Analysis of Micro Services and Serverless Architecture for Mobile Application Enablement

A thesis submitted in partial fulfillment of the requirements

For the degree of Master of Science in Software Engineering

By

Andrew Hoang

May 2017
The thesis of Andrew Hoang approved by:

_____________________________________
Professor Robert McIlhenny

__________________________
Date

_____________________________________
Professor Li Liu

__________________________
Date

_____________________________________
Professor Taehyung Wang, Chair

__________________________
Date

California State University, Northridge
Dedication

This work is dedicated to my mother and father whose wisdom, generosity, talent, and dedication are endlessly inspiring.
# Table of Contents

Signature Page ii  
Dedication iii  
List of Figures v  
List of Tables vi  
Abstract vii  

Chapter 1 Introduction 1  

Chapter 2 Microservices 4  
What is Microservices? 5  
A Microservice Reference Architecture 7  

Chapter 4 Amazon Web Services Lambda 16  
Restrictions and Limitations 16  
Testing Lambda Instances 19  
Test A 23  
Test B and C 26  
Test D 30  

Chapter 5 Implementation 33  
Next Page Android Application 33  

Conclusion 39  

Works Cited 41  

Appendix 43
List of Figures

Figure 1. Microservices Deployment Diagram 7
Figure 2. Microservices Component Diagram 9
Figure 3. Microservices Activity Diagram 10
Figure 4. An Example of Three Tier Architecture 11
Figure 5. An Example of Microservice Decomposition. 12
Figure 6. Amazon Web Services Lambda Rate Exceeded 18
Figure 7. Deployment Diagram of Tests 20
Figure 8. API Gateway Example 22
Figure 9. Test A- Quicksort – Node.js 23
Figure 10. Test A- Quicksort – Node.js 24
Figure 11. Test A- Database call – Node.js 24
Figure 12. Test A- Database call – Java 24
Figure 13. Test B- Database call – Node.js 27
Figure 14. Test B- Database call – Java 27
Figure 15. Test C- Database call – Node.js 28
Figure 16. Test C- Database call – Node.js 29
Figure 17. Test D- Java- Price vs. Performance 32
Figure 18. Next Page Use Case Diagram 33
Figure 19. Next Page Deployment Diagram 34
Figure 20. Next Page Login Screen 36
Figure 21. Next Page Book Selection Screen 37
Figure 22. Next Page Record Screen 38
List of Tables

Table 1. Resource Limits for Lambda 16
Table 2. Test A Summary 25
Table 3. Test B Summary 27
Table 4. Test C Summary 29
Table 5. Test D Summary 31
ABSTRACT

Analysis of Micro Services and Serverless Architecture for Mobile Application Enablement

By

Andrew Hoang

Master of Science in Software Engineering

The primary objective of this thesis is to analyze microservices and serverless architecture in the context of creating a back-end system to support a mobile application in order to assess the impact of new practices in software engineering. Microservices are generally defined as software components that have a single responsibility and are physically and logically loosely coupled from other software components, while serverless architecture refers to the design and implementation in a way such that components are only instantiated when necessary and deployed on third-party infrastructure. After distilling published works and industry expertise, the thesis will propose criteria for implementing microservice and serverless applications. Several tests will show the ability of Amazon Web Services Lambda, a Function as a Service platform, to deliver serverlessly designed microservices. Ultimately, the data gathered will highlight the drawbacks of using Java instances of Amazon Web Services Lambda as it compares to using Node.js instances. A demonstration implementation will involve the development of a mobile application combined with a serverless back-end to facilitate the distribution of audio, video, and e-books allowing parents to record themselves reading books for their children.
Chapter 1: Introduction

The march of progress has had a profound influence on uses for technology and the underlying systems that enable modern applications and experiences. A dramatic rate of change has advanced computing and has evolved how systems are created, maintained, and deployed. Central to developing modern applications is the idea of Service Oriented Architecture. An article by David Sprott and Lawrence Wilks of the CBDI forum defined Service Oriented Architecture as:

The policies, practices, frameworks that enable application functionality to be provided and consumed as sets of services published at a granularity relevant to the service consumer. Services can be invoked, published and discovered, and are abstracted away from the implementation using a single, standards-based form of interface (Sprott & Wilkes, 2004).

Sprot and Wilkes lay out several principals of Service Oriented Architecture, summarizing them as follows: standardized, reusable, abstracted, published, formal, and relevant. Systems that are designed in such a way are “essential to deliver the business agility and IT flexibility promised by Web Services” (Sprott & Wilkes, 2004).

A recent evolution of this concept is microservices. Martin Fowler and James Lewis suggest that the label microservices can be thought of as either a derivative of service oriented architecture or a new style altogether (Fowler & Lewis, 2014). Borrowing from the CBDI Service Oriented Architecture definition, microservices can be considered as a style that introduces new policies, practices, and frameworks to deliver a greater level of business agility and IT flexibility by utilizing advances in infrastructure technology via the internet. Literature on the subject defines microservices differently but there is some common ground. For instance, in the work Microservices: Yesterday, Today, and Tomorrow, the authors defined a microservice as “a cohesive, independent process that interacts via messages”
They also defined microservice architecture as “a distributed application in which all its modules are microservices.” Adrian Cockcroft, a software architect at Netflix, defined microservices as a service-oriented architecture composed of loosely coupled elements that have bounded contexts (Cockcroft, 2014). Tom Killalea, in his work titled The Hidden Dividends of Microservices, defined microservices as “an approach to building distributed systems in which services are exposed only through hardened APIs; the services themselves have a high degree of internal cohesion around a specific and well-bounded context or area of responsibility, and the coupling between them is loose” (Killalea, 2016).

The common elements between these definitions are: independence or loose coupling, distributed systems or applications, and bounded context or service responsibility. When using these elements an application should receive several benefits. Dragoni et al. list several, including scalability, ease of software maintenance, and resistance to failure. For example, if an application has separate independent components, resources can be allocated to meet the demand of a particular component. If a component fails, its failure should not affect other modules and their functionality. Because of the distributed, loosely coupled nature of the components, developers can make changes to one component without affecting other components.

However, implementing microservices architecture on a system is not without its drawbacks. Traditional development entails software built into monolithic applications where all components reside on one system (Richardson, Pattern: Monolithic Architecture, 2017). With microservices architecture, as large, monolithic applications with many components are decoupled into smaller microservices, there is an increase in the number of
components to manage and a larger increase in effort required to orchestrate these various components. Modules should be deployed independent of each other and therefore be on independent physical or virtual machines. Each machine will have a certain configuration and these configurations must be independently updated and maintained.

To minimize the complications that arise when using microservices, one can use the concept of serverless computing or serverless architecture. A goal of serverless computing, as described by Amazon Web Services, is “allowing you to focus on product innovation and get faster time-to-market” (Amazon Web Services, 2017). By using the serverless concept a system should require less effort to maintain and configure back-end systems because of the transfer of operational maintenance and coordination to third-party vendors.

This work has showcased a back-end system for an Android application using the Amazon Web Services Lambda platform and other web services. The premise of the application is an e-book application to allow parents to record themselves reading books to children. This application is called Next Page. Apart from the functional prototype, there is an exploration of Amazon Web Services Lambda and its various options, concluding with an analysis and comparison of different deployments of serverless functions. The ultimate goal is to have evaluated the sufficiency of microservices deployed via serverless architecture for the enablement of mobile applications.
Chapter 2: Microservices

The idea of microservices as a unique architectural style is a relatively recent invention. Martin Fowler explains that the concept was crystalized in February 2011 at an architectural workshop in Venice, California. The participants identified and acknowledged a common architectural style that they were all exploring, “an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.” By May 2012 the group decided on the term “micro services” as the most appropriate name (Fowler, Microservices a definition of this new architectural term, 2017). Although a new trend, the idea that microservices represents has a long history, but only has recently been enabled by technology innovations to become what is known today and the topic of this thesis.

Historically the microservice approach can trace its development from service-oriented architecture otherwise known as SOA. SOA is an architecture that aims to deconstruct monolithic applications into individual services (Dragoni, et al., 2016, p. 2). The Service Oriented Architecture manifesto working group’s published manifesto states that the goal of SOA is “applying service orientation to help organizations consistently deliver sustainable business value, with increased agility and cost effectiveness, in line with changing business needs” (SOA-Manifesto.org, 2017). To achieve these benefits of architecting in a service oriented manner should result in a separation of concerns between different systems, such that individual services can be optimized to serve users and other systems better while negating the effects of errors or limitations of systems onto other non-related, but otherwise functional systems.
What is a Microservice?

Microservice architecture is defined broadly by several characteristics. Dragoni et al., define a microservice architected system application as being composed of semi-autonomous loosely coupled components or services (Dragoni, et al., 2016). For example, in an application two or more software components would be independent from one another. Failures or changes in one component would not necessarily have any effects on the other.

In terms of deployment, a system designed with microservices will reside on virtual or physical hardware that is independent from other components (Dragoni, et al., 2016). For example, software components could exist in the same data center but are not run on the same machine. This would further insulate one component from another’s deployment issues.

The connection of the various components is done through the network using messages or events, otherwise known as an interface. Richardson adds that inter-service and inter-process communication can abide by common standards, such as REST or Apache Thrift (Richardson, Patter: Remote Procedure Invocation, 2017). For example, a system could use JavaScript Object Notation, or JSON, to provide a communication interface that would be widely understood by other developers.

The individual microservices within an application should also be defined around a particular area of responsibility. Richardson describes different ways to turn monolithic applications into microservices, such as decomposition by business capability or decomposition by sub-domain. Ultimately, the goal of those activities, and subsequently a microservice, is to achieve a core principal of object oriented design, namely the single responsibility principle (Richardson, Pattern: Microservice Architecture, 2017). Martin expresses the single responsibility principle as “A class should have only one reason to
change.” In other words, a software component should have one underlying reason for its existence, and modifications around this reason should be the only catalyst for change. Applying that definition to microservices produces the idea that a microservice should have only one reason to change (Martin, 2003).  

An application deployed using microservices architecture has impacts on many non-functional requirements, including availability, reliability, maintainability, performance, security, and testability (Dragoni, et al., 2016, pp. 8,9). Security is a counter-point to the positive benefits of microservices, and is treated as a casualty of the distributed nature of microservices. Microservice architecture relies on distributed components that communicate with each other and third-parties through known protocols. Dragoni, et al. describe that “an additional challenge is to provide authentication mechanisms with third-party services and ensure that the sent data is stored securely” (Dragoni, et al., 2016, p. 9). By not having components default to close connectivity through secure connections, microservice architecture opens a system up to additional vulnerabilities. Moreover, implementing microservices is not without its complications. For example, the distributed nature of services across multiple physical and virtual machines increases the complexity of trying to match users with dynamically assigned and dynamically changing services (Richardson, Pattern: Client-side Discovery, 2017). To reap real benefits of microservices, there is an implication that new components and systems must be

---

1 Microservice architecture also implies certain development practices, namely an organization whose developers are organized around the individual micro services they construct. The architecture also typically entails development with continuous integration, continuous delivery, infrastructure as code and other modern development practices to minimize service interruptions (Dragoni, et al., 2016, p. 7). The nature of this project is not organizationally focused and therefore this will not be a focus of study.
developed and maintained to orchestrate many services, the communication between them, and work that they do. One such supporting system that addresses these issues is a service registry, which is a system designed to track services, their instances, and locations (Richardson, Pattern: Service Registry, 2017). This thesis utilizes Amazon Web Services API gateway as an API gateway and as a service registry in the implementation of the Next Page Mobile Application.

**A Microservice Reference Architecture**

This work proposes reference architecture that fulfills the criteria of a microservice. The methodology we will use will be the United Modeling Language, or UML, in order to have a plainly understood and implementable architecture. This section will describe the reference architecture by presenting various diagrams followed by additional commentary and explanations.

![Microservices Deployment Diagram]

*Figure 1. Microservices Deployment Diagram*
Client or front end components speak to an API Gateway Layer which services requests from the front end via HTTP using some object notation such as JSON. These requests will be events that are deposited into a service queue that holds the information until work is to be performed. Because the various components do not exist on a single system, any implementation must define the format of communication between the various components such that each component can be implemented successfully.

For requests to be routed correctly to live services, a service registry should be created that maintains a list of live services instantiated by an automated virtual machine or container management service. In this particular reference architecture, an API gateway is used to coordinate services. The virtual machines or containers pictured in the diagram are managed in a way such that additional instances can be created or destroyed based on the amount of work to be done in a service queue to maintain a sufficient level of responsiveness to client devices or components. This can be done using Amazon Web Services Elastic Beanstalk and configuring auto-scaling within that service.

To reinforce loose coupling, each service should have its own database such that failures in one service will allow the system to continue to function. In this way, the database would not be a single point of failure for the entire system. A database failure would only affect the microservice it belongs to.

An additional orchestration tool that may be implemented is an event store. An event store coordinates work between different services. As services are loosely coupled there should not be a direct integration from Service 1 to Service 2. However, if work must be done on Service 1 and then also on Service 2, there must be some intermediary approach to coordination. The approach of an event store is to have the even information tracked in a
separate place such that many services can see the state of an event and if there is work to be done (eventuate.io, 2017).

Ultimately, client requests are responded to by the service via HTTP and the system is on a constant ready state to accept process, and respond to events.

Figure 2. Microservices Component Diagram
In the above component diagram of a microservices implementation, there are various components that must be implemented in order for a microservices architecture to be fulfilled. In the more detailed service component diagram, there is an illustration of the various components that would comprise a service. Specifically, the reference architecture contains multiple sub-components for handling the service registry, communication to the database, communication to the event store, any calls to external APIs that may contribute to the service, and the primary business logic that fulfills the responsibility of the service.

Lastly, below is an activity diagram that explains at a high level the system flow of the microservice architecture. Client interaction is routed by the API gateway to a service. Once a service does work, it publishes to the event store for other services to do work. If there is no additional work to be done it finishes.

![Microservices Activity Diagram](image)

Figure 3. Microservices Activity Diagram
Chapter 3: Serverless Architecture

A complementary approach to microservices is serverless architecture or serverless computing. The serverless concept relies on cloud technologies to dedicate computing resources and other services only when necessary, doing away with the idea of dedicated physical or virtual machines to provide application services. An evolving definition provided by ThoughtWorks.com describes serverless architecture as an “approach that replaces long-running virtual machines with ephemeral compute power that comes into existence on request and disappears immediately after use” (ThoughtWorks.com, 2017). Mike Roberts, writing on martinfowler.com, describes two types of serverless architecture. The first type, an architecture that relies solely on services provided by third-parties, he titles Backend as a Service or BaaS. The second type is a system that is an application written by a developer using stateless computation and fully managed by a third-party (Roberts, 2017). This definition falls in line with the one provided by ThoughtWorks.com and is referred to as Functions as a Service or FaaS.

This thesis will explore the Functions as a Service definition of serverless architecture and computing. An example taken from Mike Roberts explains serverless architecture clearly:

Figure 4. An Example of Three Tier Architecture. Reprinted from https://martinfowler.com/articles/serverless.html, Copyright 2017. Reprinted with permission.
In a typical three-tier system there is a client layer, business logic layer, and a database layer. This is represented in the above example through a browser, the Pet Store Server, and a Database. In the discussion in the previous chapter, services in the Pet Store Server would be liberated from the Pet Store Server and separated into microservices, such that they would be loosely coupled and operate independently from one another, typically on physical or virtual machines. However, in a serverless deployment of that scenario, these virtual machines are done away with, replaced by functions deployed to the cloud that are only instantiated when invoked by the client.

Figure 5. An Example of Microservice Decomposition. Reprinted from https://martinfowler.com/articles/serverless.html. Copyright 2017. Reprinted with permission.

In the graphic above the actual services for the pet store are done by a third party such as the Authentication Service, or via the API Gateway and dependent functions. In a monolithic system, the authentication, purchase, and search components would rest on one
machine. In a microservice architected system, each component would be built separately and reside on independent systems. In a serverless system, the purchase and search components are further decomposed into independent functions hosted on a cloud service that instantiates the functions only when necessary. The authentication service is outsourced to a third-party party provider.

**Implementing Serverless Systems**

Serverless systems fit within the primary defining criteria of microservices. Serverless systems can still have a high degree of loose coupling. Utilizing functions as a service implies that services can exist independently of one another and can also elastically scale independent of other services. Implicitly this is because the functions do not exist unless invoked and functions are treated as a separate application by cloud providers like Amazon Web Services Lambda.

Serverless systems also meet the need for microservices to be distributed. By utilizing cloud computing, serverless functions are inherently as distributed as the cloud provider they use. For example, Amazon Web Services describe their infrastructure as:

“Built around Regions and Availability Zones (AZs). A Region is a physical location in the world where we have multiple AZs. AZs consist of one or more discrete data centers, each with redundant power, networking, and connectivity, housed in separate facilities…. Each Amazon Region is designed to be completely isolated from the other Amazon Regions… Each Availability Zone is designed as an independent failure zone” (Amazon Web Services, 2017).

By opting to move away from a data center implementation, serverless applications, utilizing a function as a service provider, such as Amazon Web Services Lambda, become physically distributed very easily.
Serverless systems, by utilizing Backend as a Service, can have the components of their architecture very clearly defined, just as in microservices. In the Pet Store example above, a third-party service is completely responsible for authentication. This satisfies the idea of single responsibility as put forth by the previous chapter’s discussion on microservices.

The primary benefits of serverless systems are cost control, elasticity, and operational simplicity. Cost control can be achieved because Functions as a Service only bill per the amount of processing power used. Functions are not charged for any stand-by capacity; therefore, lightly used functions have a cost advantage by not having to pay for resources when not used. Serverless systems obtain elasticity through the use of Function as a Service providers and the provider’s ability to invoke functions as much as necessary. Also, the same functions do not hold excess capacity because the resources they use are freed once a function is complete. Lastly, operational simplicity comes from the fact that all server maintenance and deployment are done virtually through a console; no physical server management is needed. It is important to note that these benefits primarily come from the use of a robust third-party to power the application and its services.

Constructing applications using serverless architecture has drawbacks. The services that power serverless systems and Function as a Service have some computational limits. With Amazon Web Services Lambda, there is an issue of concurrent executions shared across a particular account. At the time of this writing this limit is 600 concurrent executions; however, during the testing phase the limit was 200 concurrent executions. Because this limit is across all functions across the account it is possible for one function to effectively halt the processing of other functions, leading to a multi-service and multi-system collapse. This limit
can be raised, but as one begins to implement more and more functions, the severe limitations of global account restriction can have very negative consequences. Testing on publicly available environments results in a very difficult platform to experiment on.

Because serverless systems are cloud based in nature and must be implemented on a system like Lambda they are subject to vendor lock-in. The challenge of vendor lock-in is described by Justice Opara-Martins et al. as “situation where customers are dependent (i.e. locked-in) on a single cloud provider technology implementation and cannot easily move in the future to a different vendor without substantial costs, legal constraints, or technical incompatibilities” (Justice Opara-Martins, 2016). This is because a service like Amazon Lambda has a specific deployment platform and software development kit, or SDK, that must be used. One cannot simply lift and shift production serverless systems from one environment to another.

Another major drawback of serverless systems is start-up latency. Anecdotally, Roberts suggests serverless systems may have start up times “anywhere from 10 ms to 2 minutes.” He goes on to specify that users of a Java Virtual machine “occasionally see long response times (e.g. > 10 seconds) while the JVM is spun up” (Roberts, 2017). The article’s purpose is to give a brief description of serverless applications and the author asks future readers to continue to explore this rapidly changing technology, “Whether or not you think your app may have problems like this you should test with production-like load to see what performance you see. If your use case doesn’t work now you may want to try again in a few months’ time since this is a major area of development by FaaS vendors” (Roberts, 2017). With this background in mind, this thesis will investigate Functions as a Service and Amazon Web Services Lambda rigorously to understand the impacts of start-up latency.
Chapter 4: Amazon Web Services Lambda

As discussed previously Amazon Web Services Lambda is a leading Function as a Service provider. As an enabling provider of serverless systems the restrictions and limitations of this platform can have consequences on the deployment of a system.

Restrictions and Limitations

The architecture of Amazon Lambda does not necessitate a specified amount of system resources. As more requests come in, more resources are dedicated towards completing functions. There is an upper limit, however, due to explicit restrictions on the amount of computing a function can do. Amazon Lambda resource limits are described by Amazon as follows:

<table>
<thead>
<tr>
<th>Resource</th>
<th>Default Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ephemeral disk capacity (&quot;/tmp&quot; space)</td>
<td>512 MB</td>
</tr>
<tr>
<td>Number of file descriptors</td>
<td>1,024</td>
</tr>
<tr>
<td>Number of processes and threads (combined total)</td>
<td>1,024</td>
</tr>
<tr>
<td>Maximum execution duration per request</td>
<td>300 seconds</td>
</tr>
<tr>
<td>Invoke request body payload size (RequestResponse)</td>
<td>6 MB</td>
</tr>
<tr>
<td>Invoke request body payload size (Event)</td>
<td>128 K</td>
</tr>
<tr>
<td>Invoke response body payload size (RequestResponse)</td>
<td>6 MB</td>
</tr>
</tbody>
</table>


The above table is taken directly from the Amazon Lambda website and lists the restrictions on the size of the functions invoked by the Function as a Service. The most impactful limit for the purposes of this thesis is the “Maximum execution duration per request,” which limits how long an individual function may run before being automatically
terminated. Microservice architecture does not prescribe a function duration, but any implementation using Amazon Lambda must abide by its restrictions.

Another restriction, as discussed in the previous chapter, is that Amazon Web Services Lambda limits the number of concurrent executions an account may have. This is the total computing capacity that all functions of an account may execute at any one time. As of April 2017, Amazon Web Services Lambda has a default limit of 600 concurrent executions. In February and March of 2017 the default limit for concurrent executions was 200. The calculation for concurrent executions is below, followed by examples:

- Events (or requests) per second * function duration = Concurrent Executions.
  - 10 requests per second * Function duration of 10s = 100 Concurrent Executions.
  - 500 requests per second * Function duration of 4s = 2000 Concurrent Executions.

If concurrent executions are exceeded Amazon Lambda will return a function throttle error code 429. In a production application, there are several ways in which to handle throttle errors. Superficially one would be able to explicitly raise the concurrent execution limit such that more Lambda functions will be executable. The benefit of this is that the functions can expand or contract elastically within the bounds of the concurrent execution limit. The challenge of setting an arbitrarily high concurrent execution limit is the potential worst case scenario for extremely high costs associated with runaway executions. A limit that is too low will cause errors across all functions in one’s Amazon account and disrupt services to users.

This thesis has tested a low-limit scenario by creating an Amazon Web Services Lambda function to Quicksort an array 1 million times. As the time of this particular test the concurrent execution limit is 600. In order to maximize concurrent connection usage, the function was called 1000 times at a duration of approximately 90 seconds each. This would
be 9000 concurrent executions, far beyond the default account limits set forth by Amazon. While the array sorting occurs, a test was run on another function to simulate other services or development occurring. The result was an error of “Rate Exceeded” which did not allow the function to process. This is the exact error expected and representative of the danger of how Amazon Web Services Lambda functions across accounts regarding concurrent executions.

Figure 6. Amazon Web Services Lambda Rate Exceeded

Using Amazon Web Services Cloud Watch alarms and billing notifications, a system can be engineered to notify administrations of these situations but the damage would have already been done. Alternatively, one could pre-emptively design a system and create retry logic in the initiating components such that throttling errors are caught, the calling system executes an alternative path or retries the request to the Lambda function later.

Aside from explicit platform restrictions, other considerations shape how an application can be deployed on Amazon Web Services Lambda. A particular topic of concern is the idea of “cold start,” or functions being invoked for essentially the first time. A search on Amazon Web Services forums for the term “cold start” provides many developers
asking about their particular functions having very long durations. (Amazon Web Services, 2017).

Sean Work of Kissmetrics.com, a company focused on a marketing analytics company, states that “47% of consumers expect a web page to load in 2 seconds or less” and “40% of people abandon a website that takes more than 3 seconds to load” (Sean Work). These data points support the idea that users expect applications that behave quickly and create an interesting dilemma for Functions as a Service providers. By creating a system with serverless microservices there is a possibility for unacceptable user experiences due to change in response time dictated by the underlying systems is architected on. This thesis has constructed a series of tests comparing Function as a Service implementations of Node.js and Java in order to understand the impact of designing systems serverlessly.

**Testing Lambda Instances**

Architecture of AWS Lambda dictates that a container will be created when a function is called: “when AWS Lambda executes your Lambda function on your behalf, it takes care of provisioning and managing resources needed to run your Lambda function.” Amazon express that “it takes time to set up a container and do the necessary bootstrapping, which adds some latency each time the Lambda function is invoked” (Amazon Web Services, 2017). The outcome of this is an ambiguous performance metric that, as will be tested, has profound implications on applications. If a function is called repeatedly, a function may reuse that container, but in no way, is this a certainty. The exact implementation details are hidden from AWS Lambda users. Amazon states that “depending on various other factors, AWS Lambda may simply create a new container instead of reusing an existing container” (Amazon Web Services, 2017). Thus, the way in which AWS Lambda
is provided as a platform has a dramatic drawback since it does not allow resource
customization and there are few options outside of revisiting the code base to perform
optimizations.

To probe this issue, a series of tests has been constructed. The following deployment
diagram illustrates how the test components will be deployed and what services and
machines will be used:

![Deployment Diagram of Tests](image)

Figure 7. Deployment Diagram of Tests.

Apache JMeter is a testing program that is well known tool for testing web applications. In this work, JMeter will be used to systematically create HTTP requests to the
API gateway in order for the requests to be called by our Lambda function. JMeter has
several benefits over manual testing. JMeter works at the protocol level and is not a browser;
therefore, we can remove a layer of overhead that may obscure our test results. JMeter can run automated test plans, such that we can test multiple HTTP requests in a similar manner quickly. JMeter has visualization tools and retry functionalities to help troubleshoot test results (Apache JMeter, 2017). Lastly, JMeter can simulate load, something that the Amazon Lambda console test page does not allow.

Amazon Web Services API gateway is a service provided to enable the exposure of developed software to the internet (Amazon Web Services, 2017). These APIs provide an outward facing end-point for clients to contact in order to gain access to services the APIs represent. From an application perspective, this allows different front-end components to communicate to a single URL. If the underlying service of the exposed API changes, the dependent front-end components will not necessarily need to change. Back-end services are abstracted by this API layer and can therefore be changed or optimized by development teams without substantively affecting front-end components. The API gateway allows for a great degree of loose coupling between various front and back end components, fulfilling a tenant of microservices. In regards to Lambda, the API gateway allows Lambda functions to be exposed via HTTP. Other ways to access Lambda functions would be direct integrations in other AWS services, but testing Lambda performance and functionality through additional system layers will result in data sets that need additional analysis to decouple Lambda functions from other Amazon Web Services cloud systems.
This thesis has created a total of four Lambda functions created to facilitate the data collection. The functions will be in groups of two services. One service has used the Quicksort algorithm provided by Node.js and Java to sort an array of size 100. This is a lightweight, minimal example of work a real-world function may do. The other service was a database call to query user information from the Next Page Database. One of these functions is a production function currently deployed in the Next Page Android prototype discussed later in this thesis.

Based on previous experience with Lambda functions, the initial hypothesis is that a minimum of 6 minute intervals between function invocations will be sufficient to generate a “cold start” scenario in which containers are not reused, as per previous discussions. “Hot start” functions should reuse old containers and should generally be faster. Cold start Lambda functions will be significantly slower than “hot start” Lambda functions and the difference between these two modes may have implications on mobile applications attempting to utilize
serverless back-end systems. Cold start, for example, can be considered the worst-case scenario for performance of a given function, because of how the platform is constructed.

All tests are constructed as follows. Using JMeter, each function will be invoked via HTTP request to a production API gateway URL at intervals of at least six minutes. For a sorting function call, a proper response would be a sorted array. For a database function call, a proper response would be the result from the database request. Function performance was measured in milliseconds. This metric was tracked in Cloud Watch and recorded into a spreadsheet. Network latency was automatically generated by JMeter because the program simulates requests and responses; this less applicable results will not be used for analysis.

Test A

Test A methodology entailed 100 Requests at a minimum of 6 minute intervals. The test included Quicksort and Database call types in both Node.js and Java. Function duration was measured in milliseconds. Results are reported in Figures 9-12 and Table 2.

Figure 9. Test A – Quicksort – Node.js
Figure 10. Test A – Quicksort - Java

Figure 11 – Test A – Database Call – Node.js

Figure 12. Test A - Database Call – Java
<table>
<thead>
<tr>
<th>Test A</th>
<th>Setting (MB)</th>
<th>Requests</th>
<th>Mean (ms)</th>
<th>Median (ms)</th>
<th>Max (ms)</th>
<th>Min (ms)</th>
<th>Standard Deviation (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quicksort Node.js</td>
<td>128</td>
<td>100</td>
<td>8.84</td>
<td>3.17</td>
<td>56.99</td>
<td>.27</td>
<td>11.31</td>
</tr>
<tr>
<td>Quicksort Java</td>
<td>128</td>
<td>101</td>
<td>17.91</td>
<td>1.27</td>
<td>227.71</td>
<td>.41</td>
<td>44.09</td>
</tr>
<tr>
<td>Database call Node.js</td>
<td>512</td>
<td>100</td>
<td>8.51</td>
<td>4.96</td>
<td>39.68</td>
<td>1.03</td>
<td>8.27</td>
</tr>
<tr>
<td>Database call Java</td>
<td>512</td>
<td>99</td>
<td>347.73</td>
<td>57.84</td>
<td>7177.00</td>
<td>33.81</td>
<td>1295.88</td>
</tr>
</tbody>
</table>

Table 2. Test A Summary

On average, Quicksort and Database call functions using Node.js outperformed Java. The overall data gathered had a large standard deviation, representing that the data gathered was spread out over a wide range of values. Originally this work hypothesized that the Lambda functions would revert to a cold state after at least six minutes. This hypothesis was generated by considering anecdotal information while creating and testing said functions. A closer inspection of the data, however, revealed three significant departures from the data set.

On three occasions Database call Java computation time took over 7000 milliseconds, significantly higher than the statistical mean of 347.73 milliseconds. Upon analyzing the test set-up what these entries had in common is relatively long cool off periods between Lambda invocations that occurred haphazardly due to the testing environment. The testing environment allowed three of the test to take places hours later rather than immediately after the minimum six minutes elapsed. The environment also caused an issue with a single Database call using Java which was discounted from the dataset. The data leads to the conclusion that a function invocation approximately six minutes apart, is not long enough to experience “cold start” situations.
Test B and C

The results of Test A demonstrated that a new test regime must be established in order to ensure a cold start situation. The second series of tests, which was labeled Test B, was constructed to address this requirement. The nature of Lambda is such that one container is created if a function is invoked and then it is available for reuse. Exploiting this behavior, the new test attempted to create a situation where there are no containers available. If the extreme maximum results of Test A are indicative of cold start situations, primarily because of the long nature of their cool off period, then we can hypothesize that a cold invocation of the Lambda function should execute at the extreme maximum of Test A or lower. Therefore, Test B will attempt to invoke Lambda functions at the 7000 ms timeframe of the initial Lambda call. If the invocations are begun and are successful within that time frame, there is no possibility for container reuse as no containers will have completed their initial invocation. For the purposes of focusing only on realistic applications, Test B will only test database function calls. DynamoDB, the AWS service that the Next Page application uses, has an explicit user-set limit on the number of reads and writes in each time frame. This limit is shaped by the free resources available for development purposes. At the time of this testing, the limit is 19 concurrent reads and write and therefore Test B will be limited to 19 invocations. These invocations will occur in rapid succession.

Test B methodology entails 19 Requests at no minimum interval in between functions. The test functions was of a Database call type using Node.js and Java. Function duration was to be measured in milliseconds. Results are in Figures 13-14 and Table 3.
Figure 13. Test B – Database Call – Node.js

Figure 14. Test B – Database Call – Java

<table>
<thead>
<tr>
<th>Test B</th>
<th>Setting (MB)</th>
<th>Requests</th>
<th>Mean (ms)</th>
<th>Median (ms)</th>
<th>Max (ms)</th>
<th>Min (ms)</th>
<th>Standard Deviation (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database Call Node.js</td>
<td>512</td>
<td>19</td>
<td>55.66</td>
<td>57.09</td>
<td>76.87</td>
<td>31.87</td>
<td>12.28</td>
</tr>
<tr>
<td>Database Call Java</td>
<td>512</td>
<td>19</td>
<td>9229.56</td>
<td>8831.9</td>
<td>11651.78</td>
<td>1664.79</td>
<td>2236.91</td>
</tr>
</tbody>
</table>

Table 3. Test B Summary
Test B resulted in dramatically different results, for example a multitude of difference between Test B’s Java mean of 9229.56 ms and Test A’s Java mean of 347.73 ms. This is a clear indicator that Test A’s hypothesis of six minute intervals being sufficient for cold start situations is untrue. This also leads one to surmise that Test A’s results were much closer to hot containers ready to be re-used for subsequent Lambda invocations.

To see container re-use performance, a scenario that may occur for a consistently used function, additional data must be collected. Test C was then constructed to maximize the chance of container reuse for a particular Lambda function. Following the same procedure as Test A and Test B, JMeter will make HTTP requests to the API gateway which will in turn invoke the Lambda function. In order to attempt container reuse the interval will minimally be set to 1000ms and additionally, the testing tool will only have invocations occur once the preceding Lambda Function is complete.

Test C methodology entails 19 Requests at minimum interval of function completion plus 1000ms. It will test the Database call function type using Node.js and Java. Function duration will be measured in milliseconds. Results are in Figures 15-16 and Table 4.

![Figure 15. Test C – Database Call Node.js](image)
The results of test C are much more in-line with the theorized “hot” results of Test A. In general, Node.js significantly out performs Java. For mobile applications Test C results for Java instances are of an acceptable duration for users to wait based on previous discussions in this thesis. For Node.js it is the opinion of this thesis that all function durations are of an acceptable duration for mobile applications.
Test D

Another interesting development in regards to testing was with the different resource allocations available to Lambda functions. Lambda functions have a minimum resource allocation of 128 MB and a maximum of 1536 MB. The varying memory tiers also have a complementary increase in computing power. The specific details between each option are hidden from developers. In the initial exploratory tests Database calls using Java returned an out of memory error with the 128 MB setting. 128 MB was too little memory to run the Java function. This led subsequent tests to be used under the 512 MB setting. Because there are also substantial cost differences at various tiers, this thesis has included another test to compare mean durations and what the cost would be if scaled to 1 million requests. The goal being to conclude if there are cost optimizations that may occur between increased resource and lower process times.

By comparing billable increments by rounding up mean function durations Test D will compute the cost per 1 million requests in order to estimate a particular function’s cost relative to other functions. The test entails 19 Requests at minimum interval of the function’s completion plus 1000ms. It will test the Database call function type using Node.js and Java. Function duration will be measured in milliseconds and settings are to be changed after the requests complete.
<table>
<thead>
<tr>
<th>Test D</th>
<th>Setting (MB)</th>
<th>Mean (ms)</th>
<th>Billable Increment</th>
<th>Price per 100 ms</th>
<th>Cost per 1M Requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database call Node.js</td>
<td>512</td>
<td>1.9</td>
<td>100</td>
<td>$0.000000834</td>
<td>$0.84</td>
</tr>
<tr>
<td>Database call Java</td>
<td>512</td>
<td>143.90</td>
<td>200</td>
<td>$0.000000834</td>
<td>$1.67</td>
</tr>
<tr>
<td>Database call Node.js</td>
<td>1024</td>
<td>1.74</td>
<td>100</td>
<td>$0.000001667</td>
<td>$1.67</td>
</tr>
<tr>
<td>Database call Java</td>
<td>1024</td>
<td>85.30</td>
<td>100</td>
<td>$0.000001667</td>
<td>$1.67</td>
</tr>
<tr>
<td>Database call Node.js</td>
<td>1536</td>
<td>2.36</td>
<td>100</td>
<td>$0.000002501</td>
<td>$2.50</td>
</tr>
<tr>
<td>Database call Java</td>
<td>1536</td>
<td>68.71</td>
<td>100</td>
<td>$0.000002501</td>
<td>$2.50</td>
</tr>
</tbody>
</table>

Table 5. Test D Summary

The data show a few interesting facets of serverless systems and Amazon Lambda. The most distinguishing feature of the data set was that because billable increments are in 100 milliseconds the increased performance of Node.js in all but the 512 MB instance is negated from a cost perspective. In the 512 MB setting the cost different is half of the Java instance. For the 1024 MB and 1536 MB settings the cost is the same because, though faster, the function is still billed at the rate covered by 100 ms. The mean durations of Node.js instances are very close together, with the 1536 MB setting having a higher mean duration than the other tests, demonstrating that the increase in processing power and memory did not have an impact on performance improvement. The variation is likely to be caused by the shared computing space that the AWS Lambda uses. Regarding Java Invocations, 512 MB to 1024 MB saw a maintenance of price due to billable increments and a reduction of 40% in processing time. From 512 MB to 1536 MB there was a doubling of price with a 52% reduction in processing time.
In the chart above you can see that there is a clear price performance leader. This data point corresponds with the 1024 MB setting and occurs because of how AWS Lambda bills in 100 ms increments. Functions that have 1 ms or 99 ms durations are still billed at 100 ms. Consequently and if a function has a duration of 101 ms it is billed at 200 ms increments. Due to this factor, it is reasonable to conclude that deployment of serverless functions should be tested against a range of settings to maximize cost-performance as demonstrated as possible in Test D.
Chapter 5: Implementation

Next Page Android Application

This thesis also contains a real-world application for using the prescribed serverless microservice architecture for mobile devices. Mobile devices by their nature typically do not have the computing power of desktops and laptops, and the distributed nature of their use lends itself to using the web services to facilitate applications on mobile devices. Serverless architecture is particularly advantageous to mobile developers because it allows a simplification of operations. Developers can worry less about maintaining back-end services and focus more on delivering content and applications on user devices.

The purpose of this application is to allow users to read and record read-a-longs to other users from anywhere in the United States. To this end, it is necessary to create a back-end system capable of identifying users, associating them with content, and then distributing said content to an individual’s device. These functional requirements are summarized by the below use case diagram.

Figure 18. Next Page Use Case Diagram
The diagram describes a user of the Android application having access to four different use cases. Namely: Login, Choose a Book, Read a Book, Record a Read-a-Long. These use cases are also utilized by undisclosed back-end services.

In the appendix is a list of the major user stories that drove the application development.

The following deployment diagram presents a clearer view of the back-end services and overall system architecture.

![Next Page Deployment Diagram]

Figure 19. Next Page Deployment Diagram

The Next Page application communicates with the API gateway via HTTP and the AWS S3 service via the AWS Android SDK provided by Amazon. The application could have been implemented with calls to AWS S3 through the API gateway, but the process would be complex, and as developed the integration of Amazon Web Services S3 directly into the application is an example of a serverless Backend as a Service. The Function as a Service, serverless microservice, portion is represented by various AWS Lambda functions which query information from the DynamoDB database. By using the API gateway and creating individual Lambda functions to process information, the application is able to fulfill
may of the requirements of a microservice such as loose coupling between components, demand elasticity, and services that can be independently deployed and developed.

The architecture of Next Page allows for loose couplings between services. With API Gateway, the Android application is not hard coded to communicate to a particular service, rather the application points to a relevant URL. The API gateway acts as a conduit in which services can be reached, with a publically exposed address that can redirect requests to first- or third-party services. For example, the Next Page Android application calls the “/checkUser” API and is then redirected to a particular Lambda function. Services are so loosely coupled that even individual Lambda functions are independent from other Lambda functions. Lastly the work flow is easily modifiable. Through Amazon API Gateway, API’s can be customized to communicate to another Lambda function very quickly or even directly to a database. From a microservices approach this allows iterations and evolutions of a particular service that can be independent of other services and even independent of the current service in deployment.

Next Page, as constructed, is highly elastic. Using a serverless approach computing resources are only used when there is a user requesting them. After work is done the resources are immediately released and the system is not billed for maintaining capacity. Demand is addressed precisely when demand is generated. If the application is not as popular during a particular time frame computing resources scale down to meet this new metric.

In terms of independent deployment and development further work on Next Page can be done very quickly without having effects on the current applications. New Lambda functions can be created and linked if necessary to existing functions or APIs. Otherwise, new APIs can be created linking to new and existing Lambda functions or other third-party
services necessary to fulfill additional application requirements. Below are several screen captures of Next Page in action utilizing serverless microservices.

In order to facilitate authentication, the login screen queries information from a database through a serverless microservice accessible through the API gateway. The returned information allows the user to match the application with a specific set of resources. Because Java instances of Amazon Web Services Lambda can have very slow start-up times, as demonstrated in the previous chapter, Node.js was used to facilitate authentication.

![Next Page Login Screen](image)

Figure 20. Next Page Login Screen.

The list of books and cover art are hardcoded into the application. The design choice here was to make a menu simple enough to facilitate choice. By checking or unchecking the
record mode box the user will enter a path where they can record read-a-longs or watch read-a-longs. In new iterations it is possible to have logic to handle a dynamic library of books.

Figure 21. Next Page Book Select Screen

Below is an example of recording a read-a-long. The recording is done through the Camera API and others, of the Android SDK. Implementation using the Camera2 API was
halted due to lack of hardware support. The videos are saved to local storage on the Android device and then uploaded via Amazon Web Services SDK to Amazon Web Services S3. Read-a-longs are downloaded by using Amazon Web Services S3. There was a conscious design decision not to make additional database calls to store user information because Amazon Web Services S3 can mimic a folder system such that individual user files can be selected and downloaded. This is an example of using the concept of Back-end as a Service, one part of serverless architecture.

I Will Help You

This book belongs to

Figure 22. Next Page Record Screen
Chapter 6: Conclusion

This thesis has explored the concept of microservices and serverless architecture. From various authors and literature, this thesis has outlined specific criteria that best represent a system created with microservice and serverless architecture. The reference architecture and diagrams are one attempt to encapsulate a deployable application that can be used in the real world using today’s technology.

The concept of serverless architecture relies heavily upon the use of third-party systems to facilitate serverless applications. Both the Back-end as a Service and Function as a Service require interaction with a third-party in order to be faithful to serverless architecture. Understanding and outlining the restrictions and limitations of Amazon Web Services Lambda allowed this thesis to implement the Next Page application and delve further into how Amazon Web Services Lambda performs under specific scenarios applicable to mobile applications.

The performance test of Amazon Web Services Lambda in regards to powering a mobile application using serverless microservices is enlightening. Mike Roberts suggested that serverless architecture may present latency issues as discussed in previous chapters, but left things open-ended for practitioners of the serverless architectural style to investigate for themselves. The data gathered by this thesis has shown that the start-up time of Java Lambda functions is too long for interactive applications. Use of Java functions may lead to unacceptable user experience and therefore would be better suited for asynchronous functions. Behaviors like transaction processing, transcoding, and batch processing may fall into this category. Time-sensitive applications such as communication, login and
authentication, and content delivery are not suited for Java instances of Amazon Web Services Lambda.

For applications and durations that are acceptable, this thesis has shown that multiple variations of Lambda instances should be attempted to explore settings that have optimal price and performance. This is because Amazon Web Services Lambda bills usage based on 100 millisecond durations, which causes large price increases if a function duration goes one millisecond over a multiple of 100. A future series of tests could be to experiment with different resource allocations to see if any functions fall within acceptable durations. It is possible for Java instances to redeem themselves; however, this subject is outside the scope of this thesis.

In regards to the Next Page application, the performance tests have shown that the current application should not use a Java based serverless instance to facilitate login. The long start-up times in cold-start situations makes an extremely poor user experience. In this way, Next Page uses a Node.js instance to retrieve data from the database.

Mobile application can therefore make use of microservices and serverless architecture to enable the back-end systems that ultimately power a user’s experience on a mobile application.
Works Cited


Amazon Web Services. (2017, April 1). *Discussion Forums > Advanced Search*. Retrieved from AWS Developer Forums: https://forums.aws.amazon.com/search.jspa?mbtc=437dc7a220d7ff03aad9875b08e74591ddbdfa33a2ac2d6c83c2160c91f1e79&threadID=&q=cold+start&objID=f186&userid=&dateRange=all&numResults=15&rankBy=10001


Appendix

User Stores:
- As a user I want to navigate via a main menu that shows downloaded books.
  - User Interface
    - screen layout
    - main menu icons
  - implement Rudimentary UI
  - implement API for retrieving a book
    - Design class for eBook
    - design JSON to be passed to represent eBook
  - hook app into server via API calls
  - implement ability to store downloaded books on Android device
- As a user I want to see what books are available for me to download.
  - backend to serve current collection of books
  - backend for storing info about what books have been purchased
- As a story reader I want to open a book and read it without audio
  - UI for book reading
    - design layout for screen while reading
    - develop transitions between pages
    - transitions for opening and closing books
    - develop book icon
  - implement book model as an array of image
- As a story reader I want to open a book and read it with audio and video that has been previously recorded by me or another user.
  - retrieve audio and video files from the server and playback on device
  - develop playback UI
  - use Tablet to display recorded video during playback
- As a storyteller I want to record audio and video as I read a book.
  - TBD library for recording audio and video in Android
  - save audio and video files to device and on the server (services).
- As a story reader I want to choose between available recordings for a current book
- As a user I want to share audio or video recordings with other users
  - store audio and video recordings on the cloud associated with a certain book and user
- As a storyteller I want to playback a recording before saving it.