Poznan University of Technology
Faculty of Computing and Telecommunications
Institute of Computing Science

Master thesis

The use of serverless processing and the FaaS model in web application development

Robert Banaszak

Supervisor: dr hab. inż. Anna Kobusińska

Poznań, 2021

Contents

Chapter 1

Introduction

1.1 Motivation

Serverless computing is a new approach to developing and managing applications and services, which is gaining popularity every year. It changes the perspective of the application architecture, which is based on the stateless and short-lived serverless functions, executed on demand to process the workloads. Serverless computing heavily relies on the integration of external services hosted in the cloud, incorporating them in the developed solution to provide the core application logic and additional capabilities. Both of the mentioned components are executed in the cloud environment, without the need to manage the infrastructure and with a pricing model proportional to the used resources. It makes the serverless paradigm an appealing solution to numerous companies, thanks to the benefits it can bring by reducing the cost and improving the development agility, focusing only on the business logic.

The field is still evolving, with industry-leading cloud vendors introducing new services and extending existing ones with new capabilities. However, serverless computing is still not a mature solution, lacking proper standardization. Building efficient solutions using the serverless architecture is not a trivial task, which requires broad domain knowledge and practical experience when working with the services hosted in the cloud platforms. The new approach changes the way how applications are developed, deployed and executed.

At the same time, the field of web applications expanded significantly, making them an appealing medium for various companies to modernise their businesses or startups to introduce new products. The widespread availability of the Internet, as well as the computers and mobile devices, contributed to the significant grow of available services, covering many areas of human life. The development does not only concern the number of available web applications, but also the importance and complexity of performed operations and the volumes of processed data. The provided solutions need to work efficiently, scale according to the number of active users, while at the same time ensure security and compliance, along with a rich and intuitive interface, similar to native applications.

1.2 Objective and scope of research

The key objective of the thesis is to conduct research on how the serverless processing and Function as a Service model can be applied in the web application field. To provide the comprehensive analysis several objectives need to be fulfilled.

In the beginning, it is crucial to provide the necessary context by introducing the field of serverless computing and describing the Function as a Service model. The thorough analysis of benefits and challenges of the researched domain should be identified, along with the comparison of the offerings from numerous cloud providers as well as the overview of existing use cases of serverless architecture.

Moreover, the field of web applications needs to be introduced, summarising the requirements they need to fulfill, outlining the architectures of web applications developed nowadays and discussing their emerging problems.

Thorough introduction of both fields should be sufficient to highlight the crucial aspects in the further analysis. The research of the serverless architecture application in the field of the web application should be conducted, covering the topics of applicability of the discussed architecture, highlighting the limitations and possible solutions and outlining the recommended architectures. In addition to that, providing hands-on implementations along with their further analysis should illustrate the possibilities of the serverless architecture as well as provide a deeper understanding and practical experience of the researched domain.

Finally, the results of the analysis should be concluded, highlighting the key capabilities and possibilities of the serverless architecture in the web application field.

1.3 Structure of thesis

The thesis is divided into five chapters. Each of them introduces an important step in the research process, covering the domain knowledge as well as providing more in depth analysis of the researched area.

The second chapter introduces the serverless processing field. In the chapter, origins and notion of serverless computing is defined. Next, the serverless components are analysed in more detail, providing the information about the Function as a Service model. The overview of benefits and challenges is provided to gain a deeper understanding of the researched architecture. Finally, the comparison of available cloud vendors and offerings of their serverless platform is discussed, along with the example use cases, extending the context by real-world examples.

The third chapter is devoted to the web application domain. The origins and further evolution of web application is described. Furthermore, the requirements which are put in front of the web applications nowadays are thoroughly discussed, providing a direction for further analysis. At the end of the chapter, the architectures of modern web applications are covered in more detail, highlighting the solutions used currently and discussing the trade-offs they create.

The fourth chapter is the main contribution of the thesis, providing comprehensive analysis of the serverless processing applicability in the web application field. Firstly, the research questions are formed and the research approach is specified. Furthermore, the topic of the serverless suitability for the web applications is thoroughly analysed. The practical implementations of the web applications using the serverless architecture are provided to give more hands-on insight into the researched field. Next, detailed overview of the FaaS and serverless processing model is conducted, including an extensive analysis of one of the practical implementations, in the form of a case study. Moreover, the area of suitability of the datastores in the serverless architecture and the serverless databases is presented. Lastly, the discussion of the web application client architecture and used communication patterns are covered.

The fifth and final chapter summarises the research, referring back to the formed research questions and highlighting the potential directions of further research.

Chapter 2

Serverless computing

2.1 Origins

The growth of cloud computing significantly influenced the way how server management and server application development are perceived. Around fifteen years ago, most of the companies were entirely responsible for managing their software, altogether with the hardware and infrastructure it was running on [RC17]. Around that time, first services capable of outsourcing some part of infrastructure overhead emerged, which started the idea of cloud computing.

Amazon Web Services was one of the first service providers that enabled companies to rent computing capacity by announcing the launch of Elastic Compute Cloud (EC2) in August 2006. It was the first Infrastructure as a Service (IaaS) product on the market that allowed companies to run their server applications on Amazon's machines that are billed per usage time and are available within minutes from requesting new resources.

Leveraging such a service model brings a handful of benefits. It reduces the labour cost by outsourcing hardware management to the provider and infrastructure cost by paying based on actual usage of services. Furthermore, it enables companies to scale the number and type of servers in correlation with the traffic and demand for processing. Finally, it encourages testing new solutions developed by companies and decrease the lead time, by making the required infrastructure available within minutes instead of months, utilising favourable billing flexibility.

The cost of IaaS solutions is profitable to providers, because of technical improvements done in terms of hardware virtualization and the economy of scale on which they operate. Shortly after, other vendors such as Microsoft, Google and DigitalOcean embraced the notion of public cloud by providing services and resources from their own data centers. At the same time, tools like Open Stack enabled companies to use hardware from their own data centers in the same way, forming the idea of private cloud.

The next step in the cloud evolution is Platform as a Service (PaaS). As a layer on top of IaaS, it adds operating system to the outsourced infrastructure stack, enabling to deploy the application code directly. In that model, the platform takes responsibility for managing the operating system as well as monitoring and running the application. Google App Engine, AWS Elastic Beanstalk and Heroku platform can be distinguished as most popular PaaS solutions, while one of the most frequently mentioned self-hosted variant is Cloud Foundry.

The growth of containerisation technologies introduced another type of service called Container as a Service. Technologies like Docker allowed developers and system administrators to deliberate more clearly on the application requirements and separate it from the operating system. Solutions such as Marathon, running on top of Mesos, and Kubernetes introduced a possibility to manage and orchestrate containers on self-hosted machines. The services provided by cloud vendors include for example Google's Compute Engine or Amazon's Elastic Container Service (ECS) or AWS Fargate. With the growing popularity of Kubernetes, dedicated services leveraging that platform emerged, including Amazon Elastic Kubernetes Service (EKS) and Google Kubernetes Engine (GKE).

Each of the described services are next generations of infrastructure outsourcing, which raise the level of abstraction from the development perspective and hand off more and more responsibility related to infrastructure management to the cloud vendor. Despite the fact, for each of the mentioned services the smallest unit of processing is some sort of server application or running application process in the virtual machine or within the container.

Serverless is considered as a next step in the cloud computing progression. The term serverless was one of the first used by Ken Fromm in his paper [Fro12]. It describes the notion of architecture migration from monolithic applications running on servers into distributed systems that consist of multiple components, processes and data stores with the goal to perform various tasks and process numerous flows. The serverless architecture enables developers to make a mind shift accordingly. Computing resources can be used as services, which makes it possible to shift thinking from the servers level to the tasks level, taking away the complexity of infrastructure management. The servers are still used underneath, but developers don't need to worry about managing them any longer.

Such an architecture model was leveraged firstly by mobile applications built on top of hosted database solutions, such as Parse (later acquired by Facebook) and Firebase around 2012 (obtained by Google). Nonetheless, the most significant event shaping the serverless architecture was the announcement of AWS Lambda in 2014, altogether with the introduction of API Gateway in 2015. By the middle of 2016, major cloud vendors such as Microsoft and Google embraced the serverless architecture approach and started offering their services for developing serverless applications.

2.2 Defining serverless

Despite the fact that the idea of serverless computing emerged about a decade ago, it has been already widely adopted by leading cloud providers. Currently it covers a range of technologies, components and cloud services. Nevertheless, there is no clear and concise view on what ”serverless” is. Various vendors, organisations and research groups tried to define what the serverless term means for them.

Cloud Native Computing Foundation is the organization working towards standardisation of numerous cloud-related technologies and components. It maintains a sustainable ecosystem for cloud native software by bringing together and collaborating with various members of the cloud community. According to ”CNCF Serverless Whitepaper” [Fou18].

Serverless computing refers to the concept of building and running applica-

> tions that do not require server management. It describes a finer-grained deployment model where applications, bundled as one or more functions, are uploaded to a platform and then executed, scaled, and billed in response to the exact demand needed at the moment.

Another definition considering the serverless service capabilities can be found in a booklet made by Mike Roberts and John Chapin titled ”What is Serverless?” [RC17].

> A Serverless service:
>
> - Does not require managing a long-lived host or application instance
> - Self auto-scales and auto-provisions, dependent on load
> - Has costs that are based on precise usage, up from and down to zero usage
> - Has performance capabilities defined in terms other than host size/count
> - Has implicit high availability

The cited definitions contain insightful information about features of serverless architecture and capabilities of its components. These can be summarized as follows:

- Serverless architecture defines the new model of developing and executing workloads. The application consists of multiple serverless components configured together to run the business logic within the application code, designed to be executed in a serverless environment.
- It does not require to maintain, provision and monitor servers and applications. Serverless does not mean that there are no servers — the overhead of managing them is handed off to the cloud provider.
- Deployment model is more granular. Having the application built from multiple components configured to work together, the deployment can update only selected ones.
- Platform is responsible for provisioning and executing the applications. With a large resource pool maintained by cloud provider and the possibility to quickly allocate it, the solution can be scaled automatically to the current load requirement almost instantly.
- The cost is proportional to the resource usage. Each of the components involved in performing the computation is billed granularly, with an accuracy to hundreds of milliseconds or number of executed operations, with no cost when being idle. Executing hundred operations in parallel will cost the same amount of money as running that workload sequentially.
- The performance is not related to the host size. Some of the cloud providers enable customers to choose how much memory and CPU can be allocated for the environment. Nevertheless, the configuration is abstracted from the capabilities of the underlying machine that is used as an execution environment.
- The serverless components are built with high-availability and fault-tolerance in mind. Despite the fact that developers are no longer concerned with servers, the underlying vendor’s machines can still fail. When using serverless services we expect that

cloud vendors will provide transparent high availability for its services. Although, as developers it may be necessary to handle consequences of some errors and failure occurrences properly.

2.3 Serverless components

When considering the serverless architecture, it refers to a range of technologies provided by cloud platforms. Two different areas can be distinguished, defining two distinctive components:

- **Backend as a Service** refers to third party services or generic components, capable of replacing some part of a server side application, that was previously developed internally or self-provisioned. It exposes an API which allows for integrating the component with the rest of the application.
- **Function as a Service** is an event-driven execution environment for running application code within stateless and ephemeral containers with strictly limited execution time.

Aforementioned components used simultaneously enable developers to build fully-fledged solutions utilising serverless architecture and are offered together within a single cloud platform. Even though presented areas serve different roles, they share common features and capabilities. These could be listed as follows:

- Require no resource management and are entirely provisioned by cloud provider
- Utilise the event-driven model for processing
- Underlying platform ensures automatic horizontal scaling, high availability and fault tolerance
- Billing is proportional to actual usage

2.3.1 Backend as a Service

The concept of Backend as a Service (BaaS) gathers various domain-generic, repeatable application components capable of replacing some part of application logic or providing some functionalities. They can be accessed and integrated into developed applications through API defined by its provider.

Taking into consideration most popular requirements of various applications, the majority of them need to store and manage the data. Depending on the requirements, the information should be stored in a structural way in databases or can be stored regardless of their shape in file storage service. For teams developing mobile or web applications it is convenient to rely on some third party components to store and access the data directly from the application. Services like Google's Firebase meet such requirements and give access to the database entirely managed by a cloud vendor. Depending on the complexity of developed product, it may be required to incorporate some more advanced components to process required tasks or flows, reflecting the business logic of application. Mechanisms serving as notification services, exposing some publish-subscribe capabilities for defined events, could be applicable in that case. It is essential to notice that such components are replacing previously self-hosted components such as databases or other data stores, message brokers or similar services responsible for processing data streams incoming from various sources.

Looking closer from the application logic point of view, there are also repeatable functionality implementations that can be extracted and reused across multiple developed applications. Majority of the products will require some features enabling users to manage their identity and associated permissions. Most of the time, the capabilities will be not only limited to registering and logging users, but it will also include integrations with other services as identity providers. Managing and sending emails can be considered as another functionality that can be extracted into separate component from the code perspective and reused in other applications. Products such as Auth0 (serving as fully featured authorization and user management service) or Mailgun (capable of managing and processing emails) makes it possible to replace entirely the repeatable part of business logic.

Specific components and services that can be classified as Backend as a Services will be covered in more details in chapter 2.5, when analysing services provided by leading cloud vendors. All of the mentioned components enable developers to create fully fledged applications and services similar to the solutions built using self-hosted component equivalents. The difference for given components is the fact that they characterise by capabilities of the serverless architecture. They are provided by cloud vendors, who take responsibility for managing, provisioning and scaling them depending on the demand. Moreover, the possibility to replace the repeatable application features by utilising third party services enables developers to iterate faster and shorten the lead time.

2.3.2 Function as a Service

The second area refers to Function as a Service (FaaS) that serves as an environment for executing application code. It introduces a new architectural approach in terms of developing, structuring and deploying the application logic, which is oriented towards individual tasks and operations.

Mike Roberts briefly described the idea of FaaS [Rob18] based on the definition of AWS Lambda [Ama20a], which is currently one of the most commonly used implementation of a Function as a Service model. Based on his summary several main features of FaaS can be highlighted:

- The main idea of FaaS is to run application code without managing servers and application processes. It is a common feature with other approaches like CaaS or PaaS, where the responsibility for managing applications is placed on the cloud vendor. However, the key difference with FaaS is the fact that the function execution time is strictly limited in contrast to long-lived processes existing in aforementioned services.
- The functions are invoked and executed on the underlying platform in response to specific events occurring or coming in the system. Based on that cloud provider handles resource allocation and runs the function code in an ephemeral container,

created based on runtime needs. These are destroyed shortly after the event is processed by the application logic in the function. Together with limited execution time it is a significant architectural restriction for the FaaS model.

- Given the fine-grained execution model, horizontal scaling can be easily and automatically handled by the cloud provider. When there is an increased traffic in the system, the platform responsible for executing functions allocates resources to create and run more functions, which will be capable of processing the increased traffic. An analogous situation takes place when there is no traffic, therefore there is no need to allocate resources for running functions code.
- As a consequence of greater granularity of the application code and managing the function execution by cloud provider, the deployment process differs from the traditional system. Each function can be independently packed and uploaded to the FaaS platform, which takes responsibility for executing it.
- Most of the cloud platforms do not require to use neither predefined framework nor programming language. Although cloud providers define the list of environments and languages that are supported by the platform, any process that is bundled into the artifact and can be executed from it, is capable of processing the incoming event.

Selected aspects of Function as a Service architecture are covered in more details based on "CNCF Serverless" whitepaper [Fou18] in the following sections.

2.3.2.1 Function lifecycle

Before analyzing the execution model in more details, it is important to examine the deployment process accompanying the development of a function. Alongside with providing the function code, the developer is responsible for specifying one or more events upon which function will be triggered. Additionally, metadata defining for example the function version, environmental variables, execution role and other configuration parameters can be defined. The function and specification prepared in that way is uploaded to the cloud provider and processed by a dedicated builder entity, resulting with a function artifact (depending on the cloud platform and selected runtime it can be a binary file, container image or a package). Next, it is deployed on a cluster managed by a FaaS controller responsible for provisioning, controlling and monitoring function instances based on incoming events.

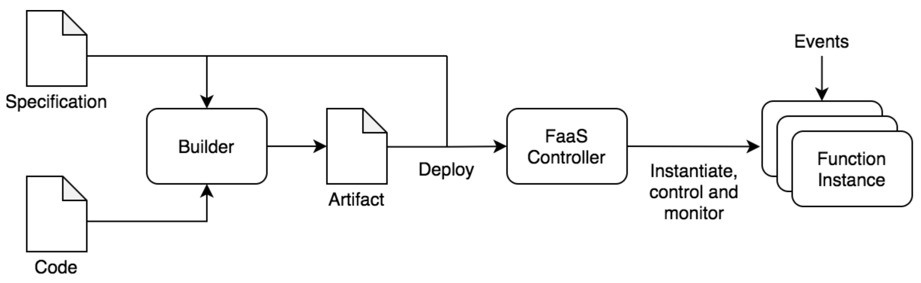

Figure 2.1: Function deployment and invocation process

Furthermore, the serverless platform may provide additional actions related to the function management such as executing, publishing, updating and deleting the function or its metadata. Also, a particular version of the function can be labeled or aliased, which can come in handy when operating the serverless system on a larger scale. Logs and statistics are gathered alongside function execution.

The function is executed in an event-driven model and within strictly limited processing duration. The whole process begins when an event triggering a particular function is dispatched. It is detected and registered by an underlying serverless platform. The controller, responsible for managing function executions, looks for functions associated with incoming events, gets their code and configuration and allocates an adequate amount of resources from the managed resource pool. The new execution environment is created inside a lightweight, ephemeral container and language runtime for the function is bootstrapped. When the environment is ready, the triggering event is redirected and processed by the application logic included in the function code.

The results of computing are sent back to the event dispatcher. Meanwhile, the function can create new events during its execution. The computing duration is limited by the majority of service providers up to a few minutes, after that the computation is completed with timeout. Finally, the lightweight container including the execution environment is destroyed.

Most of the providers delay the container deletion for longer than a couple of minutes due to optimisation related with reusing the same container instance when the next event occurs in the system. Reusing an already initialised execution environment is called a ”warm start” and allows to reduce the startup latency related to resource allocation and runtime bootstrap. The opposite situation takes place when the new container instance needs to be initialised and the host process needs to be created. Most of the time it requires additional time, impacting the request processing duration and it is called a ”cold start”.

2.3.2.2 Function environment

The characteristic of the function environment has been already drafted during description of the function lifecycle. Due to improvements in the software virtualisation field, cloud vendors can benefit from efficient and elastic management of large resource pools. It enables vendors to allocate lightweight and ephemeral containers efficiently, serving as an execution environment. Each of them is initialised to executes code of one of the functions at a time and it is destroyed shortly after, releasing the resources to the common resource pool. Even though containers can be reused for the optimisation purposes when handling subsequent events occurring in the system, it is not guaranteed by the serverless platform and should not be taken for granted by developers, that some data will be preserved for subsequent invocation.

Such an architecture leveraging stateless containers allows for automatic horizontal scaling. When multiple concurrent events occur within the system, the platform is capable of creating a separate container to process each of them independently. In order to handle spikes of traffic effectively, quick provisioning of the containers and reducing their startup latency is required and it is a purpose of many research works and improvements made by both researchers and cloud vendors

As mentioned before, ephemeral containers are dismissed shortly after the execution

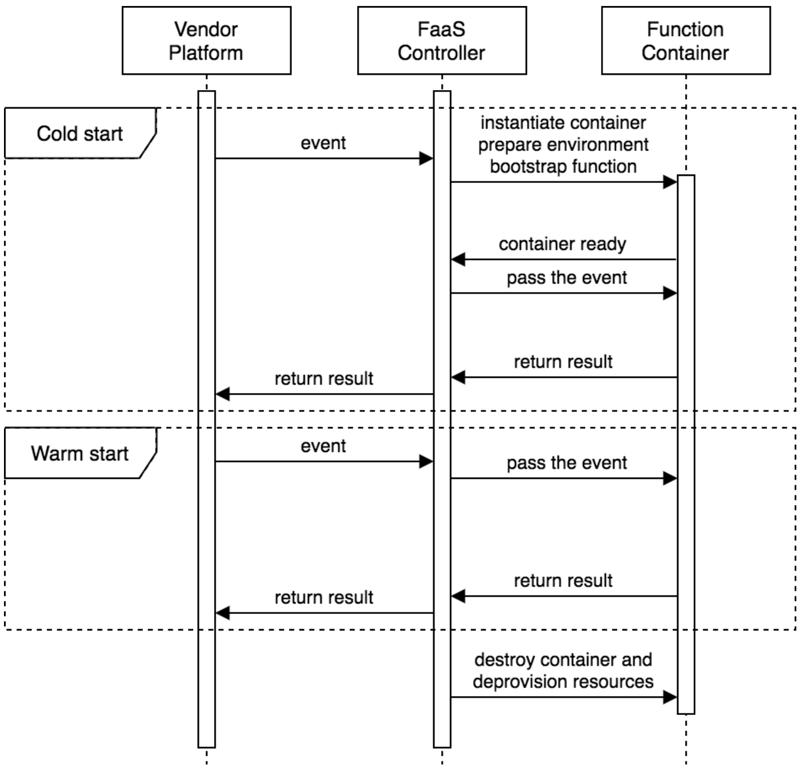

Figure 2.2: Function execution process considering "cold start" and "warm start"

alongside with their internal state. For some types of computing it would be desirable to preserve that state. To address that issue, external components need to be involved to persist the data. Nevertheless, it introduces the need to communicate with some external service and it is often associated with additional delays resulting from the communication overhead.

2.3.2.3 Function invocation

The serverless architecture utilises an event-driven processing model. The developer's task is to define the configuration that maps events coming to the system with appropriate functions. Each of dispatched events can trigger one or more functions as well as each function can be invoked by one or more predefined events — there is a many-to-many association between functions and event sources. The mapping can also refer to a particular version of the function or alias, which can greatly simplify the deployment process by replacing the function code for a given alias without modifying the configuration.

Various data sources can be divided into several categories, distinguishing among the others:

- Endpoint Services - Most of the time associated with API Gateway components, which introduces mapping between request coming from APIs (such as REST or WebSocket) and associate them with corresponding functions.
- Storage Services - Category includes numerous BaaS components provided by a cloud vendor like databases, file storage or cache services. Events can be emitted based

on operations performed on the data such as creation, deletion or modification of it.

- Messaging Services - Services providing mechanisms for data streaming, message brokers or various services sending notifications.
- Scheduled events - That category includes services capable of emitting events periodically at a given time or at a selected interval

Based on the use case, several invocation types can be differentiated:

- Synchronous Request - Includes cases when the client sends a request and waits for the response. It is used most frequently for HTTP requests.
- Asynchronous Message Queue Requests - Refers to events emitted from various data sources, when messages are published to some exchange that later distributes them to other subscribers. Messages are delivered exactly once without strict ordering.
- Event Streams - Are based on streams of messages, logs or files. Sequence of record is most of the time partitioned into several shards.
- Batch Jobs - Refers to jobs that can be split into smaller tasks and processed in parallel by multiple functions. The entire process is completed when all subsequent tasks are finished.

2.4 Benefits and challenges

The emergence of serverless architecture has met with great interest from various software architects, developers and companies that noticed numerous advantages of this approach. In addition to the promise of significant cost reduction, the development and operational opportunities and reducing time to market are just a few of the benefits that serverless introduces.

Nevertheless, the serverless architecture has also many disadvantages, inherently connected with its nature. Some of them have been addressed by various cloud vendors working towards improving their services and mitigating the problems. Despite the fact that many companies have already adopted developing services in the serverless paradigm, the technology is still not fully mature. There is a lot of research going on in various areas related to serverless computing by both researchers and cloud service providers.

Based on the reports and articles discussing the advantages and drawbacks connected with the serverless paradigm [Rob18] [JSS$^+$19] [SK19] [Bol19], the benefits and challenges of the architecture have been presented below, with the latter divided into three distinctive categories.

2.4.1 Benefits

2.4.1.1 Reduced development and operational cost

Serverless is essentially another step in the process of infrastructure management outsourcing, similar to IaaS and PaaS. In this approach, developers instead of requesting resources, provide an artifact including code to be executed and the cloud platform is responsible for provisioning the resources and executing the application logic, based on defined triggers' configuration. The responsibility of managing the servers, databases and application execution is transferred to the cloud provider [JSS$^+$19]. The favourable and competitive prices are available due to economy of scale, in which one vendor is running thousands of predefined services, effectively sharing the infrastructure among the customers.

The labour cost connected with the serverless architecture is also reduced, because there is less work related to managing the serverless solution, compared to the self-hosted alternative. There is no need to setup and maintain hardware as well as restore it to proper condition in case of failures — new resources can be allocated and be ready to work with, within a moment from requesting it. The development cost is also reduced due to the frequent usage of BaaS components, replacing the application elements that were previously developed in-house, while now can be incorporated with the application logic. An example of such approach is aforementioned Auth0 providing the authentication capabilities or Firebase, enabling client applications to directly communicate with server-side databases accordingly, providing proven authentication mechanisms for different types of users and removing much of the database administration overhead [Rob18].

2.4.1.2 Autoscaling with proportional cost

The serverless billing model refers to paying proportionally to the time resources are used, instead of calculating the cost based on the size of cloud resources as in IaaS or PaaS offerings. By tracking the load with a greater fidelity, scaling up quickly in case of increased demand and scaling down in the absence of it, customers are actually charged for the time the code was executed rather than the resources used to execute their software [JSS$^+$19].

Additionally, with the greater granularity, only the components affected by increased demand can be scaled accordingly, compared to the conventional computing. The serverless processing is characterized by the instant horizontal scaling, which is an ability to parallelize heavy workloads almost instantly on demand and deprovisioning resources shortly after, that is an automatic process handled entirely by the cloud provider. It enables companies to shift from the capital expense to the operational expenses when running their software, which removes the need to invest beforehand in costly hardware before the application or service is created and deployed [Bol19].

The cost reduction is the most visible when the service has to process the occasional or inconsistent traffic. In the first case, the processing will be executed based on the event, billed accordingly to execution time and not paying for idle, which is significantly more efficient compared to the self-hosted solution, in which some server is running the application all the time. When running the application in the latter case, it may be necessary to have enough hardware to handle the highest demand to ensure the running service is capable of handling all the requests. With the serverless solution the cloud provider is responsible for scaling to handle the demand. The customers pay only for the additional time hardware was computing to satisfy the higher traffic. Comparing to provisioning virtual machines, there is no risk of overprovisioning (when demand is handled on time, but the resources are not fully utilised) or underprovisioning (when the average processing capacity is optimised, but the highest traffic may not be handled on time or even not at all). These examples are deliberately picked to showcase the significant cost savings of serverless, but on the other hand when the traffic shape is making a good utilisation of running servers, the cost of using serverless technologies can be higher. Lastly, considering the serverless model, any performance optimisations making the function execution shorter or reducing the number of requests to some other BaaS component can accordingly reduce the cost [Rob18].

2.4.1.3 Easier operational management

Considering the scaling is automatically handled by the cloud vendor, there is no need to hire the qualified administrators to manage and scale the application and make sure that the service is running properly, which leads to further cost reduction. The packing and deployment process for serverless solutions is also simplified, compared to deployment of new versions of software to the servers. There is no need to manage some scripts or specified software responsible for performing the deployment, the serverless solution can be packed, zipped with appropriate configuration and deployed to the cloud, where the cloud vendor handles the rest of the process. In the fully serverless solutions the system administration role can be effectively minimised by the serverless platform when embracing the proper process automation [Rob18].

2.4.1.4 Development opportunities

Serverless solution is appealing, due to the possibility to reduce the cycle time. The focus of development can be put on the business logic and new features instead of managing the infrastructure.

The serverless architecture brings new architectural opportunities and provides some interesting architectural properties. For example, the automatic horizontal scaling brings yet another benefit — the software architects and developers no longer need to think about designing the solution to ensure the scalability, since it is provided out of the box by the cloud platform. Similar situation can be noticed for the implicit failover. There is no need to run another instance to pick up the computation once the first one fails. In contrast, FaaS handles implicit failover by retrying the execution on newly provisioned functions once the initial on crash [Bol19].

The cloud platform provides various BaaS components, enabling to incorporate and integrate them with the developed solution. They offer various capabilities not only related with authentication, storing data or exchanging messages in a publish-subscribe manner, but also provide some more sophisticated services including documents and image processing, generating transcriptions or incorporating machine learning for advanced prediction. This gives the opportunities for developers to look at a vast area of new possibilities with a matter of integration and configuration of some of the services.

It is especially beneficial for agile teams, gearing towards lean and agile processes by reducing time to market, which is understood as the time from an idea until the product is available for the customers. When having the better granularity of code, without the need to manage the infrastructure, the new version of a service can be released more frequently, moving towards the idea of continuous deployment. Along with the granular billing model the companies can try out new solutions with minimal cost and friction. The new experiments can be deployed within a moment since the development is completed, enabling the product owners to set the mindset of continuous experimentation. The proof of concept feature with the limited traffic will cost proportionally less or could be even free if the resource utilisation fit into the free tier provided by some of the cloud providers [Rob18].

2.4.1.5 Greener computing

The growth of awareness about environmental problems has influenced the way the data centers are built. To fulfill their energy requirements cloud providers host their data centers near the renewable energy sources to reduce the fossil-fuel emission. Contrary to the typical business and enterprise data centers, in which some of the idle servers are powered up, consuming large amount of energy and impacting environment.

Cloud infrastructure partially mitigates that problem, because the companies are renting the computing resources based on the demand, rather than provisioning the servers on their own, especially when these are run without adequate awareness about the capacity management. When using IaaS, PaaS or self-hosted solutions, most frequently the users are responsible for scaling the services, preferably overprovisioning the resources, which leads to an inefficient energy utilisation. The cloud provider takes the responsibility for the serverless solutions by provisioning the compute resources and handling the capacity decisions to fulfill the needs, leading to far more efficient resource and energy utilisation across data centers and reducing the environmental impact [Rob18].

2.4.2 Challenges related to the nature of serverless architecture

2.4.2.1 Function state management

As mentioned before, the FaaS is effectively stateless. Due to that fact it faces a significant limitation when it comes to the local state. It should be assumed that the state from one invocation will not be available in subsequent invocation of the same function, which is connected with the lack of control over the ephemeral container function it is running in. To share state between subsequent function execution, some external component like database, cache or external object storage is required to preserve the data, which introduces significant communication overhead, leading to latency increase [Rob18]

This highlights that some types of computation, relying heavily on fine-grained state sharing, may not be suitable for the serverless model. Two distinctive types of storage needs to be addressed. First of them is a low-latency ephemeral storage, enabling to transfer data between functions and maintain the application state during application lifetime. Once processing is finished, the state can be discarded. To achieve this some in-memory cache with optimised network and low latency operations can be considered. Nevertheless, the main challenge is related to providing automatic scaling, allocating and deprovisioning resources as well as ensuring access protection along with performance isolation. The second type refers to durable storage, as a long-term data storage with mutable semantics of a file system. It should also be transparently provisioned, ensuring proper level of isolation, security and performance predictability, but contrary to ephemeral storage the data should be durable and its removal should be explicit along with maintaining low cost [JSS+19].

2.4.2.2 Function communication and data transfer

The service built using the serverless architecture is essentially a composition of many functions and BaaS components working together to provide the desired functionality. To achieve this, functions need to communicate and exchange the data. In other cloud services communication can be attained through network addressing, but in serverless architecture functions are ephemeral and anonymous. The function-level addressing is not available, and they need to communicate through intermediate storage or messaging service, introducing additional latency, which can be pretty expensive with finer-grained communication patterns. Additionally, the fact that functions can be allocated and load balanced according to the current utilisation of resources in the data center, impacts the performance when the data needs to be transferred to the function. Due to that fact, the data caching in a serverless environment can be more difficult to be implemented in an effective way, since the functions are placed and executed independently [SK19].

Some of the communication patterns known from machine learning or big-data analysis software, like broadcast and aggregation, require sending more messages and can be less performant when implemented in a serverless architecture. Compared to the software running on virtual machines, the tasks can share a copy of the received data and perform local aggregation to limit the message overhead and amount of data being sent. Moreover, the serverless component utilises most frequently some intermediate component working in a producer-consumer pattern to send the data, which introduces additional communication delay. Due to lack of addressability, the functions are unable to communicate directly, for example calling one another when the data is present or coordinate some distributed operation [JSS+19].

To address that problem, cloud providers could enable developers to assign the group of functions to the same machine instance, reducing the data exchange overhead or compute the communication graph to place functions efficiently, but it would reduce the flexibility of cloud providers and data centers utilisation. Some of the offerings, such as AWS Step Function or Azure Durable Functions, try to address the function orchestration problem by running a sequence of lambda functions as event-driven workflow and efficiently maintain the application state between the subsequent invocations. Another approach proposed by various practitioners considers sort of a hybrid solution to address the problem of externalized state constraint and lack of function addressability. The low-latency application can be run as the regular, long-running server handling the requests and keeping the context in local memory. Later, the process can hand off the fully contextualized request to serverless functions, which can be executed concurrently to perform some computation without the need to lookup for external data.

2.4.3 Cloud platform and vendors challenges

2.4.3.1 Vendor dependency

As with any outsourcing technology, some control of the system is given up to the service provider, which is also the case with the serverless paradigm. The cloud vendor can put some constraints on how the clients use their services in form of unexpected limits or API changes to be more likely to deliver the reliability on its side. Similarly, when using BaaS components, developers no longer need to implement and maintain them, but it is not guaranteed that the external services will be running without some issues or unexpected failures. If the cloud platform would be to satisfy the needs of hundreds of customers or the smaller group, it will most probably choose the majority to ensure accountability of services [Rob18].

2.4.3.2 Vendor lock-in

The cloud provider selection is a significant decision when building some service using the serverless architecture, not only due to their offerings, but also due to differences in their services implementation that make the further shift harder or even almost impossible without major changes. When taking a closer look at the function invocation semantics, each of them depicts the different interface and events triggering it. The divergence also occurs on the level of BaaS components, which expose various behaviours and API that may even require to change the architecture solution in some cases. Moreover, operational tools related with deployment, logging, monitoring and configuration management will most probably differ and require migration.

When migrating the serverless solution from one of the cloud providers to another one, some parts of the system can be translated more easily, but others can have a significant impact on the whole architecture of the developed solution. For example porting of some function code between cloud platforms will not be possible without migrating other chunks of the architecture. Some companies adopt multi-cloud approaches leading to developing and operating the application in a way that is agnostic to the cloud vendor. Most frequently it is a more costly solution, which makes it impossible to apply benefits, optimisations and more specialised components provided by the cloud vendor [Rob18].

2.4.3.3 Startup latency and platform improvements

Serverless functions have much lower startup latency than the application executed on the virtual machines, but because of the frequency of using new function instances, the low startup time is crucial to provide efficient processing. The function startup time consists of scheduling and allocating the resources to run the function, establishing the environment for code execution and initialising the libraries and data structure to run the function code.

When the function is executed for the first time after a longer period of inactivity or when a new version is deployed, the function bootstrapping process will consist of all the steps forming aforementioned "cold start". Providers have already noticed that resources and environment of the function containers can be reused to save some of the bootstrap time. Nevertheless, the most noticeable latency related to the full function setup has a significant impact on the service performance from the user point of view [JSS$^+$19].

Different approaches are currently available to mitigate that problem, but they are coming with an additional cost. The long-running instances can handle low-latency applications well, because they are always running and ready to serve the traffic. Similarly, function can be pre-warmed regularly to ensure the resources and environment is ready to satisfy the incoming request. AWS Provisioned Concurrency is one of the dedicated extensions of the AWS Lambda service, while Serverless Framework includes some plugin, responsible for registering scheduled events that keep functions warm.

Another factor limiting the serverless function execution is constrained execution duration. Various FaaS offerings from the biggest cloud providers limit the execution time to elastically manage their resource pool. However, the cloud vendors could also increase the transparency of services and provide more clear expectations towards their platforms to increase the degree to which customers can rely on them [Rob18].

2.4.3.4 Multitenancy problem

To achieve efficient resource utilisation, multiple cloud functions from different applications and customers are executed on the same virtual machine, enabling the cloud platform to provide affordable services. Cloud vendors are doing their best to provide a proper level of isolation between different tenants and application execution. Most of the largest providers developed mature technology to ensure the high quality of their services, but for other less mature solutions, some problems can be noticeable. Among the most common issues robustness (error in customer's software causes failures of other), performance (one customer takes majority of the machine resources causing others to slow down) or even security (seeing data of other customers) can be distinguished [Rob18].

2.4.3.5 Security concerns

Serverless architecture opens a large number of security questions, taking into consideration the attacks surface of serverless composition which is not fully researched yet. On one hand, the cloud providers should ensure proper security level of their services, by monitoring them and patching eventual vulnerabilities. However, executing the computation within a shared resource pool extends the area of potential attacks, when some process breaks out from the ephemeral function container [Bol19].

To address this problem, different tenants and applications could be physically isolated, but it can impact the resource allocation and the startup time optimisation. The greatest challenge is to provide a proper function level sandboxing, while maintaining the short startup time enabling a shared environment between repeated function invocations. It could be possible by snapshotting the instance locally or using lightweight virtualisation technologies, and it is a subject of active research and improvements [JSS$^+$19].

Another security concern takes into account the direct access to the services directly from the client applications, which requires additional attention, since there is no protection of the server application available in traditional approach. Utilising various services from different providers makes the area of potential vulnerabilities larger and establishes multiple possible ways to breach into the system. The communication between them needs to be properly integrated and protected.

When building some service utilising different functions and components it will require cooperation between them to perform the processing. Each of the functions should have granular access policies, granting only the required permissions to the BaaS components used by the function. Maintaining and validating the security and access policies is not a trivial task, especially when the number of services, including its components and access patterns, is increasing overtime [Rob18].

2.4.4 Development and operational challenges

2.4.4.1 Development and debugging

The serverless computing introduces a relatively new approach to developing services and with the lack of knowledge and proper modeling paradigm it can lead to many development approaches, reducing the quality of service and complicating the collaboration between developers. New ideas and patterns are created and researched to utilise serverless architecture effectively, which leads to high demand for practitioners and architects aware of the best practices and providing reference architectures [SK19].

Developing the fully serverless solution introduces additional difficulties due to inability to replicate the cloud environment locally. Some of the cloud vendors provide tooling to execute the functions locally, for example AWS Serverless Application Model enables to execute functions locally and pass the event payload saved beforehand. Altogether with the execution, some tools enable function debugging, while Azure even makes it possible to run local debugging for functions triggered by remote events. Nevertheless, to verify behaviour of other services it is essential to deploy the service and verify its behaviour in a cloud environment. The idea of distributed monitoring, which enables tracing the flow of a single request across multiple components, is covered by platform services such as Amazon X-Ray and other third party offerings.

Over the recent years, the ecosystem of serverless tooling significantly evolved, providing various products and services, which helps with the development and operational aspects of managing serverless applications and services. The process of deployment and bundling improved with the introduction of Serverless Framework, AWS SAM and AWS Cloud Development Kit, which allow to define the architecture using many popular programming languages. Lastly, the serverless tooling helps with operational aspects enabling various higher level releases approach, supporting traffic shifting, A/B testing, blue-green deployments or canary releases, which are essential when releasing complex, distributed application that consist of hundreds of functions [Rob18].

2.4.4.2 Testing

The unit testing of serverless functions, which are initially pure functions that are mapping input to the output, is fairly simple. Contrary to the integration testing, because most functions use various BaaS components or other external services creating a dependency in tests. Some of the vendors supply mocked implementations or stubs that can be incorporated into tests and imitate the BaaS behaviour. Nevertheless, the idea giving the most confidence is to deploy the service into the cloud environment and conduct some end-to-end tests to verify the quality of software.

The ideal and most appealing approach would be to create a suite of automated integration tests, deploying the functions and infrastructure configuration to the cloud as part of the integration pipeline to verify whether it is working as intended. However, running such tests creates additional cost with every execution, due to resource usage for processing the test data. It may also require more work compared to writing integration tests for regular services created as stateful applications.

One radical change that could be embraced, is the idea of testing in production and monitoring-driven deployment that is related to switching subset of the traffic to the newer version and comparing the observed behaviour with the previous one [Rob18].

2.4.4.3 Monitoring and observability

Monitoring the service created by utilising the serverless architecture can be more difficult due to the ephemeral, granular and distributed nature of the processing. Aforementioned distributed monitoring is especially desired to trace the flow of a request not only to verify if the application works correctly, but also to give more insight into performance aspects directly correlated with the cost [Bol19].

Most cloud vendors provide some tooling to monitor the system and there are many third party offerings helping with achieving better observability in this area. Despite the fact that more of the operational context related with infrastructure management is handed off to the cloud vendor, there is still plenty of administration work with ensuring monitoring, proper security level and verifying that the appearance of alarms when crossing some of the established limits, does not affect the proper service execution. Introducing various checks in the deployment process can ensure that the service quality is preserved. Techniques like preemptive load testing and chaos engineering can help simulate various critical situations and verify that the system is able to cope with them [Rob18].

2.5 Cloud providers

Since the announcement of AWS Lambda in 2014, other companies followed the footsteps of Amazon Web Services and started offering their solutions for building serverless applications and services. Currently, several major cloud vendors provide large variety of services and functionalities forming complete serverless platforms. Along with the public cloud expansion, couple of open-source projects and platforms emerged as an alternative, allowing developers to use FaaS on on-premise hardware and in a private or hybrid cloud. The variety of cloud vendors offerings is significant, some of them expose the fully-fledged platforms with numerous BaaS components that can be integrated together, others provide the FaaS along with the possibility to host the applications in containers, virtual machines or incorporate serveless processing, along the statically hosted websites or cloud-managed database products. When selecting the cloud providers it is essential to consider several factors, like industry adoption, maturity and number of services, ease of integration as well as cost of the services. According to the survey conducted in June 2020 by Cloud Native Computing Foundation (CNCF) [Fou20a] the market share of FaaS platforms is presented in Figure 2.3

Amazon Web Services, **Microsoft Azure** and **Google Cloud Platform** are distinguished as three providers with the biggest market share. Their services are classified into categories and described in more details, altogether with other vendors and companies providing their services in the serverless computing field.

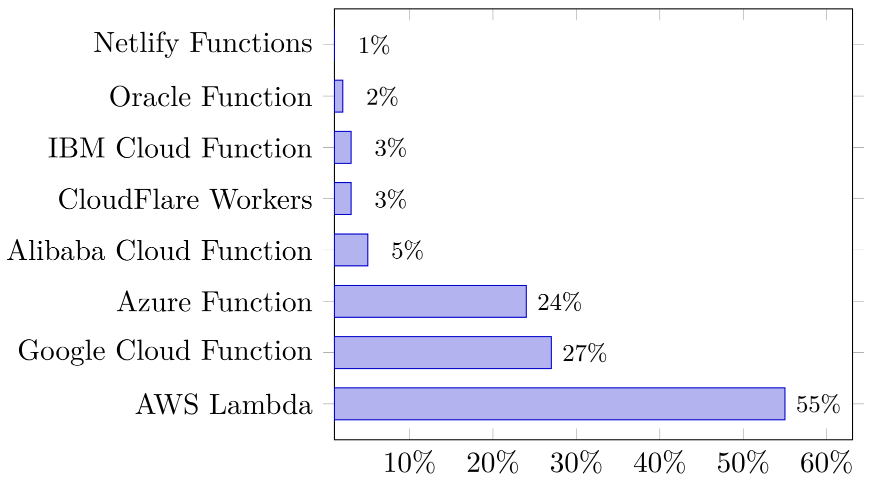

Figure 2.3: Hosted serverless platform usage according to the CNCF survey [Fou20a]

2.5.1 Amazon Web Services

As previously mentioned, Amazon was the company that pioneered the field of serverless computing. Over the years, it established the dominant market position, leading in the number of available services, characterized by high quality and undergoing constant innovation. AWS is an enterprise ready vendor, highly focused on the public cloud sector, providing most comprehensive network infrastructure and data centers.

Moreover, it comes up with rich collection of advanced tools. CloudFormation and AWS Cloud Development Kit enable developers to manage the infrastructure, utilising the Infrastructure as a Code (IaaC) paradigm. Furthermore, the AWS Serverless Application Model (SAM), that efficiently integrate with other AWS services, helps with developing, building and testing the created software. Besides, the essential services necessary to develop serverless applications (described below in more detail based on the official documentation [Ama20b]), AWS offers a wide range of services oriented towards artificial intelligence and machine learning that allow to train and deploy machine learning models, operates on documents, images or speech. Furthermore, AWS provides services capable of connecting and cooperating with various IoT devices, process multimedia or help with managing business processes.

The AWS pricing model is considered as a major difficulty when it comes to accurately estimate the cost of running the service, but most frequently the variety and maturity of services and tooling counterweight this downside.

1. Compute
    - AWS Lambda — First FaaS offering available on the market, that allows to execute the function without provisioning and managing the servers, automatically scaling the solution by running the code in a new instance in response to an event. AWS Lambda supports natively Node.js, Python, Java, C#, Go and Ruby as execution environments, with a possibility to incorporate custom runtimes or container based environments, charging for the execution with the millisecond granularity and ensuring consistent performance by enabling

Provisioned Concurrency.

- AWS Fargate — Serverless compute engine for containers, eliminating the need to manage the servers, choose instances and scale the cluster capacity, providing appropriate level of isolation and security.
- Lambda@Edge — Enable to run the Lambda code at an AWS location closer to the users of the application, utilising the Amazon CloudFront as Content Delivery Network (CDN) to reduce the latency and improve the performance.

2. Data Store

    - Amazon Simple Storage Service (S3) — Storage service for objects, offering high data availability, scalability and performance. It provides the storage of a wide range of data for different use-cases, such as application data, backups, data lake, websites and other assets, ensuring their durability.
    - Amazon DynamoDB — Flexible, scalable and distributed key-value and document database, providing high performance and low-latency data access. Enables data caching, cross-region replication, fault-tolerance and data encryption, being a suitable solution for web and mobile applications.
    - Amazon Aurora Serverless — Autoscalable and fully managed configuration for Amazon Aurora, which is a relational database compatible with MySQL and PostgreSQL, enabling higher performance and availability.
    - Amazon RDS Proxy — Fully managed and highly-available database proxy for Amazon RDS, which is a service that provides hosting and scaling for other relational databases.
    - ElastiCache — In-memory data store, compatible with Redis and Memcached suitable for data intensive applications with high-throughput and low-latency data access.

3. Network and Content Delivery

    - Amazon API Gateway — Managed service, enabling to expose APIs and connect with other AWS services, providing real-time two-way communication via REST or WebSocket API, billed accordingly to usage.
    - AWS AppSync — Another fully-managed and scalable service, capable of defining and handling the two-way communication using GraphQL API as well as the heavy-lifting and communicating with other AWS services to store and sync the data.
    - Amazon CloudFront — Fast, secure and programmable Content Delivery Network (CDN) hosting the data such as videos, images or applications, utilising the AWS edge network location to make the service globally scalable and accessible.

4. Application Integrations

    - Amazon Simple Notification Service (SNS) — fully managed publish-subscribe service enabling high-throughput, many-to-many messaging for communication between applications, microservices and application-to-person messaging.

- Amazon Simple Queue Service (SQS) — message queueing service, allowing to decouple components of the distributed systems, available in two types: standard (offering high-throughput and best effort ordering) and FIFO (designed for ordered messages delivered exactly once).
- Amazon EventBridge — Scalable serverless event bus, enabling to integrate the incoming messages from other SaaS offerings with the AWS infrastructure.
- AWS Step Functions — Serverless function orchestrator, allowing to combine AWS Lambda functions into workflows based on business processes, visualising them and maintaining the application state between execution.

5. Analytics
   - Amazon Kinesis — Offers key capabilities of scalable and fully-managed event streaming service, processing and analyzing real-time video, audio and other data streams instantly, without necessity to collect the data.
   - Amazon Athena — Interactive and managed query service for Amazon S3 based on predefined schema, using standard SQL syntax to quickly analyze large-scale datasets
6. Security and Identity
   - AWS Identity and Access Management (IAM) — Required to manage users, groups and their access to AWS services and resources in a secure way, offered without additional cost.
   - Amazon Cognito — Provides authentication service and identity management that can be easily incorporated with various clients solutions on different platforms.
7. Development Tools
   - Amazon CloudWatch — Service providing application monitoring and observability, gathering logs, metrics and capturing events within the AWS resources. Enables to analyse the environment behaviour, troubleshoot issues and take automate actions based on numerous indicators.
   - AWS X-Ray — Tool that allows to analyze and debug distributed application behaviour across multiple services based on the request tracing, gathering the execution metric to identify the application issues and performance bottlenecks, available for both development and production applications.
   - AWS Amplify — Suite of tools and services, helping developers build efficient and scalable applications for web and mobile platforms, providing possibilities to deploy the static sites, easily manage content or configure the application backends with authentication and data storage.
   - AWS CodeStar, AWS CodePipeline, AWS CodeBuild, AWS CodeDeploy — Set of tools helping with the automation, continuous integration and deployment workflows for various applications.

2.5.2 Microsoft Azure

Microsoft is the second from the biggest three vendors that started offering its serverless computing services in 2017, by integrating them into the Microsoft Azure platform. It also provides a vast number of services, giving a more flexible and configurable runtime engine, supporting more programming languages.

When investing in the serverless field, Microsoft decided to repurpose its existing proprietary software to be available and greatly integrated with the cloud. Microsoft Azure is eager to cooperate with various enterprise companies, especially using their software. Contrary to the AWS, Azure platform can more effectively interoperate with customers' data centers, enabling them to the gradual migration of on-premise software or operate in a hybrid cloud model. Azure platform also provides some powerful DevOps tooling and numerous services, integrating machine learning and artificial intelligence, exposing among the others cognitive services, chatbots and document, images and video analysis and recognition.

Nevertheless, customers frequently are not satisfied with the Microsoft Azure complex pricing model, due to complicated software licensing options that are hard to understand without outside support. Moreover, some of the services are considered less enterprise-ready, due to issues with quality of these services, technical support and lacking documentation or when managing and configuring some of them outside of the Azure platform.

The current serverless offering of Microsoft Azure is presented in more detail, according to the information available on the vendor website [Azu20].

1. Compute
   - Azure Functions — FaaS offering from Microsoft, integrating with other services, executing in an event-driven manner and autoscaling based on demand. It provides tools for building and debugging functions locally, supporting implementations in .NET Platform (C#, F#), Java, Node.js (JavaScript and Typescript), Python and PowerShell, billing the execution on a per second basis.
   - Azure App Service — Fully-managed platform for building, deploying and scaling web applicationsm written in .NET, Node.js, Java, Python and PHP, running in Windows or Linux containers, integrating with other services from Azure platform.
   - Azure Kubernetes Service — Highly-available and secure managed Kubernetes service, automatically provisioning serverless infrastructure based on the traffic, utilising open-source tools like Virtual Kubelet for provisioning nodes and KEDA for event processing from various event sources.
2. Data Store
   - Azure Blob Storage — Scalable, secure and durable object storage, capable of storing a large amount of unstructured data objects, suitable for cloud-native and mobile apps. It provides high availability and geo-replication, generating events pushed to other Azure services upon data changes.
   - Azure Cosmos DB — Fully-managed, globally distributed and multi-model NoSQL database service, guaranteeing low-latency data access and scalability,

exposing APIs for SQL, MongoDB and Cassandra.

- Azure SQL Database — Family of SQL cloud databases, providing optimized performance, data durability, backups capabilities and adapting to requirement changes.
- Azure Cache — Managed in-memory data store, which can be utilised as caching layer, providing high-throughput and low-latency operation time, scaling according to demand with geo-replication and Redis capabilities.

3. Network and Content Delivery
   - API Management — Service enabling API management across cloud and on-premise environments with unified management experience, focused on security and observability with fine-grained data exposition rules and integration with other services.
   - Azure Content Delivery Network — Secure and reliable Content Delivery Network (CDN), greatly integrated with other services from Azure platform, ensuring proper security level and analytics features.
4. Application Integrations
   - Azure Event Grid — Service handling the message routing from various event sources, working in a publish-subscribe model, ensuring scalability and high-reliability.
   - Azure Event Hub — Managed, scalable and geo-replicable real-time data streaming service, enabling to build dynamic data pipelines, capable of seamless integration with other Azure services.
   - Azure Service Bus — Reliable and scalable messaging services for a cloud, enabling to build communication between decoupled application components and on-premise systems, providing various messaging models and offline delivery.
   - Logic Apps — Service enabling developers to build various workflows within containerized environments, integrating external services and enterprise SaaS offerings along with the Azure platform components.
   - Azure Durable Functions — An extension for Azure Functions, enabling them to execute stateful computation in a serverless environment by utilising the Orchestrator that defines the state flow between subsequent functions.
5. Analytics
   - Azure Stream Analytics — Serverless real-time analytics, enabling to build streaming pipelines with similar SQL-like syntax, integrating with other platform services including artificial intelligence for more sophisticated use-cases.
6. Security and Identity
   - Azure Active Directory — Service enabling identity management, authentication and authorization capabilities for end users. Enhancing it with single sign-on to multiple SaaS applications, multi-factor authentication functionalities and integration with external identity providers.

7. Development Tools
   - Azure Monitor — Fully managed and scalable monitoring offering, integrated with numerous Azure services, providing operational telemetry to analyze the cloud and on-premise services behaviour and performance, allowing to query and visualize the data and trigger alarms based on thresholds or patterns detected by artificial intelligence to proactively notify about anomalies or issues
   - Azure DevOps — Cloud hosted service, providing a fully-fledged set of operational tools such as code repository and task management capabilities, enabling configuration of continuous integration and continuous delivery pipelines along with building artifacts and deploying software to multiple platform or on-premise services.

2.5.3 Google Cloud Platform

Google introduced its serverless products from their cloud offering in 2018 into general availability, after a long period in beta. It is noticeable that the Google Cloud Platform offering is not as rich and varied, and the services are not as mature and configurable, as those offered in the AWS or Azure catalog.

The vendor is looking to cooperate closely with companies trying to scale quickly and startups, rather than large enterprises. Besides the serverless platform, the provider is highly focused on containerization and microservice architecture, with the leading in the Kubernetes services along with strong open-source commitments and DevOps-friendly approach. Google Cloud Platform is a leader in fields of artificial intelligence, machine learning and big data, offering numerous services and capabilities based on the open-source frameworks and libraries maintained by Google.

The provider gives a better impression to its customers in terms of the ease of setup and user-friendliness of the platform. The pricing model is aimed to be more customer friendly, offering exceptionally flexible contracts, appealing to customers interested in using the cloud.

Google Cloud Platform services, based on the available offer [Clo20b], are presented below:

1. Compute
   - Cloud Function — FaaS platform from Google, enabling to run the code without server management, scalable with the size of workload. Currently it supports Node.js, Python, Go, Java, .Net and Ruby runtimes. It is priced based on number of invocation and execution duration with granularity to 100ms.
   - Cloud Run — Scalable and fully-managed container-based execution environment built on top of the Knative open-source project, triggered based on various platform events, automatically replicated across multiple regions and billed according to actual usage.
   - App Engine — Hosting platform for mostly web applications written in Node.js, Java, Ruby, C#, Go, Python, or PHP, responsible for managing the infrastructure and scaling the solution

2. Data Store
   - Cloud Storage — Object storage ensuring security, durability, data access with low latency and geo-redundancy, enabling to store data across different storage classes characterised with different parameters and cost.
   - Cloud SQL — Managed and cloud-based relational database service for MySQL, PostgreSQL and SQL Server, ensuring reliability, automate database provisioning and storage management.
   - Cloud Spanner — Database with relational semantics and strong consistency with unlimited scaling, delivering high-performance transactions across regions, automatically handling scaling and sharding
   - Firestore — Serverless NoSQL document database, scaling with the demand with no maintenance overhead, with built-in live synchronization, ACID transactions and offline support. Designed to be used with mobile, web and IoT applications with direct connection to the clients, integrating with other Google Cloud Platform services.
   - Cloud BigTable — Fully-managed and scalable NoSQL database service, designed for machine learning and big data services, seamlessly scaling to the storage needs, capable of processing high-throughput data with low latency
   - Memorystore — Low latency, scalable and secure in-memory service, compatible with Redis and Memcached, enabling high-availability, automatic failover, patching and monitoring.
3. Network and Content Delivery
   - Cloud Endpoints — Development, deployment and management tool for APIs, providing logging, monitoring, access control and integrations with third party services as identity providers.
   - Cloud CDN — Globally distributed and reliable Content Delivery Network (CDN) for images, videos, webpages integrating with other services and supporting hybrid and multi-cloud architectures.
4. Application Integrations
   - Cloud Pub/Sub — Event-driven messaging system and analytics stream, autoscalable, with no provisioning and cross-zone message replication, enabling in-order messaging with push and pull models
   - Cloud Tasks — Fully managed service that allows to handle and distribute vast number of distributed tasks asynchronously, building more decoupled applications in microservice architecture that can be scaled independently
   - Cloud Scheduler — Managed cron job service suitable for batch and big data jobs as well as cloud infrastructure operations, automating tasks management and handling retries in case of failures.
   - Workflows — Workflow orchestration service, integrated with multiple Google Cloud Platform components, focusing on modeling workflow logic, executing it in a reliable way, passing the execution state between particular steps.

5. Analytics
    - BigQuery — Serverless data warehouse based on real-time data stream with autoscaling and managed infrastructure, enabling multi-cloud capabilities, integrated with many tools and services of Google Cloud Platform in the fields of machine learning, artificial and business intelligence.
6. Security and Identity
    - Identity and Access Management (IAM) — Service provides management capabilities for fine-grained access control for Google Cloud Platform resources along with monitoring and auditing them.
    - Cloud Identity — Service providing unified Identity and Access Management for applications and endpoints, enforce strong security and access control, enabling access to thousands of applications via single sign-on
7. Development Tools
    - Cloud Build — Serves as a serverless service for building, testing and deploying artifacts and applications, utilising integration and continuous delivery pipelines
    - Cloud Logging — Real-time log management and analysis services, capable of extracting data from logs and analyzing them in real-time.
    - Cloud Monitoring — Service for collecting metrics from various sources, visualising them. It monitors applications and services behaviour, and integrate with other third party solutions to provide better visibility and observability of the system.
    - Cloud Trace, Cloud Debugger, Cloud Profiler — Set of tools helping developers with tracing events in the distributed system and collecting numerous metrics, debugging the production application to get more insight into their behavior and profiling the application performance and resource utilisation.

In addition to the Google Cloud Platform services, Firebase [Goo20] is another and independent service, backed by Google and based on their cloud platform, providing BaaS components that can be directly integrated with web and mobile clients. All of the services are auto-scalable and with the infrastructure managed on the provider side. The core feature is a real-time document database utilising Firestore underneath, integrating with Cloud Functions, Cloud Storage and providing authentication capabilities. Moreover, the offering includes other components that allow developers to monitor the performance and stability of applications as well as analyse the customers behaviour using A/B testing, boosting their engagement by in-app and cloud messaging.

2.5.4 Other cloud providers and services

2.5.4.1 IBM

IBM Cloud focuses on helping enterprises migrate to the cloud, extend their capabilities and transform their workloads to be more agile utilising cloud infrastructure. Over the years the company invested in industry-focused cloud services and hybrid cloud market, supporting diverse workloads including SAP, Oracle ERP and newer technologies including machine learning capabilities. Company promotes the open-source initiatives to produce the results, along with portable components that can be used among multiple cloud providers [Gar20].

In terms of serverless components, IBM Cloud Functions is based on open-sourced Apache OpenWhisk platform, which manages the infrastructure and scaling using Docker containers and can be deployed independently on top of popular container frameworks like Kubernetes, Mesos or OpenShift. The platform supports a programming model in which functions (called Actions) are dynamically provisioned in response to associated events (called Trigger) coming from numerous external sources or HTTP requests. Currently, OpenWhisk supports a large number of environments, such as Node.js, Go, Java, Scala, PHP, Python, Ruby, Swift or any executable running inside the Docker container [Fou20b].

2.5.4.2 Alibaba Cloud

Alibaba Cloud is a cloud provider that have a strong ties with Chinese public sector as well as cooperates with numerous companies operating in the China and Southeast Asia region. The cloud vendor supports its clients with cloud migration and helps traditional enterprises to build their presence in the digital platforms. Most of its clients reported satisfaction with the platform solutions, but the limitations can be noticed in a modest international offerings compared to the services available in China region. Moreover, the discrepancies in documentation and functionalities availability [Gar20] occur.

The Function Compute is a serverless FaaS offering with a pay-as-you-go model, managed and scaled by the cloud provider in response to events incoming from various data sources. It supports mainly Node.js, Python, Java, PHP, C# and custom runtimes in containers. The cloud platform provides two function instances — the flexible functions, and instances with the higher specifications for better performance. Additionally, vendor offers reserved instances that are always on, reducing the "cold start" [Clo20a].

2.5.4.3 Cloudflare

Cloudflare is a company providing services in areas of web infrastructure and website security, such as Content Delivery Network, domain name servers and internet security services, serving as reverse proxy for websites. It also provides numerous analytics that gives insight into transfer speed, incoming traffic from unique users and their geographic location.

From serverless offerings, Cloudflare Workers are a FaaS platform based on the edge computing, providing high-performance. The solution is also managed, auto-scalable and billed in a pay-per-use model, similar to other serverless offerings, but the difference is in the Cloudflare's edge network, that routes the request to the closest location where the function is executed, limiting the latency and providing high availability. Currently, the platform supports Node.js, Rust, C, and C++ as an execution runtimes [Clo20c].

2.5.4.4 Oracle

Oracle Cloud is expanding its worldwide presence and their cloud services into numerous regions, developing hyperscale cloud architectures that are competitive with other, more-established cloud providers. The vendor tends to cooperate more closely with enterprises, using its business applications and enable to incorporate and integrate them within the cloud platform. All of the cloud offerings are available in multiple regions with the same capabilities [Gar20].

Oracle Cloud Functions are offered in pay-as-you-go model, managed and scaled by the provider and they are based on the open-source Fn Project and CloudEvents, which is an open-source specification for describing event data. It supports Go, Java, Node.js, Python, Ruby and C# programming languages and custom runtimes using Docker to execute the functions. The platform is vendor-agnostic and can be also executed in private and hybrid cloud [Pro20].

2.5.4.5 Netlify

Netlify is offering hosting and serverless backend services for static websites and web applications. The deployment process is easy and convenient for developers, which can connect Git repositories with the platform, enabling automatic continuous deployment triggered with every code change. Netlify extended its portfolio with other services such as serverless form submission, user identity management, analytics and also their own headless content management system, which can be utilised by dynamic web applications.

Currently, the offer also includes serverless functions powered by AWS Lambda, that can be automatically provisioned when the platform detects the configuration in the linked repository. The Netlify Functions can be triggered based on HTTP request to the defined API endpoint or form submission, as well as enabled to run asynchronous processing in Background Functions with longer execution time. Altogether with the whole platform, it allows developers to quickly and easily setup the production environment for the web application or static website without any operational knowledge [Net20].

2.5.5 Open-source alternatives

The benefits of serverless architecture, such as cost reduction and eliminating the need of infrastructure management, became an appealing argument to use the technology. Nevertheless, some companies invested heavily in their hardware infrastructure, are afraid of the vendor lock-in or computation restriction of the public cloud platforms. Open-source frameworks are promising to overcome these limitations and manage the serverless components on self-hosted environment or hybrid clouds.

According to the aforementioned survey conducted by CNCF [Fou20a], the most popular solution in the category of installable software for serverless function execution is newly added Knative (claimed to be used by 27% of respondents), while OpenFaaS maintains its popularity (used by 10% of interviewee) with Kubeless (mentioned by 5% of respondents).

All of the projects are based on the Kubernetes as a portable and extensible platform, enabling declarative configuration and management for containerized environments.

Serverless frameworks rely on it to orchestrate and manage the serverless functions inside containers, schedule their executions, provide service discovery and replication. Each of the open-sourced serverless frameworks approach the container orchestration is slightly different. For example, OpenFaas provides its own API Gateway, providing access to the function, collecting metrics and scaling the solution, while others like Knative, utilise third party Ingress Controller for Kubernetes and manage the scaling by Queue-Proxy Container, passing the events to functions inside the pod. Several articles consider the viability and quality of services for the open-source serverless platforms [LKRL19], considering also interesting possibilities of utilising serverless in edge computing [PKC19].

Nevertheless, utilising open-source platforms raises questions about maturity of these frameworks and introduces another set of challenges, which need to be researched and addressed. Based on the previously mentioned papers some of the self-hosted serverless platforms show issues with scaling, lack of predictability and guaranteeing good quality of service. Each of the frameworks provides default fault tolerance for container runtime, limited to retrying the execution which sometimes may be not suitable. All of the public cloud providers mendtioned above tend to supply various tools and services that support developers with log aggregation, monitoring, alerting and distributed tracing across the serverless components, which requires manual integration in the open-source alternatives. Public cloud vendors offer not only the FaaS platforms, but also provide a large variety of BaaS components that can be integrated with serverless functions to build fully-fledged applications and services. This is in contrary to the open-sourced alternatives, which focus solely on the FaaS platforms. While the other components could be also self-hosted, integration with them as well as scaling requires manual management on the customer side, increasing the amount of work that needs to be done to maintain the services quality and going against the idea of serverless.

2.6 Example use cases

2.6.1 Application backend

Serverless technologies are an appropriate solution for building scalable application backends for all kinds of web, mobile and desktop applications. They are favorable for this use-case, because the infrastructure management can be effectively automated, ensuring dynamic scaling to meet uneven demand, with proportional and predictable billing. Despite the fact that serverless technologies are a relatively new approach to building services, some of the companies have already used it to power large applications.

To build the application backend, serverless functions are used to transform the data representing the application logic, together with other third party services capable of persisting data or exposing some desired functionalities. The API Gateway provides uniform access to various serverless functions using REST API, preserving mapping between requests and respective serverless functions, enabling integrations with other services, for example responsible for authentication and authorisation. Sending a request to the API can lead to execution of one or more serverless functions, containing custom business logic and communicating respectively with other services. It is also possible that the frontend application can communicate directly with external components, bypassing the API Gateway. Nevertheless, it is essential to handle the communication in a secure manner using the delegation tokens [Sba17].

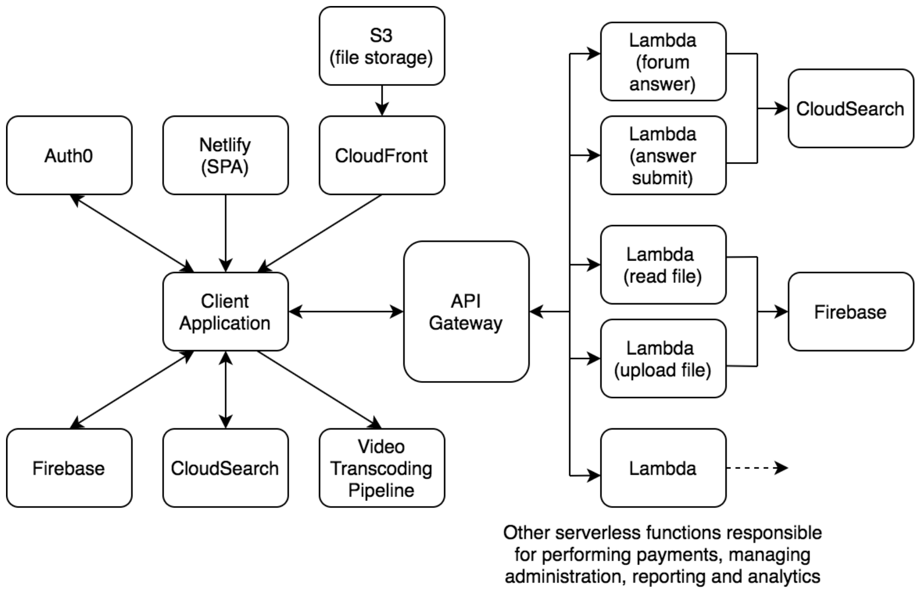

Figure 2.4: CloudGuru platform architecture

CloudGuru, which is an online educational platform dedicated to people interested in learning numerous cloud related topics, is a good example of such an use-case. Core features of the platform include streaming large selection of video courses, interactive quizzes and practice exams, real-time discussion forum and integrations with third party services, enabling users to buy access to the courses.

The architecture of the e-learning platform utilises services of several cloud providers. Frontend of the web application built as a Single Page Application is hosted on Netlify, which acts as a Content Delivery Network (CDN) for the web client. Auth0 service is responsible for authentication functionalities and provides delegation tokens for client application, communicating directly and securely with other services. Firebase is used as a primary database, which is capable of updating clients in real-time using WebSockets to push the data to the client applications. Questions and answers submitted by users to the forum are persisted in the Firebase, with the data being sent further to the AWS CloudSearch, which is indexing it for searching purpose and enabling users to later find information easier. Instructors can upload video directly to S3 bucket, that triggers the video transcoding pipeline. Users can watch the videos served via CloudFront acting as a CDN, if they call the Lambda function beforehand, which gives them the permission to access the video assets for a limited period of time.

2.6.2 Event-driven data processing

Serverless has also found application in numerous media conversion and transcoding as well as various data processing tasks. When the new file is inserted to the bucket responsible for data storage, it can trigger a serverless function passing the necessary data and file as an event to perform the processing. Such an event-driven approach is suitable for building pipelines for data-processing tasks, billed only for the function execution time and other BaaS components usage to transfer, transform or extract some information [Sba17].

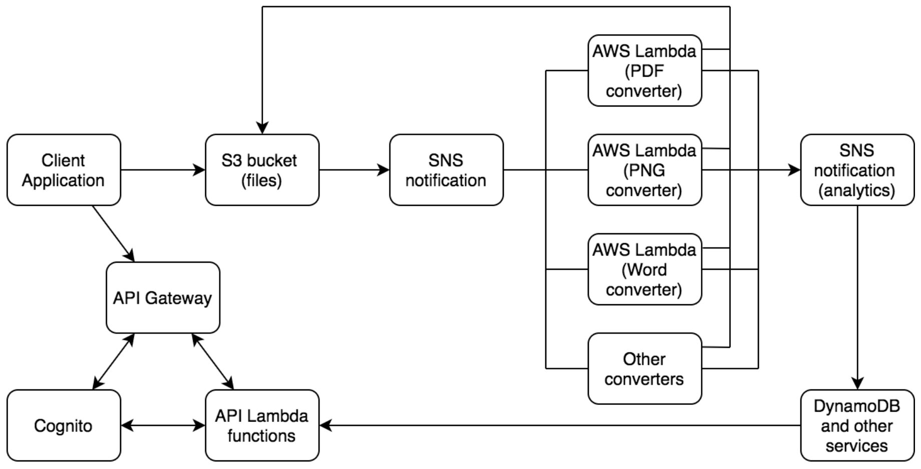
Client Application
S3 bucket (files)
SNS notification
AWS Lambda (PDF converter)
AWS Lambda (PNG converter)
AWS Lambda (Word converter)
Other converters
SNS notification (analytics)
API Gateway
Cognito
API Lambda functions
DynamoDB and other services

Figure 2.5: MindMup platform architecture for file export

An example of such serverless utilisation is MindMup, which is a popular web application serving a role of a mind mapping tool, which was initially developed by just 2 developers. The event-driven data processing is utilised when a user wants to export prepared diagram to some read-only format such as PDF, Word document, presentation or text file. The web application works as interactive user interface. Upon user requests to export the diagram, it calls the Lambda placed behind the API Gateway to generate the presigned URL, which can be used for a short period of time to insert the file directly from the client application to the S3 bucket. It sends an event to the Amazon Simple Notification Service (SNS) triggering the appropriate function responsible for export to a particular file format. Each of the exporters have different requirements, for example CPU and memory usage for PDF exporter is much higher than for others, while Microsoft Office exporters require usage of libraries written in Java. Moreover, some of the extensions such as Markdown are not as frequently selected by users. When the processing is complete, the output of the process is inserted to the S3 bucket, where the users can access it. Additionally, the event is sent to SNS which triggers other services such as DynamoDB, that are listening to that topic to save some information for analytics purpose.

The company admits that initially, the service responsible for exporting was running on the Heroku platform in the containerised environment, but after migration to the AWS Lambda, they have noticed significant cost reduction even with the growth of the platform usage by the clients [SS19].

2.6.3 Real-time stream processing

Serverless architecture also found application in the data ingestion tasks from numerous data sources, such as logs, system events, transactions or social media feeds that need to be analyzed, aggregated and stored. Utilising BaaS components capable of event stream aggregation over time with the serverless function which can be configured to run when the specific number of records (forming a batch of records) is available. Such a messaging pattern is popular in distributed systems, where it allows to decouple the services and provides reliability by storing the messages in the queue, when the consuming service cannot process more data at the moment, additionally providing retry mechanism in case of failures [Sba17].

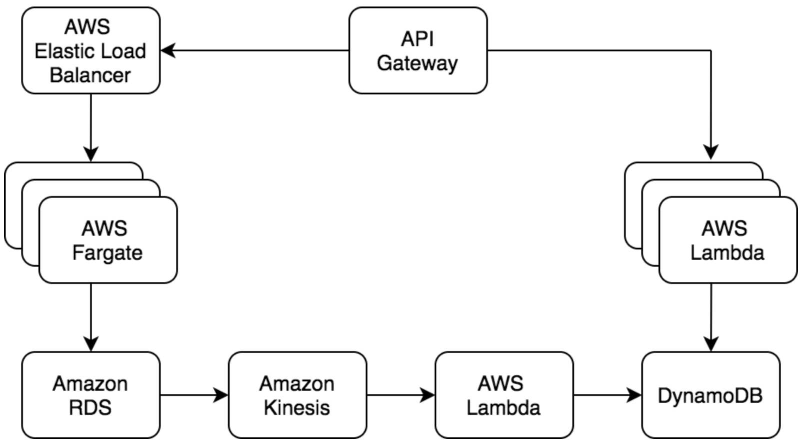

Figure 2.6: Temenos platform architecture

Temenos, as one of the largest banking software providers, is using such a pattern along with other serverless technologies in their T24 Transact banking system to achieve a highly elastic solution with low maintenance cost. It supports a large number of interactions and transactions between banking customers worldwide. Due to leveraging the platform and managed services, it reports that even the unpredictable workloads due to peaks in the market are handled efficiently. All the requests coming to the systems are getting through the AWS API Gateway. It passes them to the AWS Elastic Load Balancer (ELB), responsible for routing the requests to proper container instance running on AWS Fargate. The application running inside the container, performs the business actions based on the request and persists the result in Amazon Relational Database Service (Amazon RDS) that is an Online Transactional Processing Database (OLTP) designed to process the transactions effectively. Later, the stream of data and more complex business events is pushed from Amazon RDS to Amazon Kinesis, which aggregates the data and triggers the AWS Lambda responsible for processing it, to build a query-optimized data model which is stored in DynamoDB. When the banking system users want to read the data, they perform a request that goes from API Gateway to AWS Lambda, that queries the read-optimized data model from DynamoDB [Ama20c].

2.6.4 IoT applications

With the growth of smart homes popularity, IoT devices along with other machines that are meant to help people, started incorporating more sophisticated technologies to support people on a daily basis. iRobot is a leading company in terms of designing and building smart robots, helping people around their household, with the Roomba robotic vacuums as their most popular product. Besides robotics, the company provides iRobot HOME application which enables customers to interact with their smart home devices.

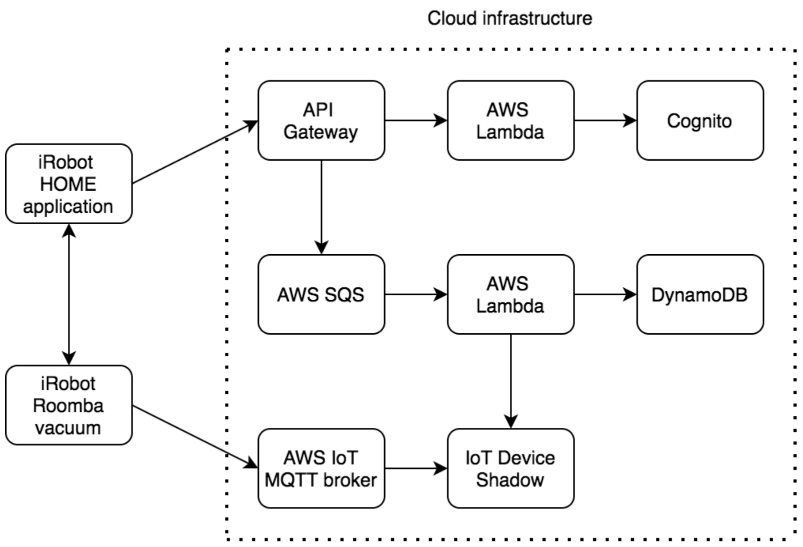

Figure 2.7: iRobot solution architecture for smart robots

AWS IoT is an entry point for the Roomba vacuums to connect them with the cloud. For the communication purposes MQTT is used, which is a lightweight publish-subscribe protocol for IoT devices. The AWS IoT Shadow Device provides asynchronous communication mechanism, enabling the robot to work properly without the constant communications with the cloud. When the smart vacuum finishes cleaning job, its data is synchronized with the cloud, which allows users to update the cleaning schedule, accordingly to changes made in the iRobot HOME application. Along with the AWS IoT, developers can define additional IoT Rules that configure the device to connect with other services in the cloud, for example to update the device status along with its parameters or informing users that the vacuum tank is full. Moreover, it gathers additional information about usage and home mapping to better plan device's tasks and manage the battery power [Keh20].

Developers and designers of the robotics solutions observed many benefits related to the utilisation of cloud vendor services and the serverless paradigm, mentioning the scalability of the solution without the need to manage infrastructure and significant cost reduction, while preserving the proper level of security and data privacy [Ama20d].

Chapter 3

Web applications

3.1 Origins

The idea of web applications has been changing over the years, alongside with the increased interest in using them. Currently, web applications can take a variety of forms, which lead to the emergence of various architectures and solutions, suitable to develop efficient and user-friendly software working in the browsers. Niels Abildgaard [Abi18] analyzed in a great detail how the perspective of web application evolved over the years, discussing the variety of web application architectures and some of the common patterns used in their development nowadays.

The initial idea of the World Wide Web was to exchange information more easily with a broader audience. To achieve this, the client-server architecture was used, in which the client received desired information from the server, responsible for storing them. The client could make a request to the server using HTTP protocol and receive some static content (in form of HTML file, other static assets) or dynamically generated content (returned as a result of program execution triggered by the server). The term web applications is more suitable for the latter one, which represents a more interactive type of website, including some business logic and returning the dynamic response based on the data.

The growth of server-side logic and its complexity, influenced the idea of splitting single server application into numerous Web Services, responsible for particular chunk of application logic and communicating with other clients or servers. Service-Oriented Architecture encouraged developers to split larger applications and enable them to cooperate using standardized communication protocols. Introduction of Representational State Transfer (REST) [FT02] refined the client-server communication to be stateless, focusing on caching to improve performance and communication via uniform interfaces using HTTP protocol.

Another idea, which enabled developers to deal with the complexity of web application, was to differentiate between specific layers in the web application architectures. Early web applications distinguished presentation layer, which communicated with the business logic layer, that in turn utilised data layer underneath for persisting the application state. Other modern frameworks used Model-View-Controller (MVC) approach, in which the visual part was represented by the View, Controller based on the request was responsible for the interaction with Model, containing necessary data and the business logic.

The next factor which led to enhancing the interactivity in the browser, was the emergence of AJAX (Asynchronous JavaScript and XML), allowing to load data dynamically and asynchronously without reloading the whole page. Growth of JavaScript popularity enabled developers to build more interactive and dynamic applications, detached from the initial markup, called Single-Page Applications. Additionally, more business logic started to be implemented on the client side of web applications, resulting in so-called thicker clients.

Alongside the technology evolution, the requirements put in front of the web applications also changed, most often pushing them to the limits to provide better user experience, while working on more demanding computations and processing larger amount of data than before. To meet such requirements, new patterns and architectures emerged, allowing to achieve the desired capabilities and satisfactory performance, but most commonly resulting with tradeoffs in different areas.

The client applications became detached, self-sufficient and more interactive, taking most frequently the form of Single Page Applications and providing user experience similar to the mobile and desktop applications. Nevertheless, to address the performance problems, other architectures for the client applications have emerged, and different communication techniques enabling efficient communication and data exchange have arisen.

To ensure the scalability and performance of the server application in face of the business logic complexity, the systems have begun to be broken down into smaller, independent components that communicate with each other, leveraging the microservice and event-driven architectures. To ensure reliable communication within such a decoupled and distributed system, message brokers have started to be used more frequently.

Along with the increase in the amount of data being processed and stored, the new approaches and models of database management systems have been introduced, to meet the expectations of high performance, scalability and resilience. However, to fulfil aforementioned requirements, other constraints needed to be weakened, introducing another area of tradeoffs in the data management systems.

3.2 Defining web application

Taking into account the various types of web applications and numerous purposes they may serve, it can be difficult to clearly present the core ideas of web application. Researchers tried to define the term web application to describe the wide range of web applications. One of the definitions can be found in ”Web Application Architecture” published by Leon Shklar and Richard Rosen [SR03].

> A Web application is a client-server application that (generally) uses the Web browser as its client. Browsers send requests to servers, and the servers generate responses and return them to the browsers. They differ from older client-server applications because they make use of a common client program, namely the Web browser

However, the perspective of web application has been changing over the years, along with requirements they face. Web servers providing mostly static content changed into more complicated applications, that consist of multiple layers, where each of them is responsible for the processing of a selected fragment of client request. For more complicated use-cases it was not enough and the server application have been split and took the form of a distributed system, that consist of multiple services communicating with each other to process the application logic. Alongside with the development of server application, the database management systems have been evolving, presenting new paradigms to meet the requirements of the web applications using them. Lastly, the presentation layer of web applications also changed rapidly, resulting in dynamic and interactive client-side applications including parts of business logic.

Niels Abildgaard also noticed the progress of technologies and architectures used in web application development and proposed another definition of web application based on his research [Abi18].

> A web application is a client-server application with any number of clients and servers, in which client-server communication happens via HTTP, in which both client and server may be execution environments, and in which the server may be any arbitrarily complex system in itself.

Taking into account both definitions, we can refer to web applications used currently as mainly two communicating with each other parts — client (preferably used by web browser) and server (which can be any complex system by itself).

3.3 Requirements

Having described the development of web applications over the years, it is relevant to consider the requirements placed on the web application, that lead to such a development process. Currently, web applications can operate in numerous areas, and based on the business needs, target various customer groups and other non-functional requirements — the desired capabilities they need to fulfill may differ. For example, a small e-commerce shop, serving a handful of customers a day within one country will have different requirements than a large social network, in which millions of people participate worldwide on a daily basis. Nevertheless, several generic requirements applicable to the majority of web applications can be distinguished, but the importance of them, and the extent to which they should be incorporated, may differ depending on the application use-case and its scale.

3.3.1 Performance and scalability

The performance of web applications can be considered from two separate perspectives — user and system point of view [Kle17]. In the first case, it can be referred to the user's interactions with the web application and the time that it needs to respond to requests made by the user. To define how the application should behave, terms such as service level objectives (SLOs) and service level agreements (SLAs) can be introduced, to characterise the expected performance and availability of web application (for example if the service median response is served within 200ms and for the 99th percentile is under 1s). Nonetheless, for most of the time the user is not using the application alone. The interactions of numerous users need to be taken into account by various components of the underlying application that need to handle them efficiently. When the load increases (in terms of the traffic, data volume or complexity of actions), the system should scale accordingly to reasonably handle the workload without performance degradation.

Scaling can be referred to as the ability to cope with increased load which can take various forms, for example number of requests per second, read to write ratio for database or number of simultaneously active users. To achieve this, two different approaches can be used: scaling vertically (using more powerful machines) and scaling horizontally (distributing the load across multiple smaller machines). Some of the systems can automatically scale by allocating new computing resources based on the metrics related with traffic or performance, which is a desirable feature especially when the load is unpredictable. By building applications servers in a stateless manner, the horizontal scaling can be easily used to meet the demand. Unfortunately, stateful components can be harder to scale. Some storage components are designed and implemented to handle scaling gracefully, for other components techniques such as data fragmentation or replication have been introduced to meet the desired performance requirements.

3.3.2 Reliability

Reliability refers to the fact that the system should continue to work correctly (perform correct processing as user expected at desired level of performance), even when the failure occurs [Kle17]. The designed system should be resilient and fault tolerant, preventing the web application from stopping providing the service to users. The aforementioned service level agreement (SLAs) metric can also refer to the uptime of the services, stating for example that not less than 99.9% of availability time within a month is acceptable.

The origin of failure may be different, but some of the major factors can be hardware faults, software bugs and human errors introduced while developing and managing the application. To prevent them, various techniques can be introduced. Hardware faults in form of hard disk crash, power grid blackout or network issues can be masked by introducing redundant components, that serve as failover until the broken ones are replaced or fixed. Software errors in the form of introduced bugs or other human errors related with configuring services can result with unexpected and undesired behaviour of application or even other components of the system. That area of errors is harder to mitigate, but the application can be rolled back to the previous, properly working version, the faulty feature can be disabled if the system is using the feature flag mechanisms or the degradation of the components can be limited by using patterns like circuit breaker. Moreover, proper manual and automation testing can prevent from introducing errors, while monitoring and analysing service behaviour in production will raise an alert if some deviation is noticed. The reliability is expected not only from larger web applications, but also from smaller ones. Downtime and software bugs in the web application can contribute to loss in revenue and damage the reputation.

3.3.3 Security and compliance

These days web applications became one of the most popular platforms for exchanging information, ranging their utilisation from numerous social media platforms to more critical areas, such as bank accounting or accessing sensitive and confidential information, stored on some servers in a digital form. The security breach of some web applications could result in compromising large amounts of sensitive data, leading to severe legal and economical consequences. Along with the growth of web application ecosystem and their complexity, the attack surface enlarges and the risk of introducing some security vulnerabilities is increasing.

It is substantial to ensure a proper level of security based on guidelines and frameworks, defined by organisations and authorities working in the field of security, hardening the web application and mitigating the possible attacks to ensure compliance with security standards. Web application design should ensure that users identity and sensitive information are properly secured, giving access to resources only when the user identity is verified along with expecting adequate permissions [XwX11].

3.3.4 Maintainability

Although maintainability may not be the obvious requirement for web application, it is an important factor for business executives, developers and people working in the operations. Majority of the costs from a product development perspective is not the initial version of application, but it is related to the maintenance, which takes form of keeping the system operational, fixing bugs and resolving failures [Kle17].

To maintain the software efficiently, the operational aspects such as monitoring, patching, deployment and maintenance of a stable environment should be easy and mostly automated to make sure the system is running smoothly. From an engineering perspective the system should be simple, by removing unnecessary complexity connected with tight coupled modules, tangled dependencies and inconsistency in terminology or architectural patterns. By reducing the complexity, the maintainability of the application will be easier. It will also increase agility and make the evolution of the system easier, which will result in reducing time of introducing new features and reacting to architectural changes, required to make the system operate efficiently.

3.4 Modern web application architecture

There are many potential perspectives, considering the architecture of modern web applications, when taking into account the areas they operate on and specified requirements that they need to fulfill. Below, the most common patterns and techniques used in modern web application architectures are discussed that enable the systems to perform efficiently and meet the desired requirement and specification.

The chapter is divided into three sections taking into consideration the server application, underlying data layer, and the web application clients, with regards to the concepts presented in the Figure 3.1. However, some of the discussed patterns goes beyond such classification, especially when they are referring to the communication between layers' boundaries.

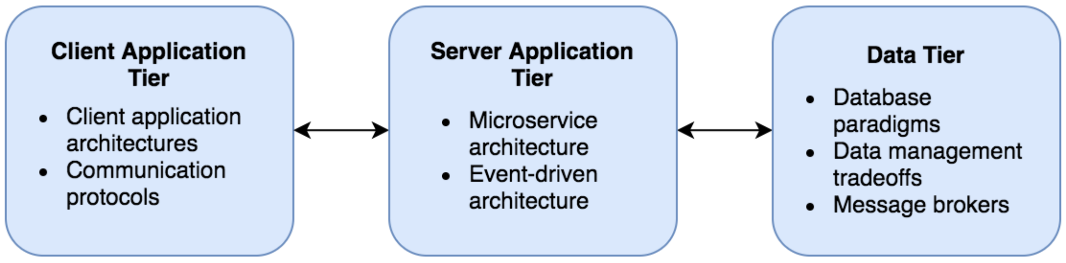
Client Application Tier
• Client application architectures
• Communication protocols
Server Application Tier
• Microservice architecture
• Event-driven architecture
Data Tier
• Database paradigms
• Data management tradeoffs
• Message brokers

Figure 3.1: Considered topics regarding the web application architecture

3.4.1 Server Application Tier

3.4.1.1 Microservice architecture

Microservice architecture is one of the trends in the modern web application development which is gaining more interest, especially when thinking about various benefits and success stories of the biggest companies using it to run their services in the web. To consider microservices it is useful to compare it with the monolithic architecture, which refers to building server-side applications as a single logical executable unit responsible for handling the HTTP requests, execute the domain logic, retrieve and update data in the database and send back the adequate response. Such architecture can be effectively applied when building simple applications, but when the system grows and evolves over time it is harder to preserve good modularisation and keep changes within the modules they should belong to, expanding unnecessary complexity into other parts of the system. Altogether with the system growth, application may require to be scaled accordingly to the load. This can be achieved by running multiple instances of the server application behind load balancer, but it requires to scale the whole application, rather than particular components that require more resources. Another downside of the larger applications built as a monolith is the fact that every change requires rebuilding and deploying the whole application, including the parts of the system not affected by changes.

Microservice architecture [FL14] is described as a solution to these problems. Microservices can be referred as an approach to developing applications as a set of small services, that each of them runs on its own and communicate via message passing, using standard data formats and protocols via well-defined interfaces. Each of the services is built around specific business capabilities, providing module boundaries within a domain it operates in — mitigating the problem of good modularisation. Moreover, each of the components can be deployed independently, reflecting the notion of agile development and encouraging to utilise continuous deployment. Scaling can be also applied in a more granular manner, affecting only the services that require more resources to perform effectively under the load.

3.4.1.1.1 Decoupling system components

Separating larger systems into independent services can help with ensuring firm module boundaries within the domain of the system. It makes it easier to maintain the components overtime and introduce required changes independently from other components. Alongside with the system decomposition, various teams can take care of selected services, focusing on them in greater details. Moreover, greater granularity enables to focus only on the requirements specified for the particular component, applying required changes to meet them, rebuilding services to be more performant or splitting into separate components when the evolution of managed context will require that.

Due to services separation, each of them can be written using different languages, frameworks and libraries that suits best for the performed task. The underlying data storage can be selected independently as well, because each of the components can manage its own database. However, most frequently the set of used technologies is limited to the predefined group for easier maintenance. Such opportunities encourage to experiment and try out new tools that can be applied through a gradual migration when they introduce desired capabilities [Fow15].

3.4.1.1.2 Microservices deployment

Another consequence of the service separation is the possibility of independent deployments. More teams and products embrace the idea of continuous integration and continuous deployment, using automation heavily to test software and verify the quality of it. Another often accompanying process is continuous delivery, enabling teams to release a new version of the product to the production environment more frequently, even many times a day. Various modern deployment techniques make it possible to introduce new versions of the software with no service downtime. Such an idea is appealing from the product management perspective, reducing cycle-time between ideas and introducing new features into the product, responding quickly to the market changes. In case of microservices the idea of infrastructure and operational automation is essential to efficiently handle the independent service management and ensure good product quality [Fow15].

3.4.1.1.3 Scaling microservices

Microservices are typically packed and deployed using containers to any platform supporting containerisation. It enables the operation teams to effortlessly relocate and replicate services across heterogeneous platforms. Each of the system components can be easily replicated, locating multiple services on the same host and dynamically scaling them according to the load, independently from other services which may not experience an increase in traffic. Due to that fact, the system can remain performant by allocating additional hosts when it is needed and deprovisioning resources when they are becoming redundant. Such a feature makes microservices a perfect technology to be used in the cloud.

The replication of services and spreading them across different hosts ensures fair distribution of the load and increases the availability of the components in case of the hardware faults, making the system robust and fault tolerant. Due to the fact that microservices create effectively a more complex distributed system compared to monolithic architecture, it is a desired feature to make the components designed for failure. Errors introduced into developed software or hardware faults are inevitable, in case of the unavailability of other suppliers the service needs to respond as gracefully as possible. Due to that, microservices value service contracts, introducing patterns like tolerant reader and other consumer driven contracts, enabling to evolve services independently, while on the other hand introducing patterns like circuit breaker to make sure that failure of one of the components will not affect the whole system. To prevent the system degradation in case of failures, the application should be tested when some of the services will be unavailable. Proper monitoring setup should detect the system failures quickly and automatically restore the services if possible. Moreover, it can be utilised to measure various business relevant metrics, providing warning and triggering alarms when some inconsistencies in the system behaviour will be noticed [DLL+17].

3.4.1.1.4 Downsides of microservice architecture

Aside from introducing additional complexity, microservices are also not free from other downsides. They can be a good solution when handling large enterprise applications, but for smaller products they can introduce unnecessary overhead. To ensure the acceptable performance in such a distributed system, asynchronous communication is used most frequently. Moreover, the decentralised data management allows each of the services to persist its state independently from others, which makes the transactional operations more complex and costly. However, from the business perspective it is important to ensure high availability of the system. Some of the business processes tolerate temporal inconsistencies, in which transactionless coordination between services or aforementioned asynchronous messaging can be applied, ensuring the eventual consistency. It enables the system to work efficiently, but requires it to be designed to tolerate inconsistency windows between updates propagation.

The frequent and independent deployments are a great advantage of microservice architecture, but handling the rapidly changing set of components is a great operational challenge. In the case of complex solutions, automation is essential to support administration of the system. From the development perspective, working on the smaller components of the system may seem to be easier, but the real complexity might not be eliminated, but rather split across multiple interconnected services. Designing and working with such a system, requires a greater skill and can benefit from using more specified tooling. In that case, embracing DevOps culture can help with greater and tighter collaboration between developers and the operation team working together on the application [Fow15].

3.4.1.2 Event-driven architecture

When designing the system it is a common practice to model the fragment of reality as a set of entities and relations between them, then map them directly to the data model. Commonly, there is an application layer utilising a set of service modules, containing and performing the application logic, which update the models depending on the result of the processing. The model capabilities are frequently limited to basic create, read, update and delete operations. This approach can be applied successfully in many simpler applications, although when the complexity of business logic grows, it may become more difficult to efficiently manage and develop it [Fow03a].

To deal with the business logic complexity, Eric Evans suggested an established set of practices and processes supporting modularization of large systems based on business context, called Domain Driven Design [Eva04]. He presents the approach of thinking about the model of a complex system as a set of many smaller models, which are representations of separate business contexts. Each component in the system exists within its bounded context which represents autonomous business domains and its model is used only within that scope, reflecting actions suitable for a given domain. It enables developers to better understand the underlying business logic domain and reason about the business processes and system behaviour, instead of modeling it as a state of particular objects.

3.4.1.2.1 Event Sourcing

Event Sourcing is another approach influencing the way application state can be perceived. Instead of modeling the application logic in a structural way, as a set of logical entities that are mapped and stored in the physical tables in the database, the events leading to the current application state can be stored. When the event occurs within the system, it is persisted and appropriate change to the application state is made, based on the event data and application logic responsible for interpreting it. The structural model saves only the current state of the system, while the Event Sourcing enables auditability in form of the sequence of events leading to particular state. The application at any time can be recreated by applying a sequence of events in an order in which they occurred.

Most frequently, to react to some events taking place within the system, some specialized component provides a mediating mechanism in form of message bus or message broker, enabling events to be transmitted. The latter one allows for indirect communication between numerous components subscribing for certain event occurrences and publishing them to all interested components when the event occurs within the system [NMMA16].

3.4.1.2.2 Command Query Responsibility Segregation

Command Query Responsibility Segregation (CQRS) is a design pattern frequently used together with Event Sourcing. It distinguished separation into two action types — commands (responsible for modifying the application state) and queries (capable of reading it). Each of the groups uses different models for updates and queries, enabling optimisations that increase performance of such a solution. The write model can take the form of events, while the read model can be formed as a more complex domain model, which can be even denormalized to some extent to minimise the number of operations required to get the data and perform queries faster. For each of the types of the operations, a different database solution can be selected, which satisfies the requirements of data model and enables independent scaling.

Despite the advantages, Event Sourcing and CQRS comes with a cost. They increase the implementation complexity of a given solution and due to different components responsible for storing data for each of the data models, the application state may not be strongly consistent [NMMA16].

3.4.2 Data Tier

3.4.2.1 Database models

Over the years, traditional relational database managements systems [Mei19] established their position and proved to be a mature and stable solution for storing and querying the structured data. Nevertheless, with the growth of Internet and data volumes transferred over the network, for example by social media or streamed from various sources, the relational model has shown some limitations related with scaling to handle such a large amounts of data efficiently and at the same time maintain high-availability and fault tolerance. NoSQL [Fow03b] has emerged as a response to the needs of scalable database management systems, handling increasing data volumes and offering higher availability, but at the same time sacrificing consistency guarantees or querying capabilities as a trade-off. Various NoSQL database management systems have some common characteristics, including among the others, less restrictive and more dynamic data model, support for running within the clusters and providing greater availability and performance metrics, which are desired capabilities for the services running in the Internet nowadays. As with many tools, NoSQL databases are not free from downsides and other undesired properties in regards to some classes of problems. Relational databases are still playing a significant role and provide a specific combination of desired properties for other types of applications and they will be used alongside NoSQL solutions, depending on the problems and requirements that the system needs to fulfill [Fow12].

The overview of the most common approaches regarding database management systems is presented below. It is divided into categories, presenting the different database paradigms, along with examples of such offerings, their application and most common use cases.

3.4.2.1.1 Relational databases

Relational model was proposed by Edgar Codd and refers to organising data into relations (tables), which consists of unordered collection of tuples (rows). It is efficient for storing and querying the data with regular and predefined structure, containing the relations between records from various tables based on their identifiers.

The data can be updated or queried using SQL, which is a declarative language, enabling to specify the pattern of the data and leave the execution to the database engine. Most of the traditional, relational databases support strong consistency, transactional processing and are ACID compliant, which makes them suitable for problems of transaction processing (banking transactions, sales) and batch processing (payroll, invoices). Besides that, the relational model is quite flexible and can be generalized for broad range of problems, which is justified by the fact that many web application use that model nowadays.

With the system evolvability and increase in its complexity, the various business domains can be harder to model using the relational databases. When the data schema is normalized to avoid the duplication and other undesired features, the queries that include joining multiple tables may require some time to be executed. Another concern refers to the fact that most applications are built using the object-oriented programming languages, which require to translate the data between the relational and object-oriented model. Numerous object-relational mapping (ORM) libraries tend to reduce the amount of overhead and boilerplate code required to translate the layers [Kle17].

The most popular databases utilising relational model include among the others PostgreSQL, MySQL and Microsoft SQL Server.

3.4.2.1.2 Key-value databases

Key-value storage enable to store a set of key-value pairs, and it is similar to data structures like hash map, dictionaries or objects available in some programming languages. The data can be stored or retrieved by passing the unique key, which points to some arbitrary value preserved in the storage. The storage does not assume any structure for the values itself, which can be an arbitrary value or more complex structure, leaving the interpretation of data to the application logic layer. It exposes most frequently a limited interface allowing to create, read, update and delete data, without possibilities to perform range queries or other more complicated operations [GWFR17].

Most frequently, such simple key-value storage persist the data in memory, which enables them to perform operations with low latency and high throughput. Such databases are used most frequently for caching purposes, making it acceptable to lose the data when the machine is restarted. However, some of the offerings provide mechanisms allowing them to preserve the data by utilising special hardware (battery-powered RAM), writing log of changes or making periodic snapshots preserved on the disk or replicate the in-memory state to other machines. Key-value storage owe their low latency to reducing the overhead related with encoding in-memory data structures that need to be written to the disk. Additionally, some of such databases provide additional data structures such as priority queues or sets, that are harder to implement leveraging disk based indexes. Redis [Red21] and Memcached [Dor21] can be distinguished as representatives of that category [Kle17].

3.4.2.1.3 Document-oriented databases

Document databases enable to store documents that are containers for key-value pairs, which form semi-structured documents that are most frequently stored in JSON format. The documents are grouped together in collections, that form hierarchical structure, and can be indexed for faster queries. The documents' data schema is implicit and it is interpreted when the documents are read, which grants greater flexibility and possibility to store arbitrary keys and values in the documents. On the other hand, it does not guarantee any document structure and attribute possession.

Instead of normalising the data and encoding the relations, the document model prefers to embed the data into single document, which is a tradeoff, granting faster reads due to data locality, but on the other hand it can introduce data duplication. It makes the writing and updating documents more complex, which may be not suitable for the relational data to form connected structures. Nevertheless, discussed approach enables developers to query the documents by matching semi-structured data, selecting parts of them or performing aggregation tasks on the underlying data. The disadvantage of the model is the fact that the whole document need to be retrieved when selecting only part of it and the entire documents need to be rewritten on updates [Kle17].

The document model can be more convenient from the development perspective, reducing the impedance mismatch by working on the JSON format, which is more similar to the objects compared to the data from relational model. The representatives of the document model databases are MongoDB [Mon21] and CouchDB [Fou21b].

3.4.2.1.4 Wide-column databases

Wide-column databases enable to store items as rows described by multiple column families. Most frequently, they are technically implemented as distributed, multi-level sorted map. The first level called row keys, identify rows that themselves consist of key-value pairs. The second level, called column keys, refers to arbitrary set of columns without defined schema, which are partitioned into column families, collocating the data that is used together on the disks. Some of the databases introduce third level, which consists of the timestamps.

Such a design enables to efficiently compress the data, collocate the information used together for faster reads as well as effectively partition and distribute the data across nodes within the cluster based on the row key value to provide easier scaling, replication across multiple nodes and decentralisation. The model provides limited data querying possibilities, most frequently allowing to constraint the records only on the row keys and does not support joins between tables [GWFR17].

The wide-column database design is sufficient for operations requiring frequent writes, but infrequent reads and updates, such as recording time series and storing IoT device measurements. Cassandra [Fou21a] and Apache HBase [Fou21c] are some of the most popular options that belong to this category.

3.4.2.1.5 Graph databases

When the data is characterized by many relations between modeled entities and connections are becoming more and more complex, it can be beneficial to start modeling the data as a graph built from vertices (modeled entities) and edges (relationships between the entities) that contains additional information about the relations.

Graph databases are designed to store and efficiently query the data persisted in such graph representation. Along with many algorithms capable of various computations on the graph structure, it can be suitable for modeling social network problems, web graphs, communication related networks or building knowledge graph. It can be also beneficial to use graph-like structure in problems related with knowledge discovery, because the underlying design allows to easily add new connections when the data model evolves and changes frequently [Kle17].

Along the graph representations, databases provide also a declarative query languages to create and traverse the information preserved in graph representations, which frees developers from specifying the execution details and make the optimizer choose the most efficient strategy to retrieve the data. Cypher is one of such graph query languages and it is created for Neo4j [Neo21] graph database.

3.4.2.1.6 Search engines

Search engines enable the possibility to perform a full-text search of the documents. It is similar to the document-oriented databases, but underneath the search engine analyzes the documents' content and creates special indexable search terms, allowing further to perform queries and find documents based on their content. Additionally, different algorithms can be used to rank and filter the documents, perform linguistic analysis, support users by suggesting other queries or handle the typos. The document indexing is quite complex and expensive operation and it can bring much overhead when the search engine is meant to run in scale [Kle17].

The most popular databases in that category, such as Elasticsearch [Ela21] and Apache Solr [Fou21d] are build on top of the Apache Lucene project.

3.4.2.2 Data management tradeoffs

3.4.2.2.1 Strong consistency vs. high availability

As described in chapter 3.4.2.1 the offering of available database management systems is diverse. Before selecting one of the options, it is essential to understand the requirements that the database needs to fulfill, based on the business problem definition and guarantees of the suitable data model for it.

Significant factors implying how the database stores and manipulates the data are the level of consistency and database transaction model. Most of the traditional, relational database management systems are compliant with the ACID semantics (atomicity, consistency, isolation, durability) that ensures strong consistency and serializability of the operations. Some of the business problems, such as financial transaction processing or some analytical processing in data warehouses, requires from the system to perform many simultaneous transactions while maintaining strong consistency. Such a model does not tolerate any invalid states and it is even preferable for that system to be unavailable, rather than relaxing the ACID constraints [Abi18].

Some of the problems does not require such strong consistency requirements and can tolerate small divergences in favour of the greater availability of the system. Most of the NoSQL databases providing a more flexible data model are designed to reflect the properties of BASE semantics (basically available, soft state, eventual consistency), which is less strict and more usable for some cases. Examples of such systems include social media and their feeds, e-commerce platforms or booking services, which tolerate small inconsistencies like stale data, leaving the product in the basket after deletion or allowing for overbooking, in favour of always responding to the user requests [GWFR17].

3.4.2.2.2 CAP theorem

Along with the tradeoff between availability and consistency it is important to consider the case of hardware failures, when the machine on which the database is running on fails or the network communication between nodes within a cluster is interrupted.

The CAP theorem [Bre10] states that at most 2 of the 3 properties from consistency, availability and partition tolerance can be achievable, denoting the upper bound on what is possible in the distributed database management system. The underlying database system can be consistent (read and writes are consistent and all clients have the same view on the data) and available (all correct nodes accept requests and return meaningful responses). Nevertheless, in case of network partition or faulty communication channels losing messages, the system needs to decide upon consistency by delaying to respond or availability by accepting the requests that can violate the consistency of data [Abi18].

However, the CAP theorem highlights the situation of the network partition, which is not as common and does not tell anything about the distributed database system when the hardware behaves correctly. The constraint can be further extended to the PACELC [Aba12], conveying the same idea in case of hardware failures, but when the system is operating properly the tradeoff can be made between the consistency of operations and the response latency [GWFR17].

3.4.2.3 Message brokers

The message brokers are frequently used to facilitate the construction of decentralized topologies, by decoupling the separate stages of the system and enable them to communicate with each other. The main goal of the message brokers is to provide a messaging middleware, which is capable of storing, validating, routing and delivering the messages to appropriate destinations, without knowing where the receivers are placed or if they are currently available. The exchanged messages are translated between formal messaging protocols in a platform agnostic way, which enables services written using different languages and technologies to communicate effectively.

To provide a greater reliability and guarantee message delivery, the message brokers rely on a group of message queues, storing the messages until they are consumed by the receivers. Such a type of inter-application communication allows to prevent the loss of valuable data and enables the system to continue running efficiently even with the connectivity or latency issues. Depending on the message broker implementation, different capabilities can be ensured, such as message ordering, exactly once delivery semantic or persisting the messages in the memory to prevent from the data loss in case of failures. Moreover, various message distribution patterns can be utilised. The most common include point-to-point communication, in which only one recipient receives the message and consumes it once, or publish-subscribe model, where producers can publish a message to a certain topic and it will be distributed to multiple consumers subscribing to it.

The message brokers are frequently utilised in the systems built based on the microservice architecture, that can benefit from greater flexibility and scalability due to intermediate communication middleware. Each component of the system can be independently scaled, dynamically increasing the number of publishers and consumers of messages as well as redeployed and restarted without affecting other services, improving resiliency and fault tolerance of the system. Message brokers find applications in various systems, whenever reliable, inter-process communication and message delivery is required, such as financial transaction and online payment processing or e-commerce order fulfillment services [Edu21].

3.4.3 Client Application Tier

3.4.3.1 Client application architectures

3.4.3.1.1 Single Page Applications

The idea of the Single Page Application [MP13] refers to the web application clients running within the web browsers, rendering the web page content dynamically, based on the user interactions and the data received from the asynchronous communication with the server. The content of web applications can be highly interactive and contain significant part of the application business logic, thanks to the JavaScript code listening to the user-triggered events and the responses from the server. Contrary to the static pages, the user interactions does not require to reload the view of the application, which enhances the user experience and makes it similar to the mobile and desktop clients [Abi18].

The client-side rendering is performed in the browser using JavaScript containing all the application logic, data fetching mechanism, the templating and routing handled entirely on the client side. On the first request, the entire application code is loaded and executed, making the website more responsive, which alleviates the need of requesting additional assets, besides the data required to be rendered as the content. Nevertheless, it comes with the significant and additional cost on the initial load, impacting the time when the application is ready to be used. Moreover, relying on the dynamic content introduces more problems, including among the others difficulties with parsing the page content by search engines. To address this problem, various techniques can be used, such as aggressive code-splitting along with the lazy-loading of the JavaScript code that is required for a given part of the application or caching the Application Shell, which is the core application logic, enabling interaction with the user [Dev21].

Despite the disadvantages, Single Page Applications are extensively used for the highly interactive web applications with the frequently changing, dynamic content.

3.4.3.1.2 Static site generators

For the web pages including mostly static content, which does not change frequently, using the static site generators can be sufficient. Every time when the data changes, the generated content is rebuilt entirely or partially, to release an updated version.

The statically generated websites can be enhanced with the JavaScript code, dynamically loading parts of the content to make the initial page structure available faster, while enabling the dynamic parts when the page is fully loaded. Moreover, it prevents from the large initial load overhead known from the single page applications, by prerendering the static pages ahead of time and making it available based on the request [Dev21].

Such approach gained additional interest and popularity recently, forming the idea of JAMstack, which refers to usage of JavaScript, reusable APIs and content preserved in the Markup files to render the websites. It highlights the performance benefits related with the prerendering of the static content, which can be further enhanced utilising Content Delivery Network (CDN) to serve the assets closer to the users with lower latency. The maintainability and ease of the deployment of such a client architecture are lower due to reduced overhead limited only to serving the static content. Prerendering the sites is also beneficial for the search engine optimisation [Net21].

The usage of static site generators may not be feasible, when the set of URLs to be rendered ahead of time is large or it is not known initially.

3.4.3.1.3 Server-side rendering

The idea of server-side rendering refers to rendering the web pages on the server side, leveraging the reliable connection and returning the initial content to the client. After the browser renders the page structure, the JavaScript code can be loaded, attaching the event listeners to the HTML elements during the rehydration process. Running the initial logic and rendering the webpage on the server side addresses the problem of the large initial load and reduces the amount of JavaScript code sent to the client, improving the performance and user experience.

Most of the modern UI frameworks are capable of executing in both browser and server environments by providing an interface for generating the initial HTML, which later can be rehydrated inside the browser. Moreover, the server-side rendering can be utilised only for the subset of pages, especially the computation-heavy, leveraging the benefits of this approach. Nevertheless, the dynamic nature of rendering pages on the server side brings additional overhead and resource utilisation to ensure it is executed in an efficient way [Dev21].

3.4.3.2 Communication protocols

With the development of the technology driving the web application clients, they started incorporating more business logic and became more sophisticated, which along with the asynchronous communication with the server application, made them more similar to the mobile and desktop clients. In order to make this possible, the server applications expose a well defined interface, establishing the communication contract between both parts, using the HTTP protocol. Web clients running inside the web browsers, along with the mobile and desktop clients can use the API to make requests to the server application, in order to retrieve the data or perform some action. Moreover, one server can itself be a client to another service, which is a common approach in aforementioned microservice architecture, when the whole system is decomposed into numerous, separate services communicating with each other.

Over the years the web application architecture has evolved, along with the communication protocols operating over the network, most frequently in a client-server manner.

Despite its various forms, one of the main task of the server application is to expose the application-specific API, enabling its clients to utilise a predefined set of inputs and outputs determined by the application logic to obtain the data and restrict the actions that can be performed [Kle17].

Different styles of building the web application APIs address different problem spaces and expose various tradeoffs. For example by granting more flexibility and customisability, the data caching can be more difficult to implement effectively. The overview of the most commonly used patterns and protocols for client-server communication in the web application field is described below.

3.4.3.2.1 REST

The Representational State Transfer (REST) is an architectural style with design principles heavily based on the HTTP protocol, including simple data formats, usage of URL addresses for identifying resources and other HTTP features for authentication, content type negotiation and cache control [FT02]. It does not define the entire communication process, but rather a set of constraints over the interaction between components and interpretation of the data elements to minimize the communication latency and maximize the scalability of the system.

Scalability and visibility of the system is achieved by stateless communication between the client and server, which alleviates the need to store the session data on the server, that simplifies the communication and reduces the amount of used resources. On the other hand, it imposes a restriction on all the requests to be self-sufficient by containing all the necessary data to process them. Moreover, the data coming from requests and responses can be marked as cacheable, which in addition to the system composed from multiple layers can provide efficient data caching, leading to the performance improvements.

To ensure simplicity and evolvability, REST introduces an idea of resources that are abstraction over the information stored in the application, their representations and operations that can be performed on them. The idea of uniform interface in the client-server communication is achieved by preserving four constraints: the resources are identified based on the URI, they are manipulated through their representation, operations on resources needs to be descriptive based on resource address (URI), action (HTTP method) and the metadata (included in HTTP headers). The last restriction refers to the hypermedia, which contains the relations between linked resources and possible interactions between them.

Thanks to the mentioned advantages and its simplicity, REST gained great popularity and it is commonly used as a primary technology for building public APIs nowadays [Eiz17].

3.4.3.2.2 WebSocket

WebSocket [FM11] is a protocol providing two-way asynchronous communication between the client and the server. It is developed as part of the HTML5 standard and natively supported in all major web browsers, with the WebSocket API available in most of the commonly used programming languages. The protocol addresses the need for efficient, full-duplex communication in web sessions, as an alternative to long-polling techniques and establishing multiple HTTP connections used for that purpose before.

WebSocket provide similar functionalities as regular TCP, but the protocol utilises HTTP as transport layer, which makes the communication backward compatible with underlying web infrastructure and inherits all its benefits, such as origin based security model as well as proxy and firewall traversal. It provides minimal framing to make the protocol frames cacheable in the communication intermediaries and support distinction between text and binary data, storing the message metadata in the application layer. Moreover, the protocol adds addressing and protocol naming mechanism, that allows it to support multiple host names on a single IP address and multiple devices on one port.

WebSocket meets the requirements of the modern and highly interactive web applications, enabling the asynchronous, bidirectional client-server communication. Examples of the protocol application requiring web-based and real-time communication include chats, games and multimedia systems [SHS14].

3.4.3.2.3 GraphQL

GraphQL [Fou21e] is a declarative query language for fetching data from APIs and a runtime to fulfill the requests with the data. It was developed internally in Facebook in 2012 to address the performance problems during Facebook's shift to native mobile clients. The specification was open-sourced in 2015.

GraphQL presents a different approach compared to REST and other web service architectures, in which clients can define the structure of the data which is returned from the server. The API schema gives a complete description of the data supported by the API, describing its structure, underlying types and relations between entities, forming a central communication point to the clients. Clients can use queries to retrieve selected fragments of the data exposed by API, by defining the entities, their relations and specifying the fields that need to be obtained. To write or modify the data, mutations can be used to change the preserved data and are allowed to cause side effects. Additionally, clients can use subscriptions, which are most commonly implemented using WebSocket, to listen for the data changes and receive them in real-time.

GraphQL enables to perform multiple queries and operations within a single request and retrieve specified properties of the resources as well as resolve the references for nested entities. Most frequently, the GraphQL API is used in conjunction with HTTP and exposed as a single endpoint, accepting the POST requests and transporting the operation payload in the HTTP body. By defining the multiple queries in a request as well as specifying only the data required, GraphQL gives great flexibility and prevents from the data overfetching, by reducing the number of requests and amount of data being sent. Nevertheless, the greater flexibility introduces additional overhead in terms of resolving the queries on the server side and makes the effective data caching significantly more difficult to implement. GraphQL is most frequently used as API for web and mobile applications to optimise the amount of data being sent to the client side [Eiz17].

Chapter 4

Serverless architecture for web applications

The growth of cloud computing influenced the way how the applications are developed and managed nowadays by outsourcing the resource management to the cloud providers as well as scaling the services within minutes by allocating new machines with proportional billing. The serverless paradigm can be perceived as a next step in the cloud computing progression, introducing significant mind shift from thinking about servers to building the applications from loosely coupled services, configured to work together to perform the processing.

There is no clear definition of serverless as described in section 2.2, however many sources describe the features of the serverless architecture, including among the others no need to manage infrastructure, which is handed off to the cloud platform, responsible for provisioning resources, executing and scaling the components to meet the demand with granular billing according to resource usage. Moreover, the architecture of the solution consists of multiple components configured to work together, running entirely in the cloud environment and introducing a fine grained development and deployment model. The idea of Function as a Service is used to execute code in an event-driven manner, inside an ephemeral and stateless execution environment, within strictly limited time. Alongside the serverless functions, the Backend as a Service components are heavily used to provide various functionalities or replace parts of existing application logic by integrating with their API, as discussed in section 2.3.

Although the serverless is considered as not fully matured technology, it is already adopted by various companies and organisations that noticed numerous advantages in using the technology, as covered in section 2.4. The development and operational cost reduction, along the scaling capabilities with billing proportional to resource usage as well as gaining agility and reducing time to market, are mentioned as desirable features of the serverless paradigm. Nevertheless, the serverless technology is not free from downsides. The stateless and anonymous nature of FaaS allows the platform to scale, but requires communication with external components to preserve the state and exchange messages as well as initiates discussions about performance. The outsourcing of the infrastructure, handed off to the cloud providers, reduces the amount of work, but on the other hand it integrates tightly with the platform, relying on it entirely in terms of reliability and security. The serverless architecture requires to make a mind shift accordingly, when developing and modeling the solution along with testing and monitoring its behaviour in the cloud environment.

Many cloud providers noticed the interest in the serverless technology and introduced numerous services in their portfolio, as described in section 2.5, enabling developers to build the serverless applications using their platform. It led to developing numerous production grade implementations, covering a vast range of use cases and serverless platform capabilities, as briefly presented in section 2.6.

The second of the discussed fields, describes the evolution of web applications over the years, introducing various architectural and development patterns to meet the requirements put on them. The idea of web application is difficult to clearly present due to numerous purposes they may serve and various architectures they implement, as mentioned in 3.2. However, the overall view can be summarised and cover any client-server application using the HTTP protocol, with the client running in the web browser environment, while the server can be any complex system by itself.

Despite the diversity in the perspective of the web applications, some of the defined requirements can be shared by the majority of them, with applicability extent based on the use case, as noticed in section 3.3. It is desired for the web application to scale according to the workload demands, without noticeable performance degradation, preserving availability and characterising with resiliency and fault-tolerance, because failures are inevitable in more complex systems. Moreover, with the increase in the data volumes processed, along with the growth of their importance and confidentiality, the system should remain safe and compliant with various regulations. Last but not least, the maintainability of the web application is important, covering the monitoring, observability and deployment processes from the operational point of view, along with the ease of development by removing complexity, inconsistencies and preserving good modularity to ensure a satisfying agility level when developing new features.

To fulfill the specified requirements, various approaches and architectural patterns have been introduced, as covered briefly in section 3.4. With the server applications becoming more complex, they started to be split into several smaller and decoupled services, communicating with each other, leveraging microservice architecture and using event-driven approaches to model the problems and process them more effectively. It introduced the need for various message intermediaries, such as message brokers and message buses, along with the new models of the database management systems to meet the requirements of the large volumes of data being stored and characterise with performance, availability and reliability, leading to further trade-offs in other areas. Simultaneously, the client application architectures evolved, providing more complex and interactive clients, communicating with the server application using the benefits of various communication protocols to fulfill the requirements and exchange information effectively.

4.1 Research

The aim of the thesis is to analyse the applicability of the serverless processing and the FaaS model in the field of web application development. Chapter 2 introduces thoroughly the concept of serverless computing, while chapter 3 describes briefly the topic of web applications and its architectures used nowadays.

4.1.1 Research questions

To understand how the serverless architecture and Function as a Service model can be applied in the web application development, the following research questions are defined and are a subject of further research.

1. Is serverless paradigm a suitable technology for building web applications?
2. How the serverless computing and FaaS model can be applied to process the web application workloads?
3. What are the characteristics of the storage components used in the web applications built in the serverless paradigm?
4. How the serverless architecture affects the communication with web application clients?

4.1.2 Research approach

To answer the defined questions, further analysis of the literature, reference architectures and articles provided by various practitioners is conducted. Based on the gathered information, the example implementations are provided to reason about different capabilities of the serverless architecture and its applicability in the field of web application development. Detailed description of the examples and the proposed solutions can be found in section 4.3.

The research focus mainly on the services provided by the Amazon Web Services to narrow the scope of the thesis and provide more in depth analysis of the Amazon's cloud platform capabilities in terms of the serverless computing. The solution has been selected due to high quality of services and development-friendly tooling provided by the cloud vendor along with numerous undergoing innovations in the field of the serverless computing, setting new standards in terms of the platform possibilities. Moreover, the detailed documentation along with reference architectures and broad community of practitioners, provide a sufficient area to investigate the topic of serverless architecture. Nevertheless, the portfolio of other cloud providers, mentioned in section 2.5, present similar services with comparable capabilities, which should meet the characteristics of the offerings provided by AWS in the near future to maintain the market competitiveness.

4.2 Serverless suitability for web applications

The serverless paradigm is becoming more popular and appealing to numerous companies due to benefits it can bring, such as reducing operational and development cost, shortening time to market as well as outsourcing some of the operational and management overhead. Numerous startups decide to adopt serverless architecture to build their products faster, while larger and more traditional enterprises want to become more agile and reduce their operational costs. Despite the fact that serverless is used extensively by some of the organisations, the applicability of the serverless paradigm is not fully comprehend and it is not entirely clear what problems are suitable to be implemented using this architecture. Cloud vendors provide guides and describe use cases of serverless applicability, but neglect to present when the serverless architecture is not suitable and has not been used effectively [Gru19].

The serverless architecture is a fairly new approach with various benefits, but also disadvantages and limitations, which are described in more detail in chapter 2.4. In order to consider the suitability of serverless paradigm, some approach would be to create an intuition about the architecture characteristics, by leveraging the benefits it brings and understanding the limitations it has. However, some of the restrictions can be overcome by providing workarounds or redesigning the software to fully utilise the capabilities of the architecture.

The section below covers the topic in more detail, summarising the knowledge about benefits and challenges along with the deeper look into use cases and examples, to answer the research question considering the serverless suitability for web based services.

4.2.1 Serverless suitability

4.2.1.1 Workload types

4.2.1.1.1 Utilisation patterns

The significant benefit of the serverless paradigm is the autoscaling capability, which enables platform to track the load with greater fidelity, scaling up quickly in case of demand increase and scaling down shortly after in the absence of it. Moreover, all of the processing is billed proportionally to the services actual usage in terms of execution time, number of processed requests or volume of transported data [JSS$^+$19].

Mentioned characteristics make the serverless architecture applicable for dynamic and irregular workloads and when the services experience little traffic, which can benefit from the automatic and rapid scaling capabilities and granular pricing model [Gru19]. It greatly mitigates the problem of underprovisioning and overprovisioning by the automatic, granular scaling handled entirely by the cloud platform [Rob18]. Nevertheless, moderate and consistent load, which does not utilise the advantage of the serverless architecture, may be more suitable to be executed on the dedicated virtual machines or containerised environment due to lower overall cost of the solution, without the need to scale based on the execution behaviour [Bol19].

When selecting the architecture it is essential to consider the cost of resources and operational work, required to handle the expected load of the service. The serverless architecture can be more cost efficient when handling unpredictable or occasional load due to its autoscaling capabilities. It can be considered as a good and generic rule when choosing architecture, but it is advised to perform more thorough cost estimation, taking into account the architecture utilisation under the expected load. However, the effective cost estimation for the serverless solution may not be trivial with different pricing models for various components and amount of data being transported, especially when handling peaking traffic or managing more complex systems [Gru19]. On the other hand, the reduction of overhead related to the operational management due to the infrastructure outsourcing can be also appealing and become a trade-off being made, from the business point of view.

4.2.1.1.2 Processing times

One of the biggest flaws pointed out to the serverless architecture is the "cold start" caused by the resource provisioning, leading to significant latency and unpredictable response time [JSS+19]. The phenomenon, caused by resource allocation and environment bootstrapping, is a subject of various studies, dedicated to understanding it better and mitigating its effects. Moreover, cloud vendors continuously work on the various solutions and virtualisation techniques to reduce the latency. Some of the vendors provide dedicated services such as AWS Provisioned Concurrency, there are also other programmatic solutions for prewarming the serverless functions, which mitigate the problem, but comes with additional cost [Rob18].

The most significant impact of the "cold start" on the response time is visible, when multiple functions are sequentially combined to perform some workflow. Along with the latency caused by the communication overhead, the compounded processing time may become significantly longer [Gru19]. To prevent these undesirable situations, it is advised to keep the user-faced function chains as short as possible to make the response time short as well.

Both factors mentioned above may indicate that serverless paradigm is not suitable for the highly-performant and mission critical workloads that need to guarantee certain response time requirements. It is more difficult to ensure the performance of the serverless solution, due to unpredictable performance of the underlying serverless platform, but depending on the characteristics of the developed application, the additional latency added to the response time may be acceptable [Bol19].

The serverless solution can be effectively utilised to perform some background tasks, which does not require to directly respond to the users. Moreover, due to scaling capabilities the workloads can be effortlessly parallelised, reducing the overall processing time.

4.2.1.2 Serverless processing limitations

4.2.1.2.1 Runtime and data restrictions

Another factors significantly impacting the serverless suitability are the runtime and data restrictions of the Function as a Service processing model. These were covered in more detail in chapter 2.4.2, when considering the challenges related to the nature of the serverless architecture. Regardless of the provider, serverless functions are designed to be effectively stateless, running code within ephemeral and isolated containers, constrained by the limited processing time. Configured amount of the memory, CPU units and network bandwidth is allocated from the shared resource pool, assigned to the container and deprovisioned shortly after processing finishes. Such architecture enables functions to scale horizontally, but on the other hand it requires to preserve state in external components, which can introduce significant overhead when large volumes of data are transported. Moreover, developers do not have control over the container execution and their addressability, which requires to use external components serving as intermediate messaging brokers [Rob18]. Lastly, various BaaS components can also have limitations, for example constraining the operation throughput or volume of the data that can be processed.

Mentioned limitations can make the serverless paradigm not suitable for some problems, which require fine-grained data sharing or significant amount of coordination and communication, due to overall messaging overhead or the cost of external intermediary services. Nevertheless, various practitioners and architects analyse numerous problems and provide ideas for limitation workarounds or redesign some of the solution architectures to fit the serverless paradigm or utilise it within hybrid solution [JSS$^+$19].

When it comes to various cloud providers, the runtime restrictions can differ and it is crucial to consider if the problem characteristics fit within the constraints of the vendor services [Bol19].

AWS Lambda [Ama21m] as one of the most developed FaaS offerings, allows to configure the memory in range of 128 MB to 10GB, with proportionally allocated CPU up to 6 vCPU cores for multithread processing [Ama21i]. The serverless function execution time is limited to 900 seconds, with invocation payload up to 6 MB for synchronous invocation and 256 KB for asynchronous. The temporal, local disk storage available for the container is limited to 512 MB. The deployment package of the function in the form of zip archive is restricted up to 50 MB when compressed. When uncompressed the limit is set to 250 MB, including up to 5 layers, which are a solution to share some common code across multiple functions [Gru19].

Lastly, major cloud providers introduced various solutions for running custom execution runtimes using container images, which may be a solution to overcome some of the mentioned limitations, but feasibility of such solution was not entirely researched yet [Ser21].

4.2.1.2.2 Operation types

When considering the serverless billing model with granular cost proportional to the used resources, it is crucial to utilise the available computing power as efficiently as possible. The CPU bound operations including the intensive calculations, such as image manipulation or some sophisticated mathematical computation, are more suitable for the serverless processing model and can fully utilise the underlying allocated resources.

On the other hand, the I/O bound operations, like loading large volumes of the data, inserting large sets of data to the database or heavy network communication, may be less suitable from the cost perspective, because the operations can be relatively slow, extending the function processing time. Moreover, when considering the communication with some external I/O components, each container invocation may require establishing new connections, introducing additional latency [Bol19].

Due to that fact, some of the dedicated services capable of I/O operations are redesigned for serverless architecture — the workflow can be transformed into a more event-driven approach. The processing using the third party service is initiated by one of the functions and when the action is performed, the incoming event can trigger another serverless function to process the response, which prevents the serverless function from idle waiting. Serverless functions can be configured to poll some of the BaaS storage components and be notified when a particular action is performed, such as data modification or file insertion [Gru19].

Another operation constraint is related to the implicit failover, provided out of the box by the serverless platform. When the function processing fails due to some error, it is retried up to several times, based on the underlying configuration. Due to the at least once delivery semantics, the operation performed by the function should be idempotent or additional mechanism needs to be implemented to prevent from subsequent processing of the event [Gru19].

4.2.1.3 Vendor dependence

As with any outsourcing technology, the serverless paradigm relies heavily on the dependence on the service provider, which was discussed in more detail in section 2.4.3. Giving up the control of a significant part of the application, on one hand enables to hand off the infrastructure management overhead, but on the other hand leads to vendor lock-in [Rob18].

API level lock-in, in which application code relies on the interface of the third party components, can be partially mitigated. Solutions such as extracting the business logic from the function handler, incorporating ports and adapters pattern to rely on abstract interfaces, rather than concrete service implementations or using some of the serverless frameworks, enables to abstract over the cloud providers interfaces to a certain degree.

The service level lock-in, in which the developed solution relies on particular feature or behaviour of the external service, can be harder to mitigate, due to the fact that there may be no alternative components with similar capabilities, blocking the migration and requiring to redesign some of the functionalities [Gru19].

Moreover, the tooling related to the deployment, monitoring, logging and configuration may require to be replaced when changing the cloud vendor platform [Rob18].

The cloud provider is responsible for maintaining, updating and protecting the underlying architecture and generally can ensure sufficient level of services, because of the knowledge, experience and qualified staff it has. However, incidents happen due to some unpredictable events or human errors, causing unexpected outages and connection disruption, even in the largest data centers [Gru19].

When deciding on the service architecture, it is essential to consider if such vendor dependence is bearable [Bol19]. The vendor lock-in in the serverless architecture can be seen as a trade-off rather than limitation, when considering whether the benefits one can obtain are sufficient to sacrifice the elasticity compared to other architectures.

4.2.2 Serverless applicability for web based services

4.2.2.1 Suitability for web based workloads

The idea of web application has been introduced and discussed in more detail in chapter 3. Modern web applications can take a variety of forms and implement numerous architectures to fulfill the requirements put in front of them. It is difficult to generalise and unambiguously answer the question whether the serverless architecture is suitable for the web applications, because the number of variables deciding on the applicability is significant.

It is considered that the serverless paradigm can be utilised successfully for building entire systems or incorporate isolated components, implementing granular or larger tasks, into the existing system [Sba17].

The examples provided in section 2.6 cover a significant amount of use cases, showing how the services were designed to utilise the serverless architecture and its benefits in the production grade services. The section below discusses in more detail the applicability of serverless architecture, covering the examples and their capabilities as well as providing additional information to claim the serverless suitability for described workloads.

4.2.2.1.1 MindMup

Mindmup, serving the role of online collaboration tool for mind mapping, described in section 2.6.2, is one of the early adopters of the serverless paradigm, which has undergone an architectural migration from the traditional architecture running on Heroku, experiencing significant operational cost reduction [AC17]. The redesign of the exporters' services enabled developers to reduce the boilerplate code responsible for managing, logging and monitoring of the processing as well as focusing only on the converter's logic running in the serverless functions. Due to the fact that some of the file formats were used less frequently, all the processing logic was initially bundled into single service for cost optimisation. The negative aspect of such coupling has been observed by exhausting all of the available resources, along with bugs propagation impacting several exporters. The re-architecturing of the flow to event-driven approach, with the logic orchestration pushed to the client as well as the further migration of the application backend, contributed to the cost reduction by 50% even with the increase in number of active users — resulting in the overall cost savings estimated to 66%. The MindMup example of exporters processing benefits from utilising the serverless architecture for the high-throughput task, utilising fine-grained serverless function scaling.

4.2.2.1.2 Yubl

Another example of the early adopter of the serverless technology was the London based, social networking company named Yubl [AC17]. Initially, their solution was implemented as a monolithic system, serving as a backend for a mobile application. The significant problem of the platform was the spiky social media traffic, exceeding up to 70 times the normal usage pattern, especially when the advertisers or influencers posted some news. The autoscaling feature of the AWS Elastic Compute Cloud has been enabled by engineers, but the time required to provision new instances was too long to handle the instant increase in demand. After lowering the threshold responsible for triggering the scaling, the rapid traffic growth could be partially handled, but in the long run it led to notable resource overprovisioning.

The migration to serverless architecture was based on the cost reduction, possibility of more frequent deployments and minimising the operational overhead related to the maintenance of the system. Despite the fact that Yubl has been closed, because of not securing another round of funding, the engineers migrated significant part of the application backend, using around 170 different serverless functions in production, with the cost savings estimated to 95%. Moreover, breaking the monolithic system into separate functions reduced the number of conflicts when integrating new features and improved the agility, enabling developers to introduce new features quicker. Working on smaller units of the system allowed to an increase in the number of production deployments from 4-6 to about 80 on a monthly basis.

4.2.2.1.3 CloudGuru

The CloudGuru educational platform is an example of a fully-fledged web application, serving the role of a scalable application backend, including numerous workloads with different characteristics, as described in more detail in section 2.6.1.

The simpler tasks include API calls, triggering the AWS Lambda to execute the code responsible for business logic, communicating with external BaaS services to perform some action or preserve the state in the database. The video processing task is an example of a more complex workflow, initiated after uploading the video directly to S3 bucket, preceded by obtaining the presigned URL granting secure access. The instructor, who uploaded the video can be further notified, when the asynchronous action is completed, following with the transcoded video being uploaded to the S3 bucket, along with the update of its status in the database which makes the video available for students. The real-time communication capabilities of Firebase are a convenient solution, enabling users to exchange messages, while the background jobs are further managed to index the forum entries for search purpose.

What is interesting, is the fact that the described architecture includes services from multiple providers — Netlify as a host for the static web application, Auth0 as identity provider responsible for authenticating and authorizing users and Firebase as a main database, directly accessed from the client. Nevertheless, all of the largest cloud providers, covered in more detail in section 2.5, include in their portfolio similar services, allowing developers to build the complete, production ready application using resources from a single platform.

The initial architecture of the CloudGuru platform can be described as a serverless, RESTful monolith based on the API Gateway and AWS Lambda for processing and Firebase as a primary database. The example describes the real, sophisticated and production-grade system, supporting thousands of users on a daily basis and scaling to the demand, while keeping low operational cost (on a monthly basis the API Gateway and AWS Lambda cost was estimated below 1000$).

Despite the fact that the system was performing well, the architecture has undergone a significant transformation, as described in the second edition of the "Serverless Architecture on AWS" book [SCN21], initially written by Peter Sbarski [Sba17]. The CloudGuru case study describes an interesting example of re-architecturing the large, serverless-based web application. As the major problems mentioned as a migration reason were tight coupling between different parts of the system as well as shallow isolation of the domains' boundaries. Making some changes to a single, primary Firebase database most frequently affected multiple functions and was leading to conflicts and overlaps during development.

The company decided to split the monolithic, serverless architecture into multiple parts, resembling the microservices approach based on different parts of the product, in which each service ensured proper boundaries and owned its data for particular business context. The GraphQL has been used to prevent from overfetching and multiple round trips when accessing the data by the clients. The Backend for Frontend pattern enabled engineers to expose different endpoints for web and mobile clients, running custom GraphQL servers in AWS Lambda and combining the responses from multiple microservices. Each serverless microservice has been split into stateless components (API Gateway, AWS Lambda) and stateful components (DynamoDB, S3) deployed independently. Common services, such as AWS Virtual Private Cloud, AWS Web Application Firewall and Amazon Redshift serving as data warehouse, have been extracted and loosely coupled with specific microservices.

The desired outcome of the migration has been achieved — teams are working more independently without affecting others, owning responsibility for their microservices and at the same time noticing performance improvements. The serverless architecture enabled developers to migrate the platform gradually, implementing new features while keeping the old one serving the production traffic. The developers could focus on the migration process without worrying about provisioning resources and their cost, by adjusting the services configuration described in the Infrastructure as a Code approach and letting the platform provide new resources [Sba17].

4.2.2.1.4 Temenos

The banking system developed by Temenos, effectively utilises the event-driven architecture, implementing to some degree the Event Sourcing approach along the Command Query Responsibility Segregation, described in section 3.4.1.2. The first part of the processing pipeline is responsible for the Online Transactions Processing (OLTP), accepting the transactions data and serving the role of command operations. It consists of the API Gateway, AWS Elastic Load Balancer, AWS Fargate which is running containerised application, including the business logic and saving the transactions in the database, hosted by AWS Relational Database Service. Later, the AWS RDS is streaming the data to Amazon Kinesis, which aggregates the records and triggers the AWS Lambda to process them, storing the results in the DynamoDB in the form of a read-optimized model, which is a subject of performing query operations.

The serverless solution developed by Themenos enabled to decouple the processing, providing proper scalability level, effectively incorporating the serverless paradigm for high-throughput data ingestion tasks performed in the background, as part of the larger hybrid solution.

4.2.2.2 Comparison with traditional architectures

The common use-case for web application servers is to provide an API used by the clients running in the web browsers. There are several research [IRD19] [FJG20] analysing the performance, scalability, reliability and cost of the API built using serverless, microservices and monolithic architectures.

Mentioned papers compare simple API services, performing basic operations using the AWS Lambda, communicating with the database or processing some CPU-bound computation. They simulate user behaviour with different traffic characteristics and workload volumes within a specified period of time, measuring the response time of the implementations, using different architectures with comparable resource configuration. The results of research are categorized and concluded below.

4.2.2.2.1 Serverless is more agile in terms of scalability and with more stable performance characteristics

During the initial phase of load testing, it was noticeable that the serverless implementation suffered from the ”cold start” impacting the response time. After the period of 2 minutes, the performance has stabilized and the response time remained almost constant until the tests completion. The implementation utilising serverless technology characterise with instant scaling, matching the increase in workloads. The higher number of requests or unexpected spikes are handled automatically, with small impact on the response time and slight performance degradation observed [IRD19].

Which is in contrast to the solution implemented using the microservice architecture, which starts to scale after reaching preconfigured utilisation criteria. Moreover, for some types of the traffic increase, the delay of responsiveness to re-balance the current workloads was greater, which lead to notable increase in the response time [FJG20].

4.2.2.2.2 Microservices suffer from load balancing and traffic distribution problems

The solution implemented in the microservice and monolithic architecture was capable of maintaining the performance level for the stable and smaller loads, but characterized with longer response time under more rapid traffic growth. After the autoscaling utilisation threshold is met, more instances are added into the cluster, improving the response time, which with further scaling could match the performance of serverless implementation [IRD19].

Moreover, the high peak response time was observed for the microservices implementation, scattered randomly during tests, which could be potentially caused by the traffic distribution within the cluster, when the new instances were added or deprovisioned. Due to that, the serverless solution, despite the initial ”cold start” period, notes more stable and favourable performance characteristics [FJG20].

4.2.2.2.3 Serverless performance is comparable with the microservices or monolithic architecture

Overall, the serverless implementation was characterised with comparable performance, leading in some of the tests over the microservices, especially when heavily affected by the load increase. However, the microservices handled the small size and repetitive tasks more efficiently, which could be a result of smaller processing overhead, related with the virtualisation stack and involving multiple components in the serverless implementation [FJG20].

Moreover, with the growth of the load the performance degradation of the monolithic implementation significantly impacted the response time, while the solution based on microservices degraded performance much slower. [IRD19]

4.2.2.2.4 Cost effectiveness depends on workload characteristics

The cost analysis conducted by one of the authors of the papers [IRD19] claims that the solution utilising the virtual machines is more cost efficient — resulting with the estimated cost to be 12 times lower without autoscaling and 6 times lower with autoscaling enabled, which still could not match the serverless performance for some tests. Nevertheless, the additional cost of the data transfer and other components, such as load balancer, storage, database are not included into the research. Some of the mentioned components are specific to the architecture based on the virtual machines, which could increase the cost, along with the additional labour related to the maintenance of the solution. On the other hand, when the traffic is stable and the virtual machines are configured to match the workload, the solution can be much cheaper and preserve the suitable performance characteristics. For one of the tests purposes, the serverless architecture used AWS Lambda configured for the 1.5 GB of memory allocation. Optimising the function configuration in terms of the performance to cost ratio, could reveal that other setting could yield similar performance with lower cost, because AWS Lambda price is established based on the memory size per execution time basis. The research concluded that there is no clear choice and one need to carefully measure the cost based on the estimated traffic characteristics and include the price of all components used in both implementations.

4.2.3 Fulfilment of the web application requirements

The discussion on the serverless suitability and further analysis of the example implementations, using the serverless architecture for a variety of web based workloads, can indicate that the serverless paradigm finds the applicability for web based workloads. Some of the thoroughly designed implementations can bring significant benefits in terms of the solution scalability, performance, cost reduction and development agility. The section covers in more detail how the serverless paradigm is applicable for the web application requirements, presented in the chapter 3.3.

4.2.3.1 Performance and scalability

The stateless and short-lived design of the serverless function makes it convenient for horizontal scaling, however the performance from the point of view of the single request is still debatable, considering the ”cold start” phenomenon, as described in sections 2.4.3.3 and 4.2.1.1.2. Moreover, the ephemeral nature of the processing and lack of addressability requires to use external components to preserve the state and communicate with other services, as noticed in section 2.4.2. The time required to allocate the resources and bootstrap the runtime of the function, along with the communication overhead between the components that make up the workload processing, can significantly impact the overall processing time. However, the architects and developers with the thoroughly designed application architecture can partially alleviate the undesired latency to make it acceptable for the developed web application. Moreover, the cloud providers tend to introduce further improvements to mitigate the unpredictability of the serverless platform and provide dedicated functionalities to make the execution and response time foreseeable.

On the other hand, when thinking about the performance of the system, the users are not using the application alone. Most commonly, the solution is utilised by several users at the same time, who perform various actions in the application. The research discussed in section 4.2.2.2 covers the analysis of performance and scalability of the serverless based systems, compared with the more traditional architectures. It results with conclusions that the performance of the serverless implementations is comparable with more traditional architectures, with more stable characteristics and lower degradation ratio, especially when the application experiences increase in the traffic and needs to scale accordingly to meet the workload demand as well as maintain the satisfactory response time.

The serverless components, based on their definition described in section 2.2, are characterized by fine-grained scaling capabilities, enabling instant horizontal scaling to meet the demand. The scaling is handled automatically by the cloud platform in response to the events coming to the system, affecting only the components required to process them, as mentioned in sections 2.4.1.2 and 4.2.1.1.1. The serverless architecture is designed to track the load and respond to it with greater fidelity, compared to scaling of the solution based on containers or virtual machines, as discussed in section 4.2.2.2. From the application examples covered in section 4.2.2.1, the Yubl social media platform leveraged the scaling capabilities to meet their spiky traffic, significantly reducing the cost when migrating their product to the serverless architecture.

4.2.3.2 Reliability

Considering the serverless as the outsourcing technology, the significant amount of the overhead and responsibility, required to manage the system and ensure it is running properly, is handed off to the cloud provider, as described in section 4.2.1.3. Having the knowledge, experience and qualified staff, the cloud provider is able to ensure the reliability level, exceeding the services clients could guarantee by themselves. Nevertheless, as mentioned in section 4.2.1.3, unpredictable events or human errors happen and the disruption and outages are inevitable. To prevent the service downtimes, the serverless solution can be deployed and replicated into multiple Availability Zones managed by the cloud provider. Each of the datacenters is independently supplied with the electricity and network access, which gives a possibility to redirect the traffic between them when some disruption happens.

The section 2.2 describing the serverless components characteristics, mentions that the services are designed with implicit high availability. Multiple serverless components provide various retry mechanics in case of errors occurring during the processing, related to issues with the application logic or hardware faults. For example, by default AWS Lambda invoked asynchronously retries the event processing 2 times upon processing failure. If the processing is not completed successfully after retries, based on the configuration the event can be placed in Dead Letter Queue (DLQ), which is a dedicated Amazon SQS instance for handling failed messages.

Along with the programmatic solutions, establishing processes including proper monitoring, observability and triggering alarms in case of the undesired components behaviour could be beneficial, mitigating further administrator intervention or even restoring the system automatically to the appropriate state. For the production running application, the solutions similar to discussed in the section 3.4.1.1 considering the microservices architecture as well as all of the efforts required to ensure proper monitoring and observability described in more detail in section 2.4.4.3 could be applied. The automated tests running as part of the deployment pipeline, on the environment mirroring the one available in production, along with the load tests can indicate various discrepancies.

The serverless solution reliability can further benefit from the higher level deployment techniques and instant rollback capability, in case of observing issues after the release. Lastly, the serverless function can utilise the services providing dynamic configuration, enabling them to alter their behaviour by toggling certain features or implementing a solution similar to the circuit breaker pattern, preventing from the whole system degradation in case of bugs introduced into the implementation.

4.2.3.3 Security and compliance

As described in section 2.4.3.5, the serverless paradigm opens a lot of security related questions, having the field not fully researched yet. The cloud vendors provide numerous guidelines and recommendations regarding their services, aiming to maintain appropriate security level for applications running in the cloud and their configuration. One of the factors that cloud providers need to take into account is the multitenancy aspect, mentioned in section 2.4.3.4, that requires preserving appropriate level of isolation between tenants' workloads to guarantee compliance. AWS introduces a shared responsibility model, in which the security of the underlying infrastructure including the hardware, default software and the network configuration is maintained and protected by the cloud vendor, while the client is responsible for ensuring the compliance of the application and its configuration.

Moreover, numerous services and their configuration help developers with ensuring security for serverless applications. As an example, the API Gateway can be configured to perform authentication and authorisation for requests, provide data protection and input validation, while the AWS Web Application Firewall can provide DDoS protection and rate limiting to restrict the undesired and dangerous behaviour [Gru19].

Nevertheless, the new attacks considering the serverless architecture are emerging and highlighting the security problems of the serverless applications. The greater granularity of the serverless function deployments, makes their dependencies updated less often, reducing the frequency of security patches of the third party libraries. The direct access to the BaaS components is restricted by default, but their improper configuration can lead to various incidents, exposing confidential information or leading to spreading a malware. Other types of the attacks include improper authentication configuration, insecure secrets storage, function event's data injection and inadequate logging and monitoring setup [Gru19].

Most of the problems with the security of the serverless solutions results from the misconfiguration, however the security awareness is developing along with guidelines on how to protect the serverless workloads and guarantee their compliance with various security regulations.

4.2.3.4 Maintainability

As described in the definition in section 2.2, the serverless paradigm transfers the need to maintain, provision and manage resources and execute applications to the cloud platform. It significantly reduces the operational costs of running the applications as well as the labour cost related with the operational aspects, as noticed in sections 2.4.1.1 and 2.4.1.3. The deployment process is also simplified, in which the developers need to provide the artifact with the proper configuration and the platform is responsible for executing the application.

Section 2.4.4.3 indicates that the granularity of the components and the distributed nature of the serverless processing requires to setup proper monitoring and ensure appropriate level of observability to make sure the application is working properly. Leveraging the tools capable of distributed tracing, along with alarming when some metrics are over the predefined limits, can give more insight into the application behaviour.

Moreover, leveraging automation allows to configure multiple checks and perform automation tests in the deployment pipelines to give confidence when deploying the serverless solution. Due to the ephemeral nature of the serverless processing, which is performed in the cloud, the end-to-end tests conducted against the mirrored testing environment running in the cloud are required to verify the correctness of the software. Along with the advanced operational tooling capabilities, the releases can be enhanced with the canary deployment or the A/B testing, driven by the appropriate monitoring and ensuring that the application works properly, as covered in section 2.4.4.2.

The developers can benefit from multiple BaaS components, introducing new opportunities and reducing the development time, as highlighted in section 2.4.1.4. On the other hand, the serverless paradigm is a fairly new approach of building services with the new ideas and patterns emerging and best practices created to further guide the development 2.4.4.1. The development process can be harder initially, because of the greater granularity of the functions compared to traditional architectures and a new approach of running and testing the applications in the cloud environment [Gru19]. Nevertheless, the ecosystem of tools evolved over the years along with the knowledge base of the cloud providers and practitioners, describing reference architectures and good patterns.

Lastly, the serverless architecture makes it easier to introduce changes and adjust the architectures to the current needs with very little effort. The example of CloudGuru migration, described in section 4.2.2.1.3, shows that serverless makes the evolvability of the application easier, along with encouraging to experiment and introduce new features in a convenient way.

4.3 Example implementations

The further research takes up two example web services implemented in the serverless architecture. The selected problems consider some of the existing web applications and tasks they perform, which are shortly introduced in the section below, along with the overview of the implemented serverless solutions. The provided examples are used to discuss some features of the serverless architecture along the implemented patterns to analyse in a greater detail the implication of the serverless paradigm in the developed solution.

4.3.1 Receipt processing

4.3.1.1 Problem description

First example considers the service similar to PanParagon[1], which is a mobile application that enables users to scan and collect fiscal receipts, which are further analysed to gather some statistics or can be used for the return or complaint of the products, or as a reminder of the return date or warranty periods. The application is capable of obtaining the information from the receipt, such as name of the shop, purchase date and amount of money. Based on the retrieved data, the receipt can be further categorised and included in the spending statistics, giving users more insight into their shopping history. Furthermore, the application provides integration with several external services, allowing users to show current offers and promotions or electronically insure their equipment.

The service can be simply reimplemented as the web application, which uses the same backend for the calculations, business logic and integration with external services. The more demanding task would be to implement the receipt processing flow. Taking into consideration the variety of services available in the cloud, some of the components could be incorporated in the application to perform the document and text analysis, or even help with the data classification using the natural languages processing techniques. Moreover, the cloud provider takes responsibility for hosting and scaling the application along with its more demanding tasks, alleviating the need of managing the services and developing them from the ground up.

4.3.1.2 Designed solution

Key points of the suggested architecture:

- **GraphQL as API** — Presented example shows the basic backend of the web application built using the serverless technology. AWS AppSync is used to expose the GraphQL API, which is configured to trigger the AWS Lambda or communicate directly with some external services, such as DynamoDB. Moreover, the integration with AWS Cognito introduces the authentication and authorization capabilities to the API. Each of the GraphQL operations, such as queries, mutations and subscriptions can be restricted, to ensure that only logged in users have access to their private data. Once the receipt processing is completed and the data is stored in the database, the user is notified in real-time about the results of the asynchronous

[1]https://panparagon.pl/

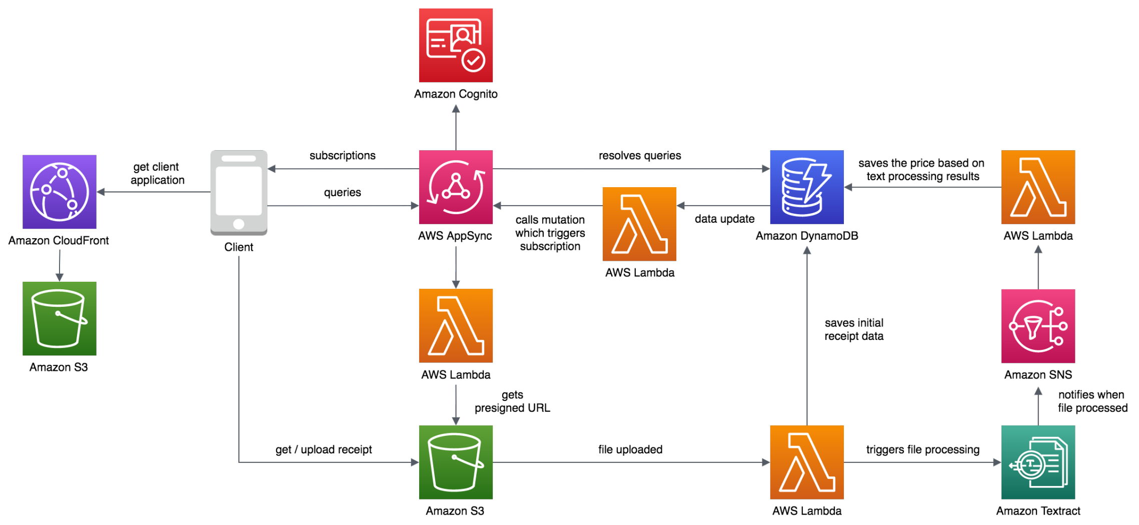

Figure 4.1: Architecture of receipt processing web application

operation, based on the established GraphQL subscription. The AWS Lambda, that is polling the DynamoDB for the event stream, including inserts and updates for a given table, triggers the GraphQL mutation, which in turn pushes the new data to the client.

- **Secure and direct access to services storing the files** — The GraphQL API, built on top of the AWS AppSync services, is not designed with intent to handle the operations on the files, including downloads and uploads. Instead, the mechanism of presigned URLs is incorporated. It enables the client application to directly communicate with the Amazon S3 in a secure way, to access or store the files with predefined names and within a limited and strictly defined period of time. It alleviates the need of using the AWS Lambda to process and insert the data into the bucket upon the user request, which could be more time consuming operation compared to generating the presigned URL.
- **Asynchronous receipt processing** — The receipt processing flow is performed in an asynchronous manner, similarly to many other processing flows, utilising the serverless paradigm. Uploading the receipt to Amazon S3 triggers AWS Lambda, which is responsible for inserting metadata about the image to the database and sending the receipt to Amazon Textract performing the document analysis task. Once the processing is completed, the Amazon SNS receives the event with the job identifier, which further triggers the AWS Lambda, subscribing to the particular topic. The serverless function retrieves the data and based on a simple heuristics, analyses the results of the text extraction process and stores the information further in DynamoDB.
- **Web application hosting** — The backend of the application, composed from the combination of configured with each other services mentioned earlier, is effectively hosted in the cloud. Similarly, the client application once built, it can be stored in a publicly available Amazon S3 bucket, in the form of the static assets. Amazon CloudFront distribution can be additionally integrated to provide the capabili-

ties of Content Delivery Network, delivering the client application to users in a more convenient way.

Along with the simple heuristics retrieving the information based on the text extraction process, the more sophisticated services could be incorporated in the receipt processing flow, for example to categorise the expenses, or provide more detailed spending statistics. Nevertheless, the other services would also operate in the same, asynchronous manner, extending the presented flow and performing the computation in a form of a distributed system, composed of several components hosted in the cloud.

4.3.1.3 Discussion

The proposed solution presents the fully fledged web application, which is built using the serverless architecture, including the serverless backend responsible for processing and storing the data along with the client application hosted in the cloud environment. The presented architecture is also a good illustration of different execution models of the AWS Lambda, covered in more detail in section 4.4.1.3, including synchronous execution to obtain the presigned URL, creating an asynchronous receipt processing flow as well as polling the stream of updates from the DynamoDB.

The asynchronous processing choreography, described in more detail in section 4.4.2.2, is a common pattern in the serverless architecture and refers to composition of multiple components, responsible for particular chunks of the processing and forming effectively a distributed system. The AWS Lambda is used to incorporate the business logic of the application, transforming the data when desired and using other services to orchestrate the flow or handle the processing, without waiting idle for the results, but rather being triggered in an event-driven manner.

The use of Amazon Textract brings new development opportunities, allowing developers to integrate the service in the processing flow with convenient billing per usage and eliminating the need to maintain similar self-hosted service.

AWS AppSync serves as a feature-rich gateway of the application backend, triggering Lambda functions or directly communicating with other services to retrieve the data, but also it allows to update the client in real-time using the subscriptions, which is a beneficial feature for the asynchronous processing. All of the processing is performed in a secure and compliant way, thanks to the authentication and authorization capabilities, introduced by the integrating AWS AppSync with the Amazon Cognito, along with the presigned URL mechanism, enabling application to safely store and retrieve the files from Amazon S3 bucket.

4.3.2 Generating interactive slideshow based on LaTeX files

4.3.2.1 Problem description

The second example covers an implementation of web application similar to the Pitagoras Korepetytor[2], which is a service playing a role of the virtual math tutor for high school students. It provides a large collection of exercises, which can be further adjusted by users

[2]http://pitagoras-korepetytor.pl/

when changing the values of the exercise parameters. Based on the user input, the service generates an interactive slideshow, which is further previewed by the user in the interactive client application. The slides present the following steps required to solve the exercise, with the additional audio commentary, explaining the performed mathematical operations.

4.3.2.2 Proposed processing flow

When inspecting the processing results of the original solution, the user is given a set of slides along with the audio files, which are later processed by the presentation player on the client side. Due to the large number of the parameters combinations, that can be specified by the user and their value domains, generating the slides and commentary for all possible input values can occupy a lot of memory and be unprofitable. Moreover, some of the presentations include advanced math equations, graphics visualising more sophisticated geometry problems or even considering some conditional logic based on parameters, upon which the solution for the exercise is generated.

Taking into consideration the problem and its requirements described above, the LaTeX [Pro21] typesetting language and its processor is one of the possible solutions for the problem, which is well-known and frequently used in the academia community. It enables the presentation creator to easily define the mathematical computations, present graphics showing the geometric tasks, which can be also extended based on the defined variables, including even some computations performed by the LaTeX processor or conditional logic.

Taking into account the selected tool, the processing flow for the user request could take the following:

1. The initial step, taking place before the processing of the file is started, can be used to validate the user input. The entered values can be compared with the constraints of the exercise, along with verifying if the user can preview the particular presentation, based on the application logic of the service.
2. Once the input is verified and the user is allowed to request the presentation, the exercise variables from the input can be injected into the LaTeX file. Next, the presentation is generated to the PDF file, which is further split into separate slides in the form of SVG files.
3. Along with the slideshow generation, the initial presentation file can include additional directives with the commentary, which are extracted and supplemented with the variables from the user input. The processed text can be transformed further into the human-like audio commentary, using the voice synthesis software.
4. Additionally, some metadata about the presentation can be retrieved, including the slide durations for the slideshow or the information when to play the audio commentary, which will be sent and used later by the client application.
5. The steps 2-4 can be performed in parallel. Once they are completed, the result can be afterwards sent to the client to preview the slideshow with the exercise to the user.

The described flow presents a rather extraordinary task incorporated in a web application, which plays a significant role in fulfilling the business logic of the proposed services. In the traditional architecture, the responsibility of performing the presentation processing could be extracted into a separate service, that triggers one of the worker based on the user requests. With the limited traffic, the approach with a separate service communicating with a set of workers would be suitable, but when a larger group of users would submit a request to process a request within a short period of time, the scalability of such a system should require to scale the number of available workers accordingly. The instant scalability of the serverless architecture, with a pay-as-you-go billing model, would be beneficial to meet the variable demand of the presentation processing requests. Moreover, the high-quality services hosted in the cloud, that are capable of performing the voice synthesis, could be integrated in the solution. However, processing the LaTeX files using the serverless function is a non-standard type of computation task, which requires the function runtime to be properly enhanced to fulfill the task.

4.3.2.3 Designed solution

The designed solution presented in the diagram, covers the presentation processing workflow and communication with the client application, responsible for displaying the slideshow based on the generated data. The practical research focused on configuring the serverless function, along with the processing optimisations of the described flow, is presented in more detail in section 4.4.3.

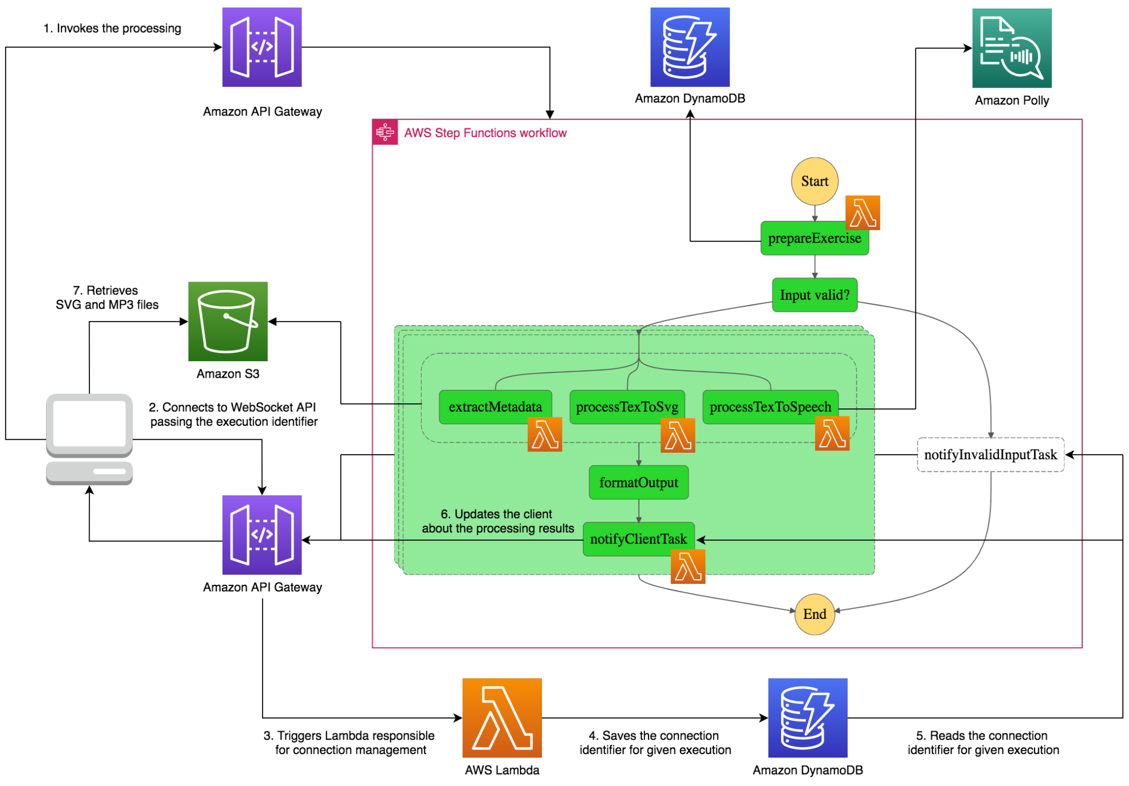
1. Invokes the processing
Amazon API Gateway
Amazon DynamoDB
Amazon Polly
AWS Step Functions workflow
Start
prepareExercise
Input valid?
7. Retrieves SVG and MP3 files
Amazon S3
extractMetadata
processTexToSvg
processTexToSpeech
2. Connects to WebSocket API passing the execution identifier
notifyInvalidInputTask
formatOutput
6. Updates the client about the processing results
notifyClientTask
Amazon API Gateway
End
3. Triggers Lambda responsible for connection management
AWS Lambda
4. Saves the connection identifier for given execution
Amazon DynamoDB
5. Reads the connection identifier for given execution

Figure 4.2: Architecture of presentation processing

Key points of the suggested architecture:

- **Function orchestration using the AWS Step Function** — The presentation processing flow is built on top of several AWS Lambda instances, with AWS Step Function orchestrating the entire process. AWS Lambda performs the initial task by retrieving the exercise details, validating the user input against the exercise constraints and defining the tasks for subsequent serverless function invocations, if the provided input satisfies the limitations. Based on the output of the first task, the further processing is executed or the user is notified about the invalid request. The presentation processing task is further separated into two chunks. The first one generates the slideshow for several initial slides, which is pushed to the client as soon as possible once the processing is completed and starts previewing it to the user. While the second part, including the remaining chunk of the presentation, is delivered to the client later, when the user is busy watching the initial portion of the slideshow. For each of the chunks, three AWS Lambda instances are executed concurrently. They are responsible for processing the LaTeX file and extracting the slides in form of the SVG files that are later uploaded to the Amazon S3, generating the voice commentary using the Amazon Polly, which are further stored in Amazon S3 as well as extracting some metadata about the generated slideshow. Once all three tasks for a given chunk are completed, the output of the functions is formated and another serverless function is invoked to push the results of the processing to the client using the WebSocket API
- **Pushing the update to the client** — AWS Step Function orchestrates the processing in an asynchronous manner, which makes it necessary to notify the client once the processing is finished. Instead of using the polling techniques to periodically check whether the presentation is generated, the push based approach is used. Initially, the client application invokes the AWS Step Function and gets its execution ID. Next, it connects to the WebSocket API and sends a message with the obtained identifier, invoking the AWS Lambda that stores the execution ID along with the associated connection ID in the DynamoDB. When the chunk of the presentation is generated, the serverless function retrieves the data based on the execution ID and sends the message with the processing results to the associated client application. Finally, the slides and audio files, stored in the Amazon S3 bucket under predefined path, are retrieved.

4.3.2.4 Discussion

The proposed solution describes an interesting and non-standard use case of the serverless processing pipeline, responsible for preparing slideshow with the audio commentary based on the LaTeX files. The predefined slideshows are requested by the user, making the desired outcome of the processing similar to the MindMup's exporters service, mentioned in section 4.2.2.1.1.

The serverless architecture is an appealing technology for the file processing tasks that are scheduled in an event-driven manner. The resource allocation based on demand, along the autoscaling capabilities with proportional billing and scaling down to zero in absence of requests, makes the serverless technology an applicable and cost efficient solution to use.

However, the processing flow for the described example is more sophisticated and consists of several steps, which require thoughtful coordination. During the research, It was re-architectured across several subsequent iterations to leverage the benefits of the serverless architecture and make the slideshow processing more efficient. The design process along with the performance comparison and cost calculation is presented in more detail in section 4.4.3.

The proposed solution relies on the orchestration of the processing, that consists of the serverless function invocations, performing various tasks on different steps of the processing. AWS Step Function enables the processing to preserve state between the functions execution along with orchestrating the flow, which can include some branching, tasks executed in parallel, mapping over a list of subtasks and gathering the results of the group of tasks. The solution brings quite some elasticity when defining the processing schema and provides visualisation of the workflow, giving better insight into the executed processing. However, the size of accumulated state in the AWS Step Function is strictly limited, which is insufficient to store the results of LaTeX files processing there, and requires the use of Amazon S3 to preserve the processed files from different stages of the computation to make them later available to the user.

Similarly to the previous example, the processing is performed in an asynchronous manner and requires some additional solution to update the client once the results are ready. The proposed solution invokes the AWS Step Function using the REST API and later connects via the WebSocket API to receive notification when the slideshow is available.

Another challenge with the proposed processing flow is related with the LaTeX files processing. The use of Lambda layer, including the additional dependencies, makes it possible to perform such custom tasks and enhance the Lambda runtime with additional capabilities.

Lastly, the use of the AWS Polly provided another development opportunity, enabling to easily enhance the slideshow with the audio commentary, without the need to host similar service or incorporate some other third party service, running outside of the cloud platform.

4.4 Server Tier

4.4.1 AWS Lambda as a compute resource

The idea of Function as a Service has been already introduced in section 2.3.2. An example of such FaaS offering is AWS Lambda [Ama20a], which is described as a compute service that requires no resource management. It provides the autoscaling capabilities, built-in high-availability and pay-per-use billing model, based on the function execution and the processing time with a granularity of milliseconds.

The following section analyses in more detail the FaaS capabilities, using AWS Lambda as one of the implementations of this model, taking into account its limitations, run-time possibilities, execution models and possible optimisations. Based on the overview of AWS Lambda functionalities, the intuition of FaaS model capabilities as an execution environment can be established.

4.4.1.1 Function limitations

The AWS Lambda execution is constrained by quotas referring to the compute and storage resource that can be used by the serverless function. Some of the AWS Lambda restrictions have been already discussed in section 4.2.1.2.1. The list below includes additional limitations along with explanation of their impact on the processing of the serverless workloads [Ama21m].

- **Concurrent executions** — 1000 executions — The serverless paradigm ensures about the infinite scalability of the underlying platform, however the default maximum number of functions that can be invoked simultaneously is limited. It is essential to keep that in mind when developing some serverless solution extensively using the Lambda, that there is some threshold above which the function execution will be throttled. The limit can be increased further by the cloud provider.
- **Function memory allocation** — from 128 MB to 10240 MB — The function memory allocation can be configured, which translates into the available CPU and network bandwidth. When the memory of the function is set above the 1769 MB threshold, it provides processors with multiple cores capable of multithreading processing. Therefore, further increase for single threaded computation will increase the memory or bandwidth, which will be suitable accordingly for memory-bound or I/O-bound processing [Mun18].
- **Function timeout** — 900 seconds (15 minutes) — The Lambda execution time is strictly limited, which makes it not suitable for processing various long-running tasks.
- **Deployment package size** — 50 MB (zipped) and 250 MB (unzipped, including layers) — The function deployment package, including the function code and its dependencies, is constrained as well. The function execution environment can be additionally extended by the Lambda layers, including additional code, libraries or dependencies. When the size constraints do not allow to pack all of the required dependencies, the function can be also based on the custom container image, limited

up to 10 GB. The topic of Lambda layer and container image is discussed further in section 4.4.1.2.

- **Invocation payload** — 6 MB (synchronous invocation), 256 KB (asynchronous invocation) — The invocation payload limit is related to the size of an event invoking the Lambda function. The synchronous invocation model requires the serverless function to return some value to the service triggering the function, for example the AWS API Gateway invoking the function based on user request, contrary to the asynchronous invocation, triggered for example by uploading some file to Amazon S3. The constraint prevents from passing large amounts of data in the event payload and requires the function to communicate with external components to retrieve or save the processing results if required.
- **Temporary directory storage** — 512 MB — Each of the invoked function environment is assigned a temporary storage, enabling the function to retrieve some data and perform the required processing in the local file system, which due to the ephemeral nature of the function containers, is further deprovisioned. The results of the processing need to be preserved in some dedicated storage component, depending on the type of the data being stored. Alternatively, under some circumstances, AWS Lambda can utilise the larger storage space when integrated with Amazon Elastic File System, which is a fully managed, shared file system service [Bes20].

Besides the appealing features mentioned initially, such as the Lambda scaling capabilities with proportional billing and the high-availability without the need for managing infrastructure, the further quotas describe how the Lambda runtime is limited. The constrained execution time shows the limitation of the serverless architecture, along with the limited invocation payload, which requires the function to communicate with external services from Lambda after the invocation. However, thanks to improvements in other areas, the Lambda can be configured to utilise greater amount of memory and multi-core processors, the deployment package can be extended when using the custom container image along with connecting the temporal disk space to the shared file system. Mentioned enhancement makes it possible to use the AWS Lambda in various, non-standard processing tasks, while leveraging its benefits.

4.4.1.2 Function runtime

AWS Lambda supports natively several programming languages and their runtime environments, including Node.js, Python, Java, C#, Go and Ruby, with some of them including different versions of the runtime [Ama21n]. Most frequently the function code and its dependencies are compressed and packed by the deployment tool into a single zip archive and further stored by the cloud provider. When the Lambda function is invoked, the serverless platform allocates the resources to form the function environment, downloads the function code, bootstrap the language runtime with function dependencies and execute the function. However, when the function has been recently triggered, the execution environment from a previous invocation can be reused, saving time required to prepare the environment and reusing some resources, such as established connections with external services or the temporary files.

Besides the standard functions' packing option and environment, AWS Lambda can use Lambda layers, providing a capability to share dependencies between several functions as well as custom container images. Both of the approaches are described in more detail below.

4.4.1.2.1 Lambda container image

AWS Lambda allows to deploy the serverless functions based on custom Docker or OCI images, which enable developers to build serverless functions arbitrary from the supported programming languages and required dependencies [Poc20]. Moreover, it allows to process other workloads depending on sizable dependencies, such as machine learning or other data intensive tasks. Container images are constrained up to 10 GB in size, contrary to the aforementioned limit of the zipped function artifact, which make container images suitable to include custom runtimes, binaries and libraries, that can be used by the serverless function. AWS provides a set of base images for supported runtimes as well as supported Linux base image, however the developers can use any container image if it supports Lambda Runtime API, which is a simple HTTP-based protocol, responsible for invoking the function and retrieving its results. The custom container images can be built using familiar tooling and later uploaded to Amazon Elastic Container Registry, from which the image will be retrieved once the function is invoked.

Compared to other services enabling to run container based workloads, the Lambda container image support allows to scale workloads on per request basis and scale to zero with no additional cost if there is no traffic, providing more granular pricing model than other services. Moreover, the Lambda function is integrated with many AWS services, which means that it can be invoked from numerous event sources. Nevertheless, the Lambda still needs to satisfy the aforementioned limitations, other than the artifact size [Shi20].

4.4.1.2.2 Lambda layer

When taking into consideration the encapsulated nature of the FaaS, several Lambda functions operating in the same part of the application logic can have some common methods, utilities and dependencies, effectively duplicating the same code in every deployed artifact. Lambda layer is effectively a zip archive including additional code, dependencies and libraries, that can be shared across multiple Lambda functions, reducing the overall size of individual function artifacts. When including the layer in a function, its content is extracted into function's local file system and can be further accessed by the function code, enabling functions to share common files and mitigating the problem of code duplication. Moreover, the Lambda layer can include other, operating-system specific libraries and binaries used by the function code as well as custom runtimes, enabling developers to add required dependencies and execute code developed in any programming language in Lambda functions. The function can include up to 5 layers, with the total size of the function code and layers summing up to 250 MB after decompressing [Ama21j].

The serverless function introduces an interesting approach in terms of developing and deploying the function code, which is further managed and executed by the cloud platform. Despite the fact that the set of predefined and supported languages along with provided libraries is quite broad, the AWS Lambda execution runtime is limited. Its extensions, including the Lambda functions based on the custom container images, allows developers to build the Lambda with known container technology, increasing the artifact limit to 10GB, extending the suitability of the serverless function to other, data-heavy types of workloads. Moreover, the use of Lambda layers makes it possible to share dependencies across multiple functions easier as well as introduce external libraries or even custom runtimes, thanks to the Lambda Runtime API. Both of the described approaches allow developers to adjust the Lambda runtime to perform numerous non-standard tasks.

The second example implementation, responsible for generating the interactive presentation based on the LaTeX files, covered in section 4.3.2, uses Lambda layers to enhance the serverless function environment with the LaTeX distribution. Moreover, the performance analysis of both approaches for the aforementioned application is covered in more detail in section 4.4.3.

4.4.1.3 Function execution

The AWS Lambda is executed in an event-driven manner. The serverless function can be invoked by each of the defined triggers, configured based on the various event sources, including numerous services from the AWS cloud platform and their operations. The generic categorisation of data sources and invocation types for the serverless processing is described in section 2.3.2.3. AWS Lambda execution model fits into the mentioned classification and distinguish the following types of the function invocation [Mun19b]. All of the mentioned execution models are present in the example implementation of the receipt processing application, mentioned in section 4.3.1.

- **synchronous (push based)** — Most frequently refers to the components acting as an API Gateway, invoking the Lambda based on the request and when the function completes the execution, it returns the results to the API Gateway, which passes the response back to the client. If Lambda fails during the event processing, the error is returned to the service calling the function. In case of the receipt processing application, the AWS AppSync invokes the Lambda functions in a synchronous manner, when user performs query to obtain the presigned URL.
- **asynchronous (event based)** — Covers all cases in which the service makes a request to the Lambda function, which is consuming an event, but once the execution is completed, it has no possibility to return the information to the event source. When the processing fails, the function is retried two more times by default. After that, the erroneous event can be placed in the Dead Letter Queue or it can be directed to other destination, if configured. The Lambda function, triggered after the receipt is uploaded to S3 bucket as well as the other one, invoked based on the notification from SNS topic, are the examples of the asynchronous function invocation.
- **stream (poll based)** — When assigning the stream based services as the data source, Lambda service runs a poller that is watching for the messages and once they are available, the stream of events forming a batch, can be passed to the function

via the asynchronous event. If the processing fails for some reason, the whole batch can be retried based on maximum record age or maximum retry attempts or the malformed data can be extracted from the rest of the shard and retried separately from the correctly processed events. When the retry limit is reached, the shard can be sent to the separate SQS queue or SNS topic. In the receipt processing application, the Lambda is polling the DynamoDB stream and based on received records, it is responsible for performing the mutation to push the update to the client.

4.4.1.4 Function optimisations

The Function as a Service model, which is heavily used in the serverless paradigm as a compute resource, at a first glance seems to be limited with the ability to configure the function and possibilities to optimise its execution, because the cloud provider takes responsibility for provisioning the resources and executing the function. However, the thorough design of the application architecture along with the proper composition of the function code and its configuration, regarding the event sources and underlying resources, has a significant impact on the performance of the executed function, which transfers directly to the cost of execution. The possible optimisations and configuration adjustment suggested by various practitioners for optimising the Lambda function execution and improving the configuration are discussed below.

4.4.1.4.1 Lambda execution

In terms of the Lambda function execution, the improvement possibilities can be divided between the cloud provider and the developers implementing the solution. Taking into consideration the function lifecycle, resource allocation along with the downloading function code is handled by the cloud provider, however the size of deployed artifact can have an impact on the cold start duration. Furthermore, the architecture of the function code can have an influence on initialising the function runtime along with required dependencies as well as executing the function code itself.

- **Concise function logic** — Focusing on developing functions with single purpose and well defined responsibility enables developers to create leaner functions with fever dependencies, which results with a smaller package once bundled. On the contrary to the ”monolithic” functions, that include some branching logic based on the invocation event to execute the particular part of the application logic, which take longer to initialise and can extend the cold start duration [Mun18].
- **Reduce the function dependencies** — Similarly to the previous point, the amount and size of dependencies transfers to the function initialisation time. Depending on the programming language, only selected modules or functions can be imported to incorporate only selected subcomponents of libraries or use techniques like tree-shaking to remove unused parts of imported dependencies. It allows to obtain smaller function artifacts, with fewer dependencies impacting the function start and execution time [Mun19b].
- **Use function to transform data, not to transport it** — Instead of using the

Lambda function to transport the data between services, some serverless components from the AWS offering can be integrated to transfer the data directly between each other. Similarly, when retrieving the data from external services to the serverless function, it is beneficial to request only the required data instead of performing additional filtering in the function code. It reduces the time, when the function is waiting for an I/O operation, for example when reading the data from DynamoDB or files from S3, that directly reduces the function execution cost [Mun18].

- **Leverage container reuse** — The concept of the serverless function assumes that every function execution is stateless and should not rely on existing state from the previous execution. However, due to optimisations of the function executions, when the subsequent event triggers the function in a short time after the previous execution is finished, the container environment can be reused along with the initialised runtime and its dependencies. The prehandler logic can be used to initialise the connection with external services, such as databases or obtain secrets required in the function handler logic, which can be performed once per runtime initialisation, instead of for every function execution. Similarly, the function global state can be used as a cache for rarely-changing data or lazy-loading variables and dependencies, reducing the work and processing time for subsequent invocations [Rad21].

4.4.1.4.2 Lambda configuration

- **Configure function resources intentionally** — As mentioned before, the amount of memory can be configured for each function, which transforms proportionally to the CPU and network bandwidth configuration. Depending on the type of workloads, the function performance can benefit from additional memory for computation, using multiple cores for multithreading processing, when the memory is configured above the 1.8 GB threshold as well as from additional bandwidth for I/O bound processing. In terms of the cost the Lambda function is billed per function invocation and execution time, proportionally to the amount of configured memory. It is essential to consider, that the function can process the event faster when having more memory, while having similar cost as the function with lower memory configuration that takes longer to finish the processing. To better understand the function execution behaviour depending on configuration and take a data-driven approach, the AWS Lambda Power Tuning can be used, which is an open-source tool invoking the AWS Step Function, executing the function with a set of predefined configurations, measuring the execution time and its cost [Mun18].
- **Select appropriate event sources** — When configuring the event trigger for the function, it is beneficial to use the configuration options to discard the uninteresting events. For example the message filter for SNS can select only messages with desired payload to invoke the function, while S3 trigger can specify the action performed on the bucket as well as prefix and suffix for the object key, that can be used to trigger the function, based on operation on a file with particular path, filename or extension [Mun18].

Despite the fact that the cloud provider takes responsibility for managing the underlying infrastructure and execution of function code, the way how the function is structured and configured, have a significant impact on its performance. Applying the knowledge of the Lambda design, by keeping the function lean and without unneeded dependencies along with configuring the application logic in an event-driven manner and leveraging the container reuse to execute the function code more effectively, it can greatly improve the performance of the serverless solution. It is beneficial to keep in mind that the CPU and network bandwidth is allocated proportionally to the configured memory, which can impact how long it takes for the function to execute. Moreover, proper function triggers configuration enable it to filter out undesired events.

Both of the provided example implementations follow the indications covered in section above, limiting the unneeded dependencies, leveraging the container reuse to maintain the connection with the cloud service clients as well as designing the serverless function to serve a single, well-defined purpose. The first example, covered in section 4.3.1, includes the asynchronous receipt processing pipeline, which combines the subsequent executions into a workflow, transforming the data each time, instead of making the serverless function wait idle for the results of the image processing task. The second example, which refers to the generation of interactive presentations, discussed in section 4.3.2, benefits from the altered resource allocation, including additionally allocated CPU to allow the function process the presentations more efficiently.

4.4.2 Serverless processing model

Having described the AWS Lambda as an example implementation of the Function as a Service model, it is essential to consider how the serverless function can be incorporated in the application logic. The event-driven architecture requires some components to trigger the serverless function and as a result receive the response or invoke other components, based on the configuration and code of the function. Moreover, the stateless and ephemeral nature of the serverless function requires it to communicate with external services to preserve its state. The serverless architecture relies on various services provided by the cloud platform, which along with the serverless function, make up the application logic [Gru19]. The simpler application can be composed from the handful of components configured to work together. However, when the complexity of the system grows, the application architecture composed from numerous independent components, communicating with each in an event-driven manner, creates a more complex distributed system, that can benefit from adhering to microservice architecture patterns, discussed in section 3.4.1.1.

4.4.2.1 Serverless microservices

The architecture of the web based services built with the serverless architecture can be divided into two distinctive categories based on their purpose [Mun19a]:

- **Public interface** — Responsible for exposing rich and flexible APIs, most frequently using HTTP protocol to communicate directly with the web based clients. It is using the API Gateway component or other services responsible for load balancing the requests, executing the serverless function, most frequently in a synchronous manner and returning its results to the client.

- **Internal services** — Supporting the public interfaces by providing various application capabilities. Moreover, the underlying architecture can utilise the asynchronous processing, passing events with more opinionated structure, leveraging other serverless components with simpler interfaces to decouple the internal services and exchange messages using intermediaries.

4.4.2.1.1 Public interface

The simpler web services can effectively be based on a handful of components to provide the basic functionalities. Services such as API Gateway can trigger the Lambda function, which in turn communicates with the downstream service, responsible for storing the data, performing certain operations and returning back with the response, as presented in the Figure 4.3.

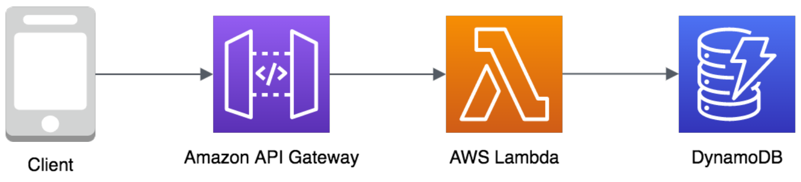

Figure 4.3: Architecture diagram of simple web service using API Gateway, Lambda and DynamoDB

There are several components provided by the AWS, which can serve a role of the public, client-faced interface of the web application, integrating with Lambda and other hosted services directly [Mun19a]. The components and their capabilities are listed below:

- Amazon API Gateway — Fully managed service enabling developers to define APIs, serving as a front door to the application, integrating with various services hosted in the cloud platform. The service distinguished several types of the API, including REST API (providing variety of additional features such as API caching, additional transformation and validation of requests, tighter integration with other services), WebSocket API as well as HTTP API (simpler and cheaper compared to REST APIs, providing only basic additional features). Moreover, The API Gateway can also provide additional capabilities such as caching, request throttling, different usage tiers for specified clients, security control as well as integration with other services incorporating authentication and authorisation capabilities.
- Application Load Balancer — Belongs to the family of the load balancer offering and it is supporting the HTTP traffic, regardless of the transmission protocol used on top of it. The requests are forwarded, based on the URL and HTTP method, to the Lambda or other services according to the configuration. Using Application Load Balancer can be cost effective when the service is experiencing high throughput and the requests can be passed directly to the associated services.
- AWS AppSync — Managed service, providing the capabilities to define GraphQL APIs, which can map the particular requests to various services, pulling the data from various data sources for a single request. Moreover, it can incorporate caching on the server and client side to improve performance, leverage GraphQL subscrip-

tions for real-time communication as well as integrate with other services to provide authentication and authorization capabilities.

All of the mentioned components provide the capabilities of exposing an API, which serves as an entrypoint for the application logic. Besides accepting the user requests and redirecting them to specified components, some of the services enable additional capabilities such as caching, traffic management, request filtering and transformation, authentication and authorization as well as throttling the requests to protect the backend services. The definition of an API can be generated along with the documentation, when using tools like Swagger [Ama21l]. The additional capabilities of mentioned components are covered in more detail in section 4.6.1.

Another frequently considered topic, when using the services like API Gateway, is where to perform the routing of the request. The Lambda code can effectively use one of the frameworks to provide some additional routing inside the handler logic, granting better portability. However, such approach introduces several challenges, which include applying the security constructs and performance configuration to the whole monolithic function as well as making the proper logging, monitoring and debugging more difficult to implement [Mun19a].

4.4.2.1.2 Internal services

The synchronous model of the public interface can work effectively with simpler services, communicating with other serverless components and returning the response back to the client. However, when the communication is more complex and it includes various services, passing messages to each other, there are several places when the errors can occur, making some of the components waiting for response [Mun19a]. When the application is experiencing heavier load, the services communicating in a synchronous manner and leveraging the serverless technologies can observe timeouts and throttling as well as increased cost related with the Lambda functions executing longer and waiting for responses from the polled services. The API Gateway is configured with maximum integration timeout for synchronous function invocation equal to 29 seconds, after which it returns an error to the client. Moreover, the function execution time is limited to 15 minutes, which along with its rapid scalability can be more challenging when communicating with some external or legacy services, that are not capable to scale accordingly, requiring to manage the number of active connections [Itu19].

One of the solutions for the mentioned problem, is to rethink how the communication between components works and redesign it to incorporate asynchronous workflows. It introduces a similar approach as with microservice architecture, making the components decoupled by using some intermediaries like message brokers to exchange and preserve the events occurring in the system in a reliable way. The API Gateway request can trigger the AWS Lambda, which in turn communicates with some other service, starting the asynchronous processing and returning the invocation status to the API Gateway, responding to the user. The processing flow can further get through several services, responsible for processing the application logic and preserving the state accordingly, using some of the serverless platform components to exchange the messages reliably, without tight coupling between services and serverless functions. The communication with external or legacy systems can be appropriately buffered and throttled to prevent from losing events and affecting their performance [Itu19]. Some of the more complex workflows may benefit from the orchestration of the processing using services such as AWS Step Function, which is described in more detail in section 4.4.2.2.

Components provided by AWS, serving the role of queues, message brokers and message buses as well as services streaming the events, enables the developers to decouple the services and redesign the flow to embrace reliable asynchronous processing, are described below:

- Simple Notification Service (SNS) — Managed publish-subscribe messaging service with an ability to reliably exchange messages with low latency, pushing and delivering the messages to other components subscribing to a predefined topic. SNS provides high throughput with autoscaling capabilities, allowing to fanout the messages, storing them durably across multiple Availability Zones, but providing no additional persistence beyond retry logic, tracking failures when lambda is not available. The events can be filtered based on the message attributes, pushing the event only to selected subscribers. The messages can trigger Lambda function or be published to HTTP endpoints, services such as Amazon SQS or Amazon Kinesis Data Firehose, email, SMS or mobile push [Woo20]. In case of serverless functions, each of the published messages translates to single Lambda execution, however its concurrency can be limited as well as additional configuration can be applied to expose parameters allowing to ensure the FIFO ordering and deduplication of events, however by default the SNS does not provide such guarantees. SNS is designed for unidirectional and fanout communication, targeted primarily to Lambda function or HTTP based endpoints.
- Simple Queue Service (SQS) — Managed message queue service, storing and exchanging the messages between various software components, reliably preserving them without requiring the service to be available. SQS is scaling automatically with configurable batch size, storing the messages durably across multiple Availability Zones with an ability to persist them from 60 seconds to 14 days. The batch of messages is polled by one of the consumers, becoming unavailable for others. Once the messages from the whole batch are successfully processed, they are removed from the queue, otherwise they become available to be picked up again after the visibility timeout expires. It enables the queue to buffer and rate limit the incoming events, securing the connected downstream services from bursty traffic. With standard configuration the SQS provides best effort message ordering, with at least once delivery, guaranteeing high throughput with lower cost. However, the service can be configured to expose additional information, enabling downstream services to order and deduplicate the incoming events. SQS finds applicability when the incoming traffic needs to be stored in a reliable way and buffered to be further consumed by downstream services [Pir20].
- Amazon Kinesis Streams — Highly scalable, managed services responsible for collecting and processing real-time streams of data coming from various sources, including data and video streams, enabling the application to preserve the data and analyse

it. The data stream is split into multiple shards, based on defined buffer capacity and time interval, passing the payload to the Lambda functions which are polling the stream, providing the data to multiple services. The data throughput can be limited by batch size or Lambda concurrency, durably storing the messages across multiple Availability Zones and persisting them up to 7 days if configured. The processing failures can be handled by retrying the events in order from a shard based on a checkpoints. The shard is processed until successful completion or it can be split to extract the erroneous records, handled separately [Woo20]. Amazon Kinesis Streams are frequently used for real-time processing with massive throughput of data, tracking the order of records and gathering the data stream from multiple consumers, such as logs, sensor data or financial transactions.

- Amazon EventBridge — Serving a role of serverless event bus, observing the events from various data sources, including AWS services, developed applications and integrated SaaS providers. The events can be further filtered out based on defined rules and routed to the downstream consumers, which receive the events reliably, while their producers can remain decoupled. EventBridge enables the application components to exchange messages together by ingesting and routing the data across underlying infrastructure, applications as well as external services. The messages are stored durably across multiple Availability Zones, however the service does not provide additional events persistence. EventBridge finds applicability when the application requires to connect a lot of different services, including external integrations, while providing granular target rules to filter out the traffic [Woo20].

The mentioned services are not mutually exclusive and can be combined together to utilise their features and provide desired capabilities to the web application architecture as well as enable effective communication by invoking serverless functions in an event-driven manner. Some of the patterns including described components and leveraging their capabilities are covered below:

- **Throttling and buffering** — The autoscaling capabilities of the Lambda function can be a problem for the downstream services, including for example the hosted relational databases, which may not be able to scale accordingly or exhaust the available connections pool. SQS and Kinesis Data Streams can be used to persist

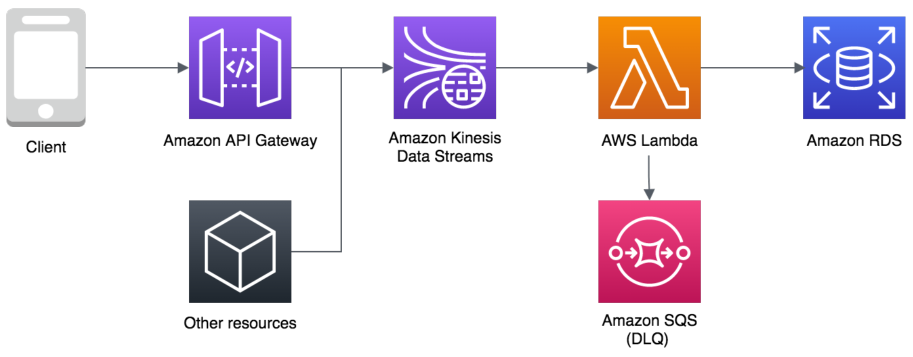

Figure 4.4: Architecture diagram of service using Kinesis Data Streams to throttle and buffer the incoming data

and batch the incoming events. Both of the mentioned approaches can limit the concurrency of incoming traffic, serving a role of buffer, aggregating the incoming requests and executing the function with a single batch or shard, leading to lower cost per function invocation. DLQ can be configured to store failed records. Alternatively, Lambda Destination can be leveraged to execute another Lambda function with an event including context of the failure [Les19].

- **Fan-Out** — When the payload does not require additional processing, the event can be sent directly to SNS, which pushes the message to subscribers based on the configured filtering. SQS can be configured to subscribe for the events published by SNS, batch the incoming data and share it further with connected Lambda functions or other services. Alternatively, for the traffic characterized with high, consistent throughput and large payloads, Kinesis Data Stream can be a more cost efficient solution [Les19].

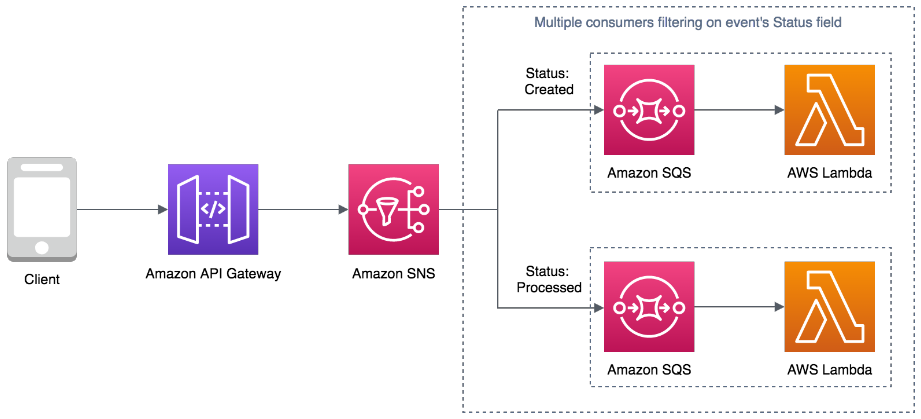

Figure 4.5: Architecture diagram of the Fanout pattern including SNS and SQS to reliably send out events

- **Streaming** — The data coming from various data sources can be effectively processed and stored for further analytical purposes. SNS can be used to discard

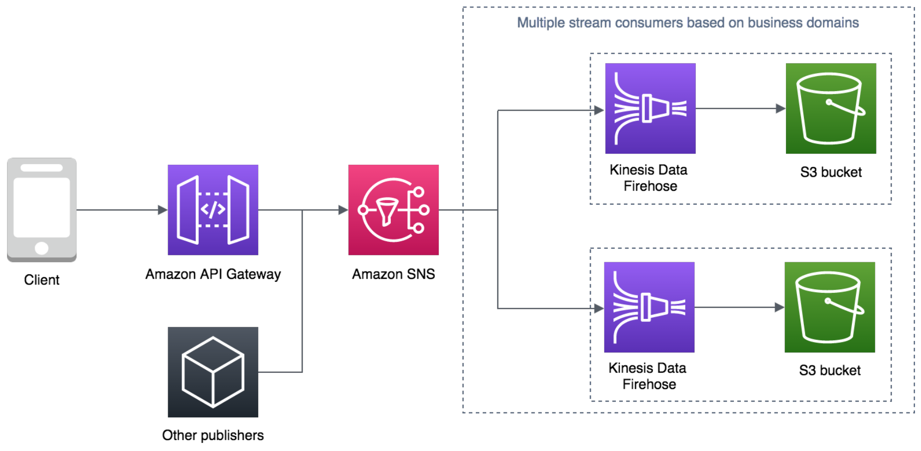

Figure 4.6: Architecture diagram of SNS and Kinesis Firehose preserving the incoming stream of events in S3 bucket

unwanted events, filter the incoming events based on the different business areas and pass the data to Kinesis Firehose to preserve the data, for example in S3 [Les19].

4.4.2.1.3 Serverless microservice principles

Besides the components serving the role of the entry point to the web application and the internal services, responsible for messaging between application components, there are also other components taking an integral role in the applications built in the serverless paradigm as well as shaping its architecture. The databases capable of preserving the data and reacting to its changes by streaming the updates to polling services are described in more detail in section 4.5. Moreover, numerous services hosted in the cloud provide various capabilities, including for example image processing, video transcoding or natural language processing, which can be easily integrated in the developed web applications.

The first of the described examples, presented in section 4.3.1, shows how the S3 and DynamoDB are incorporated in the application logic to preserve the images and results of the processing as well as notifying other components about data changes. Textract is incorporated in the asynchronous processing flow to analyze the uploaded receipts and later publish the results to SNS, fitting into the asynchronous event-driven processing.

When the serverless solution evolves and grows in complexity, it is beneficial to split the application logic into separate domains with well-defined boundaries based on the business capabilities. It resembles the microservice architectural pattern, described in section 3.4.1.1, which refers to a set of small services running on its own, communicating via message passing and focused around specific business capabilities.

The serverless microservices are going a step further, splitting the business logic across several serverless functions. Each of them is responsible for handling and responding to specific events, with the application logic isolated for particular workflow along with granular configuration in terms of access to external services, underlying resource configuration as well as individual selection of technology and libraries to serve its role effectively. However, most frequently the serverless functions working within the same service boundary and using the same data sources, define some shared library of methods, extracting the business logic from the function handler, responsible for executing the function, and abstracting away the communication with underlying resources. Each of the serverless functions include the required methods from the shared library to perform the processing, which are packed along the function handler in the deployment process, reducing the functionalities duplication.

Each of the microservices can operate on its own domain and communicate with others via message passing. To achieve that, each service should expose an interface to make other services retrieve information or perform defined actions. Moreover, if the operation in one of the services should affect others, it can publish a message to some intermediaries, notifying other components subscribing for changes to react for them appropriately. Some serverless functions can be designed to serve as a router, providing the API for other services to communicate with as well as receiver, subscribing for the messages from other components, such as SNS or EventBridge, to trigger the further processing. It enables the serverless microservices to preserve the loose coupling between different microservices, making the services know as little as possible about the implementation of others, communicate via well-defined interfaces, as well as maintain high cohesion within the same service to effectively manage the business domains [Dal20a].

The described serverless microservice architecture enables developers to leverage the benefits mentioned for the microservice architecture itself. By decomposing and separating the systems into independent components, it is easier to maintain the system over time, introduce required changes and evolve the application. Different teams can manage and develop different parts of the application, focusing on desired capabilities of the services, deciding on programming languages, libraries, data stores to meet the service requirements as well as deploying, configuring and scaling the services independently.

Moreover, various processes and design principles from the microservices architectural pattern can be also applied to the application built using the serverless paradigm. The decentralized data management allows services to preserve their state independently, but propagating the more demanding operations in an asynchronous way, requires more sophisticated coordination mechanisms to reliably process the transactions, handle failures and retries, utilising the eventual consistency. Decoupling components make it easier to develop services independently, however such a distributed system requires some communication mechanisms between its elements. Exchanging messages via intermediaries, service contracts as well as designing for failure, are necessary to make the serverless web application work properly. Lastly, the automation that is set up properly, providing continuous integration and deployment, is necessary to make the development process manageable. Automated end to end testing on the deployed environment is essential to give enough confidence that the system works properly as well as proper logging, monitoring and observability are required to track the system capabilities and trigger alarms in case of inconsistencies.

Summarising, according to Jeremy Daly in his article describing the various serverless patterns [Dal20c], the serverless microservices benefits from adhering the following principles:

- **Ownership of the service private data** — Each of the serverless microservices should own its data. If another service requires the same data, it should be replicated or it is an indication that the services should be combined together or the application architecture should be rethought.
- **Independent deployments** — The microservices should be independent and self contained. The dependencies and communication should be based on well defined interfaces and handled by intermediaries to ensure the services are loosely coupled. Maintaining this capability allows developers to independently develop, evolve and deploy the services.
- **Utilising eventual consistency** — The data replication and denormalisation are commonly used in the microservice architecture and the same solution can be applied in the serverless architecture.
- **Using asynchronous workloads** — With the Lambda function billed for the processing time, it is cost effective to consider the asynchronous processing, instead of making the function wait idle or making the processing to reach the timeout.

Frequently, the processing can be redesigned to hand off the task, which is reliably processed in the background in an asynchronous manner. For use cases that require more complicated coordination, the services provided by the cloud platform can be used, keeping the serverless function small, having a single and well-defined purpose.

- **Keeping the service small, but valuable** — With the granularity and elasticity of the serverless architectures, it is applicable to create even some smaller microservices including only a few functions, if they serve their role well and deliver business value. Based on the changing requirements and evolution of the application these can be easily relocated to suit the need.

4.4.2.2 Function choreography and orchestration

The architecture of the serverless function is suitable for the event-driven workflows, in which communication between subsequent components is managed by services such as SNS, SQS and EventBridge, is asynchronous and loosely coupled. However, some of the operations require a more fine-grained orchestration approach, with capability to preserve the state between subsequent invocations and give more insight into the business context of performed operations.

AWS Step Function is an example of a service providing capabilities of orchestrating the workflow and managing its state. Step Function enables to define a state machine, responsible for resembling the business workflow as well as triggering individual components and Lambda functions to avoid additional coordination in the code of the services. It provides necessary abstraction to define sequential and parallel tasks, branching logic, mapping over a set of tasks as well as handling errors and retry logic. Moreover, the service allows developers to reason about the state of the executed workflow by visualising the components of the orchestrated flow along their state at individual stages of processing [Ama21l].

In the sections below, the choreography and orchestration of the workflows are described in more detail, as two frequently distinguished modes of the interaction in the serverless microservices architecture.

4.4.2.2.1 Choreography

The choreography refers to an asynchronous, event-driven processing, in which services work independently, using intermediaries in form of message brokers and queues to pass the events to other components invoked further, forming the chain of component invocation performing the business logic. The interaction model is suitable for simpler operations, when services are not closely related and most frequently belong to separate business domains. The choreography approach is used in the receipt processing flow, presented in section 4.3.1.

The benefits of such an approach include loose coupling of the services, which can be changed and scaled independently. There is no single point of failure in the system, in which the events can bring additional business value, when processed in the business intelligence reports. On the other hand, monitoring and reporting in the choreography approach is more challenging, along handling errors, retry mechanisms and timeouts.

The business flow is not captured explicitly, instead it is spread across several loosely coupled services [Ata21].

An example architecture diagram of the food ordering workflow, modeled in the choreography approach based on the article written by Yan Cui [Cui20], is presented in Figure 4.7.

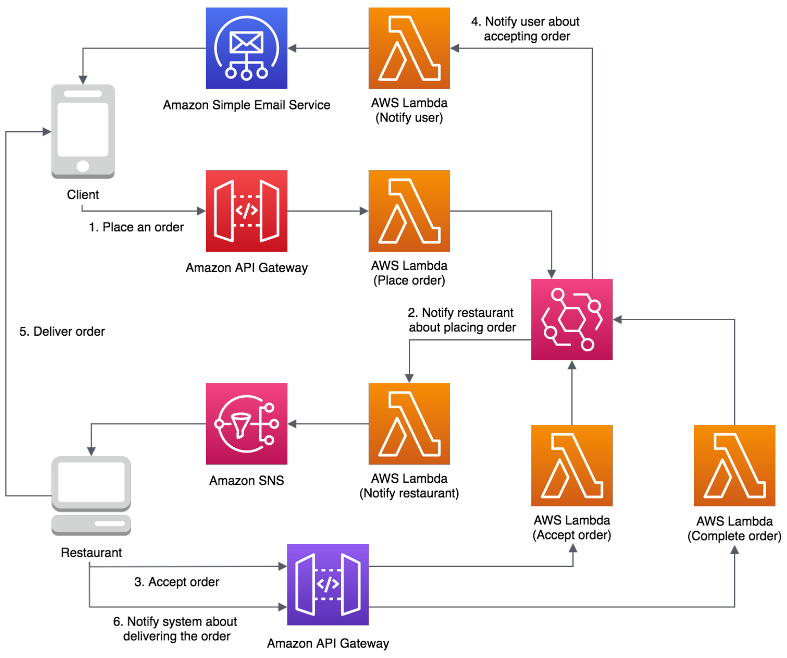
Amazon Simple Email Service
AWS Lambda (Notify user)
4. Notify user about accepting order
Client
1. Place an order
Amazon API Gateway
AWS Lambda (Place order)
2. Notify restaurant about placing order
5. Deliver order
Amazon SNS
AWS Lambda (Notify restaurant)
AWS Lambda (Accept order)
AWS Lambda (Complete order)
Restaurant
3. Accept order
6. Notify system about delivering the order
Amazon API Gateway

Figure 4.7: The food delivery workflow modeled using the choreography approach

4.4.2.2.2 Orchestration

Instead of configuring services to call each other, the orchestration approach refers to using some dedicated orchestrator component, serving a central role in the processing, coordinating the execution of components, handling interactions between them and maintaining state of the processing, along with handling retries and failures. Workflow orchestration finds applicability in more complex operations, that require sophisticated flow control, aggregation of resources from different sources and handling the cooperation between microservices in a reliable way. The orchestration approach is used in the second example implementation, considering the orchestration of generating interactive slideshows, covered in section 4.3.2.

The orchestration approach enables the developers to capture the business flow of the process, apply error handling and retry policies, along with removing the need to provide additional logic to handle it in the services. The workflow logic is contained in one place, providing additional monitoring and visualising capabilities to reason about the status of the processing. The downside of orchestration include additional cost related with work of the orchestrator, which is a single point of failure in the processing and makes the incorporated components more tightly coupled [Ata21].

Aforementioned example of the food ordering workflow, modeled in the orchestration approach, is presented in Figure 4.8.

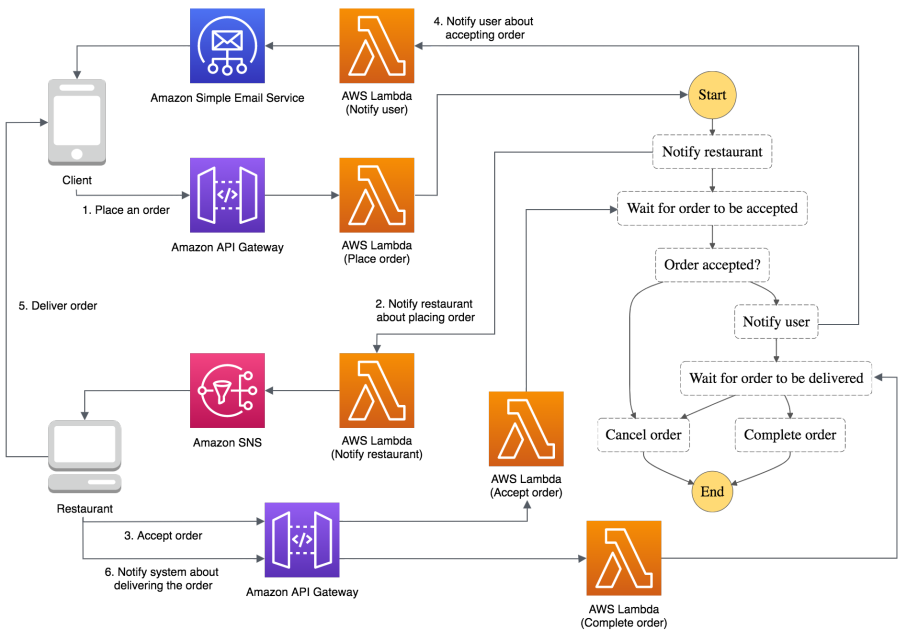
Amazon Simple Email Service
AWS Lambda (Notify user)
4. Notify user about accepting order
Start
Client
Notify restaurant
1. Place an order
Amazon API Gateway
AWS Lambda (Place order)
Wait for order to be accepted
Order accepted?
5. Deliver order
2. Notify restaurant about placing order
Notify user
Wait for order to be delivered
Amazon SNS
AWS Lambda (Notify restaurant)
Cancel order
Complete order
AWS Lambda (Accept order)
End
Restaurant
3. Accept order
6. Notify system about delivering the order
Amazon API Gateway
AWS Lambda (Complete order)

Figure 4.8: The food delivery workflow modeled using the orchestration approach

Moreover, the orchestration approach using the Step Function is suitable for using the distributed saga pattern [GMS87] to coordinate the distributed transactions across multiple microservices including independent datastores. It can be applied to coordinate some more complex processes including multiple operations, which can fail during the processing and require orchestrating a series of compensating transactions, reverting the executed operations and cleaning up the state. An example of such orchestration workflow is presented in the Figure 4.9, covering the multistep booking process.

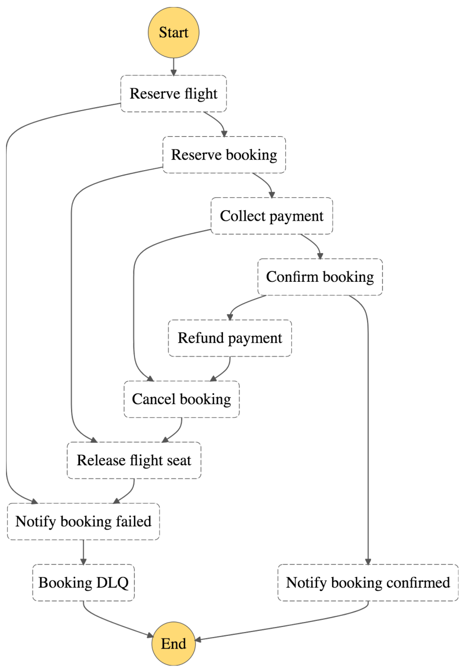

Figure 4.9: The multistep booking workflow using the distributed saga pattern

4.4.3 Case study - Generating interactive slideshow based on LaTeX files

The following section describes in more detail the optimisation process behind the second example implementation of the web application using the serverless technology introduced in section 4.3.2. The Step Function is used in the designed solution to coordinate the processing of several AWS Lambda instances that perform the processing of the individual stages in the entire flow.

In the presented research, the tests and measurements of the individual processing steps and their configurations are conducted based on the two example presentations prepared using the LaTeX typesetting system. Both of the sample presentations, described in more details below, resemble the exercises which could be used in the mentioned service and characterise with different levels of complexity due to their content.

- Exercise 1 — Covers the 71 page long presentation that includes the solution derivation for the trigonometric equation.
- Exercise 2 — Is more demanding and larger, 103 page long presentation. The first part includes graphics and describes the required steps for solving a geometry problem, with additional calculations included in the second part of the exercise.

From the initial research and experiments with the possible solutions, the process of exporting presentations to the PDF format has been identified as a bottleneck in the entire processing, taking significantly longer than the other steps when performed in parallel. The conducted performance tests consider mainly the processing duration from the client point of view, which is calculated as a time from requesting the invocation of Step Function, until the processing results are received by the client, once the slideshow generation is completed and the assets are available in the S3 bucket. Along with measuring the processing time, several other metrics have been gathered, giving more insight into the performance of individual steps of the processing. Measurements preserve the preconditions listed below:

- Each of the tests is performed 10 times and the presented results are the average of the conducted measurements.
- Initially, the infrastructure has been redeployed after each execution to ensure colds starts, however thanks to the horizontal scalability of the serverless components and further observation, the same results in terms of the processing time have been obtained when the client application performed 10 requests at the same time. Each of the functions executed in parallel included the information about cold start. Such behaviour confirms the scaling capabilities of the application based on the serverless architecture as well as resembles the situation when several users requested the presentation within short period of time, determining the worst case in terms of observable performance when the cold starts occurred.
- The LaTeX files related to generating the presentation are removed from the temporal disk space of the function execution container once the processing is completed. The operation is performed to prevent a situation, when the subsequent function

execution utilises the same instance of the serverless function and can benefit from the intermediate processing results from the previous invocation, which could affect the warm start execution results.

4.4.3.1 Custom Lambda container image and Lambda layer

As mentioned before, generating of the presentation based on the LaTeX file can be classified as a custom type of processing for the serverless function runtime, that goes beyond the standard set of its tasks. The section 4.4.1.2 discusses in more detail the possibilities of the serverless platform in terms of performing such non-standard tasks. The following section presents the results of generating the PDF file with presentation using two approaches:

- Container image — uses the official Amazon container image for Node.js runtime with additional LaTeX distribution and other libraries required for the processing.
- Lambda layer — includes the LaTeX distribution extracted from the aforementioned container image into the separate archive along with the additional Lambda layer including the Perl interpreter required for the file processing.

Both of the approaches execute the same function code running in the Node.js runtime, using the provided LaTeX distribution to generate the presentation.

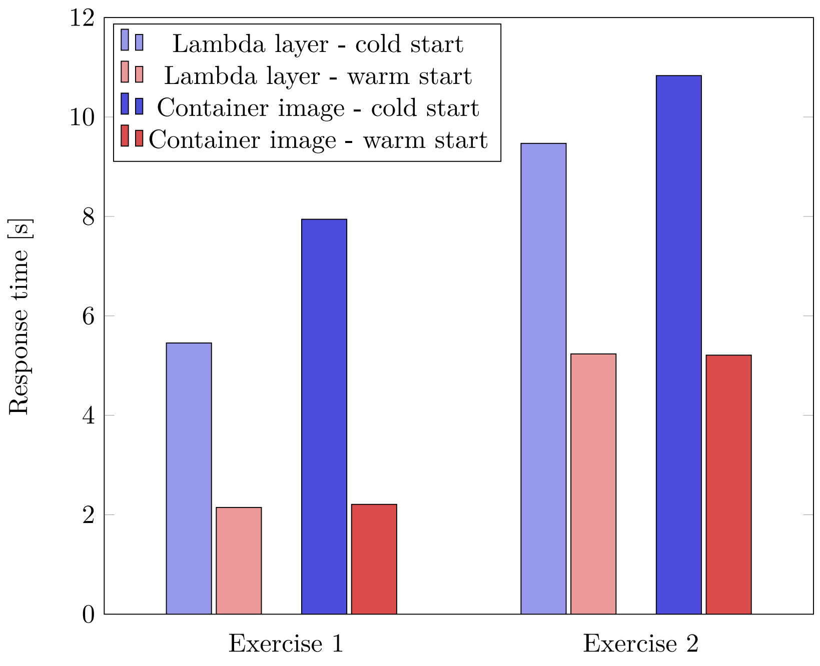

Figure 4.10: Response time comparison of the solutions using custom container image for AWS Lambda and Lambda layers with the LaTeX distribution

Key takeaways of the container image and Lambda layer processing comparison:

- **Cold start times** — The significant processing time difference can be noticed when comparing the executions with cold and warm starts. The cold start time can be further divided into two parts. The first one considers the overhead related with the resource allocation from the large resource pool hosted by the cloud provider, downloading the function code and starting new, ephemeral container to execute the function. The second part is related with bootstrapping the function runtime as well as initialising dependencies required to execute the function handler. The detailed results of the cold start analysis are presented in the Table 4.1 and 4.2.

Exercise	Execution	Function [ms]	Environment [ms]	Runtime [ms]	Execution [ms]
Exercise 1	cold start	3609	914	273	2422
	warm start	1614	34	0	1580
Exercise 2	cold start	7802	468	279	7055
	warm start	4661	52	0	4609

Table 4.1: Impact of the cold start on the presentation processing task for approach using Lambda layer

Exercise	Execution	Function [ms]	Environment [ms]	Runtime [ms]	Execution [ms]
Exercise 1	cold start	6548	637	1108	4803
	warm start	1638	45	0	1593
Exercise 2	cold start	8945	316	810	7819
	warm start	4614	35	0	4579

Table 4.2: Impact of the cold start on the presentation processing task for approach using custom container image for Lambda

The runtime initialisation time of the solution using Lambda layer is comparable for both exercises and it is equal on the average to 273 ms and 279 ms for Exercise 1 and Exercise 2 accordingly. For the processing using the container image solution, the runtime initialisation time is on the average equal to 1108 ms and 810 ms, which has a visible impact on the overall processing time. Moreover, the runtime initialisation time for the solution based on the container image is additionally billed accordingly to AWS Lambda pricing, on the contrary to the approach using the Lambda layer in which the cold start time is not included into the cost. The processing performance evaluation for the Exercise 1 preceded the Exercise 2, which could explain the decrease of the time required to download the function code and allocate resources.

- **Warm start times** — When considering the prewarmed function environments, there is no visible difference in the overall processing time between both of the tested solutions.
- **Step Function overhead** — The orchestration overhead computed as the overall processing time of the Step Function decreased by the time of particular tasks defined in the Step Function flow, ranges from 400 ms and 450 ms, for both cold and warm starts.

Based on the results of the comparison the Lambda layer is selected as a more suitable solution for further workflow optimisation.

4.4.3.2 Function chain length

In the previous section, the task of generating PDF presentations from the LaTeX files is discussed, however the flow proposed in section 4.3.2.2 describes the additional step of extracting the individual slides of the presentation to SVG files. To achieve this, the lightweight *pdf2svg* [Bar21] library is used. The following scenario measures the processing performance in two configurations:

- Single function — the conversion is performed in a single function that generates the presentation and extracts the slides, which are stored further in the S3 bucket.
- Function chain — the processing is performed in the separate functions, storing the intermediate PDF file in the S3 bucket.

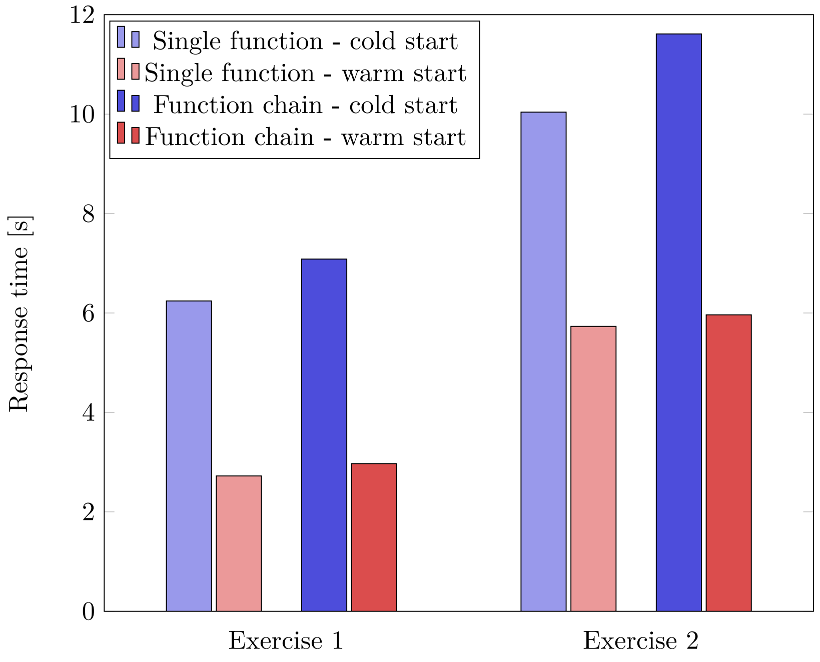

Figure 4.11: Response time comparison of the solutions using single function and function chain to process the presentation

Key takeaways of the comparison:

- **Keep the function chain short** — When chaining several functions to perform some workflow, the overhead related to the communication and the cold start compounds. Additionally, the intermediate results need to be stored in some external component, which introduces additional work and latency that can be noticeable, especially when working with large volumes of the data.
- **Lambda layers constraints** — The small size of the *pdf2svg* library makes it suitable for adding it and its dependencies to the Lambda layer, that includes the LaTeX distribution already. However, it is crucial to take into account the limitations of the approach using the Lambda layers, described in more detail in section 4.4.1.2, because at some point it may be not possible to extend the function further.

4.4.3.3 Parallel processing

The serverless architecture is characterized by the capabilities of instant autoscaling to meet the demand of the processing. Some types of the computation can be re-designed to utilise such feature of the architecture and improve the overall processing time by parallelising the workload. Generating the presentation from the LaTeX files to the PDF format is a custom type of processing task, that under some circumstances can be parallelised to a certain degree as well. The predefined range of slides from the presentation prepared using the Beamer package [Wri21], can be selected using the *\includeonlyframes* directive. However, the LaTeX files processing requires to repeat some initial and common part of the processing regardless of the selected subset of slides, which causes some part of the processing to be repeated for each of the presentation chunk.

The taken approach splits the LaTeX processing into several serverless function executions running in parallel. Each of the functions is assigned to generate the slides for predefined presentation chunks, containing 5 consecutive slides. Once the processing of all serverless functions is completed, along with the commentary transcription and metadata extraction, the user is notified about the results of the processing. The Step Function processing graph is presented in Figure 4.13, indicating that the presentation generation is divided into chunks, that are mapped to independent function calls and executed in parallel.

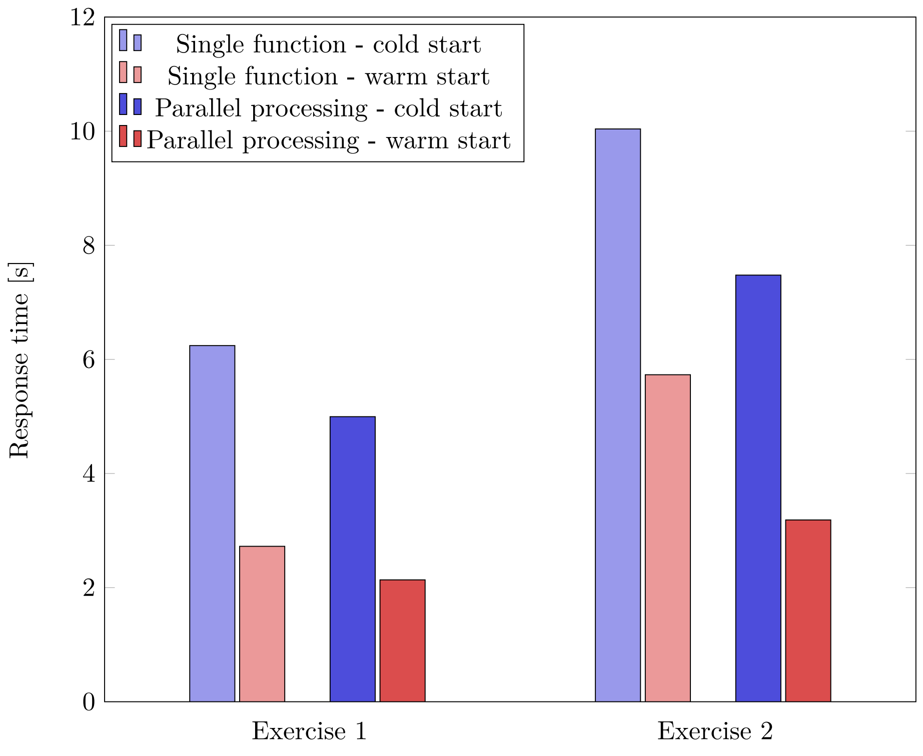

Figure 4.12: Response time comparison of the solutions using single function to generate the complete presentation and the parallelised processing of separate chunks

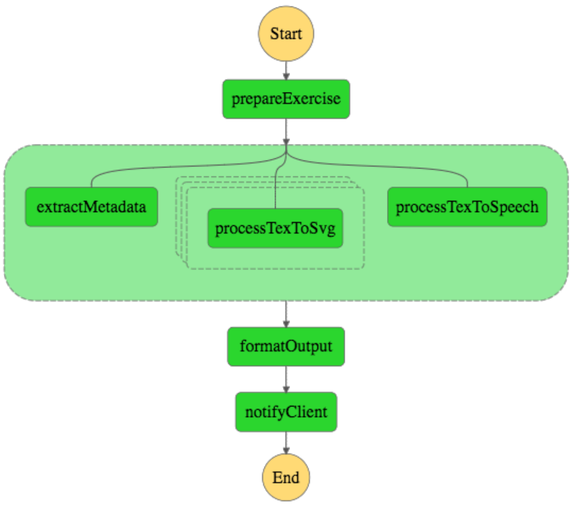

Figure 4.13: Step Function processing graph — Generating the presentation chunks in parallel

Key takeaways of the parallel processing compared to the single function execution:

- **Parallelising the LaTeX files processing** — Even though the presentation generation processing cannot be effectively parallelised, the overall processing time is reduced for both of the provided examples. The complexity of the selected examples has a significant impact on the total processing time. When measuring and comparing the processing time on the local machine, generating the chunks independently for the Exercise 1 reduced the overall processing time by 32%, while for the Exercise 2 it reached approximately 74%, that confirms the previous statement.

  The average execution time for the presentation generation task of the chunks processed in separate functions compared to processing of the complete presentation in a single function is presented in Table 4.3

Exercise	Execution	Complete [ms]	Avg. of chunks [ms]	Time reduction [%]
Exercise 1	cold start	3349	2748	17.95
	warm start	1641	1100	32.97
Exercise 2	cold start	7887	4310	45.35
	warm start	4679	1313	71.94

Table 4.3: Time reduction of the presentation processing task for the complete presentation and chunks including 5 slides

- **Orchestrating parallel processing brings additional overhead** — It is essential to consider the additional overhead related to the orchestration of several serverless functions processing the presentation chunks in parallel. In the following example, the orchestration overhead is calculated as the Step Function duration decreased by the average tasks duration. It is equal to 873 ms (cold start) and 890 ms (warm start) for Exercise 1 as well as 1765 ms (cold start) and 1745 ms (warm start) for Exercise 2.
- **Cost of the parallel processing** — Besides the processing overhead, the par-

allel execution brings additional cost for each of the executed serverless functions and its execution time. Moreover, for the given example the processing cannot be effectively parallelised and some part of the processing and retrieving the file is repeated for each invoked function, impacting the overall cost of the computation.

4.4.3.4 Updating the client with processed batches

The approach presented in the previous section enables to effectively parallelise the processing and reduce the overall processing time, but it is restricted by the fact that all of the presentation chunks needs to be processed to push the update to the client. When taking into consideration how the presentations are prepared, most frequently the first several slides include the introduction to the exercise that usually takes more than 10 seconds. From the user perspective, only the first batch is required to start presenting the slideshow, while the remaining part of the assets can be delivered later.

The further improvements of the processing flow takes into account the aforementioned assumption and parallelises the presentation processing on the higher level, to deliver the updates of the processed presentation chunks independently. The Step Function processing graph of suggested approach is presented in Figure 4.14.

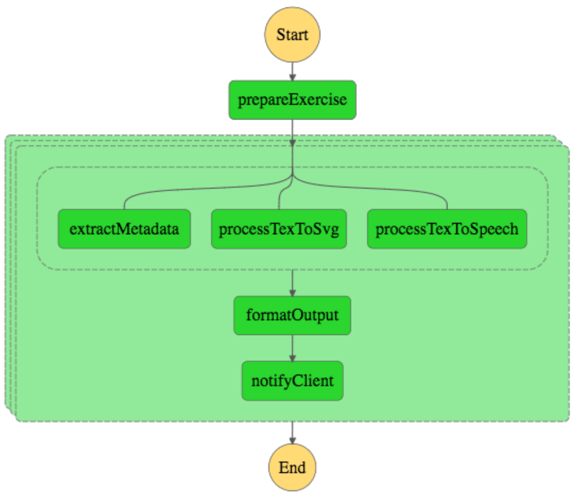

Figure 4.14: Step Function processing graph — Delivering updates to the client with independently processed presentation chunks

The further measurements reflects the response time for two configurations of the flow, described below:

- First batch with 5 initial slides processed and the second batch with the remaining part of the presentation — the processing is effectively parallelised in two branches.
- Each of the batches include information about 5 consecutive slides — the processing is parallelised depending on the size of the presentation, with each of the branches processing 5 consecutive slides.

The results of the processing are compared with processing time of the complete presentation by a single serverless function and when the processing is parallelised, as shown in the previous section. The results are presented in the figures 4.15 and 4.16.

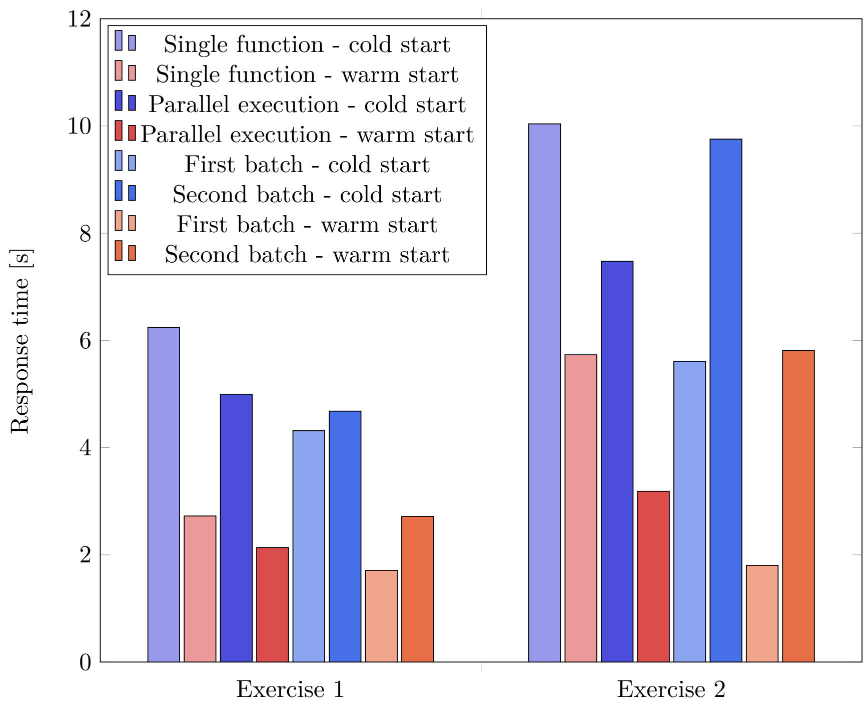

Figure 4.15: Response time comparison of the solution delivering the update to the client in the first batch including initial 5 slides and the second batch with remaining data

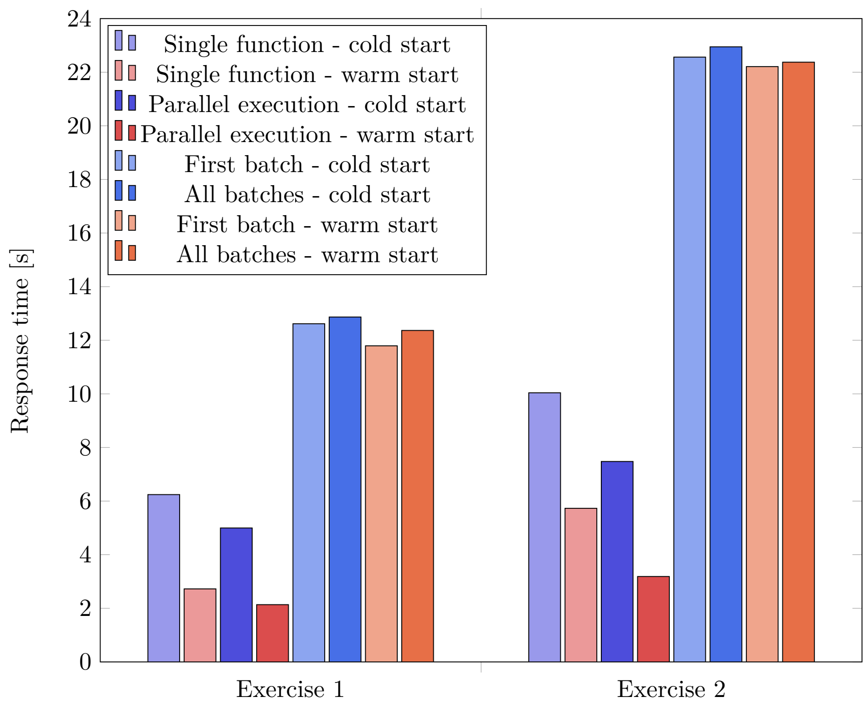

Figure 4.16: Response time comparison of the solution delivering the update to the client in the separate batches

Key takeaways of the proposed approach:

- **Improved response time of the initial batch** — For the first of the considered configurations, with the results presented in Figure 4.15, the response time for the first batch including the first slides is shorter. Once it is received by the client, the user can start previewing the slideshow, while the second batch including the remaining data can be processed and received by the client later. Such a solution enables the user to start previewing the exercise quicker, while the processing time of the complete presentation is hidden from the user perspective.
- **Increase of the Step Function overhead for the fine-grained workflow coordination** — In the second configuration that splits the presentation chunks equally and parallelises the processing, the overhead of the Step Function responsible for the orchestration makes the processing significantly longer. Moreover, the batch including 5 initial slides is frequently delivered as one of the latest, significantly postponing the time when the user can start previewing the exercise, as it is presented in the Figure 4.16.

4.4.3.5 Comparing serverless solution with the traditional architecture

When deciding on the serverless architecture, there is always a question how the solution will perform compared to the solution implemented using the more traditional architectures. To answer that question for the given problem, a similar service based on a simple server running within the container has been created and hosted on the AWS Fargate. The server is executed in the Node.js runtime, similarly to the serverless functions from the previous steps of the study. Moreover, it is reusing a significant part of the logic responsible for the initial validation of the task, processing the LaTeX presentation to PDF and then extracting separate slides to the SVG files, using the same libraries as well as uploading the results to S3 once the processing is completed. The container is configured with the same amount of memory and CPU as the serverless function used for the processing in the serverless implementation.

The figure 4.17 presents how the solution using the traditional architecture performs compared with the various configurations of the serverless solution.

Key takeaways of the serverless implementation comparison with the solution using the simple, containerized server application:

- **The performance of serverless implementation is comparable, when it is not affected by the cold starts** — The initial serverless solution, orchestrating the flow and using single function to process the LaTeX file, performs similarly when executed with the warm start, compared to the solution using the simple server hosted by AWS Fargate in the container. However, when the execution is affected by the cold starts, the response time is significantly longer as presented in Figure 4.17. It is beneficial to consider the interest in the service as well as the traffic pattern to estimate how often such a phenomenon may take place, when deciding on architecture selection or mitigating the negative influence by pre-warming the containers programmatically or by the services, such as Provisioned Concurrency for AWS Lambda.

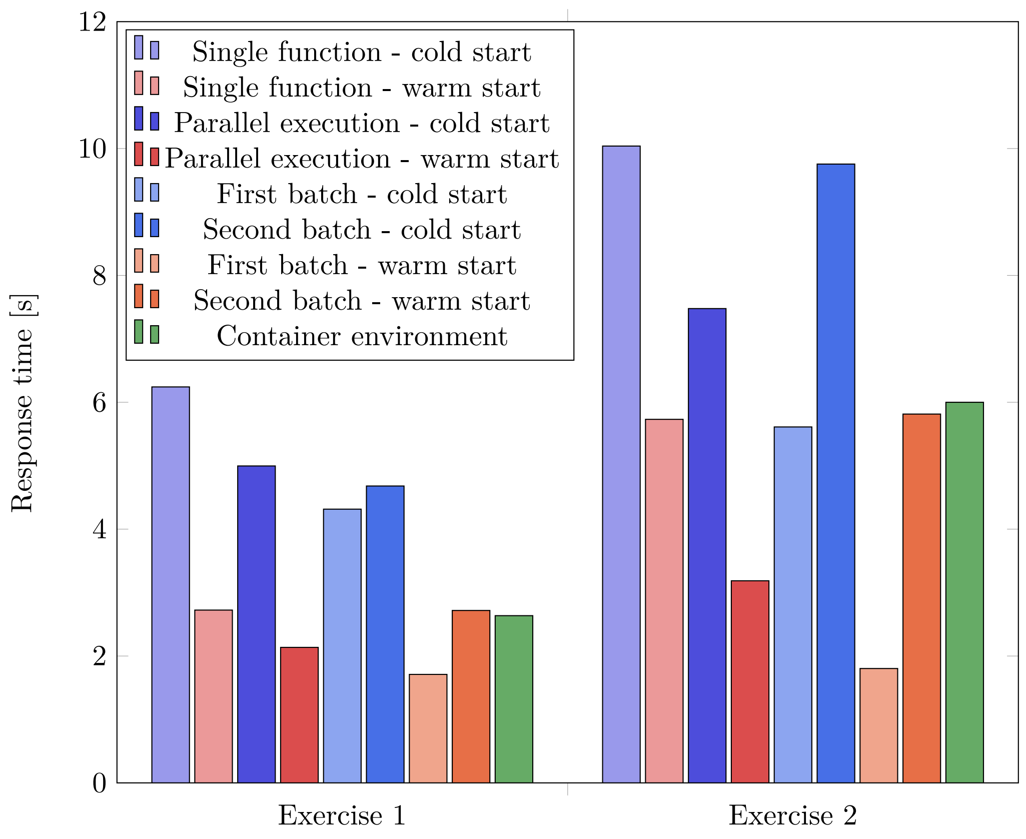

Figure 4.17: Response time comparison of the solutions using the serverless and traditional architectures

- **Re-designing the serverless processing to utilise its benefits can improve performance** — The presented configurations of the serverless solution confirms that remodeling the processing to use the benefits of the serverless paradigm can bring visible benefits in terms of the performance, compared to the solution using the more traditional architecture. However, some aspects and tasks of the proposed flow are required for the serverless solution, such as uploading the results of processing to S3, which could be further accessed by the user. The solution using the server could aggregate the results of processing locally, without the need to communicate with external component to store the files and return the results directly to the user, reducing the amount of work and cost.
- **Built-in scalability of the serverless architecture** — One of the features of the serverless architecture is the possibility of almost instant and effortless processing parallelisation, that reduces the overall processing time in that case. As mentioned in the introduction to the section, the performance tests of the executions characterised by the cold starts have been measured by sending 10 requests simultaneously, resulting with similar performance as the single executions interleaved with redeploying the infrastructure. This approach allows to infer about the behaviour of the service, when several users will use the service simultaneously, while showing the worst case in terms of the service performance. Further research could

investigate in more detail, how the implementations using both of the architectures are performing under the load, resembling the behaviour of the potential users of the system. When considering the serverless solution, the impact could be visible by increased number of cold starts, when the traffic pattern is changing or it is occasional. However, the solution using the simple server hosted in the container could note the performance decline similarly, when many subsequent requests would be submitted to a single worker instance or when the cluster of workers would need to scale to meet the demand.

4.4.3.6 Cost of the designed solution

Another, frequently mentioned factor when deciding on the architecture choice is the maintenance and execution cost. The serverless architecture makes it more difficult to precisely measure the cost of the processing, due to the number of components and the variety of granularity in pricing model for used services. However, the cost can be quite accurately estimated, when the traffic pattern is known along with the execution cost of particular workflow.

The section calculates the cost of a single execution of the entire processing flow for Exercise 1, considering two configurations described previously:

- The parallel presentation processing describes in section 4.4.3.3.
- The first configuration described in section 4.4.3.4, in which the presentation processing is divided into two branches — first with initial slides and the second with remaining data for the slideshow completeness.

The cost of services used in the implementation is summarised in Table 4.4, based on the pricing defined for region Europe (Frankfurt) and excluding the Free Tier provided by the AWS. Cost of some of the services, such as Amazon CloudWatch for storing logs, API Gateway for the incoming traffic in form of the user requests or data transfer for DynamoDB is omitted due to negligible use and very small granularity to notice the difference in the values of cost statement.

Service	Billed service usage	Pricing
API Gateway	REST API - API Calls	$3.70 (per million)
	WebSocket API - Connection Minutes	$0.285 per million connection minutes
	WebSocket API - Message Transfers	$1.14 (per million)
Data Transfer	From Amazon S3 To Internet	$0.09 per GB
DynamoDB	Read request	$0.305 per million read request units
	Write request	$1.525 per million write request units
AWS Lambda	Duration	$0.0000166667 for every GB-second
	Requests	$0.20 per 1M requests
Polly	Standard voices	$4.00 per 1 million characters
Simple Storage Service	PUT, COPY, POST, LIST requests	$0.0054 (per 1,000 requests)
	GET, SELECT, and all other request	$0.00043 (per 1,000 requests)
Step Function	State transtition	$0.025 per 1,000 state transitions

Table 4.4: Pricing summary for services used in the implementations

The usage of the services for both implementations and their cost is summarised in Table 4.5 and 4.6.

Service usage	Usage	Cost [$]
REST API - API Calls	2 requests	0.0000074
WebSocket API - Connection Minutes	2 minutes	0.00000057
WebSocket API - Message Transfers	4 messages	0.00000456
Data Transfer From Amazon S3 To Internet	0.003 GB	0.00027
DynamoDB Read request	1 Read Request Unit	0.0000000305
DynamoDB Write request	2 Write Request Unit	0.0000000305
AWS Lambda - Duration	61.057 GB-second	0.00102
AWS Lambda - Requests	22 requests	0.0000044
Amazon Polly - Standard voices	1323 characters	0.00529
S3 PUT, COPY, POST, LIST requests	86 requests	0.000464
S3 GET, SELECT, and all other request	108 requests	0.0000464
Step Function - state transitions	24 transitions	0.0006
	Total	0.00771

Table 4.5: Cost summary for the approach using parallel presentation processing

Service usage	Usage	Cost [$]
REST API - API Calls	2 requests	0.0000074
WebSocket API - Connection Minutes	4 minutes	0.00000114
WebSocket API - Message Transfers	5 messages	0.0000057
Data Transfer From Amazon S3 To Internet	0.003 GB	0.00027
DynamoDB Read request	1,5 Read Request Unit	0.0000000456
DynamoDB Write request	2 Write Request Unit	0.0000000305
AWS Lambda - Duration	11.649 GB-second	0.000194
AWS Lambda - Requests	12 requests	0.0000024
Amazon Polly - Standard voices	1323 characters	0.00529
S3 PUT, COPY, POST, LIST requests	86 requests	0.000464
S3 GET, SELECT, and all other request	98 requests	0.0000421
Step Function - state transitions	17 transitions	0.000425
	Total	0.00671

Table 4.6: Cost summary for the approach delivering the updates to the client in separate batches

Takeaways of the cost summary of designed solutions:

- **AWS Lambda processing is a small fraction of overall cost** — Cost summaries for both of the presented solutions indicate that the text-to-voice transcoding service takes a significant amount of the overall cost. The usage of the AWS Lambda processing is greater for the solution using parallel processing. Even though some part of the presentation processing is repeated in the serverless functions executed in parallel, the overall cost of the AWS Lambda related services barely exceeds 0.001$ for the analysed example per single workflow execution. Moreover, it is crucial to consider the cost of other services incorporated in the processing as well as the data transfer. For example, the summary of the second configuration indicates that the cost of uploading results to Amazon S3 and transferring them to the client exceeded the cost of the processing of the AWS Lambda.

- **Serverless pay-per-use billing model** — Considering the Exercise 2, which is a more demanding task in terms of the processing and the summaries listed above, the overall cost of the processing for a single presentation could be estimated as not exceeding 0.01$ per execution. Moreover, when taking into account the benefits of the serverless architecture, such as autoscaling with granular billing model based on usage and without paying for idle as well as the fact that the infrastructure management is handled by the cloud provider, the serverless solution with pay-per-use pricing can be an appealing alternative to more traditional architectures.

4.5 Data Tier

The perspective of data tier in the web application field is discussed in more detail in section 3.4.2.1. Despite the fact that the relational database management systems established their position over the years, the scale of modern web application showed that the relational model reaches some limitation in terms of scaling and maintaining at the same time satisfactory level of availability. NoSQL databases emerged as a response to the need of the Internet scale applications, handling scaling and the increased data volumes more efficiently as well as offering better performance and availability, while sacrificing other capabilities such as operations consistency and query flexibility.

The developer's task is to ensure that the architected web application will meet defined requirements and scale effectively. Selecting the appropriate database is one of the design choices for newly developed applications. With the growth of popularity of microservice architecture patterns, presented in section 3.4.1.1, when the web applications are broken down into smaller services, each of them should manage its own data. It allows developers to make a decision about choosing the suitable database for the service independently from the rest of the system, providing a favourable data model, which fulfills the assumed requirements [Vog18]. The serverless microservices can leverage such approach as well, drawing on the rich offer of the cloud providers to select the database which will fulfill the requirements as well as ensure high-performance and scalability, while at the same time handing off the need to manage the datastore to the cloud provider.

The offerings of cloud providers are presented in section 2.5, including the database components suitable for the serverless workloads. Nevertheless, the portfolio of the cloud providers is much broader, covering numerous database categories and their diverse data models, which are suitable for various use cases in the web application field. The extended list of the available database components provided by AWS is presented according to the current offering below [Ama21k].

- Amazon Relational Database Service (RDS) — relational database, with several database instance types optimized for memory, performance or I/O operations, supporting six familiar database engines, including Amazon Aurora, PostgreSQL, MySQL, MariaDB, Oracle Database and SQL Server
- Amazon Aurora — relational database, compatible with MySQL and PostgreSQL, fully managed by Amazon RDS, including on-demand and autoscaling configuration suitable for serverless workloads
- Amazon Redshift — relational database, designed for data warehousing
- Amazon DynamoDB — key-value database, supporting wide column structure for attributes, which can be simple types as well as more complex documents, heavily used for the serverless workloads with autoscaling and granular billing model, based on number of operations and data transfer
- Amazon ElastiCache — in-memory, key-value datastore with sub-millisecond latency, compatible with Redis and Memcached
- Amazon DocumentDB — MongoDB-compatible, document database service

- Amazon Keyspaces — wide-column, managed Cassandra-compatible database
- Amazon Neptune — graph database
- Amazon Teamstream — time series database
- Amazon QLDB — managed ledger database, providing immutable and cryptographically verifiable transaction log
- Amazon Elasticsearch Service - search engine based on Elasticsearch service

Most of the mentioned components can be also incorporated in the web application developed in the serverless paradigm, providing desired capabilities and fulfilling the requirements of various services. For example the web application serving a role of bookstore can use DynamoDB to store information about books, using simple lookups for known keys identifying the books. When the data about available products is updated, the DynamoDB Streams can replicate the data to Amazon Elasticsearch Service, providing the fulltext search capabilities. The leaderboard, including most frequently purchased items, can be implemented using Redis, hosted on Amazon ElastiCache, and its sorted set data structure. Lastly, the graph database in the form of Amazon Neptune can be utilised to provide a recommendation engine, when traversing the graph of connections for the purchased books [BI18].

However, the serverless paradigm heavily relies on the ephemeral compute in the form of Function as a Service, which leads to some complications when used with the long-lived, TCP-based connection model of traditional databases. Moreover, most of the databases do not fulfill the serverless assumption about autoscaling capabilities to meet the demand as well as granular per-per-use billing model, requiring instead to scale and configure databases on a per instance basis, billed accordingly.

4.5.1 Issues when integrating traditional databases with serverless solutions

Most of the traditional relational database management systems are not designed to work effectively with the ephemeral compute represented by Function as a Service. Such databases are more suitable to work with long-running compute instances, initialising the TCP-based connection pool and reusing the connections across the operations performed by the application. The ephemeral nature of serverless functions along its scaling capabilities, makes it not applicable to maintain the connection pool on the application level, because each of the functions is logically and physically separated from others. It leads to establishing a large number of active connections, because each executed function needs to connect to the database separately, bringing additional overhead when initially establishing the connection. Once the function execution is completed, the connection is not closed to make it available when the function container will be reused, but depending on load it may not be utilised [Deb19].

There are several approaches to alleviate the consequences of such unsuitability [Dal20b], which are listed and discussed briefly below:

- **Adjust database connection configuration** — Some of the database configuration parameters can be adjusted to affect the communication between serverless function and the database. The maximum number of available connections can be increased, however it is not a recommended solution, which can lead to dangerous spikes in database memory and CPU utilisation. Moreover, lowering the connection timeout can be used to drop the established connections more eagerly and make them available sooner.

- **Adjust function configuration** — The Lambda function concurrency can be limited, leading to reduction in number of requests to the downstream services, however when the lambda is configured to be executed in a synchronous manner, it will throttle the execution and return error to the upstream clients, which may be not convenient for them. The Lambda function code can handle establishing the connection and closing it after the execution. Nevertheless, such a solution has some downsides, including additional overhead related with establishing and closing the connection for the function as well as the database. In that case, when leveraging the warm starts, the database connection would not be reused.
- **Implementing good caching strategies** — The database service should be designed to scale by itself, however some solution could be to utilise caching services as write-through cache to mitigate the connection issues, such as Amazon ElastiCache. When the serverless function is executed, it could make a request to the cache to obtain the data and in case of cache miss, it could establish the connection to the database. When the data from the database would be retrieved, it could be stored in the cache for further function executions. It is essential to ensure proper cache invalidation and use Time To Live (TTL) parameter to prevent the data from being stale.
- **Buffering events for throttling requests and durability** — Heavy write workloads that do not need immediate feedback, can be effectively redesigned to process the requests in an asynchronous manner. The SQS queue can be used to buffer the events along with the lambda function configured with limited concurrency, consuming the events in a steady manner not to overwhelm the throughput of the downstream database. Similar approach is described in more detail in section 4.4.2.1.2. Moreover, the solution adds another level of durability and reliability when the database would have troubles processing the operations. The queue can buffer the events or return them to the associated DLQ when the processing fails.
- **Utilising proxy services** — AWS provides Amazon RDS Proxy, which is a managed database proxy for database instances running on Amazon RDS. Its responsibility is to manage the warm connection pool on behalf of the application and sharing the connections between the clients, alleviating the problem of the increase of opened database connections. RDS Proxy can handle the connectivity details of the downstream database, simplifying the logic related with connectivity management in Lambda functions, while multiplexing the function requests within the shared pool of active connections [Mao19].
- **Managing connections programmatically** — Managing the database connection programmatically is not a trivial task, however the community provides libraries,

helping with handling and managing the database connections in the serverless environment. The libraries utilise the database metadata to count the number of open connections, close them if their number exceeds predefined threshold by selecting connections with highest duration as well as provide the improved mechanism to establish the connection with the database, retrying it with exponential backoff in case of errors.

The connection limit is not the only problem with traditional database management systems when integrating them with serverless technologies. Preferably the databases are protected by strict firewall rules, restricting the clients which can access them, which is also problematic for the ephemeral compute services, bringing additional latency when initialising the communication.

The instance-based database solutions are much harder to scale, especially when considering the autoscaling nature of the serverless architecture. Most of the time, the instances are not able to scale up and down according to the demand and require to be prepared for the peak traffic, which leads to overprovisioning. Such an approach is not cost efficient, compared to characteristics of the serverless components.

Lastly, the provisioning and configuring the more traditional databases requires additional configuration on the database level, including for example additional users and roles definitions. With the serverless paradigm utilising heavily the Infrastructure as a Code approach for defining and configuring the resources, such additional database setup is harder to coordinate properly and leads to spreading the configuration across multiple tools [Deb17].

4.5.2 Serverless databases

Thanks to technology evolution, including faster network and better serialization protocols, the distributed, API-driven architectures using managed services can work more effectively. It contributes to building services, serving a role of interceptors between the databases and its clients. Some databases provide such interfaces, which makes them more suitable for the serverless workloads. The idea of serverless databases describes the datastores, which can be integrated effectively with other serverless components as well as provide additional desired capabilities, which are fitting the serverless paradigm [Deb17].

The list of features of the serverless databases is listed below:

- **HTTP-based communication** — Most frequently the serverless databases introduce other components, serving a role of proxy between the database and providing the communication interface for the clients. It alleviates the issues related with networking and managing connections to the database as well as integrates well with other services of the serverless platform [Deb21]. Moreover, some of the serverless databases such as Firebase, mentioned in section 2.4.1.1, allow web and mobile clients to communicate directly with the database and access or modify the data as well as to receive updates in real time, based on the data changes.
- **Autoscaling with no infrastructure management** — With the growth of the data volumes or the requirements for processing operations, the serverless database

should scale automatically to meet the demand, similarly to other serverless components. It also frees from estimating the proper instance size along with overprovisioning resources to meet the peak traffic [O'D19].

- **Pricing model based on usage** — Along with the autoscaling capabilities, it is desired for the database to have the granular pay-per-use billing model, according to the volume of stored data and number of operations [O'D19].
- **Security model based on IAM** — When using the HTTP based communication, the integration between the database and the serverless components making the request should be secured by the Identity and Access Management (IAM), compared to the traditional databases which provide secrets required to establish the connection directly. Such approach fits the security model of the application built in the serverless architecture, which relies on IAM rules, granting granular permissions to the various components interacting with each other and their operations [Deb21].
- **Provisioning and configuring based on IaC** — Instead of handling the additional configuration after the database is provisioned, the IaC tools should be responsible for defining the database configuration in a declarative way, similarly to other serverless components [Deb21].
- **Stream based activity log** — The feature of capturing the sequence of modified, created or removed items from the database table is a desired capability, allowing developers to incorporate it in the event-driven architecture of the serverless application and react to data changes accordingly [Deb18a]. The event-driven flow of data modification can be used to trigger various workloads, for example triggering the serverless function, responsible for notifying the web client, similarly to updates in the receipt processing application covered in section 4.3.1.

Some of the databases from the AWS platform satisfy the capabilities mentioned above, which makes them suitable solutions for serverless workloads. The characteristics of Amazon DynamoDB and Amazon Aurora are discussed in the section below.

4.5.2.1 Amazon DynamoDB

DynamoDB [Ama21h] is a fully managed NoSQL key-value and document database, suitable for workloads with high throughput, which require predictable performance. It provides autoscaling capabilities based on defined read and write capacity as well as allowing to configure on-demand scaling, that tracks the load and scales the capacity accordingly. DynamoDB stores 3 copies of the data in separate Availability Zones in a region, providing greater resiliency and fault-tolerance.

The tables in DynamoDB consist of items, which are built of attributes without predefined schema, enabling a flexible data model. Each item is uniquely identified by Primary Key, which is used to assign the item to one of the partitions after calculating a hash of it, splitting the table into smaller chunks of data handled by different nodes, which enables the database to scale and ensure predictable performance. The Primary Key can consist of a single attribute called Partition Key or combine both Partition Key and Sort Key to provide richer query capabilities. The attributes can include simple values as well as more complex structures, allowing developers to store whole documents.

DynamoDB provides lookups based on the defined Primary Key and range queries on the Sort Key if it is defined as well as projecting items to limit the number of returned attributes. The solution allows developers to configure two types of indexes — Local and Global Secondary Index, which provides the capability to define other Partition Key and Sort Key for the table, effectively duplicating the it and propagating the updates between the connected tables, which is essential to effectively query more complex data schemas.

It is billed on a pay-per-use model, based on the number of operations and used storage, including additional cost for the additional features mentioned below.

4.5.2.1.1 Additional DynamoDB capabilities

The database can be deployed in multi-region and multi-master configuration when enabling the global table feature. By default it uses the eventually consistent reads, however the query can be configured to obtain the data with strong consistency, by reading from the leader partition. Moreover, the database allows to perform queries and modification for multiple tables in ACID transactions, under some limitations in terms of number of items and amount of data being processed. To introduce an additional in-memory write through caching layer for DynamoDB, the DynamoDB Accelerator (DAX) can be configured, distributing the requests across nodes in the cluster which cache the data and reducing the response time for read intensive applications. To integrate with the event-driven serverless architecture, the DynamoDB Streams allows to trigger the Lambda function when the data is modified and react to these changes as well as integrating with other AWS services [Ama21e].

4.5.2.1.2 DynamoDB suitability for serverless based workloads

DynamoDB is suitable for serverless based workloads, ensuring consistent performance by design, even when running with large datasets as well as providing integrations with other services such as AWS Lambda or AWS AppSync. It shows the power of NoSQL databases in terms of the scalability and performance aspects, however it comes with a cost, affecting the data model and query capabilities. DynamoDB does not allow to perform joins on data tables and force to segment the data, which allows easier, horizontal scaling as well as bounds the queries by limiting the results of data retrieved with pagination capabilities, which enables to ensure predictable performance. On the other hand, it requires developers to think about the data model up front, by analysing the access patterns which are required to design the data accordingly, most frequently in a denormalized way. It requires a shift in the mental model from the relational modeling to approaches such as single-table design to store multiple, heterogeneous item types in one table to query and retrieve them with prejoined data in one request [Deb18b]. Moreover, the data duplication requires to to propagate the changes across the tables to maintain the consistent data model.

Both of the example implementations covered in section 4.3 use the DynamoDB for storing simple data models, which effectively suits the key-value lookup access pattern.

The web application performing the receipt processing, defines the Partition Key based on the username to effectively partition the items and Sort Key as insert date to order the results, which makes it convenient to query all of them for a particular user. Moreover, the DynamoDB Stream is used to trigger the Lambda function, responsible for notifying the user about the processing results, utilising the GraphQL subscription.

4.5.2.2 Amazon Aurora

Amazon Aurora [Ama21c] is a fully managed, relational database compatible with MySQL and PostgreSQL, which is designed to work in the cloud based environment. Amazon RDS is responsible for managing the database, including administrative tasks such as hardware provisioning, database setup as well as patching and performing required backups.

Aurora's architecture preserves the core features of the relational databases, however it separates the instances running the MySQL and PostgreSQL engines from the underlying storage layer, which is restructured to use a distributed storage system spread across multiple Availability Zones. The operations are replicated to 6 nodes in 3 Availability Zones, using quorum to ensure the writes consistency. Transactions are considered as committed when 4 from 6 nodes confirm the operations, while reads reacquire response from 3 nodes. It allows further operability of the database, even when one of the Availability Zones would be unreachable.

The database can be scaled vertically by allocating machines with more resources as well as horizontally by adding up to 15 read replicas. In that case, the cluster endpoint is pointing to the master node, which is responsible for writing to the distributed storage, while the reader endpoint handles the requests, balancing them across the reader nodes. The storage is allocated and scaled automatically, according to the volumes of data in 10 GB increments. Aurora can be also configured to utilise the global database, replicating the data across the regions. Amazon Aurora is billed using the Aurora Capacity Units, which is equivalent to 2 GB of memory with CPU and network bandwidth allocated proportionally, on a per second basis while the instance is active.

4.5.2.2.1 Aurora Serverless

Amazon Aurora also provides the serverless configuration, which allows an on-demand autoscaling configuration based on the application workloads, removing the need to plan the capacity. It can automatically start up, scale according to demand matching the application usage and shut down when not in use. It is designed to help with scaling and cost effectiveness for the applications with infrequent or variable workloads. Nevertheless, the first version of the service is limited in terms of additional features, which are available for the Amazon Aurora with standard configuration, including among the others configuring read replicas, cloning the database, loading the data from S3 bucket as well as invoking the Lambda functions based on the database functions [Ama21d]. However, AWS works on the second version of the database, which grants better and more granular scaling along with some of the additional features of the standard configuration.

4.5.2.2.2 Data API for Aurora Serverless

Data API for Aurora Serverless allows communication with the Aurora Serverless cluster using the HTTP endpoint instead of managing the connections, which is more suitable for accessing the database for the serverless functions. It alleviates the need to configure the services to run within Virtual Private Cloud (VPC), which is required when communicating securely with the Aurora cluster, when using the connection based approach. Moreover, it enables to integrate the database securely with other services using the IAM permissions [Ama21o].

The Aurora Serverless configuration, including its autoscaling capabilities with pay-per-use model, based on the size of provisioned resources and time they are active, is a viable solution when it comes to selecting a database suitable for the serverless workloads. Moreover, the introduction of Data API alleviates the need to establish and manage connections as well as allows use of the IAM based permission model. Currently, Amazon Aurora is under ongoing development, including the second version of the service in the preview mode, which includes additional improvements.

4.6 Client Tier

Building the web applications using the serverless architecture has a significant impact on the architecture of the compute and storage layer. The use of ephemeral and short-lived functions executed in an event-driven manner affects the client-server communication, creating the need to use dedicated components, serving the role of publicly available interfaces of a web application. Embracing the notion of asynchronous communication for more sophisticated compute workloads, introduces additional challenges for the interactive web clients, which are required to work beyond the request-response cycle and receive the updates from the application backend in real-time, once the underlying data is changed or when the operation is completed. Moreover, the cloud platforms introduce various ways to host the web application clients as well as allow them to incorporate additional client libraries to cooperate with the services hosted in the cloud more tightly. The section below includes an overview of all of the mentioned topics discussed in more detail.

4.6.1 Serverless API services

As mentioned in section 4.4.2.1.1, the architecture of web based services requires dedicated components, serving a role of public interfaces, responsible for exposing an API, which will be consumed by the web application clients as well as forwarding requests to the Lambda function or other hosted services. Amazon API Gateway and AWS AppSync belong to the most frequently used services used along other serverless components, introducing numerous capabilities which can be integrated when developing the web applications in the serverless architecture. Both services and their features are covered in more detail below.

4.6.1.1 Amazon API Gateway

Amazon API Gateway [Ama21a] is a fully managed service, enabling developers to create APIs, serving the role of public web application interfaces as well as integrate directly with other components of the cloud platform and developed backend services. It handles tasks related with maintaining, monitoring and securing APIs, integrating them with other services, providing authentication and authorization capabilities, traffic management and throttling and versioning management.

The API endpoint can be configured as one of the mentioned type, depending on the traffic origin:

- Regional — recommended for general usage and built for clients in the same region
- Edge-optimized — utilising Amazon CloudFront distribution, designed for globally distributed set of clients
- Private — accessible from within Virtual Private Cloud (VPC) and designed for internal APIs and private microservices

Amazon API Gateway supports three types of APIs, including HTTP API, REST API as well as WebSocket API, which are described in more detail below:

- REST API — Defines a mapping between REST resources according to appropriate HTTP methods to integrate with Lambda functions, AWS services or other HTTP endpoints. REST APIs allow developers to use additional capabilities, such as defining usage plans with API keys, caching responses, transforming requests as well as providing validation for the data being sent and received.
- HTTP API — Suitable for defining mapping for routes and HTTP methods, forwarding the requests to configured services of the cloud platform. HTTP APIs are more lightweight and designed to proxy the requests to the downstream services, introducing less network overhead and being considerably cheaper than the REST APIs, however the number of additional capabilities related with operating on the requests is limited.
- WebSocket API — Enables developers to define real-time, full-duplex communication between clients and servers, while maintaining and persisting the connection state as well as handling the data transfer between web clients and backend services.

API Gateway can be configured to provide authentication and authorization capabilities based on IAM policies and AWS credentials, when integrating with Amazon Cognito Pool or using Lambda authorizers to validate the tokens or request parameters, granting access to the backend services [Joh19]. Moreover, it can be integrated with AWS Web Application Firewall (WAF) to secure the API from common vulnerabilities, blocking attacks from specified IP ranges, matching particular requests based on the content or user agent. Proper logging, monitoring and usage metrics are granted by Amazon CloudWatch, based on the client interactions. The API definition can be described in OpenAPI format, including custom integration which allows to configure the routes with AWS services, and used further by the Infrastructure as a Code tools to deploy the API Gateway configuration [Ama21b].

API Gateway is billed based on the number of API calls and transferred data as well as the connection duration when using the WebSocket APIs.

The second example implementation, covered in section 4.3.2, uses API Gateway to provide simple REST API, integrating directly with AWS Step Function to start the presentation generation workflow and transform the response from the service to obtain the execution identifier. It is later used when connecting via the WebSocket API to subscribe for notification, once the particular processing flow is completed and the slideshow is available for the user preview.

4.6.1.2 AWS AppSync

AWS AppSync [Ama21f] is a fully managed GraphQL service, responsible for processing requests and mapping their different parts to appropriate resolvers, which query the data from multiple data sources, including databases, microservices or other HTTP endpoints, and aggregate it in one response. The GraphQL schema allows web clients to define strongly-typed queries, granting greater elasticity by defining desired fields, preventing from overfetching and multiple round-trips to the server.

Resolvers can integrate directly with other components hosted on the cloud platform, including DynamoDB, Aurora Serverless, Elasticsearch Service as well as use HTTP requests to invoke other services, such as AWS Step Function, or bridge with regional API Gateway endpoints, forming a facade to AWS services. The resolvers can use mappings, defined in Apache Velocity Language, when requesting the data and receiving the response, forming simple unit resolvers. Unit resolvers can be composed into pipeline resolvers, performing a more advanced sequence of operations as well as using Lambda resolvers for processing more complex application logic.

The GraphQL subscriptions allow clients to receive updates in real-time. AppSync manages the WebSocket connections, scaling according to usage and delivering the messages to subscribing clients. The subscription can be triggered based on the client operation as well as one of the cloud platform services, broadcasting the update to subscribing clients, which can additionally filter the message based on its content.

AppSync provides additional authentication and authorization capabilities when performing operations or querying the data. It can use API keys, IAM permissions, integrate with Cognito User Pool to manage users and groups as well as incorporate any identity provider, which is compliant with OpenID Connect protocol.

Moreover, results of resolver processing can be cached, based on defined caching policy including caching keys and authenticated user parameters. The feature is based on the ElastiCache instances, serving the role of cache with predefined Time To Live (TTL) values, reducing the compute cost for slowly changing data.

AppSync integrates with other services to bring additional capabilities. AWS Web Application Firewall (WAF) can be configured to enable additional protection, Amazon CloudWatch gathers logs and metrics, while AWS X-Ray grants greater observability for the processed requests [PHM19].

The AppSync can be integrated with the web and mobile applications, using the Amplify DataStore to preserve the data on the client devices and make it available for working offline, synchronising the state when the device will be reconnected to the network [Ama21g].

AppSync pricing is based on data transfer and the number of performed queries, mutations and subscriptions, with additional billing per time when the clients use WebSocket connections as well as depending on the caching configuration.

The first example implementation performing the receipt processing, described in section 4.3.1, uses AppSync to provide GraphQL API for the client application. It uses Lambda functions to resolve some of the queries along with communicating directly with other services to retrieve the data. Moreover, the communication is enhanced by the real-time subscriptions, which allow applications to update the clients, once the asynchronous processing flow is completed and the state in the database is updated. The integration with Amazon Cognito allows the application to handle user identities and ensure the returned data belongs to the particular user. The DynamoDB Stream triggers the Lambda function, which performs the appropriate mutation, resulting in sending an update to the particular client, selecting subscription based on the identity.

4.6.2 Client communication patterns for serverless architecture

Depending on the application requirements and the workflow types it incorporates, the communication with the client can range from simple, synchronous requests with one of the back-end services, to more complex setup, which requires additional coordination of the asynchronous application logic from the client side. Section 4.6.1 covers the components frequently used with the serverless architecture and their capabilities. The section below describes in more detail, some of the use cases of the discussed components along with the suggested communication patterns used in the web application implementations utilising the serverless architecture.

4.6.2.1 Synchronous requests

For simple use cases the synchronous communication with the serverless application backend is applicable, as mentioned in section 4.4.2.1.1. Figure 4.18 shows, how the API Gateway can be applied to integrate directly with DynamoDB to retrieve the data along with AWS Step Function to trigger some more advanced workflows in an asynchronous manner, returning with the response to the client to confirm the invocation. Similarly, the API Gateway can invoke the Lambda function in a synchronous manner, as covered in section 4.4.1.3, to process some more complex business logic. It is essential to be aware that the more sophisticated workflows may be restricted by the time limits of the components serving a role of the public interfaces. In that case, the processing can be redesigned to use the asynchronous processing, while the periodic, synchronous requests can poll for the status or results of the processing.

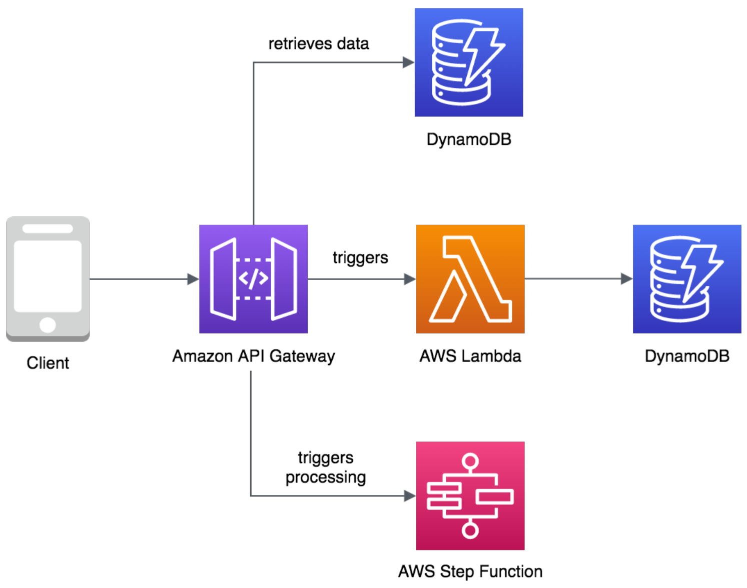

Figure 4.18: Example of synchronous communication with Lambda function, DynamoDB and AWS Step Function using API Gateway

In the application covering the receipt processing, covered in section 4.3.1, AWS AppSync resolves the queries by retrieving the data from DynamoDB as well as by triggering the Lambda function to obtain the presigned URL to S3 bucket. The second example, described in section 4.3.2, shows how the API Gateway can be used to trigger the asynchronous processing of the AWS Step Function, while returning the response with the invocation details immediately to the client.

4.6.2.2 GraphQL APIs

The AWS AppSync, which hosts the GraphQL API, can be effectively used to aggregate the data from multiple data sources in a single response. Simpler logic can be handled by resolvers written using Apache's Velocity Template Language and communicating directly with the data sources, including for example querying, inserting or modifying items in DynamoDB or communicating with Elasticsearch Service. For more complex logic, Lambda function can be configured as a datasource to perform the function code, when requesting a particular field from the GraphQL schema, returning the data back to the client. The request including communication with several datasources can be obtained when using pipeline resolvers, performing a chain action, involving a sequence of calls to multiple services and aggregating the responses. However, for more complex mutations, the AWS Step Function can be a better choice if it comes to handling errors and retries more effectively, compared to pipeline resolvers [Les19].

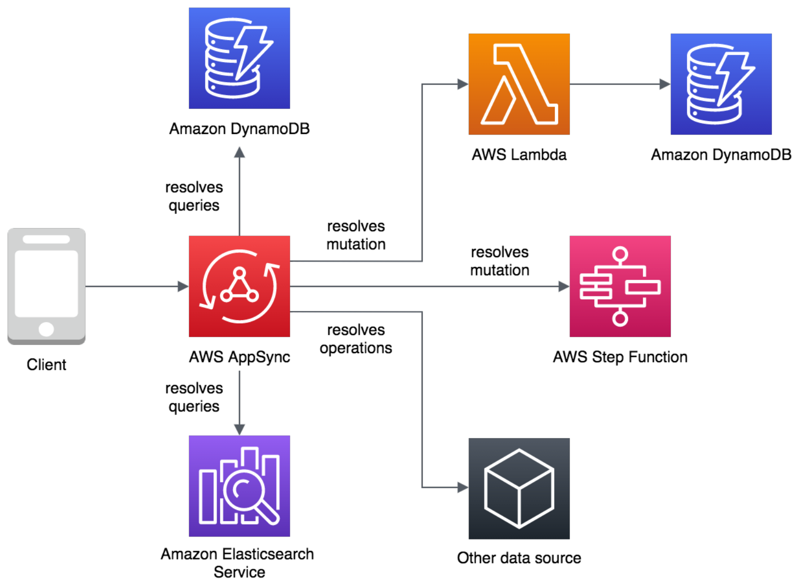

Figure 4.19: Example usage of AWS AppSync for hosting GraphQL API using various services as data sources

The GraphQL APIs are beneficial when working on larger applications, including several independent services to provide the API layer in one place, calling configured services accordingly. The underlying implementation of the services can be changed, as long as the API contract based on the GraphQL schema is maintained, enabling developers to evolve the application architecture effectively. Moreover, AWS AppSync handled and manages the WebSocket connection, allowing users to leverage the GraphQL subscriptions to receive the updates in real-time, based on the changes of the underlying data [PHM19].

The application responsible for receipt processing, covered in section 4.3.1, uses the

GraphQL API hosted using AWS AppSync to communicate with various services to perform queries and mutations as well as utilise subscriptions to notify the users when the asynchronous processing flow is finished.

4.6.2.3 Working with files securely — presigned URL

The limitations of the AWS Lambda invocation payload to 6 MB (when invoked synchronously, as mentioned in section 4.4.1.1) along with the API Gateway payload restriction to 10 MB, makes it unsuitable to upload the larger files using this serverless components. The solution for such a constraint is to upload the files directly from the client to Amazon S3 bucket. To perform this operation securely, having the control over accessing the objects in the bucket, the mechanism of presigned URLs can be used. The Lambda function can be configured to obtain the presigned URL for a given bucket, enabling clients to obtain the particular file or put it into under predefined path, making the URL additionally validated based on credentials or until specified expiration time is not exceeded [Itu19].

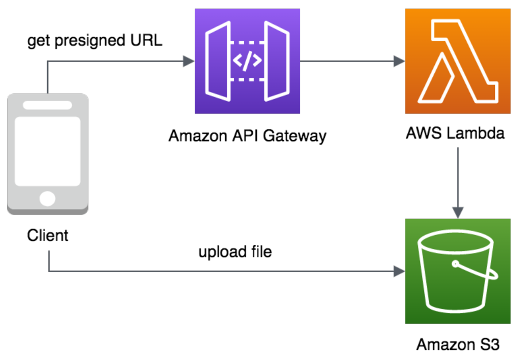

Figure 4.20: Example usage of presigned URL when uploading files to Amazon S3

4.6.2.4 Webhooks

For the asynchronous workflow invocations, the client needs to be notified when the operation finishes or the results of the processing are available. Webhook pattern can be applied when operating with clients, which are external services capable of exposing an endpoint to receive the callback, once the processing is completed. Example implementation of such a pattern is covered in Figure 4.21. The client calls the API Gateway and receives the confirmation of submitting the call as soon as the request is inserted and stored durably in SQS. Once the Lambda or some other service performs the processing, the message is sent to SNS which is responsible for performing the HTTP call to the endpoint, defined during the initial request. It removes the need for the client to poll for the processing status. Additionally, some mechanism can be incorporated to ensure the communication with the client is trusted by verifying API key or performing authentication in a different way [Itu19].

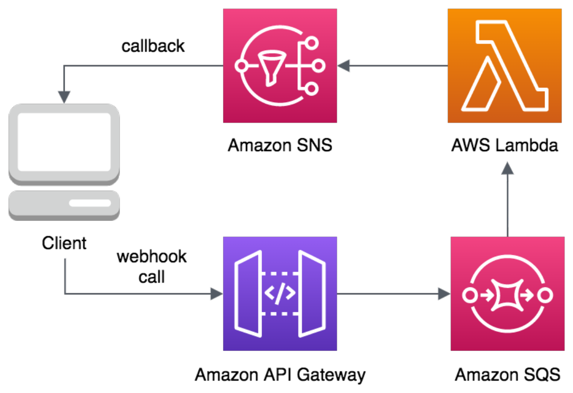

Figure 4.21: Example webhook implementation

4.6.2.5 Pushing updates to clients using WebSockets

Nevertheless, not all clients have a possibility to expose an endpoint and receive a callback once the asynchronous processing is completed. For the web clients the WebSocket communication can be established to receive the notification in real-time, containing the state of the processing. Figure 4.22 shows how the update can be pushed to the client, when the asynchronous workflow of an AWS Step Function is completed. The example depicted on the diagram uses AWS API Gateway with two types of API — REST and WebSocket API. Clients use the REST API to invoke the AWS Step Function and obtain in response the execution identifier or some other token. It is later used to connect to the WebSocket API, calling one of the tasks involved in the orchestration workflow. When the client is connected as well as the processing is finished, the Step Function invokes the last step of the processing, responsible for sending the message to the client, notifying that the asynchronous processing is completed [Itu19].

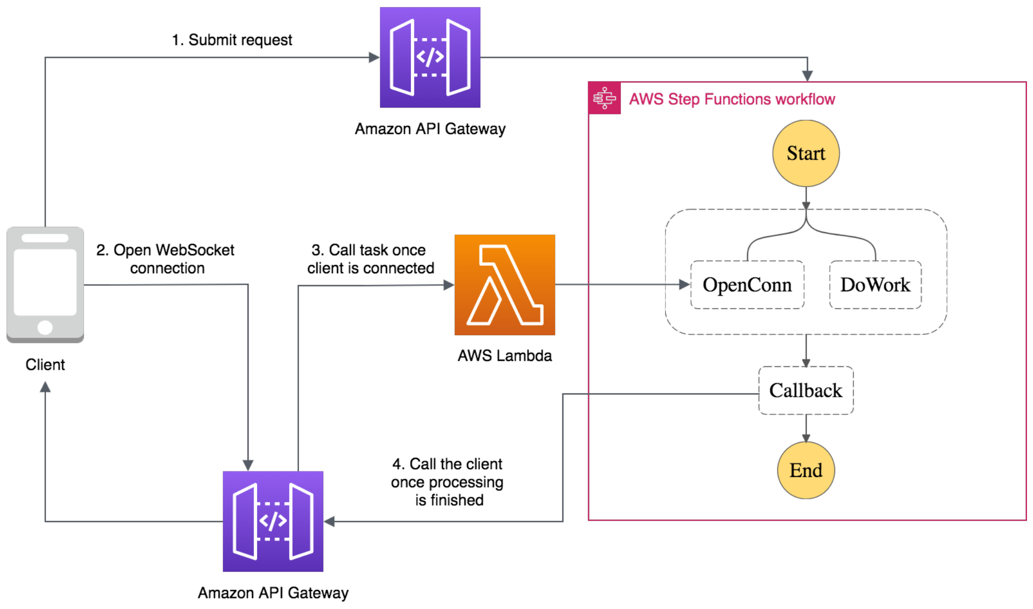
1. Submit request
Amazon API Gateway
AWS Step Functions workflow
Start
2. Open WebSocket connection
3. Call task once client is connected
OpenConn
DoWork
Client
AWS Lambda
Callback
4. Call the client once processing is finished
End
Amazon API Gateway

Figure 4.22: Example diagram showing Step Function invocation responding to client using WebSocket once processing is finished

Similar approach is used in the example of interactive presentation processing, covered in section 4.3.2. The provided implementation uses DynamoDB additionally to store the association between connected client identifier and the function execution identifier, reading the stored value for each of the chunks of parallel execution. The opened, bidirectional communication connection can be applied for richer eventing between the backend of the application and the web or mobile clients.

4.6.3 Hosting web clients

Amazon Web Services along with other cloud vendors provide various ways to host the web application clients. Depending on the type of web clients, which are covered in section 3.4.3, a different approach can be applied. Along with hosting static assets, Lambda function can be effectively utilised to render the client interfaces on the server side. Both approaches are discussed in more detail in sections below.

4.6.3.1 Hosting static assets

Majority of cloud vendors provide a possibility to host static files and distribute them to the end users, leveraging the worldwide datacenter network with additional locations serving the role of Content Delivery Network (CDN), hosting the assets from the locations closer to the users. AWS enables customers to host the files using Amazon S3 buckets, using additionally the Amazon CloudFront Distribution as a Content Delivery Network [Ama21p].

It provides necessary capabilities to host the Single Page Applications as well as the static websites. Single Page Applications, described in detail in section 3.4.3.1.1, are dynamically rendered in the web browsers and communicating with the web application backend in an asynchronous manner, providing rich and interactive user experience. During the build process, such applications are effectively processed into the resulting artifact, which is a set of static files, including mostly JavaScript scripts, but also stylesheets and other static assets. In terms of statically generated pages, covered in section 3.4.3.1.2, the results of the processing in form of the static files can be similarly hosted in the cloud environment, using automation to update the content of the S3 bucket once the generation process is completed.

An example diagram, describing how the Single Page Application can be hosted in the cloud environment along with communication with the backend services, is presented in Figure 4.23.

An example implementation of receipt processing application, described in section 4.3.1, uses this approach to host the Single Page Application in the Amazon S3 bucket, leveraging additionally the CloudFront Distribution as a CDN. The build process incorporate the Github Actions [Git21], serving the role of processes automation which configures the continuous deployment pipelines, during which the client application is built and deployed in the cloud environment, once the change is merged to the main branch of the repository in which the application code is stored.

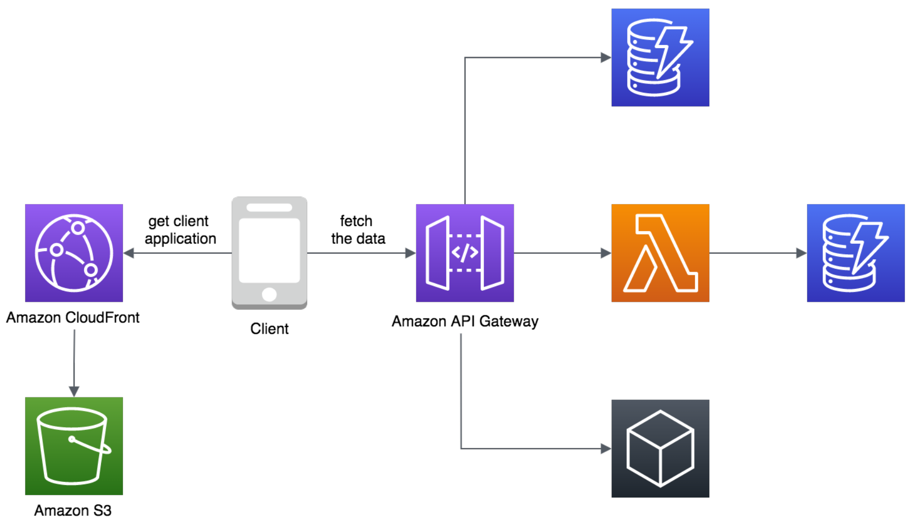

Figure 4.23: Example diagram presenting Single Page Application hosted in the cloud environment

4.6.3.2 Server-side rendering

Nevertheless, hosting the static sites is not the only possible solution for hosting web clients. Lambda function can be used to render the web pages on the server side, utilising the benefits of server-side rendering, discussed in section 3.4.3.1.3. When using the reliable connection and efficient processing, the web page can be rendered and sent back to the client, reducing the overhead required to load the whole Single Page Application as well as optimising the website for search engines.

An example implementation of the server-side rendering using the Lambda function is presented in Figure 4.24. Amazon CloudFront is configured to forward the requests to the API Gateway endpoint, which is invoking the Lambda function, responsible for rendering the web page on the server side. It can additionally call the underlying backend services to request the additional data, required to render content of the page. When the web page is generated, API Gateway returns the resulting file, utilising additionally CloudFront caching to optimize the client-server communication for the subsequent requests.

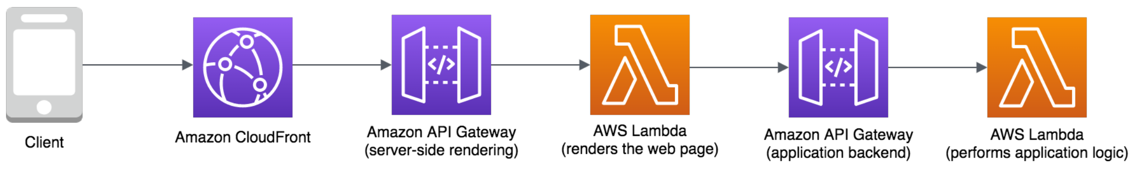
Client
Amazon CloudFront
Amazon API Gateway (server-side rendering)
AWS Lambda (renders the web page)
Amazon API Gateway (application backend)
AWS Lambda (performs application logic)

Figure 4.24: Example diagram presenting the server-side rendering of web pages using Lambda function

Moreover, the Lambda@Edge can be utilised to generate the static content at the edge locations closer to the users to reduce the communication latency [Bes21]. Diagram presenting the usage of Lambda@Edge is presented in Figure 4.25.

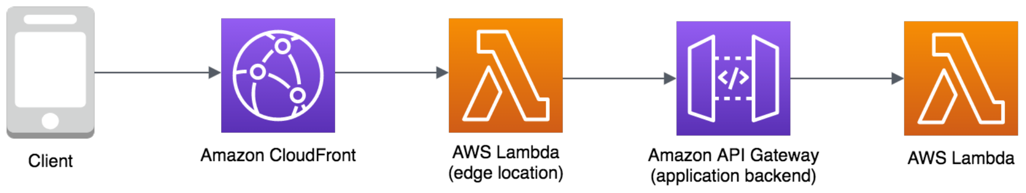

Figure 4.25: Example diagram presenting the server-side rendering of web pages using Lambda@Edge

4.6.4 Thick and thin client

The notion of thin clients refers to the application clients, which serve as a thin presentation layer, communicating with the underlying services, which handles the significant part of the application logic. Contrary to that, the richer and more interactive thick clients can incorporate some part of the business logic on the client side, making the back-end services a thinner layer, responsible for validating the performed actions as well as communicating with the database to preserve the changes durably.

At the first glance the architecture of web application clients should not be significantly affected by the changes of the underlying server and storage architecture to use the serverless architecture. However, when taking into consideration the possibility of direct communication with cloud-based services and changes in the workflows processing, gearing towards asynchronous pipelines, it makes the application clients include more complex logic, including additional integrations with serverless components to access them securely as well as moving some part of the orchestration logic to the client in terms of handling the communication with the serverless infrastructure.

The initial architecture of the CloudGuru learning platform is an example of such rich and thick client web application client, communicating directly with various third party services as well as smaller, cloud based services, as described in sections 2.6.1 and 4.2.2.1.3. The web application client integrates directly with the Firebase, querying for the data, performing updates as well as receiving the messages in real-time, when the underlying data model changes. Moreover, the integration with Auth0 as an identity provider, responsible for handling authentication and authorization, is performed on the client side, incorporating additional part of the logic into the web application client.

It reduces the role of the server application as a middleware between the client and datastores or other application modules, making the communication directly. The amount of code responsible for handling the application logic in the server layer is smaller, while some part of it is pushed to the clients, for orchestrating even some more complex workflows including communication with several components.

However, not all logic is suitable for processing on the client and some part of the logic, including working with the sensitive data or requiring additional validation from the security standpoint needs to be implemented outside of the client application logic, because the actions executed in the user's browser cannot be fully trusted for such operations. For example when finalising the transactions, granting users access to the course or when validating if the user has permissions to access the resources, the additional application logic incorporated in Lambda function is required to protect the underlying resources.

In that case the client application performs certain operations, orchestrating the whole flow, while communicating with the application logic incorporated in the Lambda functions, which validates the privileges. Similarly, when triggering emails to the groups of users, the processing needs to be performed in the cloud platform, because the emails may include sensitive user data such as emails or their personal data.

Such a shift is possible thanks to the desktop and mobile hardware improvements, making the devices more powerful to handle the tasks of feature-rich and interactive web application clients. Moreover, the wide range of third party services exposing various capabilities, including the identity management, protected operations on the files, notifications and real-time data streaming with the offline synchronisation, makes them an appealing solution to incorporate into the interactive and thick clients. The integration is possible thanks to numerous programming frameworks, enabling developers to write larger and more complex web application clients, incorporating additional libraries to integrate with services or communicating with cloud-based components, using the HTTP protocol with additional token to grant access to predefined resources in a secure manner [Kro15].

The serverless architecture encourages developers to build more interactive and thick clients, handling the direct communication with external services as well as performing some orchestration of the processing, reducing the amount of application logic, which needs to be handled in the serverless functions. However, some types of the processing needs to be executed in the cloud environment, especially when working with sensitive data, when the operation needs to be additionally validated against the application logic or external services, granting access or authorising the operation when appropriate credentials are provided. It is possible to build a thin client and incorporate most of the application logic in the server layer of the application, however it may be not suitable with the serverless paradigm, reaching various limitations of the serverless components as well as most frequently resulting with additional cost, when handling the logic and orchestrating the communication using the serverless platform.

Similar conclusions can be formed when considering the example implementations. The application responsible for receipt processing, covered in section 4.3.1, incorporates an additional library to integrate with Amazon Cognito, incorporating the authentication and authorization mechanism in the application. Moreover, it uses the presigned URL to communicate directly with the Amazon S3, to retrieve and upload the files directly, after the user is authenticated when querying for the URL. In terms of the application generating the interactive slideshows, described in section 4.3.2, it incorporates additional logic responsible for orchestrating the processing and obtaining the results of the generation process. Initially, it invokes the AWS Step Function to trigger the workflow, however later it needs to establish the WebSocket connection, receiving the updates and retrieving the assets from Amazon S3 once the processing of the serverless workflow is completed. In both cases, the web application client is enhanced with some additional logic, incorporating it with the cloud platform and adjusting the client application logic to overcome some limitations and make it more suitable for the serverless processing model.

Chapter 5

Summary

5.1 Summary of the research

The goal of the thesis was to research the use of serverless processing and the FaaS model in web application development and it has been described thoroughly during the research.

Firstly, the domain of serverless computing has been presented in more detail, including the origins of the paradigm, definition of serverless as well as covering the characteristics of serverless components, including Function as a Service to reason about the model in further research. Next, the benefits of serverless computing have been gathered to understand what capabilities makes it an interesting solution for customers along with the challenges, describing the limitations and problems related to the researched paradigm. The leading cloud providers have been identified, presenting the offering of their serverless platforms as well as the example use cases have been mentioned to build the context for further analysis.

Furthermore, the field of web applications has been described, introducing its origins, defining the web application and reasoning about the requirements they need to fulfill nowadays. The architectural patterns and problems related to the field have been covered in more detail, dividing the discussion into three categories considering the server and data layer as well as the web application clients.

The initial description of the serverless computing and web application fields introduced required knowledge for further research of the applicability of the serverless processing and FaaS model in web application development. Below there is a summary of the conclusions regarding the research questions, which have been introduced in section 4.1.1.

5.1.1 Suitability of the serverless paradigm in web application development

Serverless paradigm is an appealing solution for various companies, due to benefits it can bring, such as reducing operational and development cost, shortening time to market and removing the need to manage the infrastructure, however the applicability of the serverless paradigm is not fully mature yet and it is a subject of various practical research for numerous practitioners.

The instant autoscaling capabilities of the serverless components along with the proportional cost based on used resources makes it an appealing solution for the services characterized with little or variable traffic. The cloud-based, event-driven processing model introduces concerns regarding the performance of the critical workflows of the implemented solution, however cloud vendors actively work on alleviating the inconvenience of resource virtualisation, while architects provide reference architectures mitigating the limitations of the serverless platforms. It is essential to be aware of the limitations of the serverless components, especially the Function as a Service components, which are designed to effortlessly scale horizontally, however due to that require external components to preserve the state and communicate, which introduces additional overhead. The serverless functions are designed to perform CPU intensive tasks, nevertheless integrating them with other components, aggregating the events, and designing the workflows in an event-driven manner, which prevents functions to wait idle, makes the serverless architecture also suitable for high-throughput and network-bound tasks.

Moreover, it is crucial to consider if the tight integration with the cloud platform, which leads to vendor lock-in, is bearable. Otherwise, it could be seen as a trade-off, contrasted with the benefits that the serverless paradigm can bring.

When referring to the examples introduced in section 4.2.2.1, the serverless implementations of various web services led to appealing outcomes. Mentioned benefits include the development opportunities with significant cost reduction, fine-grained infrastructure scaling to meet the unpredictable load with proportional cost, easier evolution of the web application with increased team agility as well as decoupling the processing along with scaling according to the high-throughput processing task. When compared with more traditional architectures, including monolithic or microservices, the serverless architecture is more agile in terms of scaling, for the larger workloads the performance of the serverless solutions is comparable with the aforementioned approaches, additionally standing out with more stable performance characteristics. Nevertheless, the cost effectiveness of the serverless solution is still a debatable aspect, which require thorough cost estimation, covering all used serverless components and additional costs, including for example other services and data transfer. While at the same time, it removes the need to maintain the infrastructure and cover other operational aspects such as scaling of the solution.

5.1.1.1 Fulfilment of the web application requirements

The fulfilment of the web application requirements of the serverless architecture is covered in section 4.2.3.

In terms of performance, the serverless solutions raise various concerns, regarding the ”cold start” phenomenon and the overhead of communication overhead for workflow composition. However, both problems are under constant improvement from the side of cloud providers, providing dedicated services to alleviate the resource provisioning overhead as well as architects designing the solutions to process the workloads efficiently. The serverless solutions characterise with almost instant horizontal scaling capabilities according to the current workloads, making the solution scalable by design as well as providing satisfying performance of the system, comparable with more traditional architectures.

When it comes to reliability, the serverless solutions are configured to provide a failover mechanism, retrying the processing, which can be further extended with proper error handling and running across multiple data centers. Leveraging automation tests with proper monitoring and observability enables developers to ensure that the application is running properly, leveraging the tooling provided by the cloud platform, when utilising a similar approach when maintaining the services running in the microservice architecture.

The cloud providers introduce various guidelines in terms of security aspects, however, the field of serverless security is still under ongoing development, researching various vectors of attacks and educating customers to prevent them from the possible misconfiguration of the cloud-based solutions.

In terms of maintainability, the serverless paradigm reduces the need to work on various operational aspects along with new development opportunities, however it introduces a new approach of developing and testing the web applications in the cloud environment, which requires additional knowledge and experience. The cloud platforms provide various services, helping with maintainability, observability and deployment process as well as introduce numerous services and tooling that help with management of the serverless solution.

All of the mentioned features make the serverless architecture an appealing solution in the field of web application development. However, serverless technology requires a mind shift in terms of developing and managing the solution. Building a performant and effective serverless solution requires additional domain knowledge about the serverless architecture, significant amount of experience with the cloud-based services, along with the thorough cost estimation to decide whether the technology will be applicable to solved problems.

5.1.2 Application of the serverless computing and FaaS model in processing of the web application workloads

5.1.2.1 Function as a Service model

The Function as a Service model has been discussed based on AWS Lambda service, which is one of the most popular implementations of the serverless function. AWS Lambda is a serverless, stateless execution environment, scaling based on a number of requests, executed in an event-driven manner and constrained by several limitations in terms of resource allocation, function execution time as well as size of deployment package, invocation payload and temporary disk space. Some types of processing may be not suitable for the constrained execution environment of the Lambda function, however AWS introduces various improvements to overcome the limitations, increasing the amount of configurable resource allocation and connecting the function to shared disk space. Furthermore, the serverless function runtime can be extended beyond the set of supported programming languages, by introducing a possibility to incorporate custom runtimes based on Docker images and share dependencies using Lambda layers, which extends further the capabilities of the AWS Lambda in terms of processing various workloads. The knowledge about the function execution models, possible optimisations and configurations can have a significant impact on the performance and other characteristics of the serverless based workloads.

Mentioned capabilities make the AWS Lambda, which represents the FaaS model, a suitable and satisfying execution environment, not only for the web application workloads based on a simple request-response execution, but also for more complex and even non-standard tasks.

5.1.2.2 Serverless processing model

The serverless processing model, heavily relies on asynchronous and event-driven processing, valuing the loose coupling of its components, which makes it similar to microservices architecture. However, it pushes the limits further by splitting the service implementations into multiple functions and other serverless components, configured to work together within a single domain of application. The cloud platforms introduce various components, serving as a front door for the application, exposing numerous types of APIs, which can be integrated with other cloud-based services as well as provide additional capabilities to work with user requests and protect the downstream services. The internals of the serverless services use the serverless functions to execute the code responsible for the application logic. Asynchronous workflows are more suitable for the stateless and short-lived nature of the serverless functions, incorporating various components including services with publish-subscribe capabilities, queues and event buses to exchange the messages and coordinate the processing between subsequent function execution. The aforementioned components can be configured together to provide additional functionalities, covering the throttling and durable buffering of events, distributing the events reliably to multiple providers, aggregating the stream of data which is sent to several consumers and many more, depending on the requirements of the serverless workflows.

The serverless microservices bring similar benefits as the microservices architecture, enabling developers to split the application into a set of decoupled services with independent data stores, technology choices and deployments, adjusting the developed application to the organisation structure. The asynchronous and reliable communication is enabled thanks to the various serverless intermediaries, utilising the eventual consistency to propagate the changes in such a distributed system. Furthermore, leveraging automation, proper monitoring, observability and other patterns of microservice architecture can be applied to ensure a satisfying level of confidence, that the web application is running properly.

Services such as Amazon SNS, Amazon SQS and Amazon EventBridge find applicability in the choreography of the asynchronous, event-driven workflows, forming a loosely coupled chain of components responsible for processing the application logic. Nevertheless, some types of tasks require more thorough coordination of the processing. AWS Step Function is suitable to handle orchestration of more complex workloads, requiring reliable cooperation between various services, capturing the entire business flow, handling more complex branching logic, errors, retry policies and transactional logic across multiple services.

5.1.2.3 Example implementations

The serverless processing model has been analysed with a reference to two example implementations, covering the web application, gathering information about spending by processing receipts, and a service responsible for generating interactive presentations based on the LaTeX files, which are covered in section 4.3. Additionally, the detailed analysis of the second example is described in section 4.4.3, covering the development process, introduced optimisations along with the performance and cost analysis, presenting how the serverless paradigm can be applied to processing a non-standard task, related with the web application field.

5.1.3 Characteristics of the storage components used in the web applications built in the serverless architecture

The cloud platforms provide a wide range of available services, serving the role of databases and datastores, which characterise with different capabilities and data models suitable for various problems and requirements, that the web applications needs to fulfill. Most of the solutions can be effectively incorporated in the web application architectures built in the serverless paradigm, granting interesting features.

However, most of the traditional databases are working effectively with long-running instances, communicating using the worker pool and reusing the connections for performed operations. The ephemeral nature of the serverless function and its scaling capabilities makes it unsuitable to work effectively with the traditional databases. To mitigate the issues various techniques can be applied, such as adjusting the function configuration, implementing caching strategies, introducing additional buffering and throttling of requests along with using dedicated proxy services and managing the connections programmatically.

5.1.3.1 Serverless databases

Nevertheless, cloud providers introduce hosted databases, dedicated to work more effectively with the serverless workloads. Such services are characterised with communication based on the HTTP protocol, that alleviate the need to manage the connections, as well as autoscaling capabilities to meet the volumes of data, without the need to manage the infrastructure, with pricing model proportional to usage of stored data volumes and performed operations. Similarly to other serverless components, the security model is based on the IAM rules, along with the possibility to manage the database infrastructure using the Infrastructure as a Code approach, along with incorporating the stream based activity log which makes them suitable to incorporate in the serverless, event-driven workflows.

Amazon DynamoDB is an example of such a serverless database, providing all of the aforementioned, desired capabilities as well as integrating effectively with other serverless components. However, the NoSQL data model of DynamoDB may not be suitable for all types of workloads and requires additional effort, when modeling and evolving the business domain schema to ensure the data will be accessed effectively.

On the other hand, the Amazon Aurora with Aurora Serverless configuration seems to be an interesting alternative, providing a well-known relational data model. The service satisfies a significant subset of aforementioned requirements, however there are still some limitations when using it with the serverless workloads. Amazon Web Services works actively on the second version of the service, introducing various improvements and additional capabilities, which tends to establish the Aurora Serverless position alongside DynamoDB.

5.1.4 Client communication with the web applications utilising the serverless architecture

5.1.4.1 Client-server communication

Building web applications in the serverless technology introduces a need to use dedicated components, serving a role of public interfaces, handling the communication with clients and redirecting it to the internal services, responsible for performing the application logic. Services such as Amazon API Gateway and the AWS AppSync are frequently used with the serverless architecture to provide the entry point for the application, exposing different types of APIs including REST, WebSocket and GraphQL. The services can integrate with Lambda functions or communicate directly with the downstream services hosted in the cloud platform. Moreover, the mentioned components can be configured to use additional capabilities, covering authentication and authorization, transforming and validating requests as well as incorporating caching mechanics.

Simple use cases can be effectively implemented using synchronous communication, integrating the API Gateway directly with the services or serverless function to perform the application logic. On the other hand, more complex web applications can benefit from using the GraphQL APIs to aggregate the response from multiple services, trigger more complex workflows executed asynchronously as well as receive the updates in real-time, thanks to the established subscriptions. When working with files, web clients can access the services responsible for storing them directly. Lastly, it is beneficial for the more complex, serverless workflows to execute them in an asynchronous manner. Using patterns such as Webhooks or leveraging WebSocket connection can be used by clients to listen for updates once the asynchronous and event-driven processing flow is completed, providing satisfying user experience.

5.1.4.2 Thicker web clients

The serverless architecture encourages to build richer, more interactive and thicker clients, incorporating some part of the application logic on their side, by orchestrating some workflows as well as communicating directly and securely with the datastores and other services. However, the serverless services are still used to process more complex application logic, along with providing additional validation of the client operations as well as performing the processing which cannot be fully trusted on the client side.

5.2 Acquired knowledge and experience

First of all, the broad literature overview has been conducted to gain more knowledge and deeper understanding of the domain of serverless processing. It led to a comprehensive introduction to the field and built the context for suitability of the serverless paradigm and Function as a Service model. Next, the overview of the web application architectures helped with establishing the field of web applications as well as clarifying the requirements used in further analysis.

The thorough theoretical introduction contributed to acquiring essential knowledge to conduct further research, including the analysis of scientific research papers, services documentation, gray literature and conference presentations, including the insight from various practitioners working with the serverless architecture. This, in turn, helped with extending the required knowledge to thoroughly describe the use and applicability of the serverless architecture and Function as a Service model in the field of web application development.

The acquired knowledge is additionally supported by providing example implementations, analyzed in the course of the thesis, which introduced additional context and familiarised author with the serverless processing field, contributing to gaining the practical experience of implementing the web applications in the serverless paradigm based on the Amazon Web Services platform.

5.3 Recommendations and future work

The landscape of serverless processing is evolving rapidly, introducing new services by various cloud providers or incorporating improvements into the existing services which extends their capabilities. The domain of serverless computing will surely expand, giving a high probability that the cloud providers will continue to mitigate various problems of the applicability of the serverless technology, especially in the web application development field.

The research could be repeated in a few years and further extended, including the thorough analysis of the second version of Amazon Aurora Serverless, when it would be publicly available as well as covering the comparison of AWS Step Function Express Workflows in the example implementation, mentioned in section 4.3.2, considering the generation of interactive presentations. Some general recommendations of further extensions, regarding the problems covered in the thesis, are listed below:

- **Covering other cloud platforms in the research** — As mentioned in section 4.1.2, the research focuses mainly on the services provided by the Amazon Web Services to narrow the scope of the thesis and give a more in depth analysis of the capabilities of the serverless processing. Nevertheless, it would be beneficial to analyse other, largest cloud providers, described in section 2.5, verifying how their service offering can be applied to develop the web applications in the serverless architecture.
- **Analysis of the production-grade web applications built in the serverless architecture** — The provided example implementations, covered in section 4.3, refer to rather simple and limited implementations of the web applications, but sufficient enough to reason about the suitability of the serverless architecture for developing web applications. The comprehensive analysis of the second example implementation, discussed in section 4.4.3, gives a valuable insight into the architecture, performance and cost of the developed solution. To further investigate the implications of the serverless architecture, it would be interesting to gather metric from the production-grade application, including the more in depth analysis

of the impact of users behaviour, which most frequently is less regular, especially in context of a full day as well as providing detailed report covering the cost of the executed workloads.

Moreover, several aspects of serverless computing have been introduced in the thesis, leaving some topics not fully covered and applicable for further research. Some of the areas of future work are listed below:

- **Analysis of databases in the serverless paradigm** — The field of serverless databases can be further developed, providing more hands-on analysis, while investigating the performance, applicability and the limitations, when integrating mentioned databases with the serverless architecture. Amazon DynamoDB characterise with the interesting capabilities, however modeling complex business domains, requires leveraging approaches such as Single Table Design, to provide a model which will satisfy performant operations for all access patterns.
- **More in depth investigation of the communication with clients in the serverless architecture** — The research of communication patterns with the web application clients, including components and their capabilities discussed in the thesis, could be further extended, providing more hands-on implementations along with their in-depth analysis, regarding performance and cost implications.

Bibliography

[Aba12] Daniel Abadi. *Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. Computer*, 45(2):37–42, 2012.

[Abi18] Niels Abildgaard. *Perspectives on Architecture, Evolution, and Future of Web Applications.* Master's thesis, 2018.

[AC17] Gojko Adzic and Robert Chatley. *Serverless Computing: Economic and Architectural Impact.* In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 884–889, New York, NY, USA, 2017. Association for Computing Machinery.

[Ama20a] Amazon Web Services. *AWS Lambda*, 2020. [online] https://aws.amazon.com/lambda/ (accessed on 02.12.2020).

[Ama20b] Amazon Web Services. *Serverless on AWS*, 2020. [online] https://aws.amazon.com/serverless/ (accessed on 07.03.2021).

[Ama20c] Amazon Web Services. *Temenos: Building Serverless Banking Software at Scale*, 2020. [online] https://pages.awscloud.com/apn-tv-this-is-my-architecture-ep-214 (accessed on 11.03.2021).

[Ama20d] Amazon Web Services. *The Spikiest Time of the Year Is No Problem for iRobot and AWS IoT*, 2020. [online] https://aws.amazon.com/solutions/case-studies/irobot-iot/ (accessed on 11.03.2021).

[Ama21a] Amazon Web Services. *Amazon API Gateway*, 2021. [online] https://aws.amazon.com/api-gateway/ (accessed on 12.06.2021).

[Ama21b] Amazon Web Services. *Amazon API Gateway Features*, 2021. [online] https://aws.amazon.com/api-gateway/features/ (accessed on 12.06.2021).

[Ama21c] Amazon Web Services. *Amazon Aurora*, 2021. [online] https://aws.amazon.com/rds/aurora/ (accessed on 11.06.2021).

[Ama21d] Amazon Web Services. *Amazon Aurora Serverless*, 2021. [online] https://aws.amazon.com/rds/aurora/serverless/ (accessed on 11.06.2021).

[Ama21e] Amazon Web Services. *Amazon DynamoDB features*, 2021. [online] https://aws.amazon.com/dynamodb/features/ (accessed on 10.06.2021).

[Ama21f] Amazon Web Services. *AWS AppSync*, 2021. [online] https://aws.amazon.com/appsync/ (accessed on 12.06.2021).

[Ama21g] Amazon Web Services. *AWS AppSync Features*, 2021. [online] https://aws.amazon.com/appsync/product-details/ (accessed on 12.06.2021).

[Ama21h] Amazon Web Services. *AWS DynamoDB*, 2021. [online] https://aws.amazon.com/dynamodb (accessed on 10.06.2021).

[Ama21i] Amazon Web Services. *AWS Lambda now supports up to 10 GB of memory and 6 vCPU cores for Lambda Functions*, 2021. [online] https://aws.amazon.com/about-aws/whats-new/2020/12/aws-lambda-supports-10gb-memory-6-vcpu-cores-lambda-functions/ (accessed on 10.04.2021).

[Ama21j] Amazon Web Services. *Creating and sharing Lambda layers*, 2021. [online] https://docs.aws.amazon.com/lambda/latest/dg/configuration-layers.html (accessed on 09.05.2021).

[Ama21k] Amazon Web Services. *Databases on AWS*, 2021. [online] https://aws.amazon.com/products/databases/ (accessed on 06.06.2021).

[Ama21l] Amazon Web Services. *Implementing Microservices on AWS*, 2021. [online] https://d1.awsstatic.com/whitepapers/microservices-on-aws.pdf (accessed on 29.05.2021).

[Ama21m] Amazon Web Services. *Lambda quotas*, 2021. [online] https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html (accessed on 10.04.2021).

[Ama21n] Amazon Web Services. *Lambda runtimes*, 2021. [online] https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html (accessed on 08.05.2021).

[Ama21o] Amazon Web Services. *Using the Data API for Aurora Serverless*, 2021. [online] https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/data-api.html (accessed on 11.06.2021).

[Ama21p] Amazon Web Services. *Web Hosting*, 2021. [online] https://aws.amazon.com/websites/ (accessed on 19.06.2021).

[Ata21] Mete Atamel. *Choreography vs Orchestration in Serverless Microservices*, 2021. [online] https://youtu.be/rDSWHNdYx6E (accessed on 04.06.2021).

[Azu20] Microsoft Azure. *Azure Serverless*, 2020. [online] https://azure.microsoft.com/en-us/solutions/serverless/ (accessed on 07.03.2021).

[Bar21] David AW Barton. *PDF2SVG*, 2021. [online] https://cityinthesky.co.uk/opensource/pdf2svg/ (accessed on 30.04.2021).

[Bes20] James Beswick. *Using Amazon EFS for AWS Lambda in your serverless applications.* 2020. [online] https://aws.amazon.com/blogs/compute/using-amazon-efs-for-aws-lambda-in-your-serverless-applications/ (accessed on 08.05.2021).

[Bes21] James Beswick. *Building server-side rendering for React in AWS Lambda*, 2021. [online] https://aws.amazon.com/blogs/compute/building-server-side-rendering-for-react-in-aws-lambda/ (accessed on 19.06.2021).

[BI18] Shawn Bice and Joseph Idziorek. *Databases on AWS: The Right Tool for the Right Job*, 2018. AWS re:Invent 2018, [online] https://youtu.be/-pb-DkD6cWg.

[Bol19] R.T.J. Bolscher. *Leveraging serverless cloud computing architectures: developing a serverless architecture design framework based on best practices utilizing the potential benefits of serverless computing.* Master's thesis, 2019.

[Bre10] Eric A. Brewer. *A certain freedom: thoughts on the CAP theorem.* In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, page 335. ACM, 2010.

[Clo20a] Alibaba Cloud. *Function Compute*, 2020. [online] https://www.alibabacloud.com/product/function-compute (accessed on 08.03.2021).

[Clo20b] Google Cloud. *Serverless computing*, 2020. [online] https://cloud.google.com/serverless (accessed on 07.03.2021).

[Clo20c] Cloudflare, Inc. *Cloudflare Workers*, 2020. [online] https://workers.cloudflare.com/ (accessed on 08.03.2021).

[Cui20] Yan Cui. *Choreography vs Orchestration in the land of serverless.* 2020. [online] https://theburningmonk.com/2020/08/choreography-vs-orchestration-in-the-land-of-serverless/ (accessed on 04.06.2021).

[Dal20a] Jeremy Daly. *An Introduction to Serverless Microservices.* 2020. [online] https://www.jeremydaly.com/an-introduction-to-serverless-microservices/ (accessed on 04.06.2021).

[Dal20b] Jeremy Daly. *Building Resilient Serverless Systems with "Non-Serverless" Components*, 2020. Serverless Days Cardiff 2020, [online] https://youtu.be/coygxBg2wTY (accessed on 10.06.2021).

[Dal20c] Jeremy Daly. *Serverless Microservice Patterns for AWS.* 2020. [online] https://www.jeremydaly.com/serverless-microservice-patterns-for-aws/ (accessed on 04.06.2021).

[Deb17] Alex Debrie. *Serverless Aurora: What it means and why it's the future of data.* 2017. [online] https://www.serverless.com/blog/serverless-aurora-future-of-data (accessed on 10.06.2021).

[Deb18a] Alex Debrie. *Faux-SQL or NoSQL? Examining four DynamoDB Patterns in Serverless Applications.* 2018. [online] https://www.alexdebrie.com/posts/dynamodb-patterns-serverless/ (accessed on 10.06.2021).

[Deb18b] Alex Debrie. *SQL, NoSQL, and Scale: How DynamoDB scales where relational databases don't.* 2018. [online] https://www.alexdebrie.com/posts/dynamodb-no-bad-queries/ (accessed on 10.06.2021).

[Deb19] Alex Debrie. *Why the PIE theorem is more relevant than the CAP theorem.* 2019. [online] https://www.alexdebrie.com/posts/choosing-a-database-with-pie/ (accessed on 10.06.2021).

[Deb21] Alex Debrie. *Picking a database for your serverless application in 2021*, 2021. Prisma Online Meetup #3, [online] https://youtu.be/pGXR2vy9BJ0 (accessed on 10.06.2021).

[Dev21] Google Developers. *Rendering on the Web*, 2021. [online] https://developers.google.com/web/updates/2019/02/rendering-on-the-web (accessed on 20.03.2021).

[DLL$^+$17] Nicola Dragoni, Ivan Lanese, Stephan Thordal Larsen, Manuel Mazzara, Ruslan Mustafin, and Larisa Safina. *Microservices: How To Make Your Application Scale. CoRR*, abs/1702.07149, 2017.

[Dor21] Dormando. *Memcached*, 2021. [online] https://memcached.org/ (accessed on 27.03.2021).

[Edu21] IBM Cloud Education. *Message Brokers*, 2021. [online] https://www.ibm.com/cloud/learn/message-brokers (accessed on 20.03.2021).

[Eiz17] Thomas Eizinger. *API Design in Distributed Systems: A Comparison between GraphQL and REST*. Master's thesis, 2017.

[Ela21] Elasticsearch B.V. *Elasticsearch*, 2021. [online] https://www.elastic.co/elasticsearch (accessed on 27.03.2021).

[Eva04] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.

[FJG20] Chen-Fu Fan., Anshul Jindal., and Michael Gerndt. Microservices vs serverless: A performance comparison on a cloud-native web application. In *Proceedings of the 10th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,*, pages 204–215. INSTICC, SciTePress, 2020.

[FL14] Martin Fowler and James Lewis. *Microservices*, 2014. [online] https://martinfowler.com/articles/microservices.html (accessed on 19.02.2021).

[FM11] I. Fette and A. Melnikov. *The WebSocket Protocol*. RFC 6455, RFC Editor, 2011. [online] http://www.rfc-editor.org/rfc/rfc6455.txt.

[Fou18] Cloud Native Computing Foundation. *CNCF Serverless Whitepaper v1.0*, 2018. [online] https://github.com/cncf/wg-serverless/tree/master/whitepapers/serverless-overview (accessed on 02.12.2020).

[Fou20a] Cloud Native Computing Foundation. *CNCF Survey 2020*, 2020. [online] https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf (accessed on 07.03.2021).

[Fou20b] The Apache Software Foundation. *Apache OpenWhisk*, 2020. [online] https://openwhisk.apache.org/ (accessed on 08.03.2021).

[Fou21a] The Apache Software Foundation. *Apache Cassandra*, 2021. [online] https://cassandra.apache.org/ (accessed on 27.03.2021).

[Fou21b] The Apache Software Foundation. *Apache CouchDB*, 2021. [online] https://couchdb.apache.org/ (accessed on 27.03.2021).

[Fou21c] The Apache Software Foundation. *Apache HBase*, 2021. [online] https://hbase.apache.org/ (accessed on 27.03.2021).

[Fou21d] The Apache Software Foundation. *Apache Solr*, 2021. [online] https://solr.apache.org/ (accessed on 27.03.2021).

[Fou21e] The GraphQL Foundation. *GraphQL*, 2021. [online] https://graphql.org/ (accessed on 19.03.2021).

[Fow03a] Martin Fowler. *Anemic Domain Model*, 2003. [online] https://www.martinfowler.com/bliki/AnemicDomainModel.html (accessed on 19.02.2021).

[Fow03b] Martin Fowler. *NoSQL Definition*, 2003. [online] https://martinfowler.com/bliki/NosqlDefinition.html (accessed on 19.02.2021).

[Fow12] Martin Fowler. *Introduction to NoSQL*, 2012. GOTO 2012, [online] https://youtu.be/ql_g07C_Q5I.

[Fow15] Martin Fowler. *Microservice Trade-Offs*, 2015. [online] https://martinfowler.com/articles/microservice-trade-offs.html (accessed on 19.02.2021).

[Fro12] Ken Fromm. *Why The Future Of Software And Apps Is Serverless*, 2012. [online] https://readwrite.com/2012/10/15/why-the-future-of-software-and-apps-is-serverless/ (accessed on 02.12.2020).

[FT02] Roy T. Fielding and Richard N. Taylor. *Principled design of the modern Web architecture. ACM Transactions on Internet Technology*, 2(2):115–150, 2002.

[Gar20] Gartner, Inc. *Magic Quadrant for Cloud Infrastructure and Platform Services*, 2020. [online] https://www.gartner.com/doc/reprints?id=1-1ZDZDMTF&ct=200703&st=sb (accessed on 08.03.2021).

[Git21] GitHub, Inc. *Github Actions*, 2021. [online] https://github.com/features/actions (accessed on 19.06.2021).

[GMS87] Hector Garcia-Molina and Kenneth Salem. *Sagas*. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, SIGMOD '87, page 249–259, New York, NY, USA, 1987. Association for Computing Machinery.

[Goo20] Google. *Firebase*, 2020. [online] https://firebase.google.com/ (accessed on 08.03.2021).

[Gru19] Algirdas Grumuldis. *Evaluation of "Serverless" Application Programming Model: How and when to start Serverless.* Master's thesis, KTH, School of Electrical Engineering and Computer Science (EECS), 2019.

[GWFR17] Felix Gessert, Wolfram Wingerath, Steffen Friedrich, and Norbert Ritter. *NoSQL database systems: a survey and decision guidance. Comput. Sci. Res. Dev.*, 32(3-4):353–365, 2017.

[IRD19] Cosmina Ivan, Vasile Radu, and Vasile Dadarlat. *Serverless Computing: An Investigation of Deployment Environments for Web APIs. Computers*, 8:50, 2019.

[Itu19] Roberto Iturralde. *Serverless at scale: Design patterns and optimizations*, 2019. AWS re:Invent 2019, [online] https://youtu.be/dzU_WjobaRA.

[Joh19] Eric Johnson. *I didn't know Amazon API Gateway did that*, 2019. AWS re:Invent 2019, [online] https://youtu.be/yfJZc3sJZ8E.

[JSS$^+$19] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. *Cloud Programming Simplified: A Berkeley View on Serverless Computing. CoRR*, abs/1902.03383, 2019.

[Keh20] Ben Kehoe. *Serverless IoT at iRobot*, 2020. [online] https://www.infoq.com/presentations/serverless-iot-irobot/ (accessed on 11.03.2021).

[Kle17] Martin Kleppmann. *Designing Data-Intensive Applications.* O'Reilly, Beijing, 2017.

[Kro15] Sam Kroonenburg. *Serverless — the future of software architecture?* 2015. [online] https://acloudguru.com/blog/engineering/serverless-the-future-of-software-architecture (accessed on 19.06.2021).

[Les19] Heitor Lessa. *Serverless architectural patterns and best practices*, 2019. AWS re:Invent 2019, [online] https://youtu.be/9IYpGTS7Jy0.

[LKRL19] Junfeng Li, Sameer G. Kulkarni, K. K. Ramakrishnan, and Dan Li. *Understanding Open Source Serverless Platforms: Design Considerations and Performance.* In *Proceedings of the 5th International Workshop on Serverless Computing*, WOSC '19, page 37–42, New York, NY, USA, 2019. Association for Computing Machinery.

[Mao19] George Mao. *Using Relational Databases with AWS Lambda - Easy Connection Pooling*, 2019. AWS Online Tech Talks, [online] https://youtu.be/dgj9cvqgYYs (accessed on 10.06.2021).

[Mei19] Azhar Susanto Meiryani. *Database Management System. International Journal of Scientific & Technology Research*, 2019.

[Mon21] MongoDB, Inc. *MongoDB*, 2021. [online] https://www.mongodb.com/ (accessed on 27.03.2021).

[MP13] Michael Mikowski and Josh Powell. *Single Page Web Applications: JavaScript End-to-End*. Manning Publications, 1st edition, 2013.

[Mun18] Chris Munns. *Become a Serverless Black Belt*, 2018. AWS Online Tech Talks, [online] https://youtu.be/4nrRt0dOcFk (accessed on 08.05.2021).

[Mun19a] Chris Munns. *Building microservices with AWS Lambda*, 2019. AWS re:Invent 2019, [online] https://youtu.be/TOn0xhev0Uk.

[Mun19b] Chris Munns. *Optimizing Your Serverless Applications*, 2019. AWS Online Tech Talks, [online] https://youtu.be/DYQ8pXrktBM (accessed on 08.05.2021).

[Neo21] Neo4j, Inc. *Neo4j*, 2021. [online] https://neo4j.com/ (accessed on 27.03.2021).

[Net20] Netlify. *Netlify Functions*, 2020. [online] https://www.netlify.com/products/functions/ (accessed on 08.03.2021).

[Net21] Netlify. *Jamstack*, 2021. [online] https://jamstack.org/ (accessed on 20.03.2021).

[NMMA16] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice Architecture: Aligning Principles, Practices, and Culture*. O'Reilly Media, Inc., 1st edition, 2016.

[O'D19] Dan O'Donnell. *What Front-End Developers Need to Know about Serverless Databases*. 2019. [online] https://thenewstack.io/what-front-end-developers-need-to-know-about-serverless-databases/ (accessed on 10.06.2021).

[PHM19] Michael Paris, Nare Hayrapetyan, and Patrick Michelberger. *Develop serverless GraphQL architectures using AWS AppSync*, 2019. AWS re:Invent 2019, [online] https://youtu.be/XVU4pYeNfNo.

[Pir20] Justin Pirtle. *Scalable serverless event-driven architectures with SNS, SQS, and Lambda*, 2020. AWS re:Invent 2020, [online] https://youtu.be/8zysQqxgj0I.

[PKC19] Andrei Palade, Aqeel Kazmi, and S. Clarke. *An Evaluation of Open Source Serverless Computing Frameworks Support at the Edge. 2019 IEEE World Congress on Services (SERVICES)*, 2642-939X:206–211, 2019.

[Poc20] Danilo Poccia. *New for AWS Lambda – Container Image Support*. 2020. [online] https://aws.amazon.com/blogs/aws/new-for-aws-lambda-container-image-support/ (accessed on 08.05.2021).

[Pro20] Fn Project. *Fn Project*, 2020. [online] https://fnproject.io/ (accessed on 08.03.2021).

[Pro21] The LaTeX Project. *LaTeX – A document preparation system*, 2021. [online] https://www.latex-project.org/ (accessed on 30.04.2021).

[Rad21] Maciej Radzikowski. *AWS Lambda performance optimization*, 2021. [online] https://betterdev.blog/aws-lambda-performance-optimization/ (accessed on 09.05.2021).

[RC17] Mike Roberts and John Chapin. *What is Serverless?* O'Reilly Media, Inc., 2017.

[Red21] Redis Labs. *Redis*, 2021. [online] https://redis.io/ (accessed on 27.03.2021).

[Rob18] Mike Roberts. *Serverless Architecture*, 2018. [online] https://martinfowler.com/articles/serverless.html (accessed on 02.12.2020).

[Sba17] Peter Sbarski. *Serverless Architectures on AWS.* Manning Publications, 2017.

[SCN21] Peter Sbarski, Yan Cui, and Ajay Nair. *Serverless Architectures on AWS, Second Edition.* Manning Publications, 2021.

[Ser21] Serverless, Inc. *Container Image Support for AWS Lambda*, 2021. [online] https://www.serverless.com/blog/container-support-for-lambda (accessed on 10.04.2021).

[Shi20] Mikhail Shilkov. *Running Container Images in AWS Lambda.* 2020. [online] https://mikhail.io/2020/12/aws-lambda-container-support/ (accessed on 09.05.2021).

[SHS14] D. Skvorc, M. Horvat, and S. Srbljic. *Performance evaluation of Websocket protocol for implementation of full-duplex web streams.* In *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1003–1008, 2014.

[SK19] Hossein Shafiei and Ahmad Khonsari. *Serverless Computing: Opportunities and Challenges. CoRR*, abs/1911.01296, 2019.

[SR03] L. Shklar and R. Rosen. *Web Application Architecture: Principles, Protocols and Practices.* Wiley, 2003.

[SS19] Slobodan Stojanović and Aleksandar Simović. *Serverless Applications with Node.js.* Manning Publications, 2019.

[Vog18] Werner Vogels. *A one size fits all database doesn't fit anyone.* 2018. [online] https://www.allthingsdistributed.com/2018/06/purpose-built-databases-in-aws.html (accessed on 06.06.2021).

[Woo20] Julian Wood. *Choosing Events, Queues, Topics, and Streams in Your Serverless Application*, 2020. AWS Online Tech Talks, [online] https://youtu.be/d9Jb1WKCLd8.

[Wri21] Joseph Wright. *Beamer - A LaTeX class for producing presentations*, 2021. [online] https://github.com/josephwright/beamer (accessed on 30.04.2021).

[XwX11] Xiao-wei and Y. Xue. *A Survey on Web Application Security.* 2011.