

Once a container allocation message is found in a queue, the web service sends an HTTP request to a worker service using the URI contained in the message. The worker then executes the function and returns the function outputs to the web service, which in turn responds to the invocation call.

3.2.3 Container Allocation

Each worker manages a pool of unallocated memory which it can assign to function containers. When memory is reserved, a container name is generated, which uniquely identifies a container and its memory reservation, and is embedded in the URI sent in container allocation messages. Therefore, each message in the queues is uniquely identifiable and can be associated with a specific memory reservation within a worker service instance. Memory is allocated conservatively, and worker services assume all functions will consume their allocated memory size.

When container allocations are sent to the cold queue, they have not yet been assigned to a function. To ensure workers do not over-provision their memory pool, it is assumed the assigned function will have the maximum function memory size. Then, when a worker service receives an execution request for an unassigned allocation, it reclaims memory if the assigned function requires less than the maximum size. After the container is created and its function executed for the first time, the container allocation message is placed in that function's warm queue.

3.2.4 Container Removal

There are two ways a container can be removed. Firstly, when a function is deleted, the web service deletes the function's warm queue, which is periodically monitored for existence by the worker service instances holding containers of that function. If a worker service detects that a deleted function queue, it removes that function's running containers and reclaims their memory reservations. Secondly, in the implementation a container can be removed if it is idle for an arbitrarily set period of 15 minutes, after which it is removed and its memory reclaimed. Whenever memory is reclaimed, worker services send new container allocations to the cold queue if their unused memory exceeds the maximum function memory size.

Container expiration has implications for the web service because it is possible to dequeue an expired container from a function's warm queue. In this case, when the web service sends the execution request, the worker service will return HTTP 404 Not Found. The web service will then delete the expired message from the queue and retry.

3.2.5 Container Image

The platform uses Docker to run Windows Nano Server containers and communicates with the Docker service through the Docker Engine API. The container image is built to include the function runtime (currently only Node.js v6.9.5) and an execution handler. Notably absent from the image is any function code. Custom containers are not built for each function in the platform, instead a read-only volume containing function code is attached when starting the container. A single-image design was chosen for multiple reasons: it is simpler to manage a single image, attaching volumes is a fast operation, and Windows Nano Server container images are significantly larger than lightweight Linux images such as Alpine Linux, which affects both storage costs and start-up times. In addition to the read-only volume, the memory size and CPU

percentage of the container are proportionally set based upon the function's memory size.

The container's execution handler is a simple Node.js server which receives function inputs from the worker service. The worker service sends function inputs to the handler in the request body of an HTTP request, the handler calls the function with the specified inputs, and responds to the worker service with the function outputs. The container is addressable on the worker service's LAN because containers are added to the default "nat" network, which is the Windows equivalent of the Linux container "bridge" network.

3.3 Performance Results

I designed two tests to measure the execution performance of the implementation, AWS Lambda, Azure Functions, Google Cloud Functions, and Apache OpenWhisk. I developed a performance tool[2] to conduct these experiments, which deploys a Node.js test function to the different services using the Serverless Framework [30]. I also built a Serverless Plugin[3] to enable Serverless Framework support for the platform.

This tool is deigned to measure the overhead introduced by the platforms using a simple test function which immediately completes execution and returns. This function is invoked synchronously with HTTP events/triggers as supported by the various platforms, and through the function's invocation route on our platform. Manual invocation calls were not used on the other services as they are typically viewed as development and testing routes, and I believed a popular production event/trigger such as an HTTP endpoint would better reflect existing platform performance. A 512MB function memory size was used in all platforms except Microsoft Azure, which

[2]Available: `https://github.com/mgarrettm/serverless-performance`

[3]Available: `https://github.com/mgarrettm/serverless-prototype-plugin`

dynamically discovers the memory requirements of functions.

The prototype was deployed in Microsoft Azure, where the web service was an API App in Azure App Service, and the worker service was two DS2_v2 virtual machines running Windows Server 2016. All platform tables, queues, and blobs resided in a single Azure storage account.

Network latencies were not accounted for in the tests, but to reduce their effects I performed the experiments from virtual machines inside the same region as the target function, except in the case of OpenWhisk, which was measured from Azure's South Central US region, and from which single-digit millisecond network latencies were observed to our function endpoint in IBM's US South region. Geographically similar datacenters often have low network latencies, even between different cloud providers [22].

3.3.1 Concurrency Test

Figure 3.2 shows the results of the concurrency test, which is designed to measure the ability of serverless platforms to performantly invoke a function at scale. The tool maintains invocation calls to the test function by reissuing each request immediately after receiving the response from the previous call. The test begins by maintaining a single invocation call in this way, and every 10 seconds adds an additional concurrent call, up to a maximum of 15 concurrent requests to the test function. The tool measures the number of responses received per second, which should increase with the level of concurrency. This test was repeated 10 times on each of the platforms.

The prototype demonstrates near-linear scaling between concurrency levels 1 and 14, but sees a significant performance drop at 15 concurrent requests. This drop is due to increased latencies observed from the warm queue, indicating that the load is approaching the scalability targets of a single Azure Storage queue [24]. AWS Lambda appears to scale linearly and exhibits the highest throughput of the commercial plat-

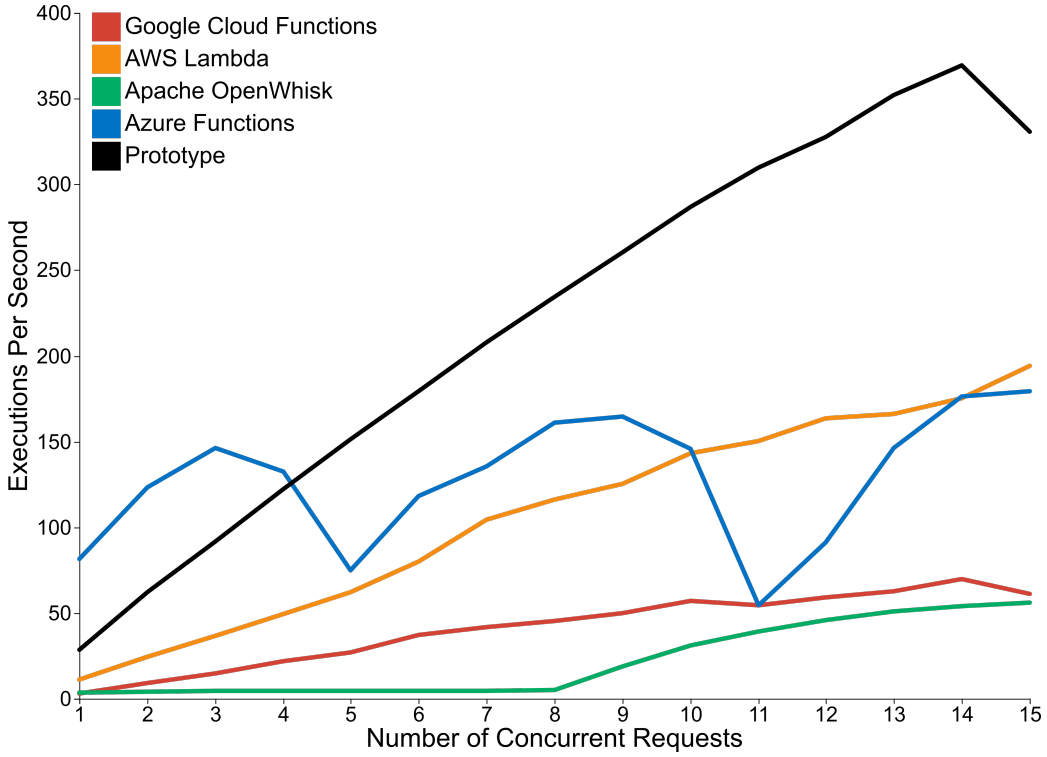

Figure 3.2. Concurrency test results, plotting the average number of executions completed per second versus the number of concurrent execution requests to the function.

forms at 15 concurrent requests. Google Cloud Functions exhibits sub-linear scaling and appears to taper off as the number of concurrent requests approaches 15. The performance of Azure Functions is extremely variable, although the throughput reported is quite high in places, outperforming the other platforms at lower concurrency levels. This variability is intriguing, especially because it persists across test iterations. OpenWhisk's performance is curious, and shows low throughput until eight concurrent requests, at which point the function begins to sub-linearly scale. This behavior may be caused by OpenWhisk's container pool starting multiple containers before beginning reuse, but this behavior is dependent on the configuration of IBM's deployment.

3.3.2 Backoff Test

Figure 3.3 shows the results of the backoff test, which is designed to study the cold start times and expiration behaviors of function instances in the various platforms. The backoff test sends single execution requests to the test function at increasing intervals, ranging from one to thirty minutes.

As described in the prototype design, function containers expire after 15 minutes of unuse. Figure 3.3 shows this behavior, and the execution latencies after 15 minutes show the cold start performance of the prototype. It appears Azure Functions also expires function resources after a few minutes, and exhibits similar cold start times as the prototype. It is important to note that although both the prototype and Azure Functions are Windows implementations, their function execution environments are very different, as the prototype uses Windows containers and Azure Functions runs on virtual machines in Azure App Service. OpenWhisk also appears to deallocate containers after about 10 minutes and has much lower cold start times than Azure Functions or the prototype. Most notably, AWS Lambda and Google Cloud Functions appear largely unaffected by function idling. Possible explanations

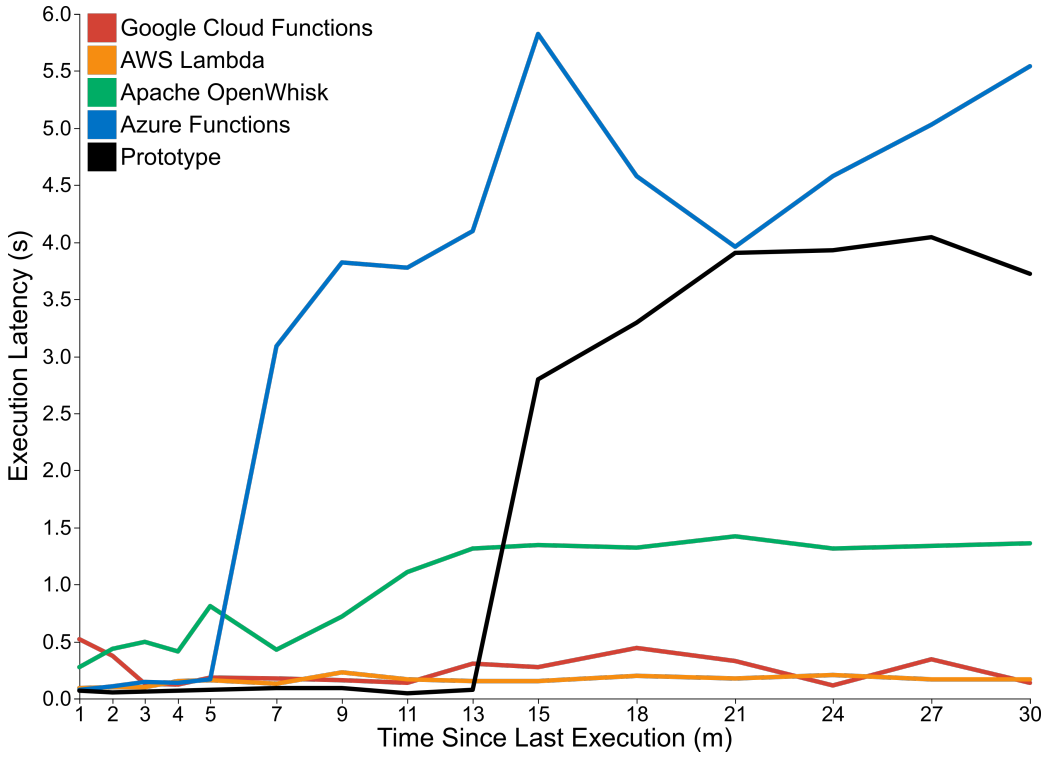

Figure 3.3. Backoff test results, plotting the average execution latency of the function versus the time since the function's previous execution.

for this behavior could be extremely fast container start times or preallocation of containers as considered below in the discussion of Windows containers.

3.4 Limitations

3.4.1 Function Immutability

Functions are immutable. In other words, only creation, deletion, and read operations are supported on function resources in the REST API. Immutability is a desirable attribute for the execution logic, but in reality it is useful to have many versions of a function and/or to update a function's existing code. This behavior could be supported by treating function versions as the objects being executed instead of functions as a whole, where functions could then be represented as a current version and a list of past version identifiers, and function versions could store the information currently contained in function metadata. Execution requests would either target a specific version identifier or the latest version by default, and because the latest version is included in the top-level function object, metadata fetching would still only require one table read. Therefore, although function updating would be a useful addition to the platform, supporting multiple function versions would not adversely affect execution performance.

3.4.2 Warm Queues

The fact that the warm queue is a FIFO queue is problematic for container expiration. For example, imagine that a function is under heavy load and has 10 containers allocated for execution. Then, load drops to the point that a single container would be able to handle all of the function's executions. Ideally, the extra 9 containers would expire after a short time, but because of the FIFO nature of the queue, so long as there are 10 executions of the function per container expiration period, all

10 containers will remain allocated to the function. The solution to this issue is of course to use "warm stacks" instead of "warm queues", but Azure Storage does not currently have support for a LIFO queue.

One possible solution is to use a Redis [27] cache to store the warm stacks, where the "list" data type supports LIFO queue operations out-of-the-box. However, Azure Storage and Redis are very different systems in both purpose and implementation. Azure Storage provides high availability and consistency guarantees, as well as message durability and multi-region replication. Conversely, Redis operations, especially when running on a Redis Cluster, are best-effort, and failure scenarios can lead to data loss, inconsistency, and temporary unavailability. Therefore, placing and relying upon data in Redis must be done cautiously by examining the persistence, consistence, and availability requirements of that data.

The argument for Redis is improved by observing that the prototype's warm containers are themselves a cache of function instances, used to avoid cold start performance on every execution. Indeed, the prototype could function without warm containers entirely, causing every execution to create a new container. Therefore, storing the existence of the warm instances in a Redis cache may be tenable, as infrequent loss of the instance data does not impact the correctness of the service, and will cause the forgotten instance to expire after a short time. Consistency is also negotiable, as the source of truth of instance status are the worker services themselves. If a Redis consistency failure occurs and a message is dequeued twice from the list, the worker service will return HTTP 409 Conflict to the second request for execution. The web service that received the duplicate message will then discard it and continue on to the next item in the list.

The availability of Redis data is a more important point, as the scenario where a function's warm stack is completely unavailable leads to cold start performance on every function execution. Redis Cluster [28] improves the availability of Redis

values by replicating data from a single master node across multiple slave nodes. If a Redis node becomes unavailable, data for which it was a master is promoted on corresponding slaves. Redis Clustering can lead to more complex failure scenarios that damage consistency, but as discussed above, this is tenable for the warm stack structures.

Given this analysis, consider a modified version of the prototype where the warm queues in Azure Storage are replaced by LIFO lists on a Redis instance. The web service finds an available warm instance by removing an item from the beginning of the list, and the worker service inserts an availability message once a container is free, also to the beginning of the list. All other storage artifacts including the cold queue remain in Azure Storage. Figure 3.4 shows the results of the concurrency test on this modified prototype, as well as the other results from the concurrency test. Redis is able to achieve improved latency compared to the original design, improving the total throughput of the prototype. However, further study is needed to determine if the Redis solution continues to perform well under failure scenarios. If Redis proves to not be a viable solution due to failure conditions, other solutions such as a custom consistent hashing [32] implementation are promising. But most importantly, the Redis solution behaves as a LIFO queue and allows for the correct expiration of function instances.

3.4.3 Asynchronous Executions

Currently the prototype only supports synchronous invocations. In other words, a request to execute a function will return the result of that function execution, it will not simply start the function and return. Asynchronous executions by themselves are simple to support, the web service can simply respond to the invocation call and then process the execution request normally. The difficulty in asynchronous execution is in guaranteeing at-least-once execution rather than best effort execution. It is

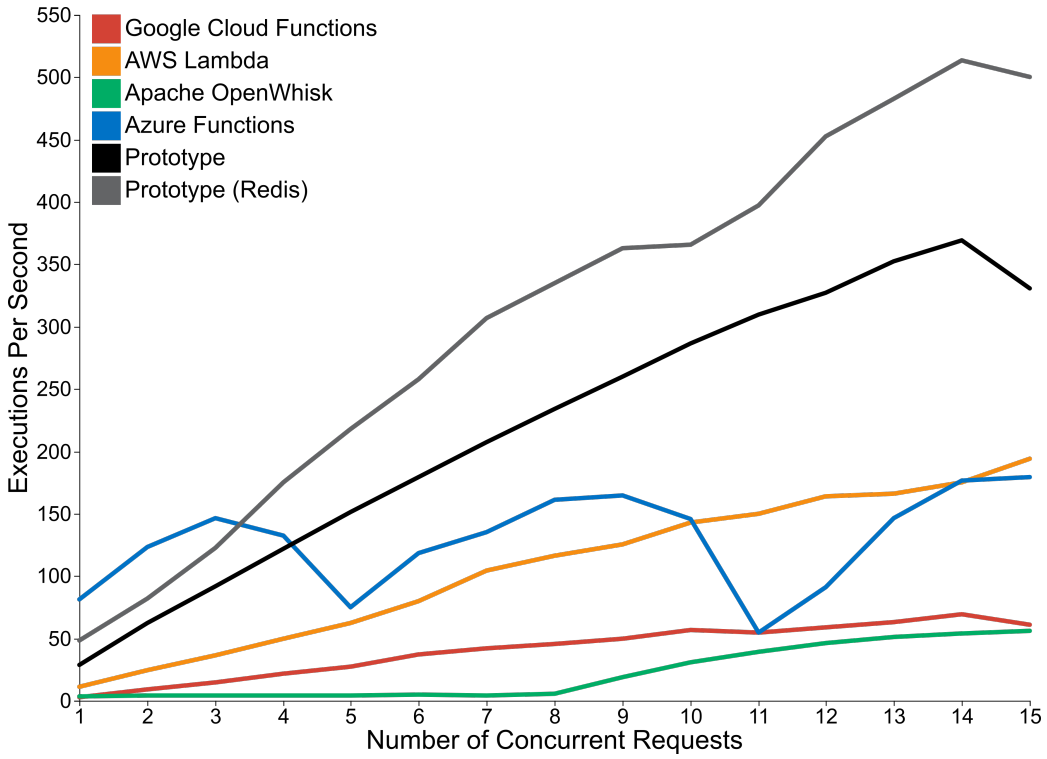

Figure 3.4. Concurrency test results for the modified Redis prototype, plotting the average number of executions completed per second versus the number of concurrent execution requests to the function.

important to understand that synchronous or asynchronous execution is only guaranteed once an invocation request returns with a successful status code. Therefore, no further work is needed for synchronous execution requests (as in the implementation), because a successful status code is returned only once execution has completed. However, asynchronous executions respond to clients before function execution, so it is necessary to have additional logic to ensure these executions complete successfully.

The prototype could support this requirement by storing active executions in a set of queues and introducing a third service responsible for monitoring the status of these queue messages. Worker services would continually update message visibility delays during function execution, and the monitoring service would detect failures by searching for visible messages. Failed messages could then be re-executed. Note that this is about handling platform execution failures and not exceptions thrown by the function during execution, for which retry may also be desired.

3.4.4 Worker Utilization

A large area for improvement in our implementation is worker utilization. Realistic designs would require an over-allocation of worker resources [16], with the observation that not all functions on a worker are constantly executing, or using all of their memory reservation. Utilization in a serverless context presents competing tradeoffs between execution performance and operating costs; however, the evaluation of utilization strategies is difficult without representative datasets of execution loads on serverless platforms. Future research would benefit from increased transparency from existing platforms, and from methods of synthesizing serverless computing loads.

3.4.5 Windows Containers

Windows containers have some limitations compared to Linux containers, largely because Linux containers were designed around Linux cgroups which support useful operations not available on Windows. Most notably in the context of serverless computing is the support of container resource updating and container pausing. A common pattern in serverless platform implementations is pausing containers when idle to prevent resource consumption, and then unpausing them before execution resumes [38, 15].

Another potentially useful operation is container resource updating. Because we reserve resources for containers before executions begin, it would be beneficial for cold start performance if we were able to start containers before they are assigned to a function, and then resize the container once an execution request is received. Future work can study how to support these semantics in Windows containers, perhaps by limiting or updating the resources to the function process itself rather than the container as a whole. Alternatively, the prototype could experiment with Linux containers to compare start-up performances and test the viability of container resizing during cold starts.

3.4.6 Security

Security of serverless systems is also an open research question. Hosting arbitrary user code in containers on multi-tenant systems is a dangerous proposition, and care must be taken when constructing and running function containers to prevent vulnerabilities. This intersection of remote procedure calls (RPC) and container security represents a significant real-world test of general container security. Therefore, although serverless platforms are able to carefully craft the function containers and restrict function permissions arbitrarily, increasing the chances of secure execution, further study is needed to assess the attack surface within function execution

environments.

Additionally, as with many cloud services the possibility of side-channel attacks must be considered [29]. This is especially dangerous when considering the Redis Cluster implementation, as Redis Cluster uses a known bucketing strategy to distribute elements. Therefore, an attacker could damage the performance of a function with a known identifier by generating significant load on a function with an identical CRC-16 hash. This can be mitigated by using securely encrypted function identifiers as Redis keys to eliminate the possibility of collisions, but is a good example of the kind of side-channel attacks cloud services must defend against. Similar precautions should be taken to ensure the placement of function instances cannot be exploited.

3.4.7 Performance Measures

There are significant opportunities to expand understanding of serverless platform performance by defining performance measures and tests thereof. This work focused on the overhead introduced by the platforms during single-function execution, but the quality of these measurements can be improved by better handling of network latencies and clocking considerations. Other aspects of platform performance such as latency variations between language runtimes and function code size, system-wide performance of serverless platforms, performance differences between event types, and CPU allocation scaling also warrant study.

CHAPTER 4

CONCLUSION

Serverless computing enables application developers to create powerful event-driven integrations within commercial clouds, while providing automatic horizontal scaling and a fine-grained billing model that closely fits application costs to resource usage. However, the newness of serverless computing presents many challenges related to application development, management, and tooling. Projects such as the Serverless Framework are easing the transition from developing traditional applications to serverless architectures, and I analyzed the abilities and limitations of deploying existing applications to serverless platforms.

Performance is important in many application domains, requiring serverless platforms to provide consistently low overheads during execution, both to encourage users to relinquish control of application scaling, and to open opportunities for higher-performance applications within serverless computing. Towards this end, I designed a novel serverless platform prototype, both to study techniques of improving serverless computing performance, and to serve as a baseline to compare existing platforms. Furthermore, I developed performance metrics to quantify function execution overheads, and gathered measurements from the prototype and existing serverless platforms. These performance results are encouraging, and present opportunities for continued development and study. I hope to see greater research interest in serverless computing, which could be fueled by an increased openness from industry leaders.

BIBLIOGRAPHY

1. Amazon Web Services. AWS Lambda. Available: `https://aws.amazon.com/lambda/`, 2017.

2. Amazon Web Services. AWS Lambda@Edge. Available: `http://docs.aws.amazon.com/lambda/latest/dg/lambda-edge.html`, 2017.

3. Amazon Web Services. AWS Greengrass. Available: `https://aws.amazon.com/greengrass/`, 2017.

4. Amazon Web Services. AWS Step Functions. Available: `https://aws.amazon.com/step-functions/`, 2017.

5. Apex. Apex: Serverless Architecture. Available: `http://apex.run/`, 2017.

6. I. Baldini, P. Castro, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, and P. Suter. Cloud-native, event-based programming for mobile applications. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, MOBILESoft '16, pages 287–288, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4178-3. doi: 10.1145/2897073.2897713.

7. O. A. Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. The resource-as-a-service (raas) cloud. In *4th USENIX Workshop on Hot Topics in Cloud Computing*. USENIX, June 2012. URL `https://www.usenix.org/conference/hotcloud12/resource-service-raas-cloud`.

8. B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043571.

9. E. d. Lara, C. S. Gomes, S. Langridge, S. H. Mortazavi, and M. Roodi. Poster abstract: Hierarchical serverless computing for the mobile edge. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 109–110, Oct 2016. doi: 10.1109/SEC.2016.37.

10. I. Dwyer. Serverless computing: Developer empowerment reaches new heights. Available: http://cdn2.hubspot.net/hubfs/553779/PDFs/Whitepaper_Serverless_Screen_Final_V2.pdf, 2016.

11. Ghost Foundation. Ghost: The professional publishing platform. Available: https://ghost.org/, 2017.

12. Google. Google Cloud Functions. Available: https://cloud.google.com/functions/, 2017.

13. E. Hammond. Lambdash: Run sh commands inside AWS Lambda environment. Available: https://github.com/alestic/lambdash, 2017.

14. J. S. Hammond, J. R. Rymer, C. Mines, R. Heffner, D. Bartoletti, C. Tajima, and R. Birrell. How To Capture The Benefits Of Microservice Design. Technical report, Forrester Research, May 2016.

15. S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Serverless computation with openlambda. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'16, pages 33–39, Berkeley, CA, USA, 2016. USENIX Association.

16. J. Heo, X. Zhu, P. Padala, and Z. Wang. Memory overbooking and dynamic control of xen virtual machines in consolidated environments. In *2009 IFIP/IEEE International Symposium on Integrated Network Management*, pages 630–637, June 2009. doi: 10.1109/INM.2009.5188871.

17. M. Holste. Building Scalable and Responsive Big Data Interfaces with AWS Lambda. Available: https://aws.amazon.com/blogs/big-data/building-scalable-and-responsive-big-data-interfaces-with-aws-lambda/, July 2015.

18. Iron.io. Iron.io IronFunctions. Available: http://open.iron.io/, 2017.

19. E. Jonas. Microservices and Teraflops. Available: http://ericjonas.com/pywren.html, 2016.

20. C. Lowery. Emerging Technology Analysis: Serverless Computing and Function Platform as a Service. Technical report, Gartner, September 2016.

21. G. McGrath, J. Short, S. Ennis, B. Judson, and P. Brenner. Cloud event programming paradigms: Applications and analysis. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 400–406, June 2016. doi: 10.1109/CLOUD.2016.0060.

22. M. G. McGrath, P. Raycroft, and P. R. Brenner. Intercloud networks performance analysis. In *2015 IEEE International Conference on Cloud Engineering*, pages 487–492, March 2015. doi: 10.1109/IC2E.2015.85.

23. Microsoft. Azure Functions. Available: https://azure.microsoft.com/en-us/services/functions/, 2017.

24. Microsoft. Azure Storage Scalability and Performance Targets. Available: https://docs.microsoft.com/en-us/azure/storage/storage-scalability-targets, March 2017.

25. OpenLambda. OpenLambda. Available: https://open-lambda.org/, 2017.

26. C. Pahl and B. Lee. Containers and clusters for edge cloud architectures – a technology review. In *2015 3rd International Conference on Future Internet of Things and Cloud*, pages 379–386, Aug 2015. doi: 10.1109/FiCloud.2015.35.

27. Redis. Redis. Available: https://redis.io/, 2017.

28. Redis. Redis Cluster Specification. Available: https://redis.io/topics/cluster-spec, 2017.

29. T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-894-0. doi: 10.1145/1653662.1653687. URL http://doi.acm.org/10.1145/1653662.1653687.

30. Serverless, Inc. Serverless Framework. Available: https://serverless.com/, 2017.

31. Sparta. Sparta: A Go framework for AWS Lambda microservices. Available: http://gosparta.io/, 2017.

32. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM. ISBN 1-58113-411-8. doi: 10.1145/383059.383071.

33. The Apache Software Foundation. Apache OpenWhisk. Available: http://openwhisk.org/, 2017.

34. K. uwe Schmidt, D. Anicic, and R. Sthmer. Event-driven reactivity: A survey and requirements analysis. In *In 3rd International Workshop on Semantic Business Process Management*, pages 72–86, 2008.

35. M. Villamizar, O. Garcs, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang. Infrastructure cost comparison of running web applications in the cloud using aws lambda

and monolithic and microservice architectures. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 179–182, May 2016. doi: 10.1109/CCGrid.2016.37.

36. R. Vojta. AWS journey: API Gateway & Lambda & VPC performance. Available: `https://robertvojta.com/aws-journey-api-gateway-lambda-vpc-performance-452c6932093b`, 2016.

37. B. Wagner and A. Sood. Economics of Resilient Cloud Services. *ArXiv e-prints*, July 2016.

38. T. Wagner. Understanding container reuse in AWS Lambda. Available: `https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/`, 2014.

39. A. Warzon. AWS Lambda pricing in context: A comparison to EC2. Available: `https://www.trek10.com/blog/lambda-cost/`, 2016.

40. M. Yan, P. Castro, P. Cheng, and V. Ishakian. Building a chatbot with serverless computing. In *Proceedings of the 1st International Workshop on Mashups of Things and APIs*, MOTA '16, pages 5:1–5:4, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4669-6. doi: 10.1145/3007203.3007217.

This document was prepared & typeset with pdfLaTeX*, and formatted with* NDdiss2$_\varepsilon$ *classfile (v3.2013[2013/04/16]) provided by Sameer Vijay and updated by Megan Patnott.*