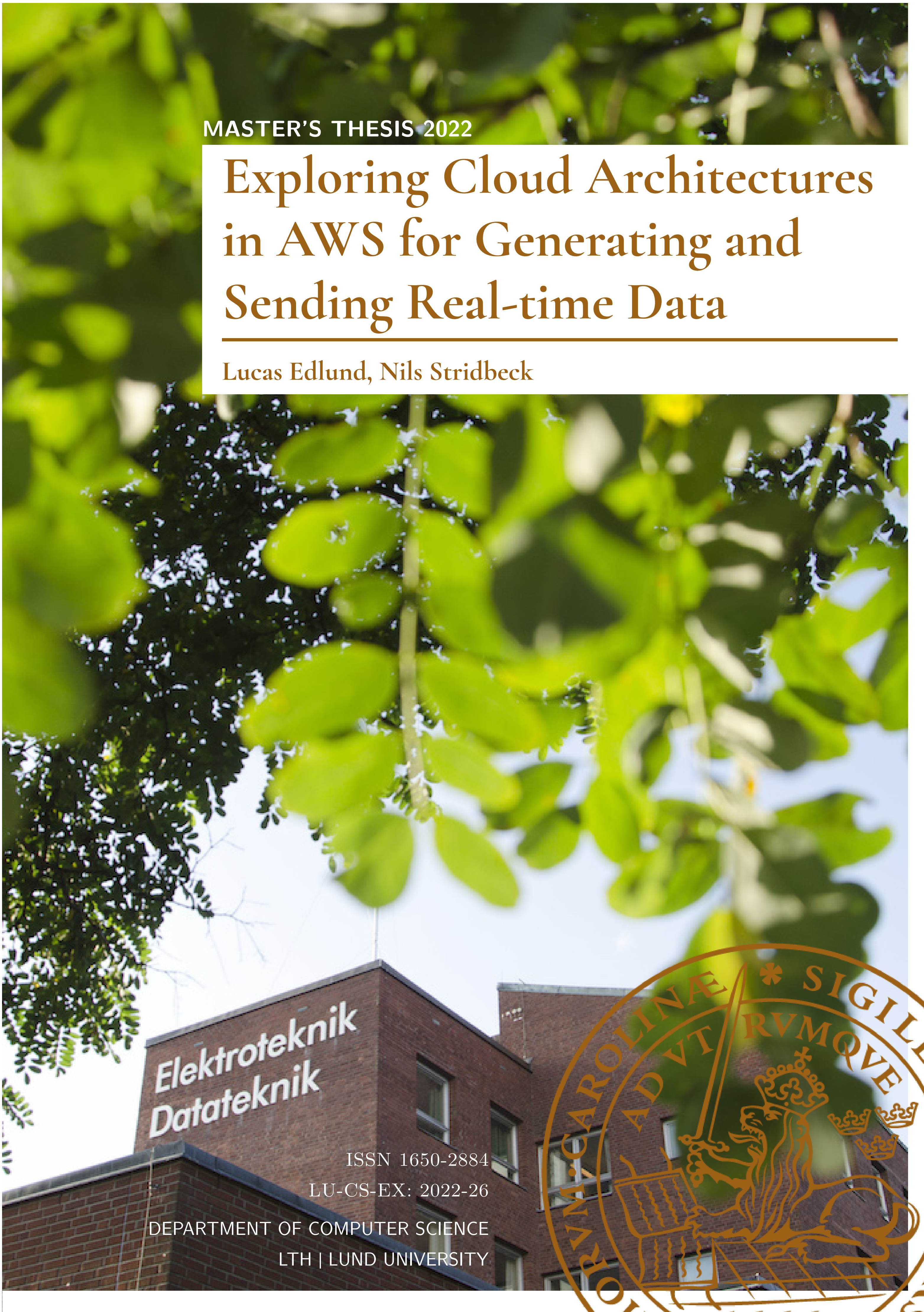


MASTER'S THESIS 2022

Exploring Cloud Architectures in AWS for Generating and Sending Real-time Data

Lucas Edlund, Nils Stridbeck

ISSN 1650-2884
LU-CS-EX: 2022-26

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2022-26

Exploring Cloud Architectures in AWS for Generating and Sending Real-time Data

Utforskandet av molnarkitekturer i AWS för generering och skickande av realtidsdata

Lucas Edlund, Nils Stridbeck

Exploring Cloud Architectures in AWS for Generating and Sending Real-time Data

Lucas Edlund
lu6512ed-s@student.lu.se

Nils Stridbeck
ni8280st-s@student.lu.se

June 9, 2022

Master's thesis work carried out at Tactel AB.

Supervisors: Magnus Wihlborg, Magnus.wihlborg@tactel.se
Per Andersson, per.andersson@cs.lth.se

Examiner: Ulf Asklund, ulf.asklund@cs.lth.se

Abstract

Several different types of architectures can be used to build a performant, scalable, and cost-effective system for delivering real-time data to planes. This paper compares three architectures built in AWS for handling such a task. Our findings show that a monolithic approach with EC2s and a load balancer gives a faster but slightly more expensive system for a low amount of planes. Our two microservices systems were slower but cheaper than the monolith. With more planes, the monolith falls behind the synchronous microservice system in speed but is still more cost-effective. The asynchronous microservice system is the slowest of them all and has the highest cost when the number of planes increases. It is, however, the easiest system to extend.

Keywords: MSc, IoT, AWS, simulation, system architecture, cloud computing

Acknowledgements

We would like to start by thanking some people that have helped us to accomplish this work. This project could never have happened without big help from Tactel who gave us the tools, budget, and delicious healthy snacks to build all three systems.

A special thanks to our supervisor from Tactel Magnus Wihlborg. You have been a constant support throughout the entire process and have encouraged us to the end.

We would also like to thank our supervisor from LTH Per Andersson. He has guided us through this journey and been a steady source of great tips.

Finally, we want to thank Amir Popaya. Always a source of inspiration and laughter.

Contents

Chapter 1

Introduction

In most successful software projects, the expert developers working on that project have a shared understanding of the system design. This shared understanding is called 'architecture'.

Martin Fowler

In an age of technology all industries are adapting their current business models to attract more customers. Technology is used to ease the customer experience and make it overall more enjoyable. The aviation business is no different and different types of technology have been introduced both before but also during flights to enhance the passengers experience. There are now self-service check-in stations and many flights now offer WiFi and entertainment options.

Tactel is a human-centered digital interaction agency that primarily works with airlines. They create software to enhance the flight experience for passengers and work with some of the largest airlines in the industry. Tactel's latest product Arc is an interactive high-definition map application for airlines. Through an in-flight tablet, passengers can see where the airplane is in the world and learn more about the different landmarks they are flying past.

1.1 Problem definition

At this moment Tactel has no system in place to support generating and sending real-time data to the airplanes mid-flight. All information about different landmarks is pre-loaded before the flight takes off and is not updated during flights. This thesis explores the possibility of sending data to the airplanes in the form of news and weather during its flight. The data sent to the airplane should be based on the airplane's geographic position in the world and the system will be built in AWS. Arc is a relatively new product and Tactel wants to quickly let new airlines connect their fleet of airplanes to it. The system must be highly scalable to handle all of these potential customers. In order to deliver data to the airplane, Tactel will use an existing antenna onboard the airplane. While this antenna enables a stable way of communicating with the airplane, data transfer to airplanes can prove costly and we therefore do not want to overuse the antenna.

1.1.1 Requirements

The requirements outlined below were a joint effort between Tactel and us.

- The system shall deliver real-time data consisting of weather and news relevant to the nearby cities of the airplane
- The airplane's coordinates and direction shall decide the relevant cities
- The relevant cities shall be chosen from the cities available in Arc
- The systems shall be built in AWS
- The data sent to the airplane shall be kept at a minimum

1.1.2 Research questions

From the requirements we identified three research questions we believed would be interesting to answer with this thesis. The questions aim to analyze different types of AWS architecture based on three different areas.

1. Which AWS architecture and resources will provide the most efficient way to supply real-time data to the airplanes?
2. What practices can be used to scale the different solutions and how well do the different AWS architectures scale?
3. Which AWS architecture is the most cost-effective and how can AWS practices be utilized to minimize the overuse of resources?

With regards to efficiency, we will be looking at the performance of each system i.e which system is the fastest. We want the systems to be able to handle different amounts of airplanes sending and receiving data so with regards to scalability we will examine how well the three systems can adapt to changing workloads. With regards to cost, we will be looking at what each system would cost each month if kept running. Since different workloads will affect the price we will examine how costs change when scaling as well.

1.2 Contribution Statement

The division of the workload was between the two authors of this thesis Nils Stridbeck and Lucas Edlund. Both Nils & Lucas have been active in development of the three systems that were built but Lucas had a strong focus on the monolithic system while Nils focused on the microservice systems.

The writing portion of the workload was divided where each author wrote the following parts:

- Nils: 1.2, 1.3, 1.3.1, 1.4, 2.2, 3.2, 4.1, 4.2, 4.4, 5.1, 5.2, 5.4
- Lucas: Abstract, 1.1, 1.3.2, 1.3.3, 1.5, 2.1, 2.3, 2.4, 3.1, 3.3, 4.3, 5.3

Conclusions and proofreading of all parts of the thesis was made by both of the authors.

1.3 Software architecture

Software architecture does not have a proper definition but one can see it as simply the organization of a system. This system includes all components, how they interact, the environment in which they operate, and the principles used to design the system. In many cases, it can also include the evolution of the software into the future.

Software architecture is designed with a specific mission in mind. It has a big impact on the behavior and structure of the final system so it is important to spend an appropriate amount of time on the design process. Software architecture is also tied closely to other disciplines and communities, such as software design (in general), software reuse, systems engineering and system architecture, enterprise architecture, reverse engineering, requirements engineering, and quality [8].

Software architecture is important for a the success of a system. It acts as a sort of plan of the system and it facilitates the understanding, the negotiation, and the communication between all the stakeholders of what the system should be and how it should look like. A lot of decisions need to be made early in the system development process and the act of software architecture is considered one of these important decisions. It ensures the system will be able to exhibit its desired behaviour. For example, if you believe that scalability will be important to the success of your system, then you need to facilitate introduction of higher-capacity replacements, and you must avoid hard-coding in resource assumptions or limits. A good architecture is necessary to ensure quality [3].

1.3.1 Brief History

Back when systems were comparatively small, you would draw abstract diagrams to understand designs and communicate them to others. These small systems did not always need abstractions to help in the design process since simplicity ruled all. The designs would be placed on the office wall and later on in a project you would find that what was on the wall had a limited relationship to what was actually being done. There was no need to update an architectural diagram because everybody in the team understood what was actually being built. Very soon, however, systems grew in size and ambition and structured descriptions became essential[2].

The first reference to the phrase software architecture occurred in 1969 at a conference on software engineering techniques organized by NATO but it would not be until 1990 that the concept of software architecture as a distinct discipline started to emerge [9]. For developers at this time software design became an integral part of their work. Two commonly cited reasons included communicating with team members, and making the code more maintainable. At this time only software architecture was mostly common in large projects and it was more easily found to be found in documentation than in the source code. It was difficult to point to anything in the implementation that was the architecture. However, drawings or descriptions of architecture could be found in documents, and these in turn were created for communicating concepts and principles to team members [25].

The following years software architecture became an important tool in all system development projects, not just large ones. Architecture styles evolved as well, from simple monolithic systems to more complicated service oriented systems.

1.3.2 Monolith

The monolithic approach has the entire system running as a single program on the same hardware [10]. This program handles all of the necessary tasks and has no type of modularity in stark contrast to the microservice pattern. Older applications were all built as a monolith and as time progresses they are harder and harder to update. Making small changes to one part of the program has the potential of breaking a different part. Trying to update one part of the program requires the entire program to be taken off production until the update has been completed. For certain critical systems this is not a preferred way of working. This can lead to an outdated program where the updates are large and far between. The speed of a monolithic system can often be faster than a microservice one since no API calls are needed between the different parts of the program. Scaling is harder to tackle since everything is on the same hardware. If one part of the program needs a faster CPU the whole program gets a faster CPU. This can lead to some parts of the program having access to an excessive amount of resources which will remain mostly unused. The cost can therefore be higher than that of a microservice system.

1.3.3 Microservice Pattern

A microservice pattern is an architectural style that keeps services in the system separate from each other [10]. Each of the services can be built using different languages or technologies and the communication between them is done via API calls. Every service has a specific task and can scale independently from the other services to handle their specific loads [6]. One of the goals of microservices is loose coupling which is achieved when the services have no knowledge of the other services in the system and changing the implementation of a service doesn't affect the rest of the system. In a production environment one of the components could be taken down for a potential bug fix while the rest of the system stays up. This favors the style of rapid development where small deployments are made regularly. Security can be tricky while working with many different services. Instead of focusing on the security of one programming language running on one system there are multiple technologies on different systems running their own operating system. The communication between the components via internet also needs to be secured.

Orchestra

This type of architecture has a central brain that controls the rest of the system, similar to a conductor leading an orchestra. The conductor has the responsibility of invoking the other services and sending data between them [10]. This central brain can sometimes be too controlling and create tightly coupled systems that all rely on one "god" service and can lean more towards a monolithic system than a microservice system. The orchestra pattern usually uses synchronous communication between the components. The central brain sends an API request to one of the components and then waits for the response. This leads to slower execution but an easier to work with system since the central component knows if the requests was successful or not and can act accordingly.

Choreography

In a loosely coupled system, the components are detached from each other [10]. All components can work independently, as part of a larger group of systems, or in close concert with multiple segmented groups of systems. Ultimately, nothing is forced into a relationship with anything else, which delivers obvious benefits in extensibility and scalability. All of the components are informed of an event and they can work out the details themselves. A component does not wait for another one to finish and instead uses asynchronous communication to send information back and forth. This usually involves a queue or a pub-sub protocol. The name refers to how a group of dancers is reacting to others around them in ballet with no central entity telling them what to do.

1.4 Amazon Web Services

Amazon Web Services (AWS) is a subsidiary of multinational technology company Amazon providing on-demand cloud computing and is the leading cloud provider in the world[22]. AWS runs data centers throughout the world and allows you to avoid hosting your private cloud in servers that you own. Instead of paying upfront costs for server halls and hardware, AWS calculates fees based on a combination of usage (known as a "Pay-as-you-go" model), hardware, operating system, software, or networking features [16].

AWS has data centers in multiple regions in the world (such as Stockholm and Ohio) and each region usually has multiple availability zones. An availability zone is a single data center or a group of data centers within a region. The availability zones in a region are placed tens of kilometers apart which means if one data center would go offline (due to power outages or earthquakes), the other data centers will usually not have been impacted [16].

1.4.1 Amazon Elastic Compute Cloud (EC2)

Amazon EC2 provides secure, resizable compute capacity in the cloud, essentially running servers. AWS allows you to launch and stop EC2 instances and offers different hardware specifications and operating systems to choose from [19].

EC2 runs on top of physical host machines managed by AWS using virtualization technology. When you start an EC2 instance you are most likely not taking up an entire machine yourself. The host machine is instead shared with multiple other virtual machines. A *hypervisor*, also known as a virtual machine monitor or VMM, is software that creates and runs virtual machines. AWS uses a hypervisor, which runs on the host machine, to allow sharing of the underlying physical resources between the virtual machines. This idea of sharing underlying hardware is called *multitenancy*. The hypervisor is responsible for isolating the virtual machines from each other as they share resources from the host. This means that even though EC2 instances may be sharing resources, they are not aware of any other EC2 instances on that host. They are secure and separate from each other [16].

1.4.2 Lambda

While running virtual machines can be great for some use cases, sometimes you only want code to run without having to worry about the underlying infrastructure. With Lambda you upload your code into what is called a *Lambda function*. You thereafter configure a trigger for the function and when the trigger is detected, the code is automatically run in a managed environment [17]. This means you can run code without thinking about maintaining any servers or worrying about scalability. You upload your code and Lambda takes care of everything required to run and scale your code.

Memory Optimization

Lambda allows you to configure the amount of memory that is allocated to the function, from between 128 MB up to 10,240 MB. How much memory you choose will impact the performance of your function, where usually higher memory results in faster run time. The amount of memory also determines the amount of virtual CPU available to a function. Adding more memory proportionally increases the amount of CPU as well [5].

This impacts cost since double the amount of memory allocated translates to double the run time costs. This means that choosing the amount of memory you want to allocate to your lambda functions will require a cost-benefit analysis.

Performance Optimization

When a Lambda function is created, the code is stored in AWS cloud storage. When the Lambda service receives a request to run a function the service first prepares an execution environment. During this step, the service downloads the code for the function from the cloud storage and then creates an environment with the memory, run time, and configuration that you have specified. Once that is finished, the lambda service finally runs your code [4]. The first step of setting up the environment of your code is usually referred to as a *cold start*.

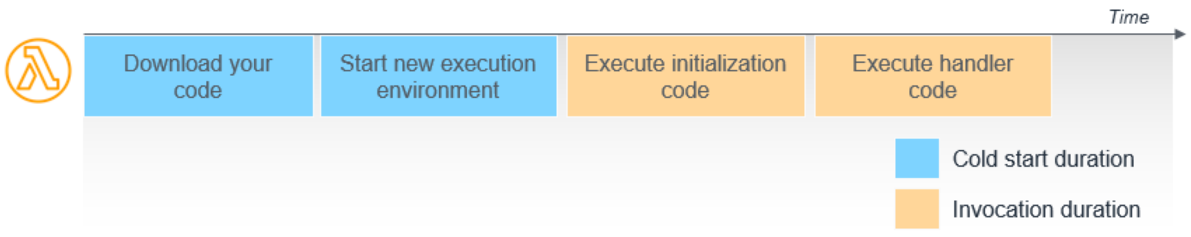


Figure 1.1: Image showing the lifetime of a lambda function that experiences a cold start.

The cold start is not included in the run time charge of the function but it will of course add latency to your overall run time.

As a step to minimize latency, the Lambda service will retain your execution environment for a non-deterministic period of time after your function has finished running. During this period if another request arrives for the same function, the service may reuse the same environment. This scenario is usually referred to as a *warm start* and will result in the second request finishing more quickly [4].

1.4.3 Simple Notification System (SNS)

SNS is a messaging service for asynchronous application-to-application communication between microservices in AWS [20]. It uses the pub/sub-messaging paradigm where a publisher can publish a message corresponding to a topic and all the subscribers who are subscribed to that topic will receive that message. Most AWS services can publish messages but only a small subset of them can receive them such as Lambda and external webhooks.

Each message sent via SNS will be in a JSON format and consists of an id, title, message body, topic, and timestamp. If a message is unable to reach its subscriber a series of retry attempts will begin and if those fail the message will either be discarded or placed in a dead-letter queue where it can be manually analyzed.

1.4.4 ElastiCache for Redis

Amazon ElastiCache for Redis is a fully managed Redis client that handles tasks such as configuration, updates, and backups. It can be used as either a cache or as an in-memory data store [14]. When used as a cache it sits between the client and the database in order to retrieve commonly accessed data from memory instead of having to query the database. This provides sub-millisecond operations and keeps the load off the database. When used as a data store it acts as its own database and mainly keeps the data in memory but can do periodical saves to a disk.

Redis is a key-value database where each entry consists of a unique key and a value of a type. The most common types are string, list, set, or hash. A hash is equivalent to a hashmap or a dictionary [11]. Each entry can be configured with a time to live (TTL) which decides how long an entry should exist in the database before it's automatically deleted [12].

1.4.5 Relational Database Service (RDS)

RDS is used to set up a relational database in AWS with support for all the major SQL database engines [15]. It manages installation, patching, and backups on its own. With RDS there are two instance types to choose from, General purpose and Memory-optimized. General-purpose databases offer a good balance of computing, memory, and network resources and are suitable for most tasks. Memory-optimized databases are used when many reads and writes are executed each second or when a higher memory cache is needed.

PostGIS

PostGIS is an extension for PostgreSQL which adds support for geographic data. Rows in PostgreSQL can now be queried based on their coordinate columns. Boundaries can be drawn up in PostGIS and select only those rows which contain coordinates within those boundaries.

1.4.6 Virtual Private Cloud

A VPC lets you provision a logically isolated section of the AWS Cloud where you can launch AWS resources in a virtual network that you define. These resources can be public-facing so they have access to the internet, or private with no internet access, usually for backend services like databases or application servers. The public and private grouping of resources are known as **subnets** and they are ranges of IP addresses in your VPC [16].

VPC allows you to secure and monitor connections, screen traffic, and restrict instance access inside your virtual network [21]. Restricting access to your AWS resources is crucial for your cloud environment since in most cases you want to control who has access to what resources.

1.4.7 Elastic Beanstalk

AWS Elastic Beanstalk is a service for deploying and scaling web applications and services developed with your programming language of choice on familiar servers such as Apache, Nginx, and others. Elastic Beanstalk automatically handles the deployment of your application on EC2 servers, from capacity provisioning, load balancing, and auto-scaling to application health monitoring.

You can configure Elastic Beanstalk to scale according to your needs. You can configure the minimum amount of EC2 you always want running as well as the maximum amount allowed. You choose which hardware specifications the servers should have and also what type of events should trigger the application to scale. Three different type of servers that Beanstalk can choose from are described below:

- t4g.nano (2 vCPU & 0.5 GB Ram)
- t4g.micro (2 vCPU & 1 GB Ram)
- t4g.small (2 vCPU & 2 GB Ram)

Elastic Beanstalk will choose a suitable hardware configuration and launch the server already pre-configured with your application and automatically adds them to the group of servers the application load is shared between [23].

1.4.8 API Gateway

The API Gateway is a service for creating and managing APIs [13]. They act as the entryway into applications in order to send and retrieve data. This service can handle hundreds of thousands of API calls each second and handles tedious tasks such as authorization and traffic management. An API Gateway can be attached to a Lambda function and automatically run that Lambda function when an HTTP request is received.

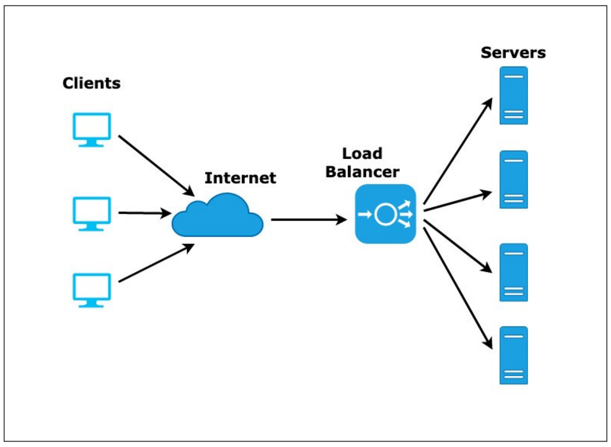
1.4.9 Amazon Cloudwatch

Amazon CloudWatch allows you to monitor your Amazon Web Services (AWS) resources and the applications you run on AWS in real-time. CloudWatch displays metrics about every AWS service you use. You can additionally create custom dashboards to display metrics about your custom applications and display custom collections of metrics that you choose.

Another feature of Cloudwatch is the ability to create alarms that watch metrics and send notifications or automatically make changes to the resources you are monitoring when a threshold is breached. You could, for example, monitor the CPU usage of your Amazon EC2 instances and then use that data to stop under-used instances to save money[18].

1.4.10 Elastic Load Balancer

When dealing with high loads of traffic to an application or service you generally would want to split the load between several servers. The problem is that when accessing a website, for example, you want users to be able to access it through one access point (IP address). Load balancing is the process of distributing network traffic across multiple servers. This ensures no single server bears too much demand.



Elastic Load Balancing is an AWS service that automatically distributes

your incoming traffic across multiple tar-
gets, such as EC2 instances. It monitors the health of its registered targets and routes traffic only to the healthy targets. Elastic Load Balancing scales your load balancer as your incoming traffic changes over time. It can automatically scale to the vast majority of workloads[24].

1.5 Related work

Albuquerque et al. compared the performance, scalability, and cost of two systems for handling geolocation data [1]. The systems were built with the microservice architecture in mind. One of them used Beanstalk while the other used Lambda functions. Their testing focused on how long it took to read and write to a database. Their findings show that the performance between the two systems was roughly the same and using higher-end hardware did not result in a proportionally faster system. The Beanstalk system was better at handling constant workloads while the Lambda system excelled at workloads with a varying number of requests each second. Beanstalk had a lower cost when the operations performed by the systems had a longer duration while the Lambda system was more cost-effective for shorter operations. They also experienced a reduced performance of the Lambda systems due to cold starts. These issues could be mitigated by restructuring the system to invoke the Lambda functions more often.

Faustino et al. looked at the effects of transforming a monolith system into a microservice system [7]. They measured the difference in performance between the two systems and how easy the migration process was. The results showed a decrease in performance in the form of higher latency and lower throughput. The new system however was easier to both scale and extend with new features.

Villamizar et al. made a cost comparison between a monolithic system running on an EC2 and a microservices system using Lambda functions [26]. Their testing setup consisted of three stress tests for both systems to determine how many requests each system could handle every second and from that calculate the monthly cost. Their findings show that the monolith system was more expensive than the microservices system and was also harder to manage.

Chapter 2
Approach

This chapter explains the approach used to answer our research questions. The first part explains the methodology used and gives insights into the three artefacts that were built during the thesis work. Lastly, the experimental setup is explained to show how we gained reliability and validity to our results.

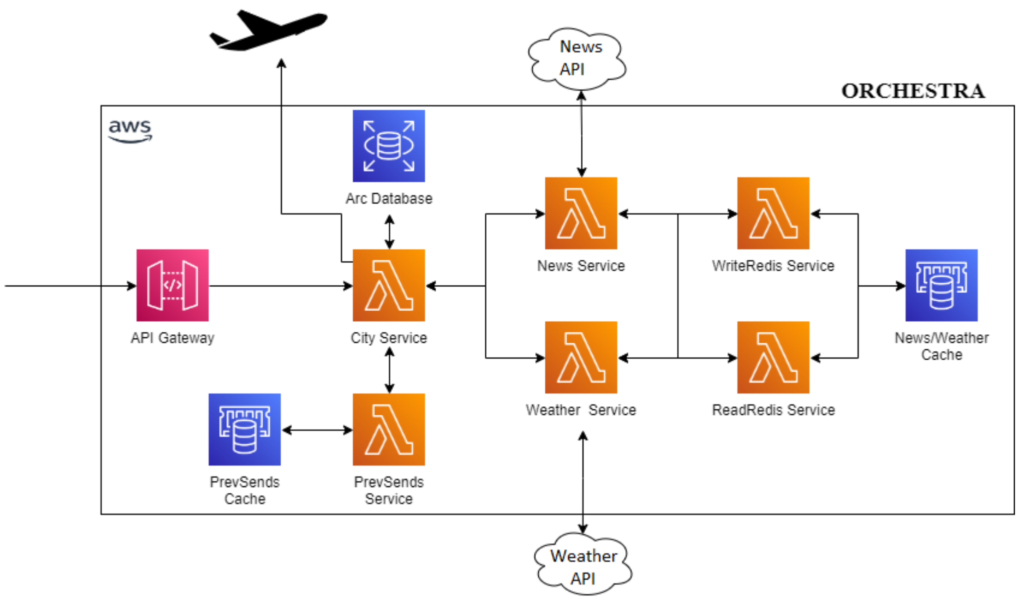
2.1 Method

To answer our research questions we aimed to compare different types of cloud architecture in AWS to measure how well they excelled in different areas. We analyzed the objectives of the system Tactel wanted to build and planned alternative solutions to reach the desired goal. This involved designing three different systems and identifying necessary AWS components for these solutions. The systems needed to be able to receive the coordinates and direction of an airplane and from that information generate relevant real-time data for cities the airplane was about to pass. We chose for the system to generate weather and news data since this seemed like data that would be interesting for passengers to see about the cities below them. We developed three artefacts in AWS that we could analyze. Once finished we evaluated each artefact by running simulations where the systems were fed location data from simulated airplanes. The simulations were performed with different numbers of airplanes to see the impacts of different types of loads on the system. From that, we could conclude where the different systems excelled and why.

2.2 Architectures

When choosing architectures we wanted to select three designs with different perceived benefits and drawbacks. From our literature study, we had identified the microservice design pattern as a powerful way of building scalable and flexible systems so we were interested in utilizing it. We decided it would be interesting to compare the microservice approach against a more traditional monolithic approach since the two approaches are considered opposites. In the end, we choose two different microservice approaches as well as a monolith approach as our three different system architectures.

2.2.1 Architecture 1 - Microservice Orchestra



This architecture operates synchronously with a central brain in charge of communicating with the other microservices. The system receives airplane data, such as airplane ID, coordinates, and direction, through an API Gateway. The city service lambda function will first determine which cities the airplane should receive news and weather for. It does this by fetching all relevant cities from the Arc database and thereafter filtering out cities the airplane has already received data for by communicating with our PrevSends cache. The remaining cities are thereafter passed in the request to our weather service and after getting a response it will send a similair request to the news service.

The news service and the weather service receive the cities it should generate data for. Before making an API request the services check the shared news & weather cache. If a city does not have weather or news data in the cache, the service will make an API request, and write the new data to the cache. The data from the cache and the new data is combined before returning it to the city service.

The city service finally joins the data from the news service and the weather service before sending it to the airplane.

AWS Resources

This system is highly dependent on AWS Lambda functions. We decided to use Lambda functions because of their event-driven property. We could structure our system such that each invocation of the API Gateway would trigger the city service and provide it with the relevant data. Lambda functions also provided functionality for triggering other lambda functions which were essential to get the system working as intended.

Potential benefits

This type of architecture results in a tightly coupled system where the city service will wait for the rest of the system to finish running before it can compile the result and send it to the airplane. An architecture of this type doesn't need a lot of outside monitoring since the central service knows when there are errors in other parts of the system.

This type of architecture also scales very well. Since the services are separated each service can scale independently from the rest of the system. This allows us to put more resources where necessary and avoid redundant resource use. For example, we predicted more heavy use of the read Redis service than the write Redis service. Our microservice approach allowed us to scale the services differently.

Allowing lambda functions to directly invoke other lambda functions also reduced the cost of having to add an API Gateway to each function which would have been necessary for a serverfull solution.

Lambda uses a pay-what-you-use pricing strategy which could also lower costs since some functions will be used less frequently than others. In production, however, the system would run all hours of the day so the benefits of this are maybe not guaranteed.

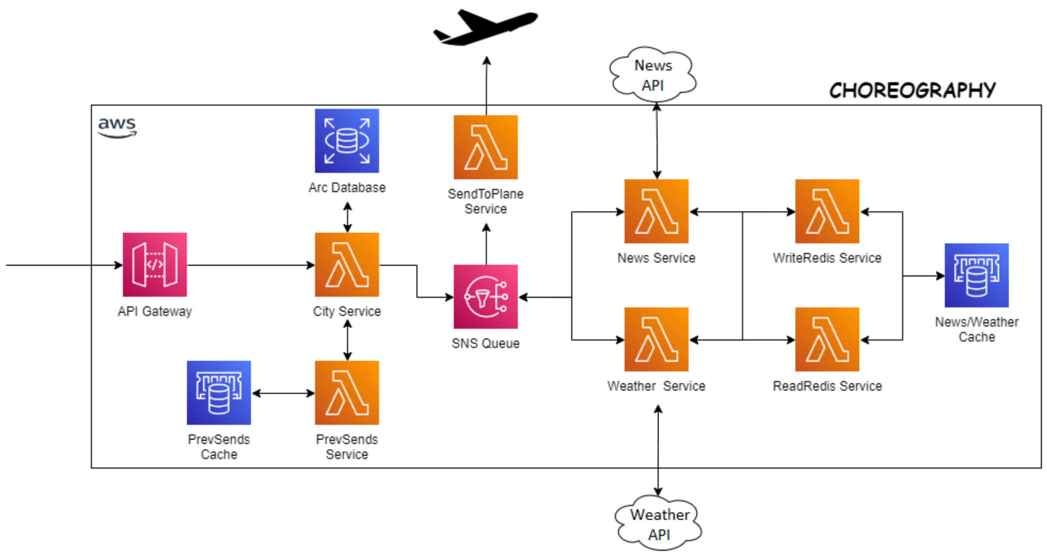
Potential drawbacks

A tightly coupled system is often perceived as being more difficult to extend. For example, if we wanted to add a service that fetches menus for different restaurants in the city. This would require us to also configure our city service so that the function is invoked and that the response is handled. This is why this system could be a bit harder to maintain and develop than a more loosely coupled system.

Another significant drawback of this system is that performance issues in one service will impact the entire system. If, for example, the news API is slow in responding with data this will lead to the entire system taking a long time to send data to the airplane. Since the services are all dependent and aware of each other this means that a failure in one of the systems will result in the entire system failing which is not a desirable trait.

The synchronous execution of the system will lead to longer run times in general and therefore greater run time costs. The city service function will wait for the news service and weather service to finish before itself finishes running which means it will rack up costs while waiting.

2.2.2 Architecture 2 - Microservice Choreography



The second architecture strives to achieve looser coupling than the first architecture. It does this by utilizing Amazon Simple Notification Service (SNS) to essentially create a job queue. Our first Lambda function will based on the coordinates and direction of the airplane, determine which cities data needs to be sent to, in a manner similar to the first architecture. The lambda function will then publish a message to SNS with information about the airplanes ID and which cities to generate data for. The weather and news services are subscribers to these messages, therefore they will automatically be triggered when a message is published. They will then generate the appropriate data to send to the airplanes, in a manner similar to the first architecture.

When the news service and the weather service have finished generating the data they will each publish a message to the SNS queue containing the generated data in the payload as well as the airplane's ID. This will trigger our SendToPlane Lambda function whose sole responsibility is to send the generated data to the airplane.

In this architecture, the components are more separated from each other. Each component has very little knowledge of the system as a whole. It only sees the parts of the system it needs for its purpose. Our first Lambda function's sole responsibility is to determine suitable cities to send data to and then add those cities to SNS. The microservices sole responsibility is to generate the data that is supposed to be sent to the airplane. The last lambda function's sole responsibility is to send data back to the airplane. The three sections of the architecture are not aware of each other and only communicate through SNS.

AWS Resources

This system is also highly dependent on AWS Lambda functions. We used this for similar reasons as the first architecture but primarily since it allows the components to scale independently.

The main component of this system is the SNS queue. It allowed the lambda functions to act as subscribers to different topics. When a message with a specific topic was published to SNS it would automatically trigger certain lambda functions. The city service publishes a message with a certain topic that triggers both the news service and the weather service.

They in turn publish a message with a different topic that triggers the SendToPlane service.

Potential Benefits

This sort of architecture is, as previously mentioned, called a Choreography. It makes it easier to modify or add services to the system without risking any failures to the system. For example, if we wanted to add a service that fetches menus for different restaurants in the city it could be done seamlessly. We could add another lambda function as a subscriber to the topic that the city service publishes and just have it publish the generated data to SNS with the same topic as the other microservices.

The services are less dependent on one another. A slow news service will not mean a delay in generating weather. City Service does not have to wait for the other microservices to finish but instead can finish executing as soon as the messages are published to SNS. This will in turn mean shorter lambda run time on average and therefore lower run time costs. Another benefit of a loosely coupled system is that failure in one part of the system is not necessarily detrimental to the health of the system as a whole. For example, the weather service would continue to work fine and send weather data to the airplane even if the news service stopped working.

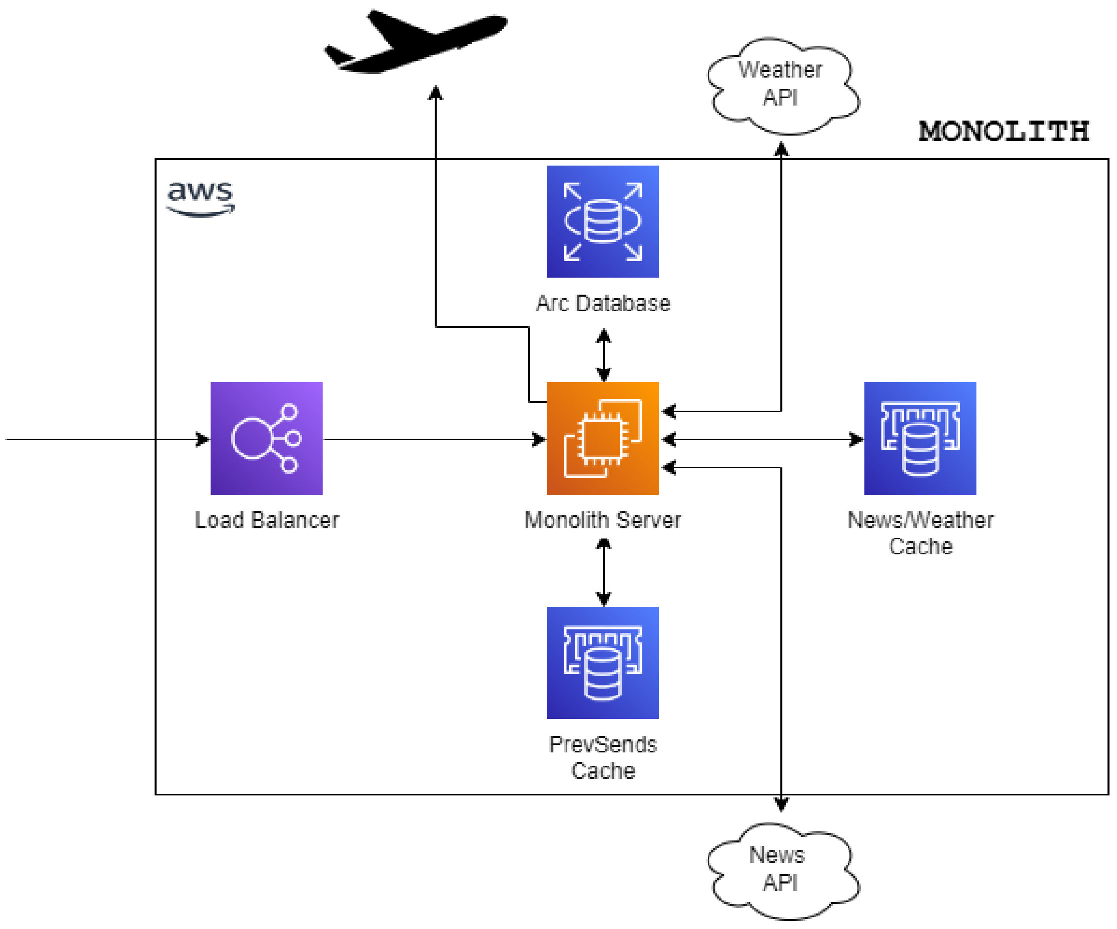
Potential Drawbacks

Some drawbacks can be perceived with this system as well. The system is larger, with more components compared to the other two systems. It contains more Lambda functions compared to the first architecture and also adds SNS as a new component. This will in turn lead to greater costs and potentially a slower system.

As the size of the system grows so does the complexity. Finding errors in the system becomes more difficult since each component needs to be monitored separately. It also becomes more difficult for new developers to learn and understand the system.

Amazon SNS also puts a 256KB limit on message sizes, exceeding this limit will cause an error. If the message size exceeds 64KB it will be automatically broken down into several messages by AWS. These two factors can cause problems when generating news since the data published contains tens if not hundreds of articles. Large payloads will be split into several messages which increase costs. Too large payloads will cause errors in the system.

2.2.3 Architecture 3 - Monolith



Our third and last architecture takes a monolith approach. What the other architectures choose to split into several services, the monolith combines into one component. This one component determines which cities it should generate data from, checks the cache, makes API requests, and lastly sends the data to the airplane. Data input and output, data processing, and error handling are all interwoven into one component.

AWS Resources

Unlike the two other systems, the monolith does not rely on Lambda functions. Instead, everything is run on EC2 servers. Each service contains all the necessary code to generate data for the airplane.

Using one server would not be enough to handle large loads. The system, therefore, utilizes an AWS Elastic Load Balancer as well as Amazon EC2 Auto Scaling to scale the system. The auto scaler is configured to automatically launch new EC2 instances when the system starts to become overloaded. These new instances are pre-configured at launch with the necessary code for the system. The load balancer evenly distributes the incoming airplane data between the available servers.

Potential Benefits

Monoliths tend to be a cheaper alternative than microservice solutions. Besides the cache and database all three systems share, this system only needs servers and a load balancer to function. AWS lets you rent servers by the hour. In production, this system would run all hours of the day so the EC2 pricing strategy fits well with a system like this.

Monolith systems also tend to perform well in regard to speed. Since it eliminates communication between components, the run time is solely spent on the crucial parts of generating data.

Potential Drawbacks

The biggest drawback of this system is the difficulty that comes with changing or extending the system. Since the system only contains one component, adding functionality to the system almost always means making changes to this one component. As the system grows this component will become more complex and more difficult to maintain. The system runs the risk of growing too large and complex to fully understand and make changes fast and correctly.

For the monolith, scaling the system to handle larger loads means adding more servers. Compared to the other architectures where we scale just the individual components that need more resources, here we scale the whole system equally. This means that a part of the system that might not need more resources will be scaled anyway if another part needs it.

Monolithic systems also have a barrier to adopting new technologies. Since changes in frameworks or languages will affect an entire system it is extremely expensive in both time and cost to keep the system updated.

2.3 Shared Components

All three of the systems have a few components in common, the PostgreSQL database handling Arc and the Redis cache handling PrevSends, news, and weather caching. Orchestra and Choreography use a few extra Lambda functions to handle the communication with Redis while Monolith handles everything from the main server.

2.3.1 Arc

The Arc database is hosted on AWS RDS and uses PostgreSQL with the PostGIS extension installed. In Arc, there is a large number of cities with their name, population, and coordinates. The relevant cities are retrieved by creating a rounded rectangle of fixed size with the same direction and starting position as the airplane. This rectangle is about 100km long and selects all the cities found inside. We chose this distance based on the average speed of a commercial aircraft and how often each airplane would communicate with the system. The 30 largest cities based on population are selected and returned. Unlike the Redis services, this one does not have a Lambda function attached to it, instead, the City Service calls on Arc immediately.

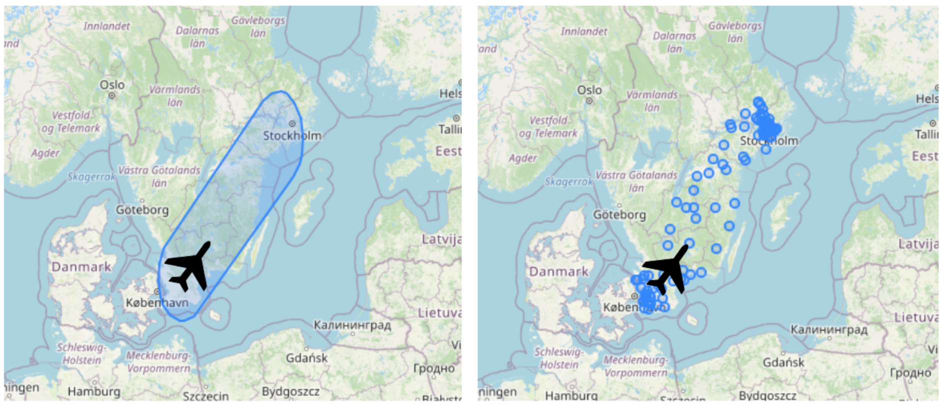


Figure 2.1: Left side: the area in which the system searches for cities. Right side: the cities found within the area

2.3.2 PrevSends

To limit the amount of data sent to the airplane, there needs to be a way of tracking what data an airplane has received. If an airplane has already received news and weather for a certain city, there is no need to send data for that city again. This is achieved by using Redis on AWS ElastiCache. After the system has determined all cities in an area 100 km in front of the airplane, it will then filter out the cities the airplane has already received data from. The system will only generate data for the cities that remain.

The key used in Redis is the airplane ID which is represented as a string of characters. The value is a JSON object where each key is a country code and the value is a set of cities in that country as shown in the example below.

```
"ABC123": {
        "SE": ["Lund", "Staffanstorp", "Ystad"],
        "DK": ["Copenhagen", "Aarhus"],
        "DE": ["Bremen", "Hamburg", "Berlin"]
    }
```

2.3.3 News and weather caching

Many airplanes will eventually fly over the same cities and to keep the API calls to a minimum weather and news data is saved in Redis for a short period of time. For our Microservice systems, this task is handled by two services, one for reading and one for writing. To make sure the data in Redis remains up-to-date news and weather for cities will eventually have to be retrieved again. Each entry will therefore have a time to live, for the weather, it's set to 30 minutes and for news, it's 15 minutes. This gives a good balance of minimizing the API calls while also having data that is mostly up to date.

ReadFromRedis Service

News and Weather Service will first send a list of the cities that the airplane has not received data for previously to the ReadFromRedis Service. This service will check if there is weather or news data for any of the cities in Redis. It returns the data it finds in the cache as well as a list of cities it did not find data for.

WriteToRedis Service

The weather and news service will use the list of missing cities to make API calls and generate new data. This data is thereafter passed to WriteToRedis which will insert it into our cache. The cache uses the country code, the city name, and the type of data as a key and the generated data as the value.

```
"SEStockholmWeather": {
        "temperature": 32,
        "description": "Clear sky"
}
```

2.4 Experimental Setup

During the implementation phase, we built each artefact in AWS. We started by building our Orchestra model since we knew that would share a lot of code with the other two systems. Later on, the Choreography and Monolith were completed.

The systems were developed in Python since it allowed us to quickly implement our ideas. Python also has a lot of documentation on how it can be used with AWS. For our Monolith, we chose to deploy the system as a web service with AWS Elastic Beanstalk. This allowed us to easily scale the system during testing. We built the web service with a web framework called Flask due to the fact that we only utilized one endpoint in our web server and didn't need the additional features of a full-stack framework.

To compare the three systems, we needed a way to test them. We built a testing environment where we could simulate airplanes sending their coordinates to our system and getting news and weather data back.

2.4.1 Simulation

Three types of airplane loads were used during simulation: 100 airplanes, 1000 airplanes, and 10 000 airplanes. Each airplane broadcasted its coordinates to the system every five minutes and then expected to receive relevant data back. For 100 airplanes this means that the system must handle a new request every 3 seconds, for 1000 airplanes every 0.3 seconds, and for 10 000 airplanes every 0.03 seconds.

A list of 10 000 airplanes was created where each element represented an airplane containing information about the airplane's id, unique starting coordinates, and a direction that would stay the same during the testing. The starting point of each airplane would be the coordinates of a randomly selected airport somewhere in the world and the airplane's direction would point towards another randomly selected airport. From this list, the program selected the first 100, 1000, or 10 000 airplanes depending on the type of test we ran.

Once the program started it would loop through each airplane and send away a request to the system. The program made sure each airplane sent away a request every five minutes. After an airplane had sent a request, its coordinates were updated and the airplane moved forward in the correct direction with a distance of 50km. Once each airplane has sent its data and updated the coordinates one iteration had been completed. The tests ran for a total of three iterations for a total run time of 15 minutes.

Tests were also performed to gauge how different hardware specifications and memory allocations impacted the systems.

2.4.2 Monitoring

All three systems would record its run times in a dictionary. This dictionary was sent between the components and updated along the way. The times of the individual components such as databases and API requests were tracked to find potential bottlenecks in the system. Once news and weather data was ready to be sent back a total run time could be calculated and the system would record the times in a separate database.

Our Orchestra and Choreography systems have an additional way of measuring performance with the help of AWS CloudWatch. CloudWatch measures how many Lambda functions were invoked and for how long they ran. For the Choreography system, CloudWatch measured how many messages were put into the queue. In our results, the run times of each of the Lambda functions were recorded using CloudWatch while the total run time of the system comes from our database. This total duration started when the request arrived at our city service and stopped when the data was ready to be sent back. The total duration includes the time between the Lambda functions such as the time needed to invoke another Lambda function or retrieve a message from SNS. Therefore the total duration will be greater than the individual services combined.

The PostgreSQL database and Redis cache was using a fixed resource that would not scale if needed, it would simply error out if the load became too big. It was, therefore, crucial to keep an eye on how much system resources were being used such as CPU, RAM, and the number of operations that the database performed each second. AWS provides different tiers of computing power for its services and a few test runs were done to find an optimal balance between price and performance. The same type of database was used across all three systems.

The Monolith uses Beanstalk with a load balancer to distribute the load across several EC2 which unlike Lambda functions doesn't scale instantly. The number of EC2 in Beanstalk was configured in a few test runs before the real simulation began to ensure that there would be a sufficient number of machines to handle all of the incoming requests. CloudWatch gives insights into how many machines are active and how often a new one begins. It also provides information about each of the machine's system health in regards to CPU, RAM, and network traffic. To determine the pricing for the load balancer we also needed to track the number of active connections, the number of new connections, and how many bytes were processed by the load balancer with the help of CloudWatch.

Chapter 3
Results

The results will be split into three different sections focusing on our three research questions. Section 3.1 looks at the total time of the entire system while the following subsections look at the times of the individual services. In section 3.2 we will explore how well the three systems scaled. The types of hardware used are described and whether vertical scaling is beneficial. Section 3.3 will talk about the costs of the three systems. Using the measurements from the simulation we calculate what the systems would cost in actual production.

3.1 Performance

The duration of a request was measured as from the start of the system receiving the request until the data was ready to be sent back. We calculated an average duration for all requests sent during a simulation and present the result below. The testing was done over 15 minutes and those results are shown in the graphs. The tables contain the average data over the last five minutes of testing to ensure that the systems had been stabilized. The table shows a clear winner for both 100 and 1000 airplanes, namely the Monolith. For 10 000 airplanes the Orchestra was the fastest. The choreography system was the slowest in all three cases.

Average run times (ms)			
NrOfPlanes	Orchestra	Choreography	Monolith
100	1278	2108	566
1000	771	928	483
10 000	223	623	300

Table 3.1: Overall average run times for the three systems

3.1.1 Orchestra

This system had the biggest percentual improvement in run time from 100 airplanes to 10 000 airplanes. Weather Service and News Service are computationally heavy with a large number of API calls sent and benefited a lot from the increased speed of the Lambda functions once they warmed up. City Service run time is determined by how fast the other services are and speeds up accordingly. The graphs are shown in figure 3.1 display the effect of this improvement in run time from the warm-up during the testing. The speed of Redis increased in some services and remained roughly the same in others. The speed of querying RDS remained constant across all three runs and was thus not impacted by different loads.

Orchestra run times (ms)							
NrOfPlanes	CityService	WeatherService	NewsService	PrevSend	ReadRedis	WriteRedis	RDS
100	1278	1255	923	24	7	13	13
1000	771	649	529	18	7	9	13
10 000	223	128	131	19	9	4	13

Table 3.2: Run times for each service in Orchestra

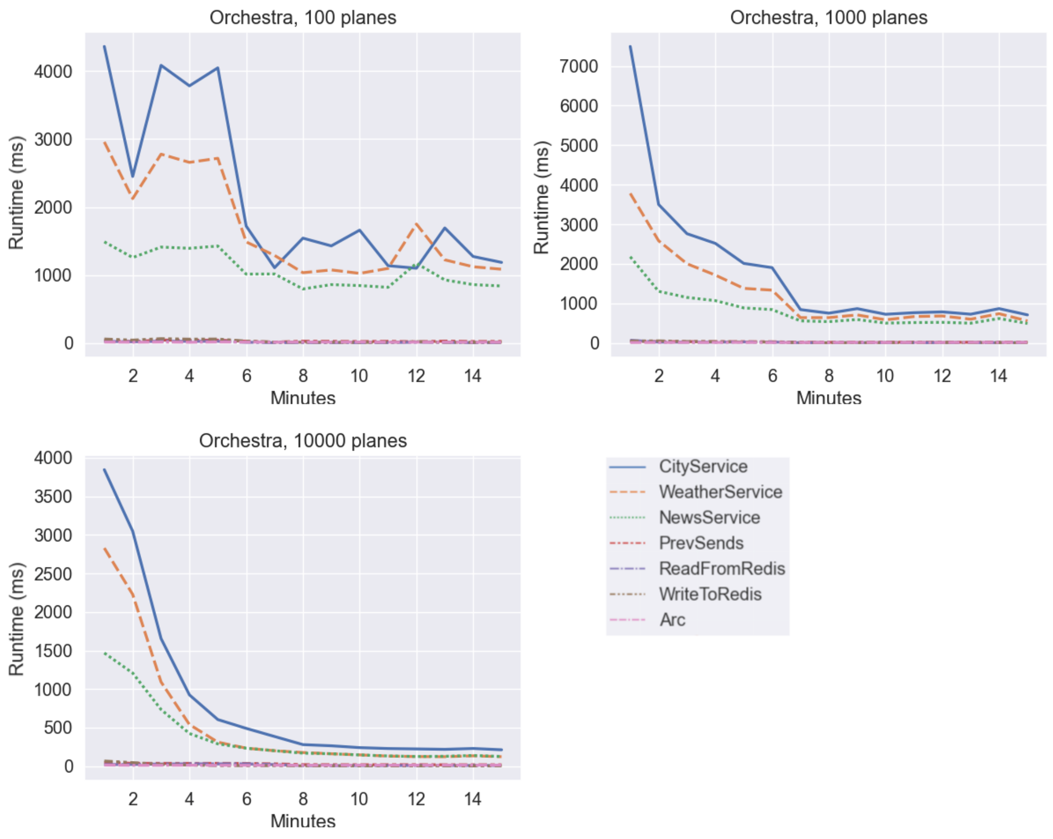


Figure 3.1: The duration for the individual services during 15 minutes of testing

3.1.2 Choreography

Overall the average run time from section 3.1 is much higher than the individual components. A large part of the run time is spent on sending and retrieving messages from SNS and then invoking the other lambda functions. City Service didn't decrease in run time as the Lambda function warmed up. The run time of both Weather Service and News Service decreased significantly when more airplanes were added. All of the database services remained mostly on the same level during the three tests. The results from the graphs in figure 3.2 are similar to those for the Orchestra system.

Choreography run times (ms)								
NrOfPlanes	CityService	WeatherService	NewsService	PrevSends	ReadRedis	WriteRedis	SendData	RDS
100	294	1438	1147	22	6	16	10	13
1000	284	826	778	21	9	8	10	12
10 000	291	340	374	23	9	5	9	13

Table 3.3: Run times for each service of Choreography

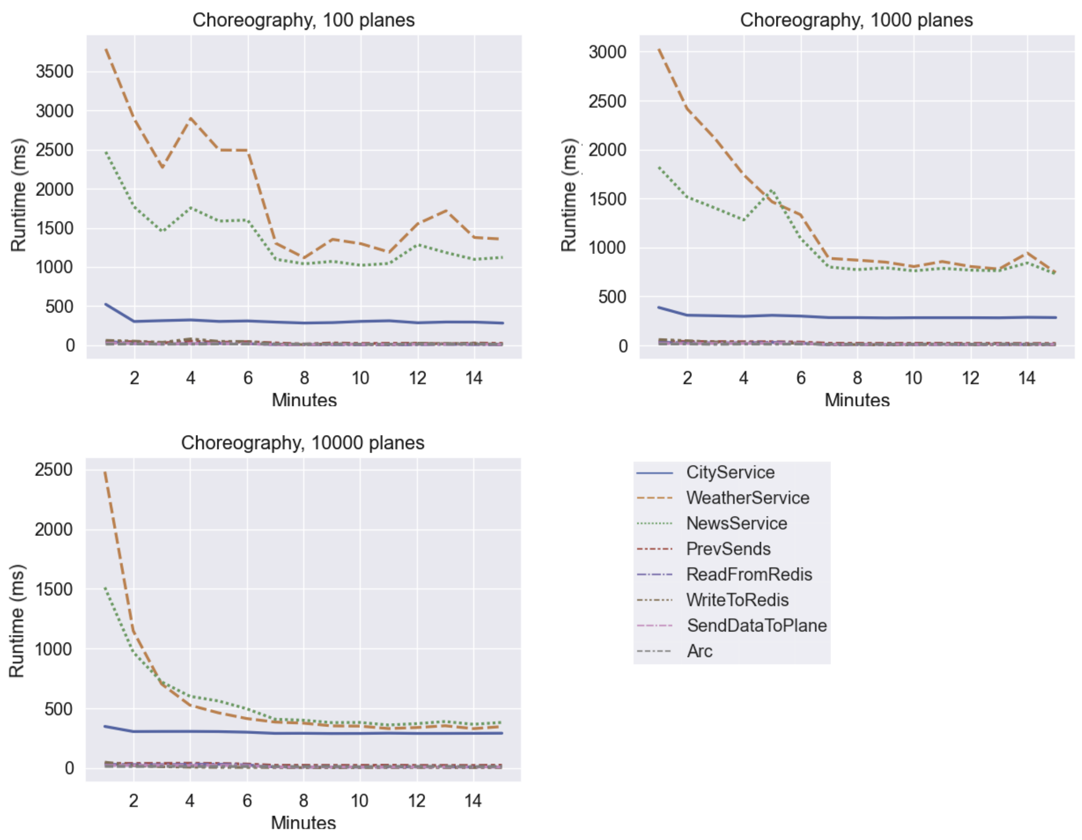


Figure 3.2: The duration for the individual services during 15 minutes of testing

3.1.3 Monolith

The individual components of the system did not experience the same increase in speed as the Lambda functions from Orchestra and Choreography. Both Redis and RDS performed at the same level as the other two systems. As the number of EC2 varied during the testing phase, the number of requests handled each second varied slightly and lead to the jagged edges shown in figure 3.3. This is a stark contrast to the graphs of the other two systems that found a mean processing time after roughly eight minutes of testing.

Monolith run times (ms)							
NrOfPlanes	CityService	WeatherService	NewsService	PrevSends	ReadRedis	WriteRedis	RDS
100	566	304	565	26	7	6	12
1000	483	326	382	18	7	9	13
10 000	300	234	269	15	3	3	13

Table 3.4: Run times for each "service" of the Monolith

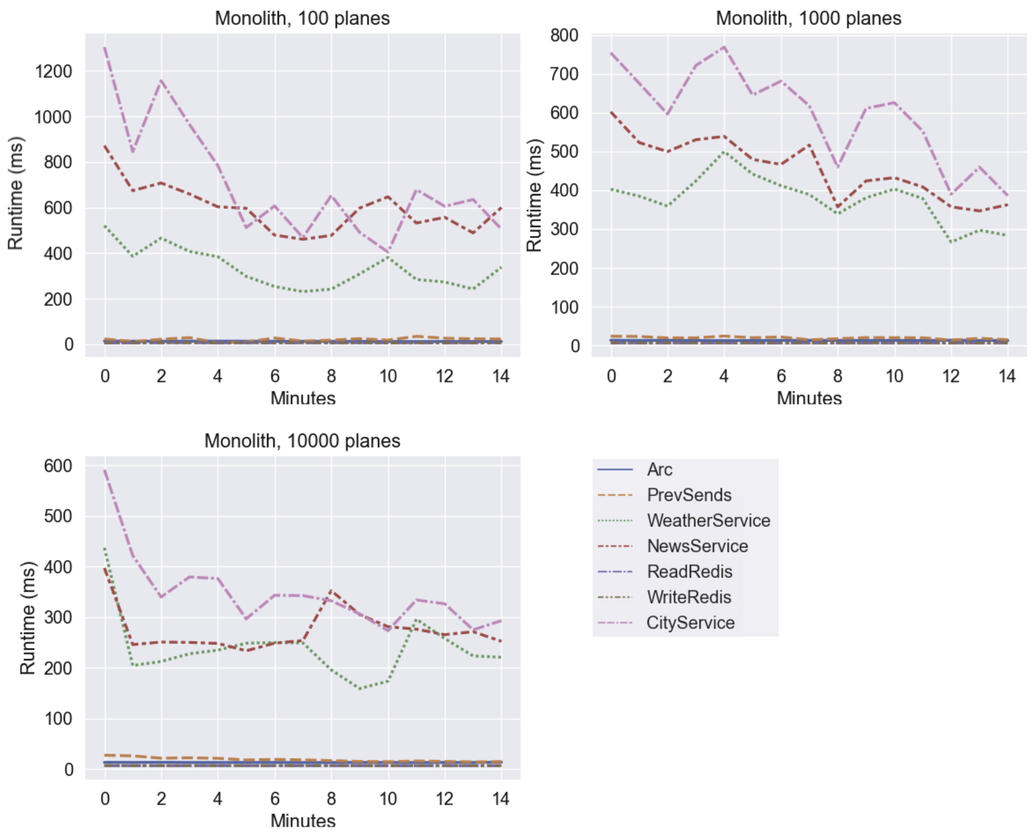


Figure 3.3: The duration for the individual services during 15 minutes of testing

3.2 Scalability

Scalability is the property of a system to handle a growing amount of work by adding resources to the system. We measured the health and performance of all three systems when experiencing three different number of airplanes. The health of the different components of the system was important because it helped us decide if we needed to scale the components.

Both RDS and Elasticache had the same amount of resource use for all three systems. They were on the lowest tier of computing and did not need a more powerful machine to handle all of the incoming requests.

Maximum CPU usage (%)		
NbrOfPlanes	RDS	Elasticache
100	4	5
1000	6	7
10 000	25	10

Table 3.5: The CPU usage for Database and Cache service

3.2.1 Microservice systems

A test was conducted where the RAM of the Lambda functions City Service, Weather Service, and News Service was increased from the standard 128MB to 256MB for the choreography system. This led to a 23% overall increase in speed while also costing twice as much for the Lambda run time part. As the Lambda functions for Orchestra and Choreography are near identical the results would be very similar for the Orchestra system.

Choreography scalability (ms)				
RAM	CityService	WeatherService	NewsService	Total duration
128MB	291	340	374	623
256MB	148	185	211	481

Table 3.6: Runtimes with different RAM configurations

3.2.2 Monolith

Our Monolith had the option of either choosing a t4g.nano EC2, a t4g.micro or a t46.small for the Beanstalk servers. It always chose to scale with t4g.nano servers and the CPU usage remained under 10% for 10 000 airplanes. Beanstalk preferred horizontal scaling instead of vertical scaling i.e. many slower machines rather than a few powerful machines. A linear increase in the number of airplanes did not lead to a linear increase in the number of EC2 servers used.

Monolith scaling	
NrOfPlanes	NrOfEC2
100	3
1000	7
10 000	19

Table 3.7: Number of EC2 used for varying number of planes

3.3 Cost

The values below show the monthly cost for each of the systems. For 100 airplanes the costs of Orchestra and Choreography are roughly the same while Monolith is higher. At 1000 airplanes Orchestra and Monolith had roughly the same cost while Choreography was much higher. At 10 000 airplanes the difference between all three systems was very clear. Both RDS and Elasticache use a fixed pricing method and their cost remained the same no matter how many airplanes were simulated.

3.3.1 Orchestra

The cost for the Lambda run times decreased during testing relative to the number of invocations as the Lambda functions speed increased. The number of invocations also decreased as the airplanes started to fly over the same cities and the cache was used more. API-gateway and the data transfer scaled linearly.

Orchestra costs (USD)							
NrOfPlanes	run time	Invocation	API-Gateway	Data Transfer	RDS	Redis	**Total Cost**
100	4.73	0.92	0.92	0.58	12.28	14.60	**34.03**
1000	26.02	8.65	9.16	5.81	12.28	14.60	**76.52**
10 000	62.38	66.23	91.58	58.14	12.28	14.60	**305.21**

Table 3.8: The different costs to run the Orchestra system for a month.

3.3.2 Choreography

The run time cost of the Lambda functions was lower than those of the Orchestra model at a lower airplane count. Invocation costs were higher than Orchestra due to the fact that Choreography has one extra Lambda function SendDataToPlane which is called upon each time the system is being run. The cost of the API gateway and data transfer is nearly identical to Orchestra. The cost of the SNS service scaled linearly and was the greatest single cost of the system for 1000 and 10 000 airplanes.

Choreography costs (USD)								
NrOfPlanes	run time	Invocation	API-gateway	Data transfer	RDS	Redis	SNS	**Total cost**
100	3.55	1.23	0.91	0.58	12.28	14.60	4.07	**37.22**
1000	21.75	10.89	9.15	5.81	12.28	14.60	38.72	**113.2**
10 000	129.43	98.48	91.58	58.14	12.28	14.60	383.19	**787.7**

Table 3.9: The different costs to run the Choreography system for a month

3.3.3 Monolith

This system had the highest cost for 100 airplanes but had the slowest increase in cost when the number of airplanes increased. The cost of the load balancer proved significant at 100 airplanes but less significant compared to other component costs when the number of airplanes increased. At 10 000 airplanes the cost of both EC2 servers and the load balancer was low which resulted in the low overall costs. Data transfer costs are identical to the other two systems.

Monolith Costs (USD)						
NrOfPlanes	Instances	Load Balancer	Data Transfer	RDS	Redis	**Total Cost**
100	9.42	17.39	0.58	12.28	14.60	**54.27**
1000	21.98	18.71	5.81	12.28	14.60	**73.38**
10 000	59.66	31.98	58.14	12.28	14.60	**176.66**

Table 3.10: The different costs to run the Monolith system for a month

Chapter 4
Discussion

The discussion will be separated into four different parts. Besides discussing the three research questions we will also touch on the challenges faced throughout the project and the following tradeoffs that were made.

4.1 Performance

While there were some differences in performance when comparing the three systems all three performed well overall. Our Choreography architecture proved the slowest in all three scenarios but that was expected considering the addition of our SNS queue. While one could argue our Choreography architecture is a lot slower compared to the other two systems we feel that it still has an acceptable run time. If you consider the end goal of the system then performance is probably the least important aspect that we evaluate. A passenger on an airplane would not notice a difference if it takes an extra second for Arc to receive data.

We can see all our three systems getting faster the more airplanes we add. This is most likely due to the fact that the more airplanes we have interacting with the system the more data will be available in the systems cache. When the system can fetch weather and news data from the cache it does not have to make any API calls which is the most time-consuming part of our data generation.

4.1.1 Lambda Performance

During our simulations, we noticed that Lambda functions would become faster as the tests progressed. Looking at figure 3.1 we can see how the average duration of the different Lambda functions decreases as the simulation progresses. This was true for both our Orchestra system as well as our Choreography system. We believe this is for two reasons:

1. As the simulation progresses our cache starts to fill up which then speeds up the process of generating data.

2. As the simulation progresses the Lambda Service understands the functions will be invoked frequently and keeps it warm until the next invocation.

The second reason makes sense since we have learned that when a Lambda function is first executed it experiences a cold start. This is because the Lambda service needs to prepare an execution environment for the function to run. At the start of our simulation process, the Lambda functions will be continuously and simultaneously invoked which will result in multiple execution environments being launched by the Lambda service. Initially, the Lambda service will launch these environments and probably not retain them long after the function finishes running. This will lead to more cold starts until the service recognizes the continuous invocation pattern. For our simulation, we noticed that after about 8 minutes the functions would be properly warmed which is why we decided to use the metrics from that point forward to calculate the performance and costs of the systems.

City Service

When building our systems we expected our City Service to be much faster in our Choreography system than in our Orchestra system. City Service in our Orchestra system has the following steps

1. Receive airplane data.

2. Calculate coordinates for the location of the airplane in 100 kilometers.

3. Fetch cities from the Arc database.

4. Check Redis to filter out cities the airplane already has received data for.

5. Call Weather Service and wait for a response.

6. Call News Service and wait for a response.

7. Send combined data to airplane.

Compared to our Choreography system which has the following steps

1. Receive airplane data.

2. Calculate coordinates for the location of the airplane in 100 kilometers.

3. Fetch cities from the Arc database.

4. Check Redis to filter out cities the airplane already has received data for.

5. Publish message to SNS queue.

In our Choreography system, the City Service does not have to wait for our news service or weather service to finish generating data, it finishes running once it has published the message to SNS. We expected our Choreography City Service to be much faster for this reason and our simulation results showed it being faster for 100 and 1000 airplanes. It was slower than its Orchestra counterpart however when simulating with 10 000 airplanes, although not by a lot. While this is unexpected, it could be due to the fact that when the system interacts with so many airplanes the cache is filled with data for more cities and therefore more useful. That would reduce the overall run time for the city service and news service and make the two systems very equal in run time. The slightly worse run time of the Choreography City Service could be due to the time it takes the function to publish a message to SNS.

4.1.2 Monolith Performance

In regards to average run time, we can see the Monolith performs well. It averaged below 1 second regardless of the amounts of airplanes and only became faster the more airplanes we added. It outperforms our Microservice systems for 100 airplanes and 1000 airplanes. This was something we have predicted since the Monolith does not have to communicate between different servers or components. Faustino et al.1.5 also concluded that a Monolithic system would be faster, although less scalable, than a microservice approach performing the same tasks. What we noticed, however, is that this is only true up to a certain threshold. For a certain number of airplanes, our Orchestra system performed just as well if not better than our Monolith system.

4.1.3 Programming Environment

For this project, we chose to build our three systems in the Python programming language. Our Lambda functions as well as our Monolith system all use Python code that we wrote. We chose Python for a few reasons. It has easy syntax, readable code, and a lot of freedom for us as developers. Python also has great libraries for working in AWS that let you fetch data to and from other services. Building three different systems was not an easy task so we wanted to pick a programming language we were comfortable with that would also help facilitate our implementation process.

Some of the factors that we liked about Python also make it slower for some tasks than other programming languages. Some languages, like Java, allow you to run code in parallel on multiple CPUs. Python, however, is single-threaded on a single CPU by design. In our News Service and Weather Service, we make asynchronous API calls to try and speed up the process of fetching data for multiple cities at once. This is a process that would have benefited from being multi-threaded.

Choosing another programming language, like Java or C++ could have resulted in more performant systems. If a system like this were to be developed for production it could be worth comparing what the optimal programming language would be for the system.

4.2 Scalability

All three systems scaled well during our testing. We expected them to fair pretty well with 100 and 1000 airplanes but were surprised to see them handle 10 000 airplanes with ease as well. While using Lambda functions can be a bit pricier than using EC2 servers, they were the easiest to scale and required very little configuration on our part. Lambda functions scale automatically so an important metric to keep track of was error rates in the system. With that, we could make sure that higher loads did not lead to failures, and using Cloudwatch we could see that we had zero runtime errors in any of the functions. Running simulations with 100 airplanes and running it with 10 000 airplanes required no manual configuration between tests. The Lambda service handled scaling seamlessly. We also monitored the metrics for our database as well as our Redis cluster to see if they could keep up with higher loads. During our simulations, we never noticed CPU usage reach higher than 25% for either instance so we felt no need to scale them. They both retained the lowest hardware specifications for the duration of our testing.

Scaling our Monolith proved to be a bit harder than our other two systems. If our Load Balancer did not have enough available servers to distribute incoming traffic between it would start to reject some of the incoming requests. We experienced this during our initial simulation of the Monolith when we had not properly configured Elastic Beanstalk yet. Some requests were immediately returned with an error message. Elastic Beanstalk is set to automatically launch new instances when traffic increases but launching a new instance could take 1-2 minutes. If Elastic Beanstalk experiences a traffic spike it could struggle to launch news instances before the system becomes overloaded. In that case, it will start rejecting incoming requests until the new instances are up and running. In our case, we managed to solve this problem by having more instances pre-launched at the beginning of our simulation. We also don't foresee this to be too big of a problem since incoming traffic would be almost constant in a production-type environment.

4.2.1 Vertical Scaling

When examining different strategies for scaling our systems we wanted to explore both horizontal scaling as well as vertical scaling to see which was more suitable for our systems. At the beginning of the project, we imagined some components would need to be vertically scaled but during our testing, it became apparent that horizontal scaling was more suitable for all our systems. We believe this is due to the fact that most tasks the systems perform are not CPU heavy. Making API requests and calling other services are time-consuming tasks, but not CPU-consuming tasks. More powerful hardware will not decrease the time it takes to make API requests for example. This aligns with the findings made by Albuquerque et al. in their paper which showed that using higher-end hardware did not result in a proportionally faster system 1.5.

Lambda Service

While scaling Lambda functions horizontally is handled automatically by the Lambda service, if you want to scale it vertically, i.e increase memory size for the function, that has to be done manually by the developer. We expected vertical scaling to have a positive effect on some of the Lambda functions, especially the news service and weather service functions. Those services make several asynchronous API requests and it seemed reasonable they would benefit from more memory.

When doubling the amount of memory we did see an overall faster run time for these functions 3.6. They almost halved their average run time which is a great result. However, since doubling the amount of memory would also mean doubling the run time cost for the Lambda function we had to consider if the change was worth it. When building systems there are always trade-offs that have to be made. Optimizing for one factor often means sacrificing another. For this system's end-use, low costs while still retaining acceptable performance seemed preferable. An increase of one second in the system's speed would not be noticeable to an airline passenger but a decrease in costs would be beneficial to Tactel. We, therefore, decided to keep our functions running at 128MB memory to reduce the costs of the system.

Elastic Beanstalk

For our Monolith system, both vertical and horizontal scaling is handled semi-automatically by Elastic Beanstalk. We could configure which hardware specifications Elastic Beanstalk could choose from when launching servers. It also allowed us to configure the minimum amount of servers we always wanted running as well as the maximum amount of servers it was allowed to launch.

The service uses a few different metrics, such as current CPU usage, network traffic, and the number of incoming requests to determine when to scale and what hardware specification to choose from when scaling. During our simulations, when the service scaled the system, it always chose the t4g.nano hardware specification. We monitored the metrics it was looking at and could see that CPU usage and network traffic always remained relatively low. We believe that since the servers ran more time-consuming tasks (such as making API requests) instead of more hardware-consuming tasks, the service decided it was more beneficial to scale horizontally with lower hardware servers.

4.3 Costs

Cost was an important aspect to focus on and a difficult aspect to measure. AWS's pay-as-you-go pricing model is great and very flexible but it also makes calculating potential costs a bit more difficult. Our goal for the three systems was to strive for low costs and acceptable performance. We kept this in mind as we were building the systems and made sure we knew the pricing of each service we were using. This allowed us to understand what parts of the service were most expensive and we tried to minimize the use of those parts if possible.

4.3.1 Comparing Costs

When looking at costs for the three systems there are a lot of shared expenses between them. All systems use a database and Redis cluster, which has a fixed cost each month. The API requests each system makes are the same amount and result in an equal data transfer charge.

None of the systems were the cheapest in all three simulations but the Monolith system got progressively more cheap than the other two the more airplanes we added. One of the reasons for this is the fact that using a load balancer proved to cost a lot less than using an API Gateway. This is due to the fact that an API Gateways cost scales linearly as it is only dependent on the number of incoming requests it receives. A load balancer, however, has a fixed cost each month that does not change depending on the amount of traffic. A load balancer also has an additional cost based on the number of LCU-Hours, often described as “the least intuitive unit known to humankind”. That unit is based on a few different factors but the resulting cost for our type of system was minimal. Our result differentiates itself from what Villamizar et al. concluded in their research comparing Monolithic systems to Lambda-focused systems 1.5. In their research, the Monolithic system proved more costly and also harder to manage. A reason for this could be due to the different use case of their system. That system implemented some CPU-intensive algorithms to generate data. For that reason, they used EC2 servers with more RAM than we did and that would result in greater costs. In their Lambda-focused system, they could separate the component responsible for the CPU-intensive algorithms into a separate service and scale it independently. They could thus allocate more memory to just that Lambda function and keep the rest of the system cheap. Our systems had no CPU-intensive parts and that might be why the Monolith had low costs.

Another reason Monolith becomes so cheap compared to our microservice solutions is that adding more servers proved less expensive than increasing the number of Lambda functions running. Since our microservice systems contain several Lambda functions, each airplane that needs data will lead to between three to seven invocations of Lambda functions. When you have 10 000 airplanes requesting data every five minutes this quickly becomes several million invocations every day. EC2 servers on the other hand have a fixed cost each month and are not dependent on the number of invocations. Our Choreography system was never the least expensive for any of the three test scenarios. It matches our Orchestra for 100 airplanes but the costs quickly escalate when adding more airplanes. The primary reason for this is the added cost that using SNS brings. SNS charges for each message published to the queue and there is also a data charge for delivering messages to our Lambda functions. When scaling the system the cost of this scaled significantly as well. If the message payload to SNS is too large then SNS will split the message up into several messages and charge for each message. Posting a message containing hundreds of articles will result in a large message payload and thus a lot of extra charges. We still feel that using SNS to let the City Service post jobs for the Weather Service and News Service to pick up is a good idea since it translates to a more loosely coupled system. However, there is probably a better way of transferring the generated data to our Send To airplane Service. One idea is to store the generated data in a database or cache and post a message to SNS that only contains the key needed to fetch the data. This would drastically reduce message size and thus costs.

4.3.2 Potential unaccounted costs

We tried to include as many costs as possible in our calculations for the system but there are some costs that we felt we could and should not include in our evaluations. These are costs that most likely would be added if the system were to be built for production. One example is the costs of APIs, such as a news API and a weather API. In our systems, the APIs are mocked since Tactel wanted to explore different API solutions themselves later on. Since we do not know which solutions they will pick it’s difficult for us to gauge eventual costs for this.

As previously mentioned we did forgo implementing security in our systems. Security would have added a lot more complexity to the project and was not necessary for our end goal so the systems were built completely on the public internet. Adding security to the system is something that will be necessary if Tactel chooses to implement the system into Arc and this could mean additional costs. Components such as NAT Gateways and Interface Endpoints could be necessary for a secure version of the system and would result in additional costs.

Another potential we have not included in our calculations is the cost of sending data to the airplane. Tactel was not sure how this would work at the moment. There is a possibility the data would first be transferred to another system before being sent to the airplane. In that case, there is no additional cost to our system since data transfer within AWS is free. However, if our system is to send the data directly to the airplane, there will be an additional data cost for sending data outside of AWS.

Lastly, one thing we have not mentioned about AWS is the AWS Free Tier. For some services, AWS provides a certain amount of free cloud computing each month. For example, AWS provides every user with 1 000 000 free Lambda requests and up to 3.2 million seconds of Lambda compute time per month. Including the free tier would lower the costs of some of the services we used in our systems. The reason we did not include it in our calculations is because Tactel most likely uses the same services for other systems so the free tier is probably used by them as well. It, therefore, was difficult to decide if our system would benefit from the free tier at all.

4.4 Challenges

When implementing the three different architectures in AWS there were a few challenges we faced that we had not anticipated. This is a common thing in IT projects and we went into the project knowing we would have to be flexible. Comparing our original architecture designs with the final systems, there are some noticeable differences. In this section, we will discuss the different challenges we faced that led to those differences.

4.4.1 VPC & Cloud Security

Cloud-based infrastructure can be a great choice for many teams. It delivers increased IT agility, on-demand IT resources, and subscription-based pricing models that give your team a lot of freedom and opportunity. When building systems in the cloud security is an important aspect to consider. When launching server- or database instances you want to be able to control who has access to these instances and what they can access. If you launch a normal instance inside a public cloud, your instance will be exposed to the Internet with a lot of potential risks for sabotage.

Amazon Virtual Private Cloud (VPC) is a logically isolated portion of Amazon Web Services that gives you a virtual network where you can launch instances with particular rules and policies to get access to the Internet. Each VPC that you create is logically isolated from other virtual networks in the AWS cloud and is fully customizable. You can select the IP address range, create subnets, configure root tables, set up network gateways, define security settings using security groups, and network access control lists. It allows you to have secure connections and restrict instance access inside your virtual network.

When we started building the three systems we wanted to make sure they were secure from outside interference. We needed a database, servers, and in-memory datastores which are components that should not be placed in the public cloud. The first thing we did before we started building was therefore to launch a VPC with an IP address range and configure a routing table. We launched all our instances inside this VPC and worked on configuring the different security rules to allow the different components to communicate inside the VPC. We quickly ran into problems, however. Getting all three systems to work properly within the same VPC was difficult due to the fact that the systems had different requirements. Launching three separate VPCs would also be troublesome since it would not allow us to reuse components (such as our database or our Redis) for the systems. When launching our Lambda functions inside our VPC we also had difficulties getting the functions to communicate with each other. This might have been due to the fact that Lambda is a serverless approach and therefore VPC configuration is limited.

We finally felt like we had hit a roadblock and after discussing it over with our supervisor at Tactel we decided to build the systems without configuring security. None of the data we were using was going to be sensitive data and the systems we were building were purely for testing and not for actual production. All our three systems were built as open as possible to facilitate our work.

The Elasticache service, however, required our Redis cluster to be placed in a VPC. Our Weather Service and News Service could not be placed in a VPC since they needed to communicate with the internet to make API requests. But for a Lambda function to be able to read and write from Redis we needed it to be in the same VPC as the Redis cluster. This required us to adapt our architectures somewhat and is why we used the ReadRedis and WriteRedis Services for our Microservice architectures.

4.4.2 API

To generate data our system needs to communicate with weather and news API. The idea for the system is to generate local data for the cities the airplanes are flying over. So, for example, say you are flying over New York you should be able to see local news for New York and the current weather there. Both weather and news APIs exist and at the start of the project, we explored different ones to find providers that were suitable for the project.

We quickly discovered that there currently is no API provider that allows its customers to fetch news for specific cities in the world. With some APIs, we could search for news from certain countries but not specify which cities we wanted news from. Most APIs had the possibility of searching for specific queries found in the headline or article text but when querying specific cities it did not prove reliable enough to be considered an option. Another issue was that all news API providers do not respond with the full article text, instead, the API response contains a summary of the article. This is probably because of copyright issues but is something that will probably remain an issue if the system was to be built for production use. After discussing it with Tactel they decided it would be best if we mocked all news data instead and that this was a problem they would solve in the future.

All weather APIs we looked at had the option of fetching the weather for certain cities. You provide just the longitude and latitude of the city and the API provider responds with weather data. We got a solution running with a certain provider but ran into problems during testing. Almost all providers had a free tier subscription which allowed you to make a certain amount of requests each month. We quickly reached that limit when running our simulations since some scenarios had us fetching weather data for hundreds of thousand cities. There were paid subscription plans that would allow us enough requests each month to run our simulations but Tactel instead wanted us to mock all weather data as well.

While this could have had some effect on our run time we tried to mock generating the data so it would behave similarly as if it communicated with actual weather and news API. This means the systems all make API calls for each city each airplane wants to generate data for. Since AWS charges for the amount of data that transfers out from AWS to the internet, making real API calls was important to get an accurate cost estimate for the data charges.

4.4.3 Database

For our simulation process, we were provided a copy of the Arc database that Tactel currently uses with their Arc system. This database amongst other things contains a table with all the cities in the world that shows up when passengers interact with the Arc map. One issue we

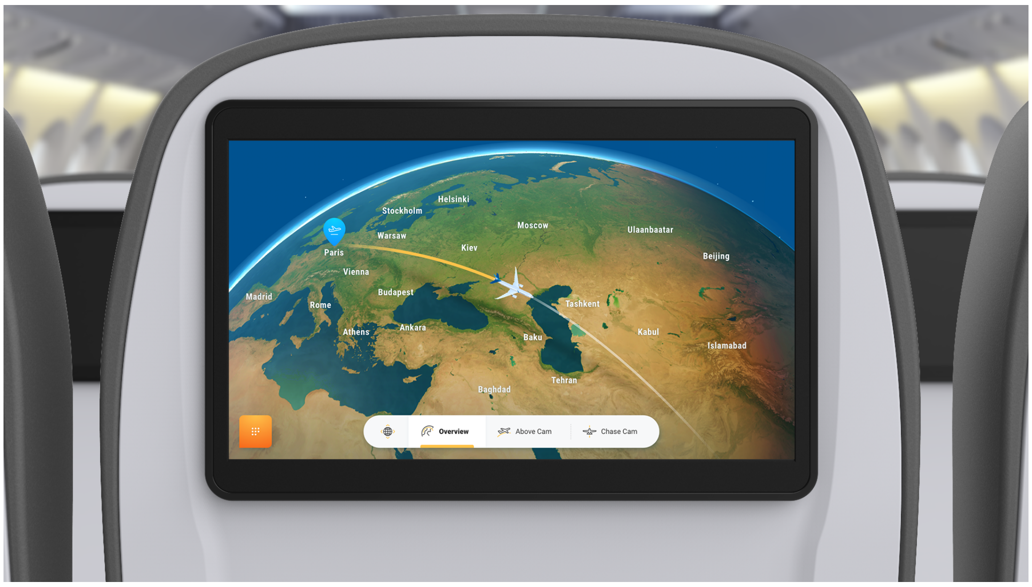


Figure 4.1: How passengers onboard airplanes will experience and interact with the Arc map

quickly noticed was that when our systems queried the PostGIS database to determine which cities to generate data for the database could return upwards of 300 cities for each airplane. We discovered that the cities table was over-saturated with entries. The table contained over 40 000 entries and many were very close to each other geographically. For instance, we noticed that Copenhagen had a separate entry for each of its suburbs and some of the entries were for cities with populations of just a few hundred. When we first started running our simulations this led to bottlenecks when generating weather and news data. Making API calls for over 300 cities took considerable amounts of time and could even lead to errors for our Lambda functions. For our Monolith, it meant that the overall run time increased significantly and a lot of servers were necessary to keep up with the load. This problem also lead to significant data charges since the number of API calls was so high.

To solve this we limited the number of cities each airplane could receive data for to 30 cities. When querying our database we selected the 30 largest cities by population and these were the cities the system generated data for. This could mean that for a passenger there is a possibility that one of the cities they interact with on the Arc map will not display local data for that city, although we do believe most of the interesting cities will be covered by our query.

Chapter 5
Conclusions

5.1 RQ1: Performance

Which AWS architecture and resources will provide the most efficient way to supply real-time data to the airplanes? The goal of this research question was to examine how well the different systems performed. We were curious if there would be any drastic differences in run time between the systems and how we could optimize performance for each system.

Looking at the results we see that all three systems had acceptable run times. For this system's intended use case a difference in the run time of a second or two is not something noticeable for an airline passenger. When comparing the three systems we see that our Monolithic system is faster for fewer airplanes (100-1000). Our Orchestra system narrowly beats it in speed when we simulated with 10 000 airplanes. This is most likely due to the fact that the system's lambda functions become optimized when heavily used. From that, we can conclude that the Lambda function should be a preferred choice for performance if the function will be frequently invoked. When a system is less frequently invoked, a constantly running server, such as an EC2, is a good choice since it does not need to be warmed before starting execution.

5.2 RQ2: Scalability

Which AWS architecture and resources will provide the most efficient way to supply real-time data to the airplanes? In all three of our systems, we used some sort of service to aid in automatically scaling our systems. AWS provides a lot of different ways to allow automatic scaling and it lets us, developers, more easily handle incoming traffic to our systems. For a monolithic system, Elastic Beanstalk was a great tool to use for deploying the system. After uploading your code the service launches a load balancer, servers, and configures your automatic scaling options. It does require some initial manual configuration of the auto scaler but once everything is up and running Elastic Beanstalk handles the rest. For our Microservice solutions, it proved very useful to use Lambda functions throughout the systems. The Lambda service requires no additional configuration after your code is uploaded. It automatically scales your function to meet demand.

An interesting observation about all three systems is that all of them become faster the more airplanes we added to our simulation. This indicates none of the systems experience any problems with scaling and that the more airplanes that communicate with the system the more weather and news data will be available in our cache.

A conclusion we can draw is that vertical scaling was not suitable for a system like this. Improving resources was not a better solution than adding more resources. While vertical scaling did provide faster run times it was not worth the extra costs to save a few seconds in run time.

5.3 RQ3: Cost

Which AWS architecture and resources will provide the most efficient way to supply real-time data to the airplanes? Since customers onboard would not notice a delay of a few hundred milliseconds extra we prioritized creating a cheap system rather than a very performant one. Our Monolith approach with EC2 and load balancers proved the most cost-effective for a high amount of airplanes. Using Lambda functions gave cheaper systems for fewer airplanes but their cost scaled much faster than using EC2. The linear cost of SNS made the Choreography system much more expensive for 1000 and 10 000 airplanes. Another solution for sending data to the airplane should probably be implemented to lower that cost. Even though the difference in cost is stark between the three systems they can all handle delivering news and weather to every airplane in the sky for a relatively low monthly price.

5.4 Future Work

The testing was done by mocking the communication with the airplane. In order to receive the actual coordinates from the airplane, a new service would have to be added with an additional cost and complexity. All of the API calls were also mocked which was necessary for the testing but is not a perfect representation of how the real system would behave. A proper test with real API calls and actual airplanes would have been interesting to find any potential pitfalls not found in our testing.

Since we did not add any additional security to our systems future developers must investigate how the system should be built inside an AWS VPC. This will most likely add additional components to the system as well as extra costs. We believe a VPC will not impact the performance or scalability of the system to a significant extent.

Throughout the project, we have used Lambda functions for two of our systems. While we have researched and worked thoroughly to optimize the performance of the functions there is still a lot more that could be researched. Concepts such as function warmers and Provisioned Concurrency are topics we have not covered in this paper but are concepts that potentially could help improve performance and reduce costs.

The three systems were built using the Python programming language. As previously mentioned there could be potential benefits to picking another programming language. Experiments should be conducted to compare performance against a faster programming language or a compiled programming language.

References

[1] Lucas F Albuquerque Jr, Felipe Silva Ferraz, RF Oliveira, and SM Galdino. Function-as-a-service x platform-as-a-service: Towards a comparative study on faas and paas. In *ICSEA*, pages 206–212, 2017.

[2] Leonor Barroca, Jon Hall, and Patrick Hall. *An Introduction and History of Software Architectures, Components, and Reuse*, pages 1–11. Springer London, London, 2000.

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice: Software Architect Practice_c3*. SEI Series in Software Engineering. Pearson Education, 2012.

[4] James Beswick. Operating lambda: Performance optimization – part 1, April 2021.

[5] James Beswick. Operating lambda: Performance optimization – part 2, May 2021.

[6] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow, 2016.

[7] Diogo Faustino, Nuno Gonçalves, Manuel Portela, and António Silva. Stepwise migration of a monolith to a microservices architecture: Performance and migration effort evaluation. *Web-based*, 01 2022.

[8] Ahad Khan. 8 reasons why software architecture is important? *LinkedIn*, Aug 2020.

[9] P. Kruchten, H. Obbink, and J. Stafford. The past, present, and future for software architecture. *IEEE Software*, 23(2):22–30, 2006.

[10] Sam Newman. *Building microservices :*. O'Reilly Media,, Sebastopol, Calif., 2015.

[11] Redis. *An introduction to Redis data types and abstractions*. `https://redis.io/topics/data-types-intro`, 2022.

[12] Redis. *EXPIRE key seconds*. `https://redis.io/commands/expire`, 2022.

[13] Amazon Web Services. *Amazon API Gateway.* https://aws.amazon.com/api-gateway/, 2022.

[14] Amazon Web Services. *Amazon ElastiCache for Redis - Global Datastore.* https://aws.amazon.com/elasticache/redis/global-datastore/?nc=sn&loc=3&dn=1, 2022.

[15] Amazon Web Services. *Amazon RDS Features.* https://aws.amazon.com/rds/features/, 2022.

[16] Amazon Web Services. *AWS Skillbuilder: Cloud Essentials.* https://explore.skillbuilder.aws/learn/course/134, 2022.

[17] Amazon Web Services. *Lambda Developer Guide.* https://docs.aws.amazon.com/lambda/latest/dg/welcome.html, 2022.

[18] Amazon Web Services. *What is Amazon CloudWatch?* https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html, 2022.

[19] Amazon Web Services. *What is Amazon EC2?* https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html, 2022.

[20] Amazon Web Services. *What is Amazon SNS?* https://docs.aws.amazon.com/sns/latest/dg/welcome.html, 2022.

[21] Amazon Web Services. *What is Amazon VPC?* https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html, 2022.

[22] Amazon Web Services. *What is AWS?* https://aws.amazon.com/what-is-aws/, 2022.

[23] Amazon Web Services. *What is AWS Elastic Beanstalk?* https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/Welcome.html, 2022.

[24] Amazon Web Services. *What is Elastic Load Balancing?* https://docs.aws.amazon.com/elasticloadbalancing/latest/userguide/what-is-load-balancing.html, 2022.

[25] S.E. Sim. A small social history of software architecture. In *13th International Workshop on Program Comprehension (IWPC'05)*, pages 341–344, 2005.

[26] Mario Villamizar, Oscar Garcés, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano Merino, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano, and Mery Lang. Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures. *Web based*, 05 2016.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2022-06-01

EXAMENSARBETE Exploring Cloud Architectures in AWS for Generating and Sending Real-time Data
STUDENTER Lucas Edlund, Nils Stridbeck
HANDLEDARE Per Andersson (LTH), Magnus Wihlborg (Tactel AB)
EXAMINATOR Ulf Asklund (LTH)

Det riktiga molnet under flygplanen

POPULÄRVETENSKAPLIG SAMMANFATTNING **Lucas Edlund, Nils Stridbeck**

Informations- och underhållnings-system är en stor del av flygupplevelsen idag. Passagerare förväntar sig mer av sin flygresa och det innebär skapandet av nya tekniska lösningar. Detta arbete utvärderar olika systemarktiekturer i AWS för att generera och skicka realtidsdata till flygplan under planets resa.

När bröderna Wright uppfann flygplanet kunde de inte föreställa sig den flygupplevelse som många upplever idag. Internet och underhållning är något som passagerare numera förväntar sig ska finnas tillgängligt på sina flygresor. De förväntas kunna hålla sig uppdaterade med de senaste nytt som händer nere på marken medan de flyger högt över molnen.

Medan de färdas över molnen på himlen, så jobbar ett annat sorts moln hårt med att generera och skicka realtidsdata till dem. I detta arbete byggde vi tre olika system i molnet för att undersöka hur olika sorters arkitekturer påverkar ett systems prestanda, skalbarhet och kostnad. AWS är en molntjänstleverantör som låter en lansera system och applikationer i molnet. Vi använde AWS för att bygga tre olika system som genererade och skickade nyheter och väderprognoser till flygplan baserat på deras geografiska position i världen. Detta betyder att om en passagerare på ett flygplan flyger över exempelvis New York så ska den passageraren kunna få aktuella väderprognoser samt de senaste lokalnyheterna för den staden.

Olika sorters arkitekturer har olika fördelar och nackdelar. System kan bete sig olika baserat på antalet plan det interagerar med och kommer vara mer eller mindre passande för vissa användningsområden. Med vår forskning ämnade vi svara på hur olika arkitekturer kan påverka ett system som genererar och skickar relatidsdata. Med resultatet kan man i framtiden enklare avgöra för- och nackdelar med olika arkitekturer för ett sådant här system samt se vad för möjliga AWS-tjänster man kan använda.

Monolitiska system visade sig ha hög prestanda och låga driftkostnader för färre plan. AWS erbjuder tjänster som låter en semi-automatiskt skala monolitiska system för att anpassa sig till olika sorters belastningar.

Ett synkront mikrotjänstsystem visade sig vara den billigaste arkitekturen för mindre mängder plan och ha högst prestanda för större mängder plan. Då systemet är uppdelat i olika mikrotjänster som kommunicerar med varandra kunde varje tjänst skalas individuellt. AWS erbjuder tjänster för att bygga serverlösa system. Dessa sortens system skalas automatiskt av AWS och är därför lätt att underhålla och skala.

Det sista systemet som undersöktes var ett asynkront mikrotjänstsystem. Detta system innebar fler komponenter och lägre prestanda än dess motparter. De adderade komponenterna resulterade även i en ökad kostnad för systemets drift. Det asynkrona systemet var däremot enkelt att skala och enklast av de tre systemen att utöka. Från en utvecklares perspektiv hade alltså denna arkitektur varit att föredra.