Evaluation of “Serverless” Application Programming Model
How and when to start Serverless

ALGIRDAS GRUMULDIS

Examiner
Mihhail Matskin

Academic adviser
Anne Håkansson
Abstract

Serverless is a fascinating trend in modern software development which consists of pay-as-you-go, autoscaling services. Promised reduction in operational and development costs attracts not only startups but also enterprise clients despite that serverless is a relatively fresh field where new patterns and services continue to emerge. Serverless started as independent services which solve specific problems (highly scalable storage and computing), and now it's become a paradigm shift how systems are built. This thesis addressed questions when and how to start with serverless by reviewing available literature, conducting interviews with IT professionals, analyzing available tools, identifying limitations of serverless architecture and providing checklist when serverless is applicable. The focus was on AWS serverless stack, but main findings are generic and hold for all serverless providers - serverless delivers what it promises, however, the devil is in the detail. Providers are continuously working to resolve limitations or building new services as solutions in order to make serverless the next phase of cloud evolution.

The thesis work has been carried out at Scania It AB, Södertälje.

Keywords

Serverless; cloud computing; AWS Lambda
Abstract


Avhandlingsarbetet har utförts på Scania It AB, Södertälje.
Acknowledgments

I would like to thank everyone who helped me get where I am today and shaped my journey. Especially I wish to express sincere gratitude to:

Regina Dalytė Šileikienė for revealing how to solve complex problems

Vilija Dabrišienė for showing the importance of seeking excellence

Dennis Borger for awakening interest in Distributed systems

Mihhail Matskin for patience and lessons in "it depends" and Web-Services

Gabija Karpauskaitė and Cosar Ghandeharioon for assisting in a timely fashion when it mattered the most

Malin Gustavsson, Rodrigo Cid and entire Scania IT team for kindness, friendliness, and support

and to my family, who sometimes disagree with my decisions but always support and never stop believing in my abilities.
# Table of Contents

1 Introduction ...................................................................................................................... 1  
1.1 Background .................................................................................................................. 1  
1.2 Problem ....................................................................................................................... 2  
1.3 Purpose ......................................................................................................................... 2  
1.4 Goal ............................................................................................................................... 2  
   1.4.1 Benefits ................................................................................................................... 2  
1.5 Methodology .................................................................................................................. 2  
1.6 Delimitations ................................................................................................................ 3  
1.7 Outline ........................................................................................................................ 3  

2 Serverless .......................................................................................................................... 4  
2.1 Definition ..................................................................................................................... 4  
2.2 Serverless Processing .................................................................................................. 5  
   2.2.1 Serverless Processing Model .................................................................................. 5  
   2.2.2 Container Lifetime ............................................................................................... 6  
2.3 AWS Serverless platform ............................................................................................ 7  

3 Deployment package size impact on cold-start ......................................................... 11  
3.1 Methodology ............................................................................................................... 11  
   3.1.1 Scenarios ............................................................................................................... 12  
3.2 Results ......................................................................................................................... 12  
   3.2.1 Different package and memory sizes, one dummy file .......................................... 12  
   3.2.2 Big package, lots of small files ............................................................................. 14  
   3.2.3 Big package, lots of small files, modules are loaded ............................................. 15  
3.3 Conclusions ................................................................................................................. 15  

4 Serverless Application Programming model ............................................................ 16  
4.1 Development ............................................................................................................... 16  
   4.1.1 Infrastructure management .................................................................................... 17  
   4.1.2 Deployment .......................................................................................................... 17  
   4.1.3 Local development ............................................................................................... 17  
4.2 Frameworks ................................................................................................................ 18  
   4.2.1 Chalice ................................................................................................................... 18  
   4.2.2 Apex ..................................................................................................................... 18  
   4.2.3 AWS SAM ............................................................................................................ 19  
   4.2.4 Up ......................................................................................................................... 20  
   4.2.5 The Serverless Framework (Serverless.com) ....................................................... 20  
   4.2.6 Summary ............................................................................................................. 22  
4.3 Architecture and design .............................................................................................. 23  
   4.3.1 Code patterns ...................................................................................................... 24  
   4.3.2 Files upload ......................................................................................................... 26  
   4.3.3 State management ............................................................................................... 26  
   4.3.4 Emulation of server-client communication ......................................................... 28  
4.4 Security ........................................................................................................................ 29  
4.5 Observability ............................................................................................................... 30  
4.6 Optimizations ............................................................................................................. 31  
4.7 Serverless applicability .............................................................................................. 32  
   4.7.1 Use cases ............................................................................................................. 33  
   4.7.2 Characteristics of serverless use-cases ............................................................... 35  
   4.7.3 Applicability checklist ......................................................................................... 37
List of Abbreviations

API – Application Programming Interface
AWS – Amazon Web Services
DDD – Domain-Driven Design
CDN – Content Delivery Network
CLI – Command-line Interface
FaaS – Function as a Service
IoC – Inversion of Control
VPC – Virtual Private Cloud

1 Introduction

An exciting trend emerges in cloud application development named serverless computing. It enables to deploy an application without worrying about infrastructure. The cloud provider takes care to load and scale the application when some specific event occurs and then unloads if not used. The serverless model lets to pay only for the resources application actually consumes.

1.1 Background

Serverless computing or in short serverless affects not only how applications are run but also how they are built. The serverless application programming model is based on best practices of development in a cloud computing environment. Cloud computing began with services offering to rent infrastructure instead of buying it. However, managing virtual servers still requires a lot of labor of intensive work, but that work can be automatized. Automatization was the primary driving force of DevOps culture. DevOps people write scripts to deploy, provision, scale, and monitor virtual servers and applications. Those scripts become tools like puppet and chef. Afterward platforms such as Heroku and kubernetes emerged. DevOps experience, knowledge, and best practices become part of technology, and the latest product of that is serverless.

Scania IT is interested in serverless because this new architecture can simplify some types of systems and it can provide economic benefits by reducing operation and infrastructure costs. At the moment Scania owns and maintains 3 data centers across the world, in total there are about ~500 physical servers and ~5500 virtual servers. However, their business model is not to take care of servers, and the company’s strategy is to move from infrastructure on-premise to a cloud. Serverless Computing is the ideal solution because there is no management of server hosts or server processes.

This thesis describes Serverless application programming model, in other words, how applications can be built and hosted in Serverless environment. The primary focus of this thesis is on Amazon Web Services (AWS) because Scania IT has chosen AWS as their main cloud computing platform. It is worth to mention, that thesis is not about AWS Serverless Application Model (AWS SAM) which provides a standard for expressing Serverless applications on AWS. However, section 4.2 Frameworks includes an analysis of AWS SAM.
1.2 Problem
Serverless computing is a new cloud computing execution model, and the problems of it are still in an exploration phase. Organizations are not fully aware of models, benefits and challenges building applications with Serverless architecture. That leads to economic losses when serverless is chosen and it can’t meet requirements. However, more often, opportunity to reduce costs is missed due misunderstandings of serverless limitations, such as cold-start and stateless execution.

1.3 Purpose
The purpose of this thesis is to present serverless application programming model, describe the advantages and challenges from a software engineering, development productivity (both design, coding, testing, and debugging), performance, scalability and security perspectives. Presented material should help to understand when serverless architecture is the right choice and give the necessary knowledge to avoid common pitfalls when working with the serverless stack.

1.4 Goal
The goal of this project is to gather necessary knowledge which would help successfully use serverless computing. That includes:
1. Describe how serverless computing works.
2. Analyze deployment package size impact on cold-start
3. Compare serverless development frameworks.
4. Identify when serverless architecture is applicable.
5. Describe challenges and best practices of serverless application programming model.

1.4.1 Benefits
This research is beneficial to everyone who uses serverless computing services because it helps to understand better what serverless is, how it works and when it is the right choice to use.
Results of this project are especially beneficial for the developers. It provides necessary information to start working with serverless stack and gives instructions on how to determine if application requirements can be fulfilled using serverless architecture, thus it reduces learning-curve and wasted efforts due to missing knowledge about serverless.

1.5 Methodology
This thesis tries to provide the necessary knowledge needed to start working with Serverless stack, therefore, applied research method was chosen. Data were collected using literature study, two qualitative research methods: interviews and case study and quantitative research to analyze cold-start problem. The methodology of it is presented in Section 3.1.
An extensive literature study was needed to describe the Serverless Application Programming model because serverless is a new phenomenon which is driven by industry. Thus, available information is limited in order to protect trading
secrets and sometimes outdated because technology evolves rapidly. Literature study includes material not only from previous researches but also from whitepapers, conferences and blog posts due to freshness and completeness of information.

Multiple informal interviews were conducted with various people to learn how they understand and how they are using serverless and what are the impressions and concerns about it. Interviewees were developers, infrastructure engineers, software architects and managers at Scania IT, also solution architect from AWS.

The case study was carried out to evaluate the following serverless frameworks: Chalice, Apex, AWS SAM, Up and the Serverless Framework. To evaluate eight simple APIs with one endpoint was built in Java 8, C# (.NET Core 2.0), Go, Node.js (6.10, 8.10) and python3 languages and deployed to AWS. User experience was evaluated looking at how easy it is to set up a framework, prepare project and deploy the application. It was done using a computer with Windows 10.

Additionally, some observations are from personal experience gained during one year of developing REST API on AWS serverless stack.

1.6 Delimitations

Serverless field evolves quickly, new tools are released, and existing ones are improved constantly, therefore features of serverless frameworks described in section 4.2 Frameworks might be inaccurate at the time of reading. The same applies to AWS features and measures because AWS is constantly working to improve performance and user experience of the serverless platform and keeping details in private. Therefore, measures and descriptions how it works are only to provide an overall view about a serverless platform and give a general idea what to expect.

1.7 Outline

This report consists of three main parts. The first part is a general introduction to serverless, serverless computing and AWS serverless platform. It is presented in Chapter 2.

The second part, presented in Chapter 3, analyzes the impact of deployment package size on cold-start. That chapter describes experiments, presents results and discusses them.

Chapter 4 presents the third and the main of the project - Serverless Application Programming model. It covers topics about serverless application development, architecture, security, observability, optimizations and limitations.

Lastly, Chapter Error! Reference source not found. summarizes results of the project, provide some conclusions, and give suggestions for possible future work.
2 Serverless

The serverless is a new generation of platform-as-a-service which distinguishing feature is no maintenance of servers. Also, Serverless is driven by industry, AWS in particular, which is visible looking at popularity between “Serverless” and “AWS Lambda” search terms. Figure 1 depicts raise of “AWS Lambda” searches followed by “Serverless” in google.

![Figure 1 Popularity of Serverless and AWS Lambda search terms in google between 2014-2018-04](image)

2.1 Definition

There is still no stable definition what serverless is, however most widespread description what serverless means is provided by Mike Roberts:

1. No management of server systems or server applications
2. Horizontal scaling is automatic, elastic, and managed by the provider.
3. Costs based on precise usage
4. Performance capabilities defined in terms other than size or count of instances
5. Implicit high availability.

Serverless can be divided into two overlapping areas: Function as a Service and Backend as a Service (managed services).

Serverless Function as a Service, also known as Serverless computing, is a service which allows running application logic on stateless compute containers that are fully managed by a third party. Serverless computing is the type of Function as a Service, which is part of event-driven computing, Figure 2 depicts the relation.

![Figure 2 Relation between Serverless computing, Function as a Service and Event-driven computing](image)
Fintan Ryan, an industry analyst at RedMonk, described [1] serverless computing as:

- programming model to deal with event-driven architecture
- abstraction layer, which abstracts underlying infrastructure

Applications running on serverless computing most of the time relies on other provided services, which falls under Backend as a Service category. CNCF Serverless Working Group defines Backend as a Service (BaaS) as “third-party API-based services that replace core subsets of functionality in an application” [2].

To sum up, despite how it sounds servers are and continue to be necessary to run serverless services, the same as wires are needed for Wireless networks. The “less” just emphasize the idea that servers are hidden from business who pay for services and developers who uses them. Those services scale automatically, operate transparently and do not cost when idle.

2.2 Serverless Processing

This section describes how a serverless compute service (FaaS) executes the code, what are lifecycle of function and different invocation types. Model description is based on Serverless White Paper published by CNCF Serverless Working Group [2]. Additionally, section 2.2.2 provides a description of FaaS implementation - a high-level description of how AWS Lambda operates.

2.2.1 Serverless Processing Model

The key elements of the serverless processing model are FaaS Controller, Event sources, Function instances, and Platform services. Figure 3 shows interaction between them.

![Figure 3 Key elements of FaaS solution](image)

FaaS Controller is responsible for managing and monitoring function instances. Managing includes executing code, scaling instances depending on demand and terminating them when idle. Event sources emit, or stream events and event can trigger one or more function instances. Also, functions can be executed synchronously or asynchronously calling Invoke operation via AWS Lambda SDK. Such method when Lambda is invoked right away is called Push model. Another Pull model is used in streams, which batches new records together before invoking Lambda.
The executed function receives event data and context information as inputs. Event data depends on event source, for example, it can be an S3 event that an object has been stored and includes objectID, path and time. Context provides information about resources (e.g., memory/time limits) and execution environment (e.g., global and environment variables).

The function can communicate with platform services (BaaS) and they can emit new events.

### 2.2.2 Container Lifetime

Each function instance is executed in a container (sandboxed environment) to isolate them from one another and provide resources such as memory or disk. The first execution of function always takes longer time due to need of spinning up the container and starting a function instance. This process is called "cold start." According to AWS Lambda Overview and Best Practices document [3] initiation speed also depends on language, code package size, and network configuration. Reducing the size of code package helps to speed up initial execution because the package is downloaded from S3 each time before creating function instance. Best practices [3] suggests using interpreted languages instead of compiled ones to reduce the time of cold start.

When the function finishes execution, a container is suspended, and it is reused if the same function is triggered again. Background threads or other processes, spawned before the suspension, are resumed too.

A container is terminated when it stays idle for a longer period of time. AWS does not provide official information about how long it keeps inactive containers. However Yan Cui carried experiments and concluded [5] that most of the time container is terminated after 45–60 minutes of inactivity, although sometimes it is terminated earlier. Frederik Willaert analyzed lifecycle of AWS Lambda containers and wrote an article [6] which includes intriguing observation, that requests are always assigned to the oldest container. This implies that when containers are idle newly created containers are terminated sooner. Tim Wagner, general manager of AWS Lambda, revealed more interesting details in [6] article comment:

- a container can be reused up to 4-6 hours
- a container is considered as a candidate for termination if it has not been used in the last five minutes
- quickly abandoned containers are primary candidates for termination
- high levels of recent concurrency are scaled-down incrementally, in conjunction with policies that pack work into fewer containers at low rates of use.

The serverless function is stateless in theory, however, in practice it has state and that state can be used to increase performance, for example, keeping the database connection open between function invocations. Additionally, serverless providers are working to reduce cold-start frequencies and time.
2.3 AWS Serverless platform

AWS offers a wide range of managed services for building applications or processing data without owning servers. Using them instead of building a custom solution might greatly reduce development costs and time. Therefore, it is necessary to get familiar with provided services before developing something new. AWS serverless services are shown in Figure 4.

The core of AWS serverless computing is *AWS Lambda*. This service lets execute code without provisioning servers. Lambda can be invoked by an event from other services and manually using AWS SDK or AWS console. The price depends on execution time and available RAM.

*AWS Step Functions* is a service to orchestrate the components of distributed applications using visual workflows. This service can be used to implement distributed transactions, coordinate complex business logic, manage long-running business processes and handle errors with built-in retry and fallback. The price depends on a number of state transitions.
Amazon API Gateway is a service to create, maintain, secure and monitor REST API. The backend can be either EC2 instance or AWS Lambda. API Gateway allows invoking AWS Lambda function through HTTP call. Additionally, client SDK can be generated based on API definition for JavaScript, iOS, and Android. The price depends on a number of requests to API plus the cost of data transfer in and out.

AWS AppSync – real-time data synchronization layer via GraphQL. It can expose data from multiple sources such as DynamoDB table, Elasticssearch, and AWS Lambda. AWS AppSync can be used to build social media, chat, and collaboration applications. It offers real-time synchronization with built-in offline support, conflict resolution in the cloud and flexible security model. The price depends on a number of operations, connected time (real-time updates) and the amount of transferred data.

Amazon Cognito provides identity and access control services. It allows users to sing-up, sign-in using identities in Cognito User pools and federated users via social and enterprise identity providers. Security features include role-based access control, multi-factor authentication. Moreover sign-up, sign-in flow can be augmented using AWS Lambda triggers. The price depends on monthly active users.

AWS offers multiple serverless services to store data persistently.

Amazon Simple Storage Service (Amazon S3) – exposes secure, durable, highly-scalable object storage via simple REST API. The object can be any type, and any amount of data and objects can be grouped in buckets. Access to data is controlled per object and bucket policies. By default, all data is private. The price depends on the amount of stored and transferred data plus a number of requests. Data in Amazon S3 can be analyzed running SQL queries using Amazon Athena. The pricing is based on the volume of scanned data.

Amazon DynamoDB – low-latency NoSQL database supporting document and key-value storing models. Access to data can be configured per field granularity. The price depends on the size of written, read, and indexed data per Capacity Unit (scalability measure).

Amazon Aurora Serverless - on-demand, auto-scaling configuration for Aurora. Amazon Aurora is a relational database which is compatible with MySQL and with PostgreSQL. Serverless configuration automatically start-up, scale up/down database based on load and shuts down when a database is not used.

All storage services except for Amazon Aurora have integration with AWS Lambda.

Streaming data can be handled using Amazon Kinesis. Amazon Kinesis is a group of services designed to collect, process, and analyze real-time, streaming data like application logs, IoT sensors data, audio, video, and other continuously generated data. Kinesis Data Firehose is for capturing, preparing and loading streaming data into data stores and analytics tools; the price depends on the volume of ingested data. Kinesis Video Streams is to ingest and process video streams for analytics and machine learning. The price is calculated based on the volume of ingested, stored and consumed. Kinesis Data
Streams is service to process or analyze streaming data and price depends on Shard Hour (processing capacity) and PUT Payload Unit (25KB). Kinesis Data Analytics allows to process streaming data using SQL queries and pricing is calculated by Kinesis Processing Unit (processing capacity).

AWS has two services for inter-process messaging. Amazon Simple Queue Service (SQS) is a fully managed message queuing service. There are two types of message queues. Standard queues guarantee at-least once delivery and give maximum throughput. SQS FIFO guarantees exactly once delivery in the exact order that they are sent, however, throughput is limited to 300 transactions per second. The price depends on a number of requests and type of queue, plus bandwidth. Amazon Simple Notification Service (SNS) is a fully managed publish/subscribe messaging service. Published messages can be pushed to Amazon Simple Queue Service (SQS) queues, AWS Lambda functions, and HTTP endpoints. The price depends on the number of published and delivered notifications.

AWS allows having whole continuous integration pipeline without managing any servers using AWS Code services which are described below. AWS CodeCommit – is a fully-managed source control service to host private Git repositories. Pricing is based on a number of active users who access the repository during the month. AWS services like AWS CodeBuild and AWS CodePipeline which access repositories are counted as an active user. AWS CodeBuild provides service to compile source code, run tests and produce artifacts to deploy. Builds are processed in parallel, therefore, no waiting queue. The price is calculated per build duration and capacity of build instance. AWS CodeDeploy provides automatic deployment service to EC2 instances and on-premises servers. Services are updated gradually, via configurable rules deployment is monitored and in case of error deployment can be stopped, and changes rolled back. This service costs only deploying to on-premises and price is calculated per number of updated instances. AWS CodePipeline is a continuous integration and continuous delivery service. The pipeline is highly customizable and can be integrated with various third-party services not only AWS. For example, source code can be stored in GitHub, Jenkins can be used to build, Apica to test and XebiaLabs to deploy. There is a fixed monthly $1 charge per active pipeline for using this service. AWS CodeStar is a service to provide an entire continuous delivery platform for simple projects. Code changes are automatically built, tested, and deployed using AWS CodeCommit, AWS CodeBuild, AWS CodePipeline, and AWS CodeDeploy. Project templates help to setup deployment infrastructure for AWS Lambda, Amazon EC2, or AWS Elastic Beanstalk. This service is very easy to use, however, it not so advance and flexible as AWS CodePipeline. AWS CodeStar is free, only services used by its costs.

AWS has two services to monitor serverless applications - AWS X-Ray and Amazon CloudWatch.
Amazon CloudWatch – is a general monitoring service for AWS services. The service allows collect and track metrics and logs, set up alarms and react to any change in AWS resources. For example, CloudWatch Events can invoke AWS Lambda function in the response of the change in Amazon EC2 instance, like connect to the instance and install necessary software when a new instance with a specific tag is started. CloudWatch provides configurable dashboards for metrics visualization and logs explorer with filtering functionality. The price depends on the number of dashboards, number of custom metrics, number of alarms, the volume of logs and number of custom events.

AWS X-Ray – is a service to debug a distributed application. It helps to analyze, and debug working system, identify and troubleshoot the cause of failures and performance degradations. The service visualizes communication between system components, and it works with applications written in .NET, Java, and Node.js that are deployed on AWS.
3 Deployment package size impact on cold-start

Cold-start is one of the most misunderstood limitations of Serverless computing which has room for performance optimization by users themselves. At first, it happens more rarely, than people assume because warm functions are reused and lingering up to 4 hours [7]. Cold-start occurs once for each concurrent execution of a function and duration depends on allocated memory, runtime, and package size.

Yan Cui compared cold-start time in different languages, memory and code sizes. Most notable observations [8] showed that:

1. memory size improves cold-start time linearly;
2. C# and Java have over 100 times higher cold-start time compared to dynamic languages. Cold-start time of 95th percentile of a 1024MB function were 712ms in C#, 677ms in Java, 2ms in NodeJS and 0.5 in Python.
3. code size improves cold-start time

The last observation that code size improves cold-start time is counter-intuitive because bigger package requires more time to download and extract the package, but it has a positive effect on overall cold-start time. The subsequent analysis [9] of cold-starts between different serverless providers by Mikhail Shilkov. The analysis has a very comprehensive introduction into the cold-start problem, and it’s worth reading the full article. However, the results of it couldn’t confirm Yan Cui observation regarding code size. Both performance tests have the same flaw in methodology – they measure code size impact on cold-start by loading libraries. This basically measures how fast a runtime loads particular libraries but not how deployment package size impacts cold-start on a serverless platform. Therefore, additional experiments are carried out for this thesis to better understand the impact of package size. Knowing typical performance helps to identify parts which can be optimised.

3.1 Methodology

A serverless platform creates a sandbox, load user code, prepare a runtime and execute a function. Figure 5 depicts this process.

![Figure 5: Execution model of serverless function](image)

Experiments are targeting steps performed by a worker which is responsible for executing user’s code. NodeJS v8 was chosen as runtime because it is lightweight. Experiments were done using AWS Lambda on 17-18 November 2018. AWS doesn’t have an explicit way to destroy all warm containers, therefore to ensure cold-start the functions are redeployed after each execution. Serverless Framework is used to deploy and execute functions in all scenarios. Each case is executed 200 times. The function itself doesn’t do any calculations.
and just returns “Your function executed successfully!” . The execution duration is parsed from CloudWatch Logs.

### 3.1.1 Scenarios
The first two scenarios are orientated at measuring workers tasks and the last one at the whole cold-start cycle.

#### 3.1.1.1 Different package and memory sizes, one dummy file.
This is the main scenario dedicated to analyzing the impact of deployment package size and allocated memory. The scenario includes cases with package sizes of 1 MB, 10 MB, 25MB, 49MB and allocated memory sizes of 128MB, 256MB, 512MB, 1024MB, 3008MB for AWS Lambda function. For each memory configuration, functions of different size were deployed and executed. A package contains function code and specific size dummy file which was generated using an online random file generator\(^1\). Benchmarking script is presented in Appendix C.

#### 3.1.1.2 Big package, lots of small files
This scenario helps to determine the impact of lots of small files in the deployment package. The package consists of function code and node modules which are presented in Appendix C. During runtime modules are not loaded. Deployment package size is 44MB, and it contains 26k files. Allocated memory for AWS Lambda function – 1024MB.

#### 3.1.1.3 Big package, lots of small files, modules are loaded
This is the scenario used in experiments by Yan Cui and Mikhail Shilkov however node modules aren’t the same. The idea is to find out how much loading of modules effects cold-start. The package consists of function code and node modules which are presented in Appendix C. During runtime modules are loaded. Deployment package size is 44MB, and it contains 26k files; allocated memory for AWS Lambda function – 1024MB.

### 3.2 Results
This section presents the results from the experiments to find out the impact of deployment package size on cold-start. Results in form as table can be found in Appendix C.

#### 3.2.1 Different package and memory sizes, one dummy file.
As expected, the biggest difference in cold-start has allocated memory. Figure 6 illustrates that allocating 256MB reduces cold-start time twice to 14ms for 80%-tile compared to 128MB. Increasing memory to 512MB and more yelled

---

\(^1\) Online random file generator: https://pinetools.com/random-file-generator
cold-start time to ~3ms. Also, the lowest cold-start time doesn’t depend on memory, and it is ~2ms.

The results of different package sizes do not vary significantly as visible in Figure 7 and Figure 8. Cold-start time is usually 1ms slower for 80%-tile when 1mb deployment package is used compared to 49MB. However, it’s bigger when deployment packages are 10MB and 25MB.

Cold-start of 80%-tile is 36-41ms when 128MB of memory is allocated and 14-16ms when 256MB.
Cold-start duration dependency on package size and allocated memory of 512MB, 1024MB and 3008MB

Cold-start of 80%-tile is 3-5ms when 512MB of memory is allocated, 3-4ms when 1024MB, and 2-3ms when 3008MB. The complete results in form as table is presented in Table 5, Appendix C.

One scenario of 512MB was run without redeploying to have a hunch how long it takes to execute the warm function, and it showed that 80%-tile completed in ~0.5ms. The complete results are presented in Table 7, Appendix C.

### 3.2.2 Big package, lots of small files

Results presented in Table 1 show negligible difference between deployment package with multiple small files and one big, respectively cold-start for 80%-tile is 3.1ms and 2.8ms, the difference is 0.3ms.

<table>
<thead>
<tr>
<th>Package</th>
<th>std dev</th>
<th>avg</th>
<th>Min</th>
<th>Max</th>
<th>80%-tile</th>
<th>95%-tile</th>
<th>99%-tile</th>
</tr>
</thead>
<tbody>
<tr>
<td>44mb (small files)</td>
<td>3.619745</td>
<td>3.2305</td>
<td>1.66</td>
<td>36.03</td>
<td>3.146</td>
<td>7.3545</td>
<td>9.2406</td>
</tr>
<tr>
<td>49mb (one big)</td>
<td>2.400699</td>
<td>2.9298</td>
<td>1.68</td>
<td>19.48</td>
<td>2.806</td>
<td>6.352</td>
<td>13.0747</td>
</tr>
</tbody>
</table>

The results of the package with small files have nearly twice longer (36ms) slowest execution time, however, 99%-tile shows faster execution (9ms compared to 13ms), this leads to believe that it is outlier value due a load of the serverless platform.
3.2.3 Big package, lots of small files, modules are loaded

Figure 9 illustrates the results. Cold-start of 80%-tile when modules are loaded is 78ms, that is 26 times slower compared to the case when modules are included into the package but not loaded.

![Figure 9](image)

**Figure 9 Cold-start times when modules included in deployment package are loaded and skipped**

It is visible that the initialization of modules takes much more time than preparing a sandbox. The results in form as table are presented in Table 6, Appendix C.

3.3 Conclusions

Results confirmed that the biggest impact on cold-start does the size of allocated memory and initialization of runtime, also showed that not deployment package size nor a number of files don’t affect significantly. Runtime part depends on users. Therefore there is room for optimizations. Traditional approaches can be applied, for example, more lightweight runtime can be chosen, usage of libraries can be reduced, libraries should be lazy-loaded, more performant external services should be used, for example Aurora Serverless database can be accessed via HTTPS API instead of requiring database connection and that can shove ~300ms from cold-starts when database is rarely used.

Lastly, in order to have stable cold-start duration, a function should have at least 512MB allocated memory.

4 Serverless Application Programming model

Serverless computing emerged to reduce operational costs and tasks, such as provisioning of infrastructure. However, it also affects the development of applications. The main architectural change is the common way to invoke the application. It does not matter where data comes from (HTTP request, scheduled event, an event of resource change, etc.) the invocation of function which processes data is agnostic on the platform level. Also, CNCF Serverless Working Group is working to provide a specification for describing event data in a platform-agnostic way\(^3\). Another profound serverless idea to move state and coordination to a client is described in the article [8] by Gojko Adzic and Robert Chatley. Historically, only server-side applications were accessing backend resources to ensure data integrity, traceability, and protection. However, in AWS serverless stack, all services share the same identity and access model – AWS IAM. AWS services treat external requests from a client in the same way as they were from AWS Lambda. It is possible safely and securely use backend services directly without an intermediate layer because most of AWS services offer fine-grained access control over resources. For example, DynamoDB access policy includes conditions which can be used to limit users’ access only to items they created, or items are assigned to them [7]. Access to S3 can be given for certain timespan, specific IP and restrictions on file size, the type can be applied.

Direct communication to backend services is faster, more scalable and cheaper instead of going through additional layer - AWS Lambda. Also, the state management is simpler because the client keeps track of just own state, comparing to the server which manages the state of multiple clients.

One of the disadvantages using serverless services directly is tight coupling with server platform. The client becomes more fragile due to API changes in backend services. Also, most of the time updating client requires more effort than a server.

4.1 Development

Serverless applications heavily rely on services provided by the serverless platform. Most of the time it is not possible to run those services locally because they are proprietary. Therefore, running functions in the cloud are the recommended way of developing and testing. Deployment step is included in all starting tutorials, and practically developers are starting with production-ready environment.

Lack of debugging capabilities is the most noticeable drawback especially for developers who used to work with compiled languages because those languages have the most mature debugging tools. Serverless functions can be debugged

\(^3\) Working draft documents are available at: https://github.com/cloudevents/spec
by adopting test practices like stubbing managed services and writing drivers which call serverless functions.

4.1.1 Infrastructure management
AWS provides web console, AWS CLI and AWS Serverless Application Model (AWS SAM) for managing serverless resources. First two are convenient while exploring AWS Lambda functionality because they are designed to work with individual resources and AWS SAM is designed to manage a serverless application which contains multiple resources (some popular third-party alternatives are mentioned in 4.4 Frameworks section). AWS SAM is an extension to AWS CloudFormation, which is the primary approach to define AWS resources. However, AWS CloudFormation has a steep learning curve, and configuration of the serverless application can quickly become more complex than the actual application. Since the implementation of SAM was open sourced, missing features can be added by the community; therefore usage of AWS CloudFormation for serverless applications most likely is going to decline.

4.1.2 Deployment
Deploying distributed software is a troublesome task because the system works correctly only if all services are running. However, the desire is to update services frequently and do not impact the availability. The AWS serverless platform takes over complexities of deployment and offers zero-downtime deployments out-of-the-box. A new version of function can be deployed via AWS web console or AWS CLI seamlessly, without interrupting existing connections. Additionally, using AWS CodeDeploy, it is possible to shift part of the traffic to a new version and rollback automatically if an error occurs. Additionally, the main purpose of third-party serverless frameworks is to automate deployment and configuration tasks to improve the development experience.

4.1.3 Local development
Serverless architecture heavily relies on managed services, therefore, working with remote services makes development and debugging harder due to additional latency, reduced observability and lack of management flexibility. Therefore, various tools emerge to emulate cloud functionality. All reviewed frameworks in case-study emulate AWS API Gateway or at least have a command to execute a function using CLI. AWS SAM CLI\(^4\) stands out among frameworks because it can execute functions in all runtimes available in AWS Lambda. Most complete AWS stack emulator is called Localstack\(^5\) by Atlassian, and it emulates 17 AWS services.

\(^4\) https://github.com/awslabs/aws-sam-cli
\(^5\) https://github.com/localstack/localstack
Local development can be easier by following an agnostic development approach [10] which recommends separating business logic from the serverless provider-specific code. This approach is developed to mitigate the impact of vendor lock-in. Additionally, separation of concerns has a positive effect on unit testing. Since the code is loosely coupled, mocking and isolating cloud provider services are much easier. In general, unit testing for serverless architecture is easier because serverless functions tend to be smaller in size and less coupled with other services. Integration and end-to-end testing are more accessible compared with the traditional environment because with serverless it is cheap to launch testing environment identical to production and customers pay only when tests are running. Also, it is beneficial to use managed services instead of mocking to detect regressions. Managed services are always improving, and the user isn’t in control of version, so there is a small possibility of an unexpected issue caused by a change in managed service, however breaking changes are rare in managed services and their APIs.

4.2 Frameworks
Since serverless computing is done in the cloud, initial tools were dedicated to making deployment to cloud more user-friendly. Most of them are from third-parties, however, on August 2017, AWS team released the official tool to manage serverless application called SAM Local. Additionally, there are more tools to build, test and deploy serverless applications which will be analyzed in this section. We start with the most specialized framework Chalice and end with the most flexible – Serverless Framework.

4.2.1 Chalice
Chalice is a python micro-framework for serverless applications by the AWS team, and it is licensed under the Apache License 2.0. The only supported language is python and AWS is the only supported cloud provider. Chalice CLI has functions to create, deploy, and delete an application, also to run it locally. The main distinguishing feature of this micro-framework is that the configuration is defined using annotations in code. It can be convenient for small and simple applications, however annotations tightly couple framework with code. Another significant drawback that the framework does not expose every feature of API Gateway and Lambda. For example, there is no way to use versioning of Lambda functions. Third party extensions are possible due to the dynamic nature of Python like Domovoi allows handling other AWS Lambda event sources not only HTTP requests. However, the setup process is not trivial and includes using domovoi CLI instead of the chalice, requires changing app entry point and use domovoilib directory instead of chalicelib to stage files into the deployment package.

4.2.2 Apex
Apex is a lightweight toolkit to build, deploy, and manage AWS Lambda functions. It is designed for event-driven pipelines on AWS and released under MIT license. The supported runtimes are Node.js, Golang, Python, Java, Rust, Clojure. Apex only manages Lambda functions. Therefore, it is integrated with
Terraform to manage additional resources. Apex uses “project” and “function” abstractions to organize a project and enforces strict structure. Details about the project, environment configuration and defaults for functions should be specified in project.json file placed in the root directory. The functions should be placed in sub-directory of “functions” directory. Sub-directories can have function.json file to define function specific configuration.

Apex offers only essential features to work with AWS Lambda like deploy, execute, rollback, display of logs and metrics, however, those features are polished and high quality. For instance, it can autocomplete commands & function names, apply command for multiple functions using globbing (ex: apex deploy api_*), deploy in parallel and only updated code (checksum is used to skip already-deployed code), and execute commands via hooks. The most significant disadvantage is that Apex does not provide any tools to run functions locally, that complicates development and testing. Besides, integration with the system could be better. It does not have an installer, modifying PATH environment variable is needed to access it system-wide. Also, the tool does not work with temporary STS credentials. This is a real problem for enterprises like Scania because due security reasons most of the time temporary security credentials are used. The workaround is to load credentials to environment variables whenever they change.

Integration with Terraform is very loose, and Apex does not offer any tools to make it easier. Therefore additional Terraform knowledge is necessary to start working with Apex.

4.2.3 AWS SAM

The AWS Serverless Application Model (AWS SAM) is an extension to AWS CloudFormation to provide a simplified way of defining serverless resources, though it can only be used within the AWS ecosystem. The specification is developed by AWS team and available under Apache 2.0 license. SAM supports definitions of API Gateways, Lambda functions, and Amazon DynamoDB tables. The files describing a serverless application are JSON or YAML formatted text files. SAM description includes a global section, resource, event sources, and properties. The global section contains the configuration of environment where Lambda function runs, for example, runtime, memory, VPC Settings, environment variables, etc. Resources section declares the AWS resources that will be to include in the stack, and it can be Lambda function, API Endpoint or DynamoDB table.

4.2.3.1 AWS SAM CLI

SAM Local is a tool to improve the development experience of Lambda functions in a local environment. It allows test serverless applications locally and deploys them to AWS. Additionally, it has validator of SAM templates, generation of sample event source payloads, supports local debugging of

---

6 Issue reported in: [https://github.com/apex/apex/issues/553](https://github.com/apex/apex/issues/553)
functions, can serve static assets, and store captured logs from the function’s invocation into a file. SAM Local is written in python and has binary versions for all modern OS, and recommended installation is via npm. However, it requires Docker (software container platform) to run.

### 4.2.4 Up

Up is deployment tool by Apex Software for HTTP servers. It supports deployments only to AWS platform; however, it is designed to be platform-agnostic. The framework is licensed as GPLv3, and it comes in two editions: community edition which is open source and pro edition which costs USD 20 per month. Up supports Node.js, Golang, Python, Java, Crystal, Clojure, and static sites. It allows deploying applications powered by frameworks such as Express, Django, Koa, Golang net/http, and others. Additionally, this tool has a local development server, can serve static files, can show log output and rollback to a previous deployment. Moreover, it supports managing and purchasing of domain names from AWS Route53 and managing resources of a stack, like show resource changes, apply them, and delete the stack resources. Configuration can be written only in JSON. The tool provides “hooks” to invoke shell commands at specific points within the deployment workflow.

Up might be used in a lift&shift scenario for existing web applications since no code changes are required for Lambda compatibility. That reduces the learning curve for developers because the tools to run and test the application locally are the same. Additionally, it prevents vendor lock-in because no AWS specific code is needed to handle requests. There are two main disadvantages of such approach - latency and customization. The first concern is the increase in latency. Up uses a reverse proxy to point all requests from AWS Lambda to an application HTTP server. Up’s reverse proxy introduces an average of around 500µs (microseconds) per request. Additional overhead depends on a framework used by an application. The second disadvantage is the loss of some customization of API Gateway functionality, such as security control and validations of data. Serving everything from one Lambda makes it hard to define allowed input parameters or configure access control using IAM.

### 4.2.5 The Serverless Framework (Serverless.com)

The Serverless Framework is an open-source toolkit for deploying and operating serverless architectures. It is a provider agnostic, actively maintained, and released under MIT license. The framework provides a configuration DSL which is designed for serverless applications and the files describing a serverless application are YAML or JSON formatted text files. An application can be deployed to a single provider. The framework supports AWS, Microsoft Azure, Apache OpenWhisk, Google Cloud, Kubeless, Spotinst, and

---

7 Based on official FAQ: https://up.docs.apex.sh/#faq
Auth0 Webtasks cloud providers. For Scania interests, the main focus of this thesis is on AWS cloud provider. The Serverless Framework using the AWS provider generates a single AWS CloudFormation stack per application. Central concepts describing a serverless application are service, provider, function, event, and resource. A single application can have multiple services, and service is used to define a function, the events that trigger them, and resources used by functions. Environment configuration can be specified at provider and function levels. Such flexibility allows to specify single consistence environment per service and customize per function when needed, for example, use different run-time language for some functions. DSL provides dynamic variables which support following sources of variables:

- environment variables
- CLI options
- other properties defined in serverless.yml
- external YAML/JSON files
- variables from S3
- variables from AWS SSM Parameter Store
- CloudFormation stack outputs
- properties exported from Javascript files (sync or async)
- pseudo parameters reference (predefined by AWS CloudFormation)

### 4.2.5.1 Functionality

The framework completely covers the development cycle. It provides CLI to:

- create a new service using predefined templates or installed locally by downloading the Github repository and unzipping it.
- deploy a service specified stage
- display information about the service
- invoke functions - locally and on cloud
- stream all logs to console for a specific function
- show metrics for a specific function
- rollback the service to a specific deployment
- destroy application stack from a provider (service and all resources)
- manage plugins of the framework

The most significant advantage of the Serverless framework is the support of plugins. They are written in Node.js and have broad adoption in the serverless community. The most popular plugin serverless-offline, which emulates AWS λ and API Gateway locally, was downloaded 570K+ times. Existing plugins provide rich functionality:

- emulates AWS like IoT and Kinesis Events, DynamoDB, Scheduler, and SNS services locally.
- adds support for services, like Step functions, manage custom domains with API Gateways, provides CLI for encrypting and decrypting secrets using KMS, add CloudWatch alarms to functions.
- adds support for languages: deploy WSGI applications (Flask/Django/Pyramid), Typescript, Haskell, CoffeeScript, Clojurescript.
• automation - deletes old versions of functions, keeps functions warm, provides tools for automating tasks in development workflow.

4.2.6 Summary

All overviewed frameworks were created for the same purpose to improve the deployment process in the cloud environment, however, each of them has a niche where it suits best. The concise list of features and main details about each framework are displayed in Table 2.

AWS Chalice offers the fastest way to build python APIs, routes configuration is defined in the code using annotations, IAM policies are generated automatically, and the deployment process is astonishing fast in a matter of seconds.

Apex provides shim for running arbitrary languages which aren’t supported by AWS Lambda, and it is integrated with Terraform. This option is great if you are already using Terraform for managing infrastructure and have a small use-case for serverless.

Up enables a straightforward way to lift and shift existing HTTP application to the serverless environment. This framework trades customizability on platform level to speedy migration so that it can be used as a first step moving away from servers.

AWS SAM has a specification to describe Serverless applications and CLI tool to deploy and run an application on local machine. AWS SAM Local offers AWS Lambda like runtime environment and supports the same languages. This high-quality framework allows creating any size serverless applications.

The Serverless framework is the most mature toolkit for deploying and operating serverless applications. It supports multiple serverless platforms, has powerful DSL to describe serverless applications and it is extensible via plugins. Also, this framework has the biggest open source community.

<table>
<thead>
<tr>
<th>Name</th>
<th>Serverless Framework</th>
<th>Apex Owner</th>
<th>Up Owner</th>
<th>AWS SAM (Local)</th>
<th>Chalice Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner</td>
<td>Serverless Inc.</td>
<td>Apex Software Inc</td>
<td>Apex Software Inc</td>
<td>AWS</td>
<td>AWS</td>
</tr>
<tr>
<td>License</td>
<td>MIT</td>
<td>MIT</td>
<td>GPLv3</td>
<td>Apache-2.0</td>
<td>Apache-2.0</td>
</tr>
<tr>
<td>Price</td>
<td>Free</td>
<td>Free</td>
<td>Free / Pro-20 USD/mo</td>
<td>Free</td>
<td>Free</td>
</tr>
<tr>
<td>Releases</td>
<td>86</td>
<td>30</td>
<td>75</td>
<td>8</td>
<td>23</td>
</tr>
<tr>
<td>Contributors</td>
<td>372</td>
<td>96</td>
<td>27</td>
<td>32</td>
<td>48</td>
</tr>
</tbody>
</table>
4.3 Architecture and design

Serverless architecture is event-based; therefore traditional tools can be used to design and model it, however, they might lack specific icons for AWS services. AWS provides icons for its services in multiple formats and suggests several online tools for creating diagrams⁸. LucidChart stands out because it has an import tool which creates a diagram from existing AWS infrastructure. It is worth to mention “Serverless By Design” – an open source serverless design web application⁹ created by Danilo Poccia. This tool allows not only to make architecture diagram but also generate AWS SAM or Serverless.com templates. At the moment of writing, there is no high-quality software oriented to modeling serverless architecture. However, the heavy upfront low-level design isn’t a trait working with serverless because serverless function by design is stateless, has single

---

⁸ Icons and tools available at: https://aws.amazon.com/architecture/icons/
⁹ A live version available at: https://sbd.danilop.net and source code at: https://github.com/danilop/ServerlessByDesign
responsibility with well-defined input and output. A serverless function should be easy to connect to event source and replace when needed. On conceptual level serverless architecture relies on well-integrated services provided by the serverless platform. These services are scalable, robust, evolving constantly however sometimes can be limited in functionality. Serverless design patterns are emerging, and several are described in this section in order to explain how serverless application can be build using stateless functions and various services.

4.3.1 Code patterns

Patterns in this section describe how code base can be organized when working with serverless. This section is based on the article “Serverless Code Patterns” [7] by Serverless Inc.

4.3.1.1 Monolithic Pattern

In the monolithic pattern, one serverless function processes all events, and it executes different logic based on payload. If it is an HTTP application, all requests are processed by one serverless function, and a custom router is responsible for executing necessary logic based on the HTTP path, method, and other parameters. Figure 10 shows the implementation of the monolithic pattern using Serverless framework.

![Monolithic pattern in Serverless framework](image)

Monolithic is a good starting point because only one entry point is AWS Lambda specific and an old application can be ported to AWS Lambda without a complete rewrite; the only adapter is needed which transforms AWS event to application specific call. Running and testing the application locally is also easier and can be done in a traditional way. However, bundling everything in one function makes deployment package bigger, and that increases cold start duration. Also, it is harder to observe and optimize running application because logs, metrics are grouped by serverless function. The rationale of putting multiple functions together is to make development and deployment faster. However, function should be separated if it significantly impacts a serverless function.

4.3.1.2 Service Pattern
In the service pattern, each serverless function is responsible for a different business area. In DDD terms, one serverless function handles requests related to one sub-domain and represents one bounded context. Figure 11 shows the implementation of the service pattern.

```
const customersHandler = require('customersHandler');
exports.handler = (event) => {
    const {path, httpMethod, body} = event;
    return customersHandler(path, httpMethod, body);
};
```

**Figure 11 Service pattern in Serverless framework**

Such approach creates real boundaries between sub-domains. Deployment package contains artifact of one sub-domain, therefore, it is smaller, and that leads to faster deployments and shorter cold starts. Testing and running locally are similar to the monolithic approach. However separate repository might be needed for storing shared code between services. Also, services can be deployed separately and managed by different teams. Disadvantages are similar to monolithic: requires routing layer and it’s harder to observe and fine-tune because serverless function performs multiple operations.

### 4.3.1.3 Nano-service Pattern

In the Nano-service pattern, serverless function exposes just one functionality. It is the most simple way to start with serverless because no custom routing is needed, the configuration is enough. Figure 12 shows implementation.

```
const customersRepo = require('customers.repository');
exports.handler = (event) => {
    return customersRepo.create(event.body);
};
```

**Figure 12 Nano-service pattern in Serverless framework**

Since the serverless function does just one function, it makes it easy to determine what inputs and results should be, therefore debugging is easier. Also, observability is better because logs, metrics are collected per function. Fine-tuning is possible because resource allocation and environment settings are also defined per function. However, such flexibility costs more when functions are accessed infrequently due to cold-starts. Deployment package size is similar to package size when services pattern is applied because most of the time dependencies are bigger than a code of business logic. Additionally,
deployments are slower because resources must be provisioned for multiple functions. Also, it is harder to manage a lot of functions due to cogitative overhead.

Nano-service pattern suits best for non-typical use cases when usage of dependencies and resource are different than the rest of application. For example, if we have a web application which the main function is to query a database and pass data to the client, report generation should be separated, because working with XML/PDF requires more RAM/CPU and specific library. Having a separate function for export allows fine-tuning environment like allocate more RAM to keep other deployment packages smaller.

4.3.2 Files upload
Files upload is common functionality in web applications and implementation in the serverless environment is significantly different because the serverless function has time constraints and cost depends on execution time. Therefore we never want to send a big chunk of data to serverless function. Instead of that, data should be sent directly to object store such as S3 and processing invoked by an event.
It is a good practice to keep buckets private and explicitly grant access only when it is needed. Signed URLs are a convenient way to give more control over access to data. They allow granting access to specific IP addresses, timeframe, and objects.

4.3.3 State management
Serverless functions are considered to be stateless, however, in reality, they persist state between invocations on the same container. However, it is not possible to determine how long such state is going to be stored, because containers management algorithm is not public and developers are constantly improving it to reduce the frequency of cold starts. In this section, we will analyze different ways to keep state when working with AWS Lambda.

4.3.3.1 AWS Lambda
AWS Lambda uses containers to run programs, and when the request is finished, the container is suspended. Therefore, process state and ephemeral disk can be used to keep some data between invocations.
It is recommended to keep and reuse external connections between function invocation because creating a new connection increases execution time. For example, Figure 13 shows how to reuse database connection between invocations of the function.
Additionally, it is worth caching working data in process memory to improve performance, for example, settings, which normally stored in database or currency exchange rates which most of the time is provided by third-party service.

Bigger files or files which are updated more frequently than deployments happen can be stored in S3, downloaded when the function is invoked and cached in the disk. An example of such data can be IP geolocation data. However, keeping data in memory or temporary disk creates an invalidation problem. There is no direct way to terminate container; the closest way is to redeploy a serverless function. Another way is to refresh data manually after x minutes.

### 4.3.3.2 AWS Step Functions

Sometimes business processes involve multiple actions which can last days. Coordination of such processes requires to store state in persistent and reliable fashion and to do that using serverless functions it is tricky. Therefore, AWS offers dedicated service named AWS Step functions which provides workflow coordination for distributed components. Workflow and handling of errors are modeled as finite state-machine, where you can connect serverless functions and specify what data from previous Lambda results should be passed as input to the next one. Intermediate results of one step must not exceed 1MB and whole execution time of workflow is limited to 1 year. Using AWS Step functions, it is easy to automate tasks reliably, for example, synchronize S3 buckets or implement long running processes like workflows which require human interactions, for example, approval by email. AWS Step functions allow to reduce coupling and improve robustness by minimizing the amount of custom try/catch and retries in code. The biggest hurdle is testing locally because AWS does not provide any tool. There are third-party projects like

---

10 The implementation is available at:  
stepfunctions-local\textsuperscript{11} which is 100\% compliant with AWS Step functions API but using it daily via CLI is not convenient.

4.3.3.3 Amazon Cognito

Amazon Cognito service is a good place store session data, for example, details about the user like email and name. Also, metadata like the date of the last visit, failed login attempts and similar information which is needed for the current user. Since client authenticates to Amazon Cognito service directly, session data can also be fetched and altered without invoking backend. Access control allows specifying which attributes can be read and modified by the user. Also, users can be placed in groups. It is possible to search for users however custom attributes are not searchable.

4.3.4 Emulation of server-client communication

Developing web applications sometimes it is needed to pass data from server to client as fast as possible. For example, to inform the client that server completed long running task like video decoding or notify the client about some event occurred in the system like a new message in the chat application. This can be achieved using WebSockets or Server-Sent Events (SSE). However, both options require to keep persistence connection, which is not optimal using serverless functions because pricing is based on duration. Such technologies can be emulated by polling. Long-polling would be worse than SSE because only one response could be sent using the same connection. Short-polling would cost less, but latency would be higher.

This can be solved using other services instead of serverless function. AWS IoT service offers message broker over multiple protocols including WebSockets \textsuperscript{[11]}. Server-client communication can be done by subscribing web application to topic and publishing to the same topic from the serverless function. A more elaborate example is presented in the article “Serverless beyond Functions”.

Sometimes it is worth to poll service directly, for example, query DynamoDB or S3 from the client without calling the serverless function. It would be a poor choice for real-time communication, however, to track the progress of a task might be a cheaper solution than AWS IoT.

Cost efficient short-polling mechanism can be built around S3. To get data, client calls S3 instead of AWS Lambda. When some event occurs, AWS Lambda puts results to S3. Access can be restricted using IAM roles or signed URLs. Data can be automatically removed after some time using object expiration. If there are multiple clients reading data, performance can be improved, and costs reduced by using CDN – AWS CloudFront with caching.

\textsuperscript{11} Emulator of AWS Step functions. https://github.com/airware/stepfunctions-local
4.4 Security

Serverless applications tend to have a bigger attack surface because each function is a new entry point with potentially different configuration and different security measurements. One might argue that serverless architecture increases security by obscurity. An attacker doesn’t know what machine is executing the request and can’t tell if the next request is going to be executed on the same machine and can just guess that all requests are handled by functions written in the same language. Since security by obscurity should never be the only security mechanism, applications should be design and implemented with security in mind. AWS defines shared responsibility security model [12] where it is responsible for securing an infrastructure (hardware, software, networking) and the user is responsible for securing application and data. The only exception to securing of infrastructure is users’ responsibility to ensure network boundaries when VPC is used.

Additionally, AWS managed services can help to protect an application. For example, API Gateway can be used for authorization and authentication, data protection (TLS) and input validation (JSON Schema) of serverless API. However, it is always recommended to encrypt sensitive data and implement application-specific deep validation. It is worth to mention that leveraging API Gateway security features help to reduce costs because invalid requests aren’t handled by AWS Lambda. Therefore AWS changes just for usage of API Gateway.

Interesting turn serverless architecture has on DDoS attacks because they become a billing issue instead of security. AWS has a web application firewall called AWS WAF to deal with DDoS and common web-based attacks. AWS WAF has predefined config and tutorials[12] for common scenarios such as blocking an IP address that exceeds request limits and submits bad requests. Also, from an application point of view, reasonable timeouts should be set for calls to external services.

Another serverless specific problem is stale dependencies. The serverless function is deployed with its dependencies, and since the function is responsible for small functionality, it is less likely that function will be changed. Therefore, dependencies are going to be updated less often compared to traditional architecture. Over time, vulnerabilities are discovered in dependencies, and stale functions become vulnerable. Also, it is very easy to deploy new functions and removing them isn’t a priority because non-used functions barely costs, also it is hard to track which functions are not used. Serverless functions are deployed separately and all dependencies bundled together, that leads to multiple versions of dependencies which might have various vulnerabilities. At the time of writing, there are no widespread practices and solutions how to track and identify vulnerable dependencies in already deployed serverless

---

12 AWS WAF preconfigured rules and tutorials: https://aws.amazon.com/waf/preconfiguredrules/
applications. The only tool to track and scan dependencies for serverless architecture is Snyk\textsuperscript{13}. S3 buckets stand out considering risk criticality and needed efforts to fix. Media reported multiple incidents when personal data [13], financial data [14], classified NSA data [15] were exposed, and even malware was spread [16] via unprotected S3 buckets. Therefore, working with managed services, it is always useful to doublecheck who has access and automate validation when possible. Constant logging and monitoring help to know what is normal and detect when a legitimate attack happens.

Puresec study [17] identified the top 10 most critical security risks in serverless architecture and provided guidance on how to mitigate them. The document\textsuperscript{14} is very well written and worth reading whole. Therefore bellow is just a list of identified risks in order of criticality, where SAS-1 indicates the most critical risk and SAS-10 the least critical risk. Top 10 risks are:

- SAS-1: Function Event-Data Injection
- SAS-2: Broken Authentication
- SAS-3: Insecure Serverless Deployment Configuration
- SAS-4: Over-Privileged Function Permissions and Roles
- SAS-5: Inadequate Function Monitoring and Logging
- SAS-6: Insecure 3\textsuperscript{rd} Party Dependencies
- SAS-7: Insecure Application Secrets Storage
- SAS-8: Denial of Service & Financial Resource Exhaustion
- SAS-9: Functions Execution Flow Manipulation
- SAS-10: Improper Exception Handling and Verbose Error Messages

### 4.5 Observability

Observing a serverless system is difficult because user request/data flow are processed by multiple functions which can be called asynchronously. There are multiple inputs and outputs to monitor, and monitoring isn’t trivial because there is no access to the underlying OS and install agent or daemon to gather metrics isn’t possible. Moreover, the serverless function cannot be used to do some actions after the request is processed, all actions should be finished before returning response because the container is suspended as soon as a response is returned. Therefore, monitoring increases request time. Asynchronous monitoring is possible by-passing metrics as log message and afterward process logs and sends metrics. Such an approach does not prolong requests however it adds delay to monitoring, increases costs due to additional logs processing and

\textsuperscript{13} Snyk launches serverless support: https://snyk.io/blog/launching-snyk-for-serverless/

increases concurrently executed functions. Maximum limit of concurrently executed functions can be reached faster due to processing of logs. Therefore, it’s better to use third-party service, such as Datadog\textsuperscript{15}, when complex log analysis is needed. Otherwise, the optimal solution is to write log messages in JSON format and analyze them using Amazon CloudWatch Logs. The syntax of filtering and pattern matching is provided in the documentation\textsuperscript{16}.

### 4.6 Optimizations

Serverless architecture at first glance seems to be very limited at possibilities for optimizations because platform takes care of provisioning. However, nearly every managed service has some features and configurations which impact performance. Table 3 presents optimizations for common scenarios.

<table>
<thead>
<tr>
<th>Table 3 Common optimizations of AWS serverless stack</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cold start</strong></td>
</tr>
<tr>
<td>Prefer simpler libraries</td>
</tr>
<tr>
<td>Increase RAM</td>
</tr>
<tr>
<td>Optimize functions to reduce execution time</td>
</tr>
<tr>
<td><strong>Processing</strong></td>
</tr>
<tr>
<td>Use execution context</td>
</tr>
<tr>
<td>Use async operations when working with I/O tasks</td>
</tr>
</tbody>
</table>

\textsuperscript{15} https://www.datadoghq.com/
\textsuperscript{16} Amazon CloudWatch Logs documentation - Filter and Pattern Syntax: https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/FilterAndPatternSyntax.html
\textsuperscript{17} The tool available at: https://github.com/alexcasalboni/aws-lambda-power-tuning
<table>
<thead>
<tr>
<th>Increase RAM more than 1.5 GB when multithreaded CPU heavy tasks are executed</th>
<th>AWS allocates more than one CPU core starting from 1.5 GB RAM [18].</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduce data processed by a serverless function</td>
<td>Managed services most of the time can filter events before passing to the serverless function. For example, AWS IoT has a topic filter, Amazon Aurora – query filters and AWS S3 – select.</td>
</tr>
<tr>
<td><strong>Invoking AWS Lambda functions</strong></td>
<td></td>
</tr>
<tr>
<td>Invoke using AWS-SDK</td>
<td>AWS Lambdas can be invoked directly without API Gateway. When API Gateway functions aren’t used, it’s overhead can be avoided by invoking AWS Lambda functions directly.</td>
</tr>
<tr>
<td>Use API Gateway regional endpoints</td>
<td>By default, Amazon API Gateway endpoint is edge-optimized, however, if an endpoint is mainly called from the same region (EC2, Lambda), latency can be reduced by specifying the type to regional [19].</td>
</tr>
<tr>
<td><strong>Costs</strong></td>
<td></td>
</tr>
<tr>
<td>Optimize functions to reduce execution time</td>
<td>Execution time is directly associated with costs, AWS Lambda bills per 100ms.</td>
</tr>
</tbody>
</table>
| Delete obsolete S3 data | • CloudWatch Logs  
• Old versions of Lambda functions |

### 4.7 Serverless applicability

Serverless architecture is on the way to finding it is niche. Startups are fully adopting serverless to develop a new product quickly, and even traditionally conservative organizations are trying it attempting to become agiler and reduce costs. However, applicability and limitations of serverless are still not fully comprehend. Cloud providers present how serverless can be used mostly by use cases. The drawback of it is that use cases do not help to understand when not to use this architecture.

In this section, we review how serverless architecture is used, identify characteristics when it is used and define criteria when serverless architecture is applicable.

---

4.7.1 Use cases
The following sections present some of the most common use cases of serverless architecture.

Web applications
Web applications can be built relying only on AWS serverless stack. Front-end code and any static content can be hosted in the S3 object storage, API Gateway routes user requests to Lambda function where they are processed, and DynamoDB can be used to store data persistently. Additionally, adding AWS CloudFront CDN would improve access speed by caching content and distributing it closer to a user. The illustration of this architecture is presented in Figure 14.

![Figure 14 High-level architecture of a serverless web application](image)

Mobile backend and APIs
A mobile backend is similar to Web applications. A mobile application calls directly to API Gateway which routes request and executes lambda to process it. Lambda communicates with managed services like Dynamo DB to store data and SNS to push notifications. Figure 15 displays this architecture at a high level.

![Figure 15 High-level architecture of serverless mobile backend](image)

Data processing
AWS Lambda can be used to process various type of data. For example, Figure 16 depicts architecture of images processing pipeline. A picture is taken and uploaded to S3, S3 triggers Lambda function which extracts metadata of image and stores it to DynamoDB.

![Figure 16 High-level architecture of a serverless image processing](image1)

IoT generated data can be filtered by AWS IoT rules engine and processed by different Lambda functions, such use-case displayed in Figure 17.

![Figure 17 High-level architecture of serverless IoT data processing](image2)

Real-time streaming data for log analysis, activity tracking, data transformation, etc. can be collected and processed using Amazon Kinesis. Amazon Kinesis streams ingest data and trigger AWS Lambda passing data in size of the predefined batch.

**Automatization**

The serverless stack can be used in automating processes, like in Figure 18. When an error or event reaches CloudWatch, it can generate an Alarm to SNS topic, and SNS topic triggers a lambda function which takes countermeasures and sends a notification.

![Figure 18 High-level architecture of serverless automated provisioning](image3)

Another use case is scheduled tasks, for example, lambda function can be triggered each day to do an automated backup, clean up the filesystem, generate a report and similar tasks which runs from time to time.
4.7.2 Characteristics of serverless use-cases
This section defines characteristics of Serverless architecture based on previously mentioned use-cases and limitations of a serverless platform.

4.7.2.1 Bursty workloads, highly available and scalable
Serverless is an appealing choice to handle dynamic and irregular workloads because it scales automatically and rapidly. The cost is calculated per execution time, so if there is no traffic, no computation costs occur. This cost model is preferable when an application is lightly used in general but sometimes usage peaks for some time, the peak can be once a day or several times per month, such as applications in development which are inactive during the night or registrations for a highly anticipated event when load bursts after tickets got released.

Since serverless functions are scalable, third party services used by the function must be scalable or at least have the capacity to handle thousands of concurrent requests.

When an application is heavily I/O bound, and the load is constant, it is worth to consider dedicated computation instance. When most of the execution time is waiting for I/O actions to finish it is more cost effective to perform those actions in parallel. AWS Lambda cost is calculated based on total execution time, hence all waiting time for input/output to finish is charged. Additionally, AWS Lambda functions suffer from a cold start. Depending on language startup times are of the order of seconds [19] and if it uses VPC time can increase 10+ seconds [20].

4.7.2.2 Higher level abstraction
Sometimes a higher-level abstraction is needed to combine smaller units. For instance, gather data from multiple microservices and expose part of it or use multiple services to implement some high-level feature. Lambda function is a good place for composition because by design it is a standalone deployable unit. A developer can implement new functionality using existing microservices without reaching out different teams who own microservices. Additionally, he can choose different language than the language used in implementing other microservices.

As an example, Netflix uses serverless functions like a higher level abstraction; it was described in a podcast with Vasanth Asokan [21]. He told that Netflix has Dynamic Scripting Platform which is a serverless-like platform built to be agiler. It allows altering the behavior of the application without a full-scale deployment. Frontend developers at Netflix use the platform to write their scripts to implement simple API functionality instead of communicating with backend team which owns the microservice to get a new API added, or an API changed. That allows easy deployment and reduces time spent on communication and planning of changes.

Such usage of serverless functions is convenient however it couples the system, and a business logic begins to be interdependent. Therefore, a system can become “distributed monolithic” which is harder to maintain and fragile.

4.7.2.3 Fine-grained
Serverless functions tend to be more granular than microservices. Microservice has a single purpose and function has just a single action. Granular service, focused on a specific action, is easier to understand, change, replace and test and deploy compared to heavy service which has dependencies. Serverless functions create hard boundaries between services, therefore, coupling them becomes costly either in implementation or usage (the longer execution time is, the higher is the cost). Thus, cutting corners due to time pressure becomes not very attractive and serverless can lead to more clean architecture.

Fine-granularity kicks back when working with complex systems because the cognitive load is greater to reason about interactions of many small functions than several larger functions. Additionally, interacting services increase traffic volume in a network which can cause congestion.

4.7.2.4 Share nothing
Serverless functions are executed in containers which isolate processes running inside. Each container has own memory, ephemeral storage and runs limited time, therefore there is no stable way to know how many containers are running and connect to them.

4.7.2.5 Small number of dependencies or small in size
Serverless functions are stateless and ephemeral. Additionally, they have various limitations which make building large application tricky. For instance, AWS Lambda limits deployment package size to 50MB, and it should not exceed 250MB after extraction. The bigger deployment package is, the longer cold start takes. Moving data in is also restraining because request payload size is up to 6 MB when an invocation is synchronous and 128K when it is asynchronous. Moreover, functions have limits in execution time and consumption of resources. AWS Lambda has the limit of 900 seconds of maximum execution duration per request and access to 500MB temporary disk space [22]. Finally, download a large set of data is not efficient because cost depends on execution time and network I/O is slow comparing to CPU executions. Therefore, most of the time would be spent on waiting for data.

4.7.2.6 Vendor lock-in
In a serverless context, vendor lock-in is more like a tradeoff than limitation. Every cloud offers similar base services (network, computation, storage) the main difference is functionality. Therefore, it's a choice how much to depend on a serverless platform. Choosing to use serverless services lock-in can occur at the API and service levels. API level lock-in happens when the application code heavily relies on the interface of a vendor's infrastructure. However, the solution to this problem is well known - wrap a 3rd party service. There also are products such as zenko.io, Serverless Framework to avoid lock-in.
Service level lock-in happens when there are no alternatives which provide similar functionality. For example, S3 supports multipart upload, and if an application uses this functionality, the application needs to be modified to have multipart upload logic in order to migrate to Google Cloud Storage. Also, migration can be another way around in that case application logic will become smaller. In most cases, it’s not worth trying to avoid service level lock-in unless you already know that migration to another provider will be necessary.

4.7.2.7 Additional exposure to risks / reduced overall control
Going serverless requires complete trust in providers because they are responsible for protecting, updating and maintaining an underlying infrastructure. 99.999% of times providers take care of infrastructure better than clients could do themselves on-prem. In general, serverless platforms are updated continuously without service disruptions however accidents happen [23] due to force majeure [24] and human-errors [25]. Human-errors can't be avoided on-prem, but an update could be scheduled on most appropriate time, and at least it's known when things might get awry. In case of unpredictable events, it depends on the situation and nature of an application. For example, it can be a hospital management system. If it is hosted in a cloud and outage happens there, new patients cannot register, and work stops at a hospital because patients’ data cannot be accessed. On the other hand, if power is lost at the hospital, no one cares about management system because there are bigger problems than registration of new patients. However, in case of flood, a cloud provider has better chances to secure the data and continue operations.

When using serverless services, it’s worth taking into account that you can be punished for other customers faults. For example, service X could be inaccessible from some AWS IPs due to the abuse of other customers. Also, services hosted on the cloud could become inaccessible, as it happened in Russia when Google and AWS IPs were blocked in Telegram crackdown [26]. The bright side is that providers are interested in helping to solve and mitigate such issues. And sometimes you can keep abusing third-party services because blocking AWS would be too much for them ;)

4.7.3 Applicability checklist
Based on the information above the Table 4 Applicability checklist for Serverless architecture was prepared to help to determine if an application should be built using serverless technologies.

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic workloads</td>
<td>A predictable load is cheaper and faster to process using traditional architecture.</td>
</tr>
<tr>
<td>Idempotent operations</td>
<td>A request/event is retried in case of failure; therefore operations triggered by those must be idempotent</td>
</tr>
<tr>
<td>Costs hard to predict / can fluctuate</td>
<td>Predicting costs is hard for complex applications because services have different pricing models and most of the time load is unknown.</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>

**Time constraints**

<table>
<thead>
<tr>
<th>Latency-insensitive</th>
<th>Cold start adds ~1s, VPC ~10s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short running (up to 900 s)</td>
<td>Maximum execution duration per request is 900 seconds</td>
</tr>
</tbody>
</table>

**Data constraints**

<table>
<thead>
<tr>
<th>No more than 3008 MB RAM</th>
<th>Memory can be allocated with 64 MB increments starting from 128 MB, up to 3008 MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>No more than 512 MB disk space</td>
<td>Ephemeral disk capacity is 512 MB</td>
</tr>
<tr>
<td>Binary size less than 250 MB</td>
<td>Compressed deployment package size should not exceed 50 MB and uncompressed up to 250 MB</td>
</tr>
<tr>
<td>HTTP request / response less than 6 MB</td>
<td>Body payload size for synchronous Lambda requests is limited to 6 MB. Bigger files can be managed using S3.</td>
</tr>
</tbody>
</table>

**Third party services/services on premises used by Lambda**

<table>
<thead>
<tr>
<th>Scalable/able to handle thousands of requests</th>
<th>Services used in Lambda are scalable and by default 1000 functions can run simultaneously.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency/bandwidth insensitive</td>
<td>Calls to services outside AWS take longer and transferring data from/to AWS costs.</td>
</tr>
<tr>
<td>Vendor lock-in is acceptable</td>
<td>The serverless architecture relies on managed services, therefore, the application is tightly coupled with services provided by the platform.</td>
</tr>
</tbody>
</table>

**Runtime constrains**

<table>
<thead>
<tr>
<th>Direct access is not needed</th>
<th>No customizations of OS is needed; also the code is hardware-agnostic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Share nothing</td>
<td>The runtime environment is short-lived, and local changes are not durable</td>
</tr>
</tbody>
</table>
5 Discussion

At first glance serverless seems to be trivial because it is described by familiar terms however, they have a slightly different meaning. For example, stateless function in the context of serverless computing actually is short-lived stateful execution of any code (the function can contain a whole application) and amount of allocated memory impacts CPU performance. Additionally, calling backend services such as data stores and databases directly from a client is a good practice. Such odd architecture gives nearly unlimited scaling capabilities, and it costs depends only on the time and resources used. Serverless platforms solve difficult parts of distributed systems, such as reliability, scaling, consistency, and observability, exposing to users just valuable functionality in the form of managed services. This thesis provides an introduction to serverless section to understand serverless and Application Programming model section to help start working with it, for example, serverless design patterns show how to connect those services into a coherent system.

5.1 Development and operations

Development is harder in serverless architecture comparing to microservice architecture due to additional latency, fragmented documentation, reduced observability and lack of debugging tools and management flexibility. However, available managed services save a significant amount of time because you don’t need nor implement functionality nor run them. Unit testing tends to be easier compared to the traditional environment because a function has clear inputs and outputs. End-to-end testing is more accessible and it’s easy to launch a testing environment identical to production because the infrastructure is described as code and it's cheap since a customer pays only when tests are running. Benefits of operations outweigh all hurdles of development. Development starts with a production-ready environment which is robust, scalable, highly-available and publicly accessible. Also, advanced deployment models such as blue/green and canary deployments can be ready in a matter of hours instead of months on own infrastructure.

5.2 Tools

Tooling is still an issue in serverless architecture, especially for monitoring and security. Observability tools provided by the serverless platform aren’t as advanced as third-party tools available in the market since there is no direct integration with different logging systems, logs need to be preprocessed and transferred, thus additional lag and costs occur. IDEs for serverless is in early phase of development. On the other hand, deployment tools are really good and mature, tools like AWS SAM and The Serverless framework allow to describe complete application model with additional resources, can prepare serverless infrastructure in a platform and deploy an application without the need of user input.
5.3 Security and observability

Security awareness is still developing in the serverless environment. In general, security practices are the same as in traditional architectures, and most widespread security threats occur due to misconfiguration. Serverless providers allow fine-grained access control and that enables implementation least privilege. However, a large number of resources, policies, services hinder the ability to monitor and make sense of all of the data. For example, security policies can be applied per function (AWS Lambda), object (S3), item and attribute (DynamoDB). Small and medium applications implemented using serverless architecture are very convenient to work with but to manage bigger application can be challenging.

5.4 Pricing

The financial aspect of serverless is tricky. On the one hand, you can access a detailed billing report and get a clear view of current operational costs, and on the other, it’s tedious work predicting scaling costs when managed services are involved because there are multiple pricing models, such as execution time, allocated memory, a number of requests, bandwidth, a number of state transitions, and more. Serverless computing is cheaper than virtual machines when working with unpredictable workloads. Also, when the workload is known and can be parallelized, serverless computing can be used to reduce overall processing time due to high concurrency but costs would be higher than processing on VMs. From personal experience, I have noticed that it is better to observe costs and optimize when needed because based on scale different services are cheaper to use and most of the time costs are much smaller than predicted.

5.5 Transparency

The need for this thesis project is partly due to the lack of transparency from serverless providers. They publicly disclose just general characteristics of serverless leaving skew between how services are marketed and how they actually work. It’s understandable that they are guarding commercial secrets but those implementation/optimization details affect how applications are built in serverless architecture. Also, providers tend to keep in dark about new services and that causes re-thinking about the architecture of existing serverless applications after each AWS re:Invent, Google I/O, etc. They do not fully disclose changes in services and historical information about outages. Serverless is a platform which requires fully depend on provider, therefore, more transparency is necessary and community should be more vocal about it.

5.6 Conclusions

Serverless architecture is a great choice for dynamic workloads. The cold-start impact can be mitigated by allocating more resources and focusing on the runtime layer. Deployment package size and a number of files don’t affect significantly.
The development process has various disadvantages, but operational capabilities outweigh them. Lack of transparency from serverless providers is a subject that isn't discussed enough. Serverless is easy to learn but difficult to master due to peculiar solutions, large scope, and fast evolution. Despite the challenges, the future of serverless is bright because serverless providers offer high-quality, low-cost services for building applications and continue improving services based on feedback provided by customers.

5.7 Future work

Further work is needed to find a better solution for handling logs produced by the serverless stack. Standardized logging interface would be the best outcome, but it depends only on serverless providers. Additional case-study is needed to explore and evaluate existing solutions to observe serverless applications. Since serverless functions are fine-grained, many functions use the same dependencies, and they need to be in sync. A case study is needed to determine the best approach to structure and share code written in interpreted languages. Lift and Shift to a serverless environment is not possible due to the execution model, limited runtimes and execution time. The only method is to re-architect a legacy system. There are various types of systems which might have different re-architecting strategies, therefore, this topic deserves separate research. This thesis focused on AWS serverless platform but there are multiple open source platforms. It would be worth to look into state and compare most mature open source serverless platforms.
References


Appendix A

Examples of framework templates

SAM
AWSTemplateFormatVersion: 2010-09-09
Transform: AWS::Serverless-2016-10-31

Resources:
ExampleJavaFunction:
  Type: AWS::Serverless::Function
  Properties:
    Handler: com.example.HelloWorldHandler
    CodeUri: ./target/HelloWorld-1.0.jar
    Runtime: java8
  Environment:
    Variables:
      STAGE: prod
  Events:
    HelloApi:
      Type: Api
      Properties:
        RestApiId: !Ref ApiGatewayApi
        Path: /hello
        Method: GET

The Serverless Framework
service: aws-java
provider:
  name: aws
  runtime: java8

functions:
ExampleJavaFunction:
  environment:
    STAGE: prod
  package:
    artifact: ./target/HelloWorld.zip
  handler: index.handler
  events:
    - http:
      path: hello
      method: get
Appendix B

Example of script used for performance testing

```javascript
const util = require('util');
const fs = require('fs');
const {exec} = require('child_process');
const sizes = ['size1', 'size10', 'size25', 'size40'];

async function deploy() {
  const { stdout, stderr } = await exec('sls deploy');
  if (stderr) {
    console.error('error: ' + stderr);
    throw new Error(stderr);
  }
}

async function invoke(fnc) {
  const { stdout, stderr } = await exec(`sls invoke -d '()' -f ${fnc}`);
  if (stderr) {
    console.error('error: ' + stderr);
    throw new Error(stderr);
  }
  const res = stdout.split('Duration: ')
    .map(ell => ell[0]);
  return res;
}

async function measure() {
  await deploy();
  const timings = await Promise.all(sizes.map(invoke));
  const str = timings.join(', ');
  fs.writeFileSync('results_102k.csv', `${timings}[${f}], {'flag': 'a'}}`);
  return str;
}

async function main() {
  for(let i = 0; i < 100; i++) {
    const timings = await measure();
    console.log('i: ' + timings);
  }
}

main();
```

Package.json used for performance testing

```json
{
  "name": "coldstar-1024-deps",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "license": "ISC",
  "dependencies": {
    "aus-nki": "2.32.0",
    "babel-core": "5.25.2",
    "bluebird": "3.5.0",
    "body-parser": "1.17.2",
    "bower": "1.8.0",
    "chalk": "1.1.3",
    "cheerio": "1.0.0-rc.1",
    "class-validator": "2.2.5",
    "co": "4.6.0",
    "commander": "2.9.0",
    "express": "4.15.3",
    "es": "1.1.1",
    "glob": "7.1.2",
    "googleapis": "3.8.2",
    "grunt": "1.0.1",
    "gulp": "3.9.1",
    "gulp-util": "3.0.1"
  }
}
```
### Results of performance testing

Table 5 Different package and memory sizes, one dummy file

<table>
<thead>
<tr>
<th></th>
<th>std dev</th>
<th>avg</th>
<th>median</th>
<th>Min</th>
<th>Max</th>
<th>65%-tile</th>
<th>80%-tile</th>
<th>95%-tile</th>
<th>99%-tile</th>
</tr>
</thead>
<tbody>
<tr>
<td>128mb-1mb</td>
<td>18.94</td>
<td>23.16</td>
<td>18.03</td>
<td>2.06</td>
<td>130.02</td>
<td>20.93</td>
<td>36.79</td>
<td>58.99</td>
<td>98.95</td>
</tr>
<tr>
<td>128mb-10mb</td>
<td>18.88</td>
<td>24.39</td>
<td>18.19</td>
<td>1.78</td>
<td>80.96</td>
<td>23.26</td>
<td>40.03</td>
<td>59.29</td>
<td>62.61</td>
</tr>
<tr>
<td>128mb-25mb</td>
<td>18.03</td>
<td>28.83</td>
<td>27.43</td>
<td>1.88</td>
<td>112.31</td>
<td>33.99</td>
<td>39.91</td>
<td>60.61</td>
<td>71.27</td>
</tr>
<tr>
<td>128mb-49mb</td>
<td>16.16</td>
<td>24.12</td>
<td>27.40</td>
<td>1.81</td>
<td>96.42</td>
<td>31.47</td>
<td>35.32</td>
<td>40.34</td>
<td>94.72</td>
</tr>
<tr>
<td>256mb-1mb</td>
<td>7.72</td>
<td>8.14</td>
<td>3.79</td>
<td>1.87</td>
<td>38.59</td>
<td>8.19</td>
<td>14.84</td>
<td>21.94</td>
<td>27.37</td>
</tr>
<tr>
<td>256mb-10mb</td>
<td>8.39</td>
<td>7.82</td>
<td>4.29</td>
<td>1.79</td>
<td>55.84</td>
<td>7.61</td>
<td>14.31</td>
<td>19.60</td>
<td>39.16</td>
</tr>
<tr>
<td>256mb-25mb</td>
<td>7.05</td>
<td>9.68</td>
<td>8.45</td>
<td>1.73</td>
<td>39.69</td>
<td>13.10</td>
<td>15.65</td>
<td>19.64</td>
<td>25.60</td>
</tr>
<tr>
<td>512mb-1mb</td>
<td>3.18</td>
<td>3.71</td>
<td>2.80</td>
<td>1.55</td>
<td>20.60</td>
<td>3.82</td>
<td>4.22</td>
<td>14.54</td>
<td>19.69</td>
</tr>
<tr>
<td>512mb-10mb</td>
<td>3.24</td>
<td>3.60</td>
<td>2.73</td>
<td>1.76</td>
<td>21.50</td>
<td>3.46</td>
<td>3.78</td>
<td>11.73</td>
<td>15.13</td>
</tr>
<tr>
<td>512mb-25mb</td>
<td>3.58</td>
<td>3.47</td>
<td>2.69</td>
<td>1.81</td>
<td>20.09</td>
<td>3.20</td>
<td>3.28</td>
<td>11.42</td>
<td>19.49</td>
</tr>
<tr>
<td>512mb-49mb</td>
<td>2.40</td>
<td>2.83</td>
<td>2.34</td>
<td>1.68</td>
<td>19.91</td>
<td>2.59</td>
<td>2.65</td>
<td>6.35</td>
<td>13.07</td>
</tr>
<tr>
<td>1024mb-1mb</td>
<td>2.73</td>
<td>3.46</td>
<td>2.66</td>
<td>1.67</td>
<td>24.70</td>
<td>3.41</td>
<td>3.93</td>
<td>6.69</td>
<td>15.01</td>
</tr>
<tr>
<td>1024mb-10mb</td>
<td>2.65</td>
<td>3.27</td>
<td>2.89</td>
<td>1.80</td>
<td>20.18</td>
<td>3.17</td>
<td>3.69</td>
<td>7.39</td>
<td>17.88</td>
</tr>
<tr>
<td>1024mb-25mb</td>
<td>1.18</td>
<td>2.72</td>
<td>2.48</td>
<td>1.69</td>
<td>11.12</td>
<td>2.62</td>
<td>2.81</td>
<td>4.96</td>
<td>8.03</td>
</tr>
<tr>
<td>1024mb-49mb</td>
<td>0.58</td>
<td>2.56</td>
<td>2.41</td>
<td>1.59</td>
<td>19.21</td>
<td>2.57</td>
<td>2.80</td>
<td>3.28</td>
<td>4.85</td>
</tr>
<tr>
<td>3008mb-1mb</td>
<td>0.21</td>
<td>2.08</td>
<td>1.94</td>
<td>1.62</td>
<td>3.97</td>
<td>1.97</td>
<td>2.22</td>
<td>2.14</td>
<td>2.42</td>
</tr>
<tr>
<td>3008mb-10mb</td>
<td>0.32</td>
<td>2.35</td>
<td>2.01</td>
<td>1.61</td>
<td>7.08</td>
<td>2.04</td>
<td>2.63</td>
<td>2.29</td>
<td>2.58</td>
</tr>
<tr>
<td>3008mb-25mb</td>
<td>0.30</td>
<td>2.65</td>
<td>2.70</td>
<td>2.07</td>
<td>4.75</td>
<td>2.77</td>
<td>2.80</td>
<td>3.03</td>
<td>3.54</td>
</tr>
<tr>
<td>3008mb-49mb</td>
<td>0.34</td>
<td>2.27</td>
<td>2.25</td>
<td>1.75</td>
<td>5.01</td>
<td>2.30</td>
<td>2.40</td>
<td>2.50</td>
<td>2.97</td>
</tr>
</tbody>
</table>

Table 6 Big package, lots of small files, modules are skipped/loaded

<table>
<thead>
<tr>
<th></th>
<th>std dev</th>
<th>avg</th>
<th>median</th>
<th>Min</th>
<th>Max</th>
<th>65%-tile</th>
<th>80%-tile</th>
<th>95%-tile</th>
<th>99%-tile</th>
</tr>
</thead>
<tbody>
<tr>
<td>modules loaded (44mb)</td>
<td>24.98</td>
<td>66.95</td>
<td>72.45</td>
<td>2.05</td>
<td>133.88</td>
<td>75.17</td>
<td>77.97</td>
<td>89.52</td>
<td>112.60</td>
</tr>
<tr>
<td>modules skipped (44mb)</td>
<td>3.62</td>
<td>3.23</td>
<td>2.52</td>
<td>1.66</td>
<td>36.03</td>
<td>2.73</td>
<td>3.15</td>
<td>7.35</td>
<td>9.24</td>
</tr>
<tr>
<td>one big file (49mb)</td>
<td>0.58</td>
<td>2.46</td>
<td>2.41</td>
<td>1.59</td>
<td>5.21</td>
<td>2.57</td>
<td>2.90</td>
<td>3.28</td>
<td>4.85</td>
</tr>
</tbody>
</table>

Table 7 Different package, allocated 512MB memory sizes, warm function

<table>
<thead>
<tr>
<th></th>
<th>std dev</th>
<th>avg</th>
<th>median</th>
<th>Min</th>
<th>Max</th>
<th>65%-tile</th>
<th>80%-tile</th>
<th>95%-tile</th>
<th>99%-tile</th>
</tr>
</thead>
<tbody>
<tr>
<td>1mb</td>
<td>3.18</td>
<td>3.71</td>
<td>2.80</td>
<td>1.55</td>
<td>20.60</td>
<td>3.82</td>
<td>4.22</td>
<td>14.54</td>
<td>19.69</td>
</tr>
<tr>
<td>10mb</td>
<td>3.24</td>
<td>3.60</td>
<td>2.73</td>
<td>1.76</td>
<td>21.50</td>
<td>3.46</td>
<td>3.78</td>
<td>11.73</td>
<td>15.13</td>
</tr>
<tr>
<td>25mb</td>
<td>3.58</td>
<td>3.47</td>
<td>2.69</td>
<td>1.81</td>
<td>20.09</td>
<td>3.20</td>
<td>3.28</td>
<td>11.42</td>
<td>19.49</td>
</tr>
<tr>
<td>49mb</td>
<td>2.40</td>
<td>2.83</td>
<td>2.34</td>
<td>1.68</td>
<td>19.91</td>
<td>2.59</td>
<td>2.65</td>
<td>6.35</td>
<td>13.07</td>
</tr>
</tbody>
</table>