Cloud Computing: Serverless

Arne Koschel
Faculty IV, Department of Computer Science
University of Applied Sciences and Arts Hannover
Hannover, Germany
akoschel@acm.org

Samuel Klassen
Faculty IV, Department of Computer Science
University of Applied Sciences and Arts Hannover
Hannover, Germany
samuel.klassen@stud.hs-hannover.de

Kerim Jdiya
Faculty IV, Department of Computer Science
University of Applied Sciences and Arts Hannover
Hannover, Germany
kerim.jdiya@stud.hs-hannover.de

Marc Schaaf
Institute of Information Systems
University of Applied Sciences Northwestern Switzerland
Olten, Switzerland
marc.schaaf@fhnw.ch

Irina Astrova
Department of Software Science, School of IT
Tallinn University of Technology
Tallinn, Estonia
irina@cs.ioc.ee

***Abstract*—A serverless architecture is a new approach to offering services over the Internet. It combines BaaS (Backend-as-a-service) and FaaS (Function-as-a-service). With the serverless architecture no own or rented infrastructures are needed anymore. In addition, the company does not have to worry about scaling any longer, as this happens automatically and immediately. Furthermore, there is no need any longer for maintenance work on the servers, as this is completely taken over by the provider. Administrators are also no longer needed for the same reason. Finally, many ready-made functions are offered, with which the development effort can be reduced. As a result, the serverless architecture is very well suited to many application scenarios, and it can save considerable costs (server costs, maintenance costs, personnel costs, electricity costs, etc.). The company only must subdivide the source code of the application and upload it to the provider's server. The rest is done by the provider.**

Keywords*—serverless architecture, cloud computing, scaling, serverless functions, service models, BaaS (Backend-as-a-service), FaaS (Function-as-a-service)

I. Introduction

Helmut Balzert described software architectures as "a structured or hierarchical arrangement of the system components and description of their relationships" [1]. The components of these architectures can be grouped by tiers, which are:

- **Presentation tier:** This tier is responsible for showing the interface of the application to the user. To do this, it calls the application tier. The input of the presentation tier is typically made by the user.
- **Application tier:** This tier is often the biggest part of an application and contains all the business logic. In the application tier, the core features of an application are implemented. The input comes either from the presentation tier (the user) or from the data tier (persistent data).
- **Data tier:** This tier contains the persistent data of an application (e.g., user data, invoices, product information for a web shop. etc.). The inputs of the data tier typically come from the application tier and are also retrieved by it afterwards. The data layer is usually realized via one or more databases, but it can also be realized via a simple text file.

The well-known classical software architectures are two-tier and three-tier architectures.

A. Two-Tear Architecture

A two-tier architecture consists of two layers. One represents the client side and the other the server side. A typical distribution of the tiers is shown in Fig. 1.

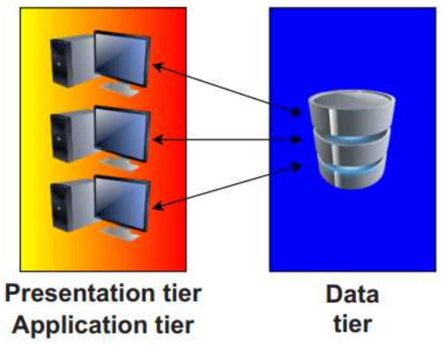

Fig. 1. Typical distribution of tiers in two-tier architecture.

In this configuration, the client is a so-called "fat client". It contains the presentation and application tiers. This means that the entire application logic is also executed on the client and the client must have sufficient power to do this. Otherwise, there is a possibility that the user experience will be negatively affected. The server side only contains the data tier and is therefore responsible for storing the data persistently. Another aspect, which should be considered, are malicious clients. Since the entire application logic is located on the client, it is possible that the client will change it for its own purpose. So the client can do things that are (probably) not allowed and we should check the input from the client on the server side. But this aspect is also a big advantage of this architecture, because all the (computationally intensive) calculations of the application logic are done by the client. This leads to a reduction in server load and a more responsive user experience. A further disadvantage is the weak encapsulation, which means that each client must install an update whenever the application logic needs to be updated [2].

B. Three-Tear Architecture

A three-tier architecture consists of three layers. One represents the client side and the other two the server side. A typical distribution of the tiers is shown in Fig. 2.

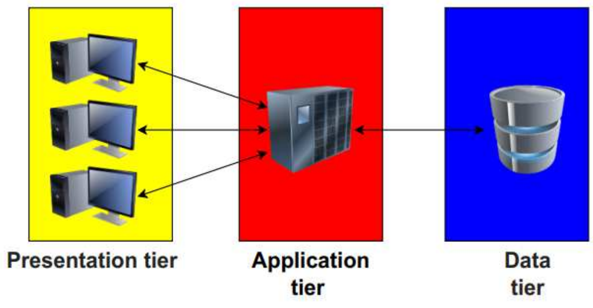

Fig. 2. Typical distribution of tiers in three-tier architecture.

The main difference from the two-tier architecture is that the application tier is part of the server side. In most cases, the application tier resides on a separate server. Therefore, it is not necessary that the clients have strong hardware to run the application. However, this leads to an increased server load. The three-tier architecture also makes it more difficult for the clients to manipulate the application, since all the application logic resides on the server. Nevertheless, the input of the clients must be checked before processing them. Furthermore, there is a stronger decoupling, which leads to better maintainability. It is also possible to change the application logic without the need for an update on the client side. The data tier is located "behind" the application tier, which means that direct accesses by the clients are not possible. Only the application tier can access the data tier [2], [3].

C. *Issues with Classical Architectures*

The classic architectures described above have many disadvantages. However, with the help of cloud computing, the following disadvantages could be eliminated [4]:

- **Purchase of servers:** Servers can cost a lot of money, but they are needed to provide services (e.g., websites, APIs, file servers etc.) over the Internet. Assumptions about their usage must be made to dimension the servers accordingly. For this purpose, it is necessary to know how many users will potentially use the service(s) and how many resources each one of them needs. Moreover, a question must be answered whether the processes can be parallelized, but it is not easy to answer this question prior to commissioning.

- **Scaling:** Besides the first purchase of servers, there is a question of how much growth can be expected. Can a constant load be expected or are there load peaks? If there are any, will additional servers be needed, or will a worse user experience be accepted because these spikes are rare? And if more power is needed, is it reasonable to scale horizontally or vertically? Especially the decision about scaling can slow down the growth of the company, because if the number of users increases significantly and quickly, a negative experience for them can be fatal for the company's reputation.

- **Administration and maintenance:** After the servers have been purchased, they require continuous maintenance. In addition, regular updates are necessary, which must be coordinated with the running applications and must be installed. Furthermore, there is always the risk of hardware failures. Thus, the hardware must be replaced quickly to avoid any risk of downtime. All these things must be done by employees. Therefore, in addition to the hardware costs, there are ongoing employee costs. Depending on the size of the company, the sum of these costs can be a significant burden on the financial situation of the company if it decides to operate its own servers.

II. Serverless Architecture

A serverless architecture is aimed at solving the drawbacks of classical architectures. The main reason for those disadvantages is that an own server must be administrated and maintained. With the serverless architecture. the company does not have to worry about the server, just because there is no server.

A. *Definition*

A statement that is often read in the context of serverless architecture is: "Run code, not server". This is indeed the core idea of serverless architecture. The focus of developers should be more on code and business logic, than on managing and administrating the infrastructure and resources around it.

To concentrate on the code and not on the server, third-party services are used. These services help the company to accomplish tasks, which are otherwise taken care of by servers. Thus, there must be services that give the company an abstraction of the backend and allow the company to develop its application, without thinking about the required platform, hardware or infrastructure. The company does not need a server and it even does not need to think about the server [5].

B. *Serverless in Cloud Computing*

In cloud computing the term serverless has two different definitions, which often confuse. Although the focus of this paper will only be on one type of serverless, we describe both definitions here for a better understanding.

- **Backend-as-a-Service (BaaS)** describes applications that significantly uses third-party (cloud-hosted) applications and services to manage server-side logic and state. The business logic is then mostly in the client. Such applications (e.g., single page applications or mobile apps) are often called "rich client" applications. An example for BaaS is FireBase. It is a cloud-hosted database system, which can communicate directly with the client. So there is no webserver in between and the database system takes care about all resources and management issues.

- **Function-as-a-Service (FaaS)** is the other definition of serverless. The big difference with BaaS is the possibility of deploying own code (called functions) in the cloud. Thus, the company can use its own code, without managing the hardware by its own. In other words, FaaS is the concept of serverless computing via a serverless architecture.

Since BaaS and FaaS are related in their operational attributes (e.g., no ressource management), they are often used together [4], [5].

C. *Definition of FaaS*

Amazon is one of the largest providers of cloud platforms, like the FaaS platform. Their platform for FaaS is called AWS Lambda. On the website of AWS Lambda, the following definition of FaaS is given: "AWS Lambda lets you run code without provisioning or managing servers. (1) With Lambda, you can run code for virtually any type of application or backend service (2) - all with zero administration. Just upload your code and Lambda takes care of everything required to run (3) and scale (4) your code with high availability. You can set up your code to automatically trigger from other AWS services (5) or call it directly from any web or mobile app (6)" [6]. The following properties can be deduced from this definition:

(1) FaaS allows the company to run its own code, without managing its own server. Thus, it gives the developers a complete abstraction of servers. The special thing about FaaS is that functions are running in stateless compute containers. This means that the state of the container, where the code is running on, is not guaranteed. This is because of the nature of functions. They are running for one request and are terminated afterwards. Therefore, the state of the server, where they are running on, can change between running functions. If persistence is required, the state must be stored outside the server (database, cross-application cache, network file store, etc.). Monitoring is also difficult, as the functions do not necessarily run on the same server [4].

(2) Code for FaaS does not depend on a specific framework or library. The only dependency is the supported programming language by the platform provider. Other external dependencies can be chosen freely by the developers.

(3) To deploy the function's code, the company only must upload it. The provider does everything else that is necessary to run it properly. Code modification effort and speed are optimized, because only small functions that are isolated from each other are modified and directly deployed.

(4) In case of many calls, the required horizontal scaling is managed by the provider, automatically and elastically. Furthermore, the billing is based on the consumption and executions, not on the instance size of the server. However, it must be noted that there is a maximum runtime for functions (e.g., 5 mins for AWS Lambda).

(5) Functions in FaaS are typically invoked by triggered events. These events are defined by the provider. Thus, the company can call the functions from other services by the provider with one of the defined events.

(6) As the company can invoke the functions from cloud services, most providers allow the company to trigger the functions as a response to inbound HTTP requests, e.g., from an arbitrary client.

The following use cases result from the above definition and the properties of FaaS [7]:

- Isolation of super high-volume transactions for better scaling and performance.
- Functions that can run dynamically or burstable, e.g., once per day or month. There is no need to pay for a server around the clock.
- Scheduled tasks are perfect to run a certain piece of code on a schedule.
- Processing of a single web request as well as an unexpectedly sudden high number of requests
- Processing individual messages from a message queue, as well as an unexpectedly sudden high number of requests.
- Manual triggering of a function.

D. Cloud Computing Service Models

There are three well-known service models of cloud computing, namely, Infrastructure-as-a-Service (IaaS), Platform-as-a Service (PaaS) and Software-as-a-Service (SaaS). Each of the three models describes how cloud services can look like and what kind of abstraction it can provide.

IaaS is the first level of abstraction and manages the whole hardware. It provides running hardware, so the company does not have to worry about it anymore. But for that the company must take care of the whole software, from the operating system to the application.

The next level of abstraction is **PaaS**. If there are no special requirements for the runtime environment, an offered runtime environment can be used. However, the necessary application (including the data) must still be managed.

The full abstraction of the backend and the software is offered with **SaaS**. These are complete applications that can be used remotely, without any additional effort by the user. The limitation, however, is that the company has no influence on the application.

Now there are two further components, namely, BaaS and FaaS, which can be summarized to the term Serverless. The classification of Serverless in the service model is shown in Fig. 3. Serverless must be before SaaS, because of the possibility to deploy own code, because not everything is taken over by the provider. In addition, the abstraction is somewhat stronger with Serverless than with PaaS, because the provider also takes over the administration of the data (e.g., the state of the server).

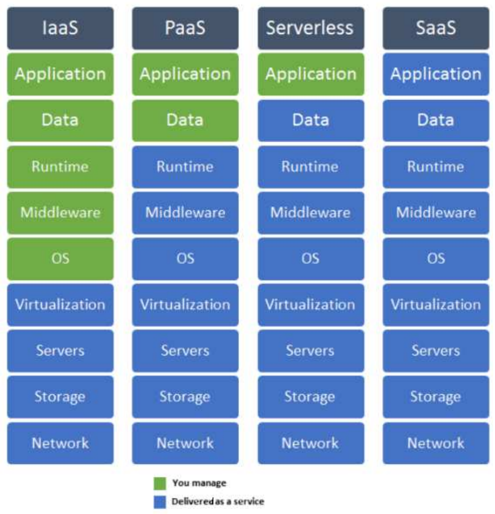

Fig. 3. Differences between the service models [8].

One additional difference between PaaS and Serverless is the characteristic of applications built for PaaS. Adrian Cockcroft says: "If your PaaS can efficiently start instances in 20 milliseconds that run for half a second, then call it serverless" [4]. Thus, one difference in the characteristics is runtime. An application on PaaS is typically running for all times. Functions on FaaS instead run only on demand.

Another big difference between FaaS and PaaS is scaling. With PaaS the company still needs to think about how to scale. With an FaaS application this is completely transparent. Entire applications are no longer deployed on FaaS, but only individual functions. It also follows from the above two differences that FaaS is more accurately billed and therefore often cheaper than a PaaS application.

E. Performance of FaaS

FaaS functions start very fast and run only for a short time. However, the performance of a function is not always the same and depends on the actual state of the function.

1) ***Cold start:*** This means that there is actually no instance of the function. So first an instance must be created before the function can be executed. How quickly an executable instance can be created depends on following aspects:

- **The number of libraries:** The more external libraries are used and needed, the longer it takes to create an instance.
- **The amount of code:** This can also negatively influence the creation of an instance.
- **The used programming language:** For example, if the Java programming language is used, the slow JVM must be started first, which can extend the cold start. A lightweight scripting language, e.g., Python could be executed directly and would not delay the cold start.
- **The configuration of the function:** A bad configuration can also negatively influence the creation of an instance.
- **The connection establishment to external resources:** If a connection to an external resource must first be established, a cold start will take longer.

2) ***Warm start:*** Unlike the cold start, a warm start already has a function instance from a previous function call. So the company does not have to create an instance first. Rather, the company can use the existing one directly. There is also no need to start runtime environments or establish connections to external resources.

The performance of creating a new instance can usually be controlled by the developers, e.g., when few libraries or a lightweight scripting language can be used. But, generally, the performance is worse with a cold start than a warm start. Thus, if a function is called regularly, there is rarely a cold start. However, if calls are infrequent and quick responses are required, the company should consider how to keep the function alive. Or the company does not use a cloud service, but its own server, so that the function runs around the clock with few resources [4].

F. API Gateway

To better understand the architecture and the interaction of all individual components in a serverless architecture, this section describes a central, often used, component. This component is called the API gateway, which is an HTTP server that forwards requests to a specific function. Either the function is called with the specified parameters in the request or the request is forwarded as a JSON object. The API gateway then forwards the function's response back to the caller as a HTTP response. This creates a loose coupling between the client and the cloud backend.

An API gateway can have the following additional functionalities:

- Authentication of users.
- Input validation of calls. Any kind of control and limitation of calls.
- Caching of calls to be able to process further calls faster.
- Logging of actions and calls to comprehend and understand behavior.
- Aggregation of results for more efficient communication and processing.

These additional functionalities can be used to perform tasks that would otherwise be performed by a server. However, since a server no longer exists, these tasks must be performed by other components (e.g., the API gateway or the client). In AWS, the API gateway is realized via BaaS that can be easily configured by the developers or administrator. This means that no additional development effort is required for these functionalities [4], [9].

G. Migration Example

To understand the difference between a classical architecture (viz., three-tier architecture) and a serverless architecture, a practical example will be used. This example is an online shop where articles for pets can be searched and purchased. An authentication of the client is also carried out. The rough architecture of this system is shown in Fig. 4. Assuming the business logic on the server side is written in Java and on the client side, HTML and Javascript are used. Most of the business logic (authentication, page navigation, searching, transactions) can be implemented on the server, so we can have a very thin client.

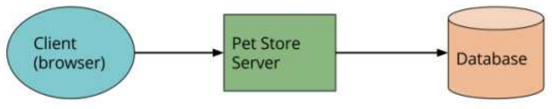

Fig. 4. Example application with a three-tier architecture [4].

The result of transforming the three-tier architecture is shown in Fig. 5 and next explained:

1) The first step is to handle the authentication of users via an authentication service (e.g., Auth0). There can be a direct communication with the authentication service and there is no need for other services.

2) Then the database could be split. One database contains the products, which can be accessed directly from the client. We can also have different security profiles for the client accessing the database than for server resources. The other database contains the purchases and can only be accessed via a special function. Both databases are hosted on dedicated servers or via BaaS in the cloud. They are not part of the functions.

3) Due to the direct database access, the authentication at the authentication service and the missing application server, a part of the business logic is shifted to the client. In this way the client can quickly become a Single Page Application.

4) Compute intensive tasks or accesses to a significant amount of data should not be run on a mobile device due to lack of resources. Therefore, it may make sense to perform such tasks in a function in the cloud, that runs only on demand. When AWS Lambda is used as FaaS platform, there is also no need of rewriting the function code, since Lambda supports Java - the original implementation language.

5) Another reason for performing tasks in functions in the cloud, are security aspects. If accesses to a database are critical for safety reasons, the access should be controlled by a function, rather than implementing it in the client.

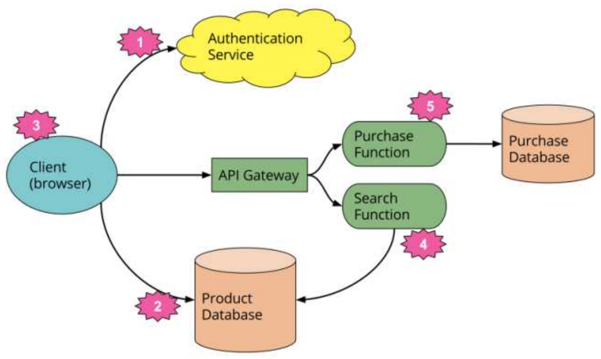

Fig. 5. Migration of the sample application to the serverless architecture [4].

After this transformation the developers must fill the database with products only, upload the code for both functions and develop the client. The authentication is done by the authentication service, the functions are scaled automatically, and the hardware is managed by the provider. This example demonstrates another very important point about serverless architectures. In the original version, all control and security flow were managed by the central server application. In the serverless version, there is no central arbiter of these concerns. Instead we see a preference for choreography over orchestration, with each component playing a more architecturally aware role – an idea also common in a microservices approach.

There are many benefits to such an approach. Systems built this way are often more flexible and amenable to change, both as a whole and through independent updates to components. There is a better separation of concerns and there are also some significant cost benefits. Of course, such a design is a trade-off: It requires better distributed monitoring and more reliance on the security capabilities of the underlying platform. More fundamentally, there are a greater number of moving pieces to get our heads around than there are with the monolithic application we had originally. Whether the benefits of flexibility and cost are worth the added complexity of multiple backend components is very context dependent [4].

III. Evaluation

In this section, the practical suitability is examined first. Then the benefits and drawbacks of serverless architectures are listed and described. Finally, some features are described which are currently not available, but which would be desirable in the future.

A. Practical Suitability

To determine whether serverless architectures are suitable for a given application, the following aspects must be considered [4]:

- **Number of requests:** The number of requests is relevant if cold starts should be avoided. Depending on the provider, a certain number of calls must be reached within a fixed time interval to prevent the application from being “frozen”.
- **Traffic volume:** The traffic volume must be considered when deciding whether it makes more sense, from a financial point of view, to run the applications in a serverless architecture. If the traffic volume is constant, this approach does not always make sense because the servers can be dimensioned accordingly to ensure efficient utilization. If load peaks can be expected, on the other hand, it can really make sense, since no additional server capacities are required just for them.
- **Confidentiality of data:** Another aspect is the confidentiality of data. Is the company allowed to outsource the processing of data to external servers or is it not allowed, e.g., due to data protection regulations?
- **Type of requests:** It is also necessary to consider the respective application. As described above, serverless functions are (so far) stateless. Thus, it would not make sense to use them with session-heavy functions.

Use case 1: Inconsistent traffic – An example of inconsistent traffic is a ticket shop. Most of the time it receives an even number of requests. But as soon as a ticket pre-purchase, e.g., from a famous band starts, the requests increase rapidly. This leads to load peaks and usually in such a scenario the servers would crash, and the customer gets an error message. This circumstance is undesirable, because on the one hand the ticket shop does not earn money from the customers and they migrate to competing companies. And on the other hand, this has a negative effect on the reputation of the company if it is not able to serve the customers in such situations (which often occur in a ticket shop).

One solution for the company could be to scale horizontally or vertically to use the additional resources when they are needed. The disadvantage of vertical scaling, however, is that there is a fixed limit up to which scaling is possible (e.g., because there is no stronger processor available). With horizontal scaling, additional servers are purchased, and the load is distributed. The additional servers can be used in two ways:

- **Keep the server running continuously:** The advantage here is that the servers are immediately available during load peaks. This leads to an immediate scaling and the customer does not notice anything. The disadvantage, however, is increased power and maintenance costs.
- **Starting the server during load peaks:** Another possibility is to start the servers only when they are needed. This reduces the costs during the time when the servers are not needed. The disadvantage, however, is that once they are needed, they take some

time to get up and running. In addition, servers are not designed for frequent startup and shutdown, so this solution also has a negative effect on the longevity of the servers.

With serverless architectures, the functions that are relevant in these situations could be outsourced via FaaS, so the company no longer must worry about scaling. During peak loads, more resources are simply allocated to these serverless functions and scaling occurs automatically and immediately [4].

Use case 2: Occasional requests – An example for occasional requests could be a new website. For example, it receives and answers a request every few minutes. The website is hosted on a dedicated server and the average CPU power required is below 5%. Most of the time, however, it is in idle and only consumes power.

When using a serverless architecture, it could be hosted on a dedicated server along with other similar applications, sharing resources without being slowed down by them (due to automatic scaling). This approach saves a considerable amount of power and thus costs that are only incurred for the actual use of resources with the serverless architecture [4].

B. Advantages

If the decision is made to use serverless architectures, there are several benefits. The most important ones are [4]:

- **Reduced operational costs:** Since no more own servers have to be purchased and the wage costs for the administration and maintenance are eliminated, the costs can be reduced significantly. This aspect is particularly important for applications that must deal with high load peaks and would require additional hardware to handle them. In addition, sharing the infrastructure leads to cost savings.
- **Reduced development costs**: Many infrastructure providers offer additional functions that no longer need to be implemented by the company. One example is AWS Lambda's authentication service (AWS Cognito), which includes the registration of new users as well as the login and management of passwords [6].
- **(Nearly) no scaling costs:** The big advantage of serverless architectures is the scaling. It happens automatically and load peaks are simply cushioned. The only cost to the company is paying for the additional hardware resources which are then used.
- **Performance optimization directly reduces costs:** Since the company pays for the hardware resources they have really used, performance optimizations on the source code have a direct effect on costs. For example, if the runtime of an application is reduced from 100 to 50 milliseconds by optimizations, the costs to be paid are also halved.

C. Disadvantages

Serverless architectures also have some drawbacks. The most important ones are [4]:

- **Vendor control:** Through vendor control, the company must anticipate sudden failures, cost changes, additional limitations, or the loss of functionality.
- **Vendor lock-in:** The use of the additional (exclusive) features (e.g., tools, architectures, libraries) of the provider can be helpful. However, the use quickly leads to a vendor lock-in, as a migration to other providers becomes increasingly costly. One possible solution would be the so-called "multi-cloud approach", in which the development of the application is designed in such a way that the majority is developed in-house and thus no reliance on the provider is necessary. Of course, this approach increases the cost of development, but in return reduces the cost of a possible future migration of the software.
- **No in-server state (FaaS):** For applications that must deal with sessions, the use of serverless functions is not recommended because they are (so far) stateless (and for good reasons, such as efficiency).
- **Limited execution duration:** Another disadvantage can be the limited execution time of individual functions. For example, the limit for AWS Lambda is 5 mins.
- **Cold starts:** Cold starts occur if a function is not called for a few minutes (depending on the provider). As a result, the subsequent call takes longer and there is an increased latency.

D. Security Aspects

An important part of the evaluation is also the listing and description of the safety-relevant aspects. The following aspects are of particular interest:

- **Increased attack surface:** If a software is distributed on the infrastructure of several providers (e.g., due to the use of special features, more independence, etc.), this increases the attack surface of the application. This is because each vendor has its own implementation and therefore there is a greater risk that one of them may contain a security problem and thus serve as a gateway for attackers to get into the application.
- **BaaS database without protective application tier barrier:** As can be seen in the migration example, the use of the serverless architecture leads to a division of an application and the distribution of the individual functions. This can also result in the client being able to access the database server directly, so that access to the database server cannot be controlled by the application tier (as it is the case with the three-tier architecture). For this reason, it may be necessary to perform additional access control on the database server (and other components of the application). This significantly increases the complexity of the application and can also lead to security problems.
- **Loss of overview:** An application can consist of many individual functions. If serverless architecture is used, this can quickly lead to a loss of overview of the serverless functions and thus might lead quickly to security and overall maintenance problems.

E. Other Aspects

Three important additional aspects to consider when choosing serverless architecture are [4]:

- **Testing:** The testing of individual serverless functions can be easily done via unit tests. However, these and integration tests are more difficult if the application has many dependencies.

- **Debugging:** Some providers support debugging directly on the server. But many others do not (yet) and this can lead to more complex debugging.

- **Monitoring:** Nowadays, monitoring the application is very important to be able to create metrics. By means of these metrics it is possible to identify and optimize critical points of the application. In addition, hard to find errors can be found. Unfortunately, many vendors only offer basic metrics that the company must use. Mostly these are not sufficient and more information (e.g., via open APIs) would be desirable.

IV. Future Work

Serverless functions are still relatively new and immature. They still lack some features to be used on a large scale. Some of the most important features are [4]:

- **Deploying groups of components:** Currently, it is possible to deploy functions individually. Since many functions of an application are based on others, it would be desirable if a collection of functions could be deployed together.

- **Remote debugging:** Many providers do not offer direct debugging of their servers. However, to test functions and discover errors, this is a feature that should not be missed.

- **"Meta operations":** Many providers offer management features for each individual deployed serverless function. For large applications, however, it would be more desirable if settings could be made for self-defined groups of functions.

- **State management:** Many applications use sessions, e.g., to store a user's shopping cart for a longer period of time. To implement such applications, serverless functions should be no longer exclusively stateless in the future. An alternative could be a dedicated "state server" that manages the states and injects the individual state before calling a function.

- **Permanent availability:** As soon as a serverless function is not called for a longer period of time, it will be frozen. This leads to a cold start next time it is called. In the future, it would be desirable if serverless functions were available permanently and without a cold start.

- **Patterns:** Patterns represent a standardized solution for similar problems. For serverless architectures, however, relatively few such patterns exist. Patterns that give a specification for the size of individual serverless functions are desirable. If it is possible to deploy groups of serverless functions in the future, a pattern could also specify how to determine such groups from the application. Furthermore, it would make sense for monitoring that no log is created and can be viewed for each function. Instead, it would be more helpful if the logs of the individual functions were aggregated in such a way that an overview of the application could be obtained more quickly. Patterns would be useful for the definitions of such aggregations.

- **Standardization:** Currently, the offers of the providers differ in almost all respects. This means that before deciding on a provider, all features must be examined and weighed against each other. A standardization with a defined feature set would help with this decision in the future. In addition, portability can be estimated before using a provider, so that no sudden extensive new developments would be necessary, thus increasing the planning ability for a company.

In the future, one more service model of cloud computing could be evaluated, viz., Storage as a Service (STaaS) [10], [11], [12].

References

[1] H. Balzert, Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb, 3rd ed. Berlin Heidelberg New York: SpringerVerlag, 2011.

[2] G. Reese, Database Programming with JDBC and Java, 2nd ed. O'Reilly & Associates, 2000, relevant chapter available via https://web.archive.org/web/20110406121920/http://java.sun.com/developer/Books/jdbc/ch07.pdf

[3] J. Dunkel, A. Eberhart, S. Fischer, C. Kleiner, and A. Koschel, Systemarchitekturen fur Verteilte Anwendungen - Client-Server, Multi-Tier, ¨SOA, Event Driven Architectures, P2P, Grid, Web 2.0, 1st ed. Munchen: ¨ Hanser, 2008.

[4] M. Roberts. Serverless Architectures, 2018, [Online]. Available: https://martinfowler.com/articles/serverless.html

[5] B. Janakiraman. Serverless, 2016, [Online]. Available: https://martinfowler.com/bliki/Serverless.html

[6] Amazon Cognito User Pools. [Online]. Available: https://docs.aws.amazon.com/cognito/latest/developerguide/cognitouser-identity-pools.html

[7] M. Watson. What Is Function-as-a-Service? Serverless Architectures Are Here! [Online]. Available: https://stackify.com/function-as-a-serviceserverless-architecture/

[8] G. Peipman. Short introduction to serverless architecture. [Online]. Available: https://gunnarpeipman.com/serverless-architecture/

[9] L. Rowekamp. Serverless Computing, Teil 1: Theorie und Praxis. ¨ [Online]. Available: https://www.heise.de/developer/artikel/Serverless Computing-Teil-1- Theorie-und-Praxis-3756877.html?seite=2

[10] E. Zoumi, E. Skondras, N. Tsolis, A. Michalas, D. Vergados, "A Storage as a Service scheme for supporting Medical Services on 5G Vehicular Networks", International Conference on Information, Intelligence, Systems and Applications (IISA), Piraeus, Greece, July 15-17, 2020, pp. 1–6, doi: 10.1109/IISA50023.2020.9284339.

[11] E Skondras, A. Michalas, D. Vergados, "A Survey on Medium Access Control Schemes for 5G Vehicular Cloud Computing Systems", Global Information Infrastructure and Networking Symposium (GIIS), Thessaloniki, Greece, October 23-25, 2018, pp. 1–5, doi: 10.1109/GIIS.2018.8635661.

[12] E. Skondras, A. Michalas, N. Tsolis, A. Sgora, D. Vergados, "A Network Selection Scheme for Healthcare Vehicular Cloud Computing Systems", International Conference on Information, Intelligence, Systems and Applications (IISA), Larnaka, Cyprus, August 28-30, 2017, pp. 1–6, doi: 10.1109/IISA.2017.8316378.