

# FedCSIS\_2023\_paper\_3893.pdf

# A Serverless ETL process: Experiences from an Implementation of a FaaS-Architecture in the Core Banking Domain

3 Jens Kohler  
University of Applied Sciences, Worms  
Worms, Germany  
kohler@hs-worms.de

**Abstract**—The goal of this paper is to analyze experiences from a FaaS-based (function as a service) cloud native implementation of an ETL process based on AWS Lambda functions. The actual implementation is outlined and the experiences from that implementation are evaluated. This results in an overview of pros and cons of a cloud native implementation in general and determines best practices for other implementations in other cloud environments or business domains.

**Index Terms**—FaaS, AWS Cloud Services, Cloud Native, Infrastructure as Code, ELT/ETL Process, Data Lake, Core Banking Domain

## I. INTRODUCTION

Traditional ETL processes and their implementations stem from data warehouse approaches. Still, current data warehouses [21] [29] [1] offer possibilities to implement such ETL processes. But, although these implementations also include cloud-based capabilities, they often result in big monoliths.

To overcome such monolithic architectures, *Function as a Service* (FaaS) approaches offer the possibility to implement (business) functions that are operated natively in a cloud environment. The main difference compared to other cloud native services is, that resource pooling and scaling of the cloud resources is done automatically by the cloud provider. The cloud consumer only implements a function, deploys it to the cloud and defines the maximum amount of resources (i.e. RAM, CPU, etc.). Furthermore, these functions can be loosely coupled together to implement business processes, ETL processes, and the like.

The overall goal from a business perspective is changing the core banking system from the monolithic PASS Core Banking System [28] to the SaaS core banking system Mambu [15]. Within the scope of the migration, the connection to other involved source systems (e.g. Oracle Netsuite [27]) has also to be considered due to regulatory reporting obligations. All in all, data is read from the source system, transformed into the required target format and then pushed to the target system, which is in this case BAIS [15]. For this, the entire ETL process is implemented on Amazon Web Services, using the respective AWS cloud native services (Lambdas [10], CloudWatch [8], StepFunctions [11], etc.).

At this point it has to be noted, that the customer already used other services from the AWS cloud ecosystem. Therefore,

the FaaS architecture outlined in this paper was nailed to AWS ecosystem as well. Of course, in other contexts a dedicated cloud provider selection with a preceding evaluation has to be taken into consideration.

The goals of the entire FaaS architecture can be stated as follows (with no specific order or prioritization):

- import data from the source systems into an AWS S3 bucket [4]
- the imported data represent the foundation of a data lake
- transform data such that the target system accepts the data
- push the transformed data to the target system

Generally, this can be considered as an ETL process [18] with additional storage of data in a data lake.

## II. RELATED WORK

In this paper, the implementation of several AWS Lambda functions and their interconnection is outlined. Therefore, the different lambda functions are considered similar to *Microservices* [25], which implement the *Single Responsibility Principle* [20]. Hence, other architectural approaches that define *Microservice* architectures are considered feasible for the FaaS approach of this paper as well. An exhaustive overview of different migration approaches from monolithic to distributed *Microservice* architectures is given in [24]. Such a migration was also the main motivation for the work outlined in this paper.

Besides the afore-mentioned monolithic data warehouse approaches [21] [29] [1], often batch systems are used to extract, load and transform data from source to target systems. Nowadays, these batch approaches can also be implemented in *Microservice* architectures as outlined in [23] with the current state-of-the-art framework *Spring Batch*. However, the cloud integration with such frameworks has also to be considered here as extra effort, whereas in FaaS the cloud is inherent.

Besides the AWS universe, other cloud providers offer similar FaaS services for various programming languages (Java, Python, .Net, Scala, etc.), e.g. *Azure Functions* [22] or *Google Cloud Functions* [16].

Yet, there are dedicated cloud native batch systems, e.g. *AWS Batch* [7], that seem to be more suitable for processing large data volumes. However, the main difference to the more general FaaS approach is that the scaling and resource pooling

of these cloud native batch systems have to be defined in a more detailed level, which eventually means more effort.

### III. APPROACH

In order to address problem definition above, the following figure illustrates the entire architecture and its components in a high-level overview.

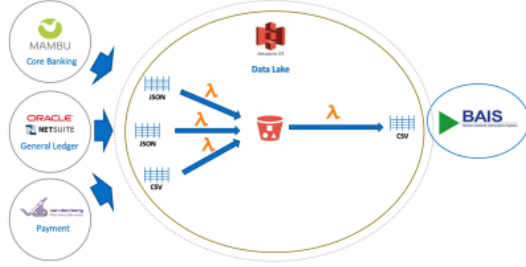


Fig. 1. High Level Architectural Overview

Basically, the AWS Lambdas query data from the source systems' APIs and store them in an AWS S3 bucket. The data formats of the source systems are imported into the S3 bucket *as-is*, meaning the different structures or formats are not changed. These data are the foundation of a data lake which later can be used for further integration projects, analysis, or for the ETL process that transforms the data such that the target system BAIS accepts them.

Furthermore, the data import to the data lake is triggered by the source systems via API call. Therefore, the AWS lambdas are connected to the AWS API Gateway [6]. This offers a well-defined HTTP gateway (via POST request, i.e. webhook) for the source systems for the data import. It also offers, in contrast to traditional batch systems which are often time-based triggered [23], this offers greater flexibility, as in principle the source systems are able to control when data are transferred to the AWS S3 bucket.

### IV. IMPLEMENTATION

This section gives a more detailed overview of the technical context in which the FaaS-based ETL process is operated.

Currently, the following source systems are integrated:

- Mambu [19]
- Oracle NetSuite [27]

The core components are structured along with the 3 proxies:

- Lambda-Proxy 1 (*s3-importer-proxy*) for the data import from the source systems into the AWS S3 bucket
- Lambda-Proxy 2 (*s3-transformer-proxy*) for the data transformation
- Lambda-Proxy 3 (*s3-exporter-proxy*) for the export of the transformed data to the target system BAIS

These core components communicate via AWS SQS Queues [3], which define the control flow for the lambdas.

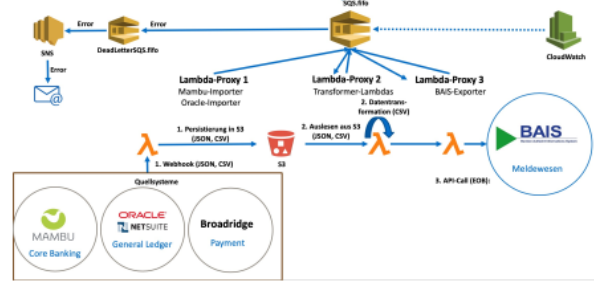


Fig. 2. Implementation Overview

**1** All lambda functions are implemented in Java 8. Generally, there are 3 main projects: *s3-importer*, *s3-transformer*, and *s3-exporter*, and these projects are structured in so-called Multi-Maven Projects [14].

These 3 main projects each contain a so-called parent pom.xml, which defines build and configuration parameters for the respective project. In this file, the version number among other configuration parameters is maintained. With respect to the Maven nature of the projects, a semantic versioning [26] of the components was introduced, following a major, minor, and incremental versioning schema.

As soon as the data import is completed, the *s3-importer-proxy* sends a notification to its queue: *DatalakeSQSImportQueue.fifo*. This notification starts the *s3-transformer-proxy* with the transformation of the data. As soon as it has finished the transformation, again a notification is sent to its FIFO queue: *DatalakeSQSTransformQueue.fifo*. Finally, this notification starts the *s3-exporter-proxy*, which transfers the transformed data into the target system BAIS. For the sake of completeness, again a notification is sent to its queue: *DatalakeSQSExportQueue.fifo*. The reason for the usage of different queues instead of one centralized is outlined in greater detail below in Section V. In short, the goal was to keep away as many logic as possible from the queues to not filter the messages in the respective lambdas. Such filtering would lead to a tighter coupling between the lambdas and the queue(s) and thus, it was avoided.

Here also comes a major restriction concerning the AWS Lambda functions into play. The lambda execution time per lambda is restricted to a maximum of 15 minutes (with a maximum of 10 GB RAM for each lambda). Hence, AWS Step Functions [11] were introduced to implement a subsequent ordering for the execution of the lambda functions. AWS Step Functions act as a state machine that allows the definition of an execution order for up to 25K transitions (i.e. lambda functions, errors and success messages, etc.).

In case of failures or timeouts, another communication channel via a dedicated so-called *Dead-Letter AWS SQS queue* was established. In contrast to the above-mentioned queues, here, one single queue for all errors is considered appropriate. As soon as this *dead-letter queue* receives a notification, an error notification is sent to an AWS SNS topic, which then

informs its subscribers via email that a failure during the ETL process has occurred.

Moreover, it has to be noted that in case of failures or timeouts, the current implementation does not contain any rollback or compensation mechanisms. This is considered feasible, as a restart of the ETL process will simply override every failing state or files stored in the S3 bucket.

Eventually, all log messages of the lambdas or proxies are written to AWS CloudWatch, which therefore acts as a central log where all activities during the ETL process can be viewed.

#### A. Typical Structures, Patterns, and Models

1) *Communication via Message Broker*: The 3 core components of this FaaS architecture communicate via AWS SQS queue in a synchronous manner. The communication is one-way, meaning that the *s3-importer-proxy* sends a notification to the *DatalakeSQSImportQueue.fifo* which triggers the *s3-transformer-proxy*. Then the *s3-transformer-proxy* sends a message to the *DatalakeSQSTransformQueue.fifo* which triggers the *s3-exporter-proxy*. In case of an error or failure, the *DatalakeSQSQueueDeadLetter.fifo* is notified, which sends a notification to an AWS SNS topic which sends a notification email eventually. This is also implemented synchronously, to have an easier and faster failover mechanism, than it would have been with asynchronous communication. However, the main reason is that different transformer lambdas rely on transformation results of other transformers. Thus, the synchronous calls allow the specification of a defined ordering of execution of the lambdas.

2) *Request/Response Communication*: The previous section determined the communication as synchronous, apart from asynchronous communication. Above that, the communication is implemented in a request/response model based on the HTTP status codes. Here, all HTTP status codes except 200 indicate errors and result in a notification via AWS SNS topic. This refers to all lambdas and their proxies, the API Gateways, and the Step Functions that are operated within the AWS environment.

3) *Persistence*: The basic persistence layer in the architecture is an AWS S3 bucket. This is also the foundation for a data lake. Generally, a bucket can be considered as a traditional (network) file system. Furthermore, it has to be noted, that the bucket has the AWS Versioning feature [13] turned on. This was introduced for the sake of better comprehensibility of the data in the buckets and because of backup and security reasons.

4) *Security & Privacy*: There is no customized implementation of any security or privacy related features. Yet, the standard security and privacy capabilities of the AWS cloud infrastructure were used. Namely, these are a 256 bit AES encryption on all used S3 buckets in combination with a blocking of all public internet access. Furthermore, all lambdas communicate within a virtual private network (VPN) with the source and target systems and with each other. Thus, a connection from outside the VPN or eavesdropping the communication is not possible.

1

5) *Graphical User Interfaces*: Finally, the implemented FaaS architecture does not implement customized graphical user interfaces. Here, the architecture fully relies on the AWS environment, namely the AWS Lambda Console, AWS CloudWatch, and the AWS S3-Bucket web interfaces that are accessible via web browser.

6) *Infrastructure as Code*: Last but not least, it has to be noted that the entire implementation and all developed lambdas can be deployed in a fully automated manner. Therefore, Terraform [17], an Infrastructure as Code framework was used. This framework offers 2 advantageous capabilities, first, the entire infrastructure and its components can be deployed automatically, and second, as the entire infrastructure is defined as code, it can be versioned as well.

This closes the implementation section of the paper and now this implementation is analyzed to gain pros and cons and to present best practices for similar implementations and architectures.

## V. EVALUATION

This section outlines and analyzes the experiences gained from the implementation of the previously described FaaS architecture.

#### A. Architectural Decisions

##### 1) *Single Bucket VS Different Buckets For Every Import*:

The following reasons led to the decision to use one single AWS S3 bucket instead of several ones:

- The bucket in which the FaaS architecture imports data is considered as the foundation for a data lake architecture. Thus, it is considered as a single source of truth. Here, the distributed character of several buckets (in extreme cases one different bucket for every import) is not considered feasible, as the advantages of such an architecture are not visible.
- The architecture uses one single bucket, but the data are structured analogously to a (distributed) file system with folders and subfolders. Thus, the current date (YYYY-MM-DD) is used as a guiding folder structure for the ETL imports.
- The AWS restrictions for buckets (e.g. 5 TB file size per bucket) [12] do not interfere with any requirements or with the bullet items depicted above.

##### 2) *AWS Lambda Architecture VS AWS Batch, Glue, etc.*:

This section shows the decision for the AWS Lambda architecture:

- AWS Lambdas can be locally tested on developers' laptops or development environments before they are pushed to the final AWS cloud environment.
- Compared to AWS Batch [7], AWS Lambdas have several advantages:
  - more lightweight (no provisioning (Docker, EC2, etc.) in advance required)
  - no dedicated execution environment (Docker, EC2, etc.) required

- more elastic: max. amount of storage and run time can be defined, scaling is done automatically by the AWS Lambda execution environment
- no additional licensing costs
- predefined execution environments (Java, JavaScript, Python) do not require additional configuration
- Compared to data warehouses like Matillion [21], Redshift [1], or Snowflake [29], AWS Lambdas have several advantages:
  - more lightweight: no initial set up required
  - different architectural approach: AWS Lambda are FaaS, Matillion, Redshift, etc. are complete data warehouses
  - highly adaptable and more customizable: lambda: develop your own function, data warehouse: develop an entire ELT-process
  - more elastic: max. amount of storage and run time can be defined, scaling is done automatically by the AWS Lambda execution environment
  - no additional licensing costs
- However, AWS Lambda suffer from one severe disadvantage:
  - the max. run time is restricted to 15 minutes

Especially for long-running transformation steps, this is a big disadvantage. The solution we have come up with to overcome this restriction is, that we have analyzed the transformation workload for each transformation step with respect to its data size. So generally, with an overall data size  $n$  for a transformation step, we know that within 10 minutes (with 5 minutes buffer) we are able to transform a data volume  $v$  (with  $v \subset n$ ). Then we store the current state  $c$  of the lambda (the current index, which is a counter which data has just been transformed) in a temporary bucket and call the same lambda again, that now begins its transformations at state  $c$ . As soon as all subsets  $v$  (of  $n$ ) have been transformed, we know that we are done and can continue with the next transformation step.

1  
3) *Data Import AS-IS VS Uniquely Transformed*: As the imported data serve as a foundation for a data lake, the imported data are kept in their source format. This reflects the common understanding of a data lake architecture, that is able to include several different source formats. The following reasons led to the decision to keep the data formats *as-is*:

- The imported data are already in common standard formats (JSON and CSV) and thus a transformation before the actual import might introduce another component, which might introduce errors during the import run.
- The advantages of a uniquely transformed data structure are not visible at the moment. It also helps to keep the 3 core components (*s3-importer*, *s3-transformer*, and *s3-exporter*) loosely coupled, as the importer imports data, the transformer transforms, and the exporter exports data (as their respective names suggest).

- Finally, this approach refers to the Single-Responsibility Principle, where every component serves exactly one purpose.

4) *Different SQS Queues VS Single SQS Queue*: As outlined in the architecture above, several AWS SQS Queues instead of a single centralized one were used. The main reasons for this decision were:

- Single responsibility principle  
With dedicated queues for each lambda core component, the lambdas themselves do not need any communication logic or logic that determines which message is for which lambda. Thus, every lambda can serve its dedicated purpose (either import, transform, or export)
- Avoid overloading the queue  
A single centralized queue would have to serve as a communication or message bus. This bus would have to analyze incoming messages and would have to guarantee that the messages were delivered to the correct recipient (in the correct order, in a defined time-frame, etc.). In contrast to this, several queues that serve exactly their dedicated purpose, are loosely coupled to their corresponding lambda proxy and establish a dedicated communication link between the two respective proxies. Thus, the communication is less error-prone and no advanced communication protocols, message exchange patterns, etc. are required.
- Flexibility  
Not only are the queues used for the communication between the lambda proxy components, but also for notifications in case of errors. Here, a single, centralized queue would have to implement logic to distinguish between different errors and failures. In contrast to this, dedicated queues can be attached easily to further error notification mechanisms and no logic for determining error causes, failures, etc. are required.
- Maintenance

Finally, an argument against the decentralized queuing approach could be maintenance. At first sight, maintaining several queues is more expensive than maintaining a single one. However, the advantages above show that this effort is well-invested and above that, all infrastructure components are maintained via Terraform scripts. This introduces a certain level of automatism which not only eases the setup, but also the maintenance of the components.

5) *Data Export: SQL VS CSV*: At the moment, the *s3-exporter* pushes imported and transformed data to the target system BAIS directly into the BAIS database via SQL. However, it is also possible to load data via CSV files into BAIS. The following reasons determine why the export via SQL was preferred:

- BAIS Environment  
Currently, the target system BAIS is operated within an AWS Workspace [12], which can be regarded as a remote desktop environment operated in the AWS cloud. This

AWS Workspace uses a dedicated file system, which is isolated within the AWS environment. Moreover, in and outbound traffic to this workspace is restricted. Thus, the problem of how to transfer files to the workspace occurred. Here, a script-based file copy (e.g. rsync, sftp, etc.) were considered as workarounds. Yet, these solutions would have needed an extra FTP server and apart from that, a script-based import of the copied or synchronized files into BAIS would also have been required. Therefore, pushing files to the BAIS database directly via SQL seemed more promising and expedient.

- Transaction Handling

Transaction handling via SQL (i.e. JDBC interface) offers more fine-grained transaction management capabilities. Thus, the exporter is able to manage the transaction handling on database, table, or even row level. In contrast to this, the CSV import of BAIS implements a plain all-or-nothing logic.

- Performance

Tests with different data structures and sizes showed, that the import via SQL is faster compared to the import via CSV.

- Flexibility

- A push mechanism that relies on CSV files would have resulted in a more human-readable and therefore more comprehensible format than the actual export via SQL does. However, the comprehensibility is still given, as the *s3-transformer* transforms the imported data into CSV (as a foundation for the data lake architecture). The preceding SQL statements are then derived from these CSV files.
- Another issue deals with the tight coupling of the exporter to the BAIS database. For this, dedicated and well-known JDBC standard drivers are used and as there exist for all commonly used database management systems respective drivers, the target database remains exchangeable with a minimum of required adaptations in the *s3-exporter* (ideally only the used JDBC database driver must be reconfigured).

6) *Single CodePipeline VS Dedicated CodePipelines*: The build-processes for the lambdas are managed by AWS CodePipelines [9]. For this, there is a dedicated CodePipeline for the *s3-importer*, another one for the *s3-transformer*, and a third one for the *s3-exporter* lambdas. This approach was preferred compared to a single centralized AWS CodePipeline that would have built all lambda components in one place:

- Flexibility

- The decentralized approach offers greater flexibility with code changes, adaptations, and bug fixes. Yet, the *s3-importer* can be adapted without touching the other core component lambdas (and vice versa). However, integration tests must prove that all changes are valid and working throughout the entire ETL process.

- Another issue concerning the greater flexibility comes with the development of features, bugfixes, etc. Here, every core component lambda can be developed, tested, and deployed independently.
- Separation of Concerns: different lambdas can be operated under different responsibilities (maintenance, test, development, etc.).
- Lambdas can be separated from their environment (e.g. development, staging, or production) more easily.

- Performance

- As the build processes are separated, each build process is able to run faster, compared to a centralized solution. Especially the development, testing, and deployment of hot fixes is faster.

7) *Scalability*: The data volume that is transferred between the source and target systems is  $\sim 3$  GB per day (note that the implemented ETL process is operated in an end-of-day interval). Currently, the entire process consists of  $\sim 35$  lambda functions. All in all, the daily operational costs sum up to  $\sim 80$  USD.

Scalability with respect to higher data volumes for the ETL process is given due to the subset-based processing of the data (stemming from the 15 min. lambda restriction). Yet, we have to evaluate how much a growing data volume affects the performance of the overall ETL process. The basic (naive, but realistic) assumption is, that the more data have to be processed, the more time will be required. This was considered a low priority issue, as our application domain (core banking) system with customers and their bank accounts is considered not to grow or shrink very fast.

Another issue regarding the overall performance is the transformation step. Here, independent data volumes can be processed in parallel. This is also a possibility that will be considered in the near future to increase the transformation performance and with that, the overall ETL performance.

## 1 VI. CONCLUSION

Based on the previous sections, the key characteristics of the FaaS implementation can be summed up to the following criteria and be compared to a traditional ETL process implementation, based on a data warehouse or a batch processing system.

TABLE I  
FAAS CHARACTERISTICS

| Criterion       | FaaS         | Traditional ETL |
|-----------------|--------------|-----------------|
| Flexibility     | higher       | lower           |
| Scalability     | higher       | lower           |
| Learning Effort | low          | high            |
| Performance     | <sup>a</sup> | <sup>a</sup>    |

<sup>a</sup>Depends on data volumes, source/target API performance, cloud network bandwidth, etc. In our case, described in this paper the overall ETL performance of the FaaS implementation was equal to the performance of a

traditional data warehouse ETL process, which eventually was 6.5 hours per ETL run.

Moreover, the following issues emerged during the course of the implementation:

- AWS Lambda triggers

With respect to the AWS SQS Queues that trigger the respective lambda functions after a notification, it must be noted that this communication mechanism has to be defined and configured very carefully. Otherwise, it might occur that lambda *A* triggers another lambda *B* which triggers another lambda *C* which eventually triggers lambda *A* again. This is a classical infinite loop known from all programming languages, yet it can become very costly, as every individual lambda call is billed.

- Monitoring the ETL process

Although, AWS CloudWatch is the centralized logging component, investigating single lambda or other AWS components can become cumbersome, as the only GUI is the AWS Console. Here, the browser-based handling with several tabs, partially misleading translations (if used) or the names of the metrics are things that a user has to get used to.

- Failure Handling

A common issue with ETL processes is that restarts in case of failures are cumbersome or not even possible. The main reason for this is to avoid complex and expensive error handling mechanisms. So a common approach is to restart the entire ETL process or sub-processes like the extraction, transformation or load processes of it. Unfortunately, also the FaaS architecture does not provide a more sophisticated approach for this issue. Here similarly, in case of a failure either the entire ETL or the sub-processes (extract, transform, load) must be restarted. For this, a more fine-grained solution would be desirable.

## VII. FUTURE WORK

Last not least, this section mentions issues that could be regarded as further improvements or as nice-to-have requirements and could be implemented in future versions.

### A. Rollback or Compensation Mechanism in Case of Failures During the ETL Process

Currently, there is no rollback or failure compensation mechanism implemented. If an error or failure during the ETL (*s3-importer*, *s3-transformer*, *s3-exporter*) process occurs, a notification is sent to an AWS SNS topic which then notifies its subscribers via e-mail. In such a case, partly imported or partly transformed data remain in the respective S3 bucket. They will be overwritten with the next ETL process. The *s3-exporter* implements a transaction logic, which pushes all transformed data or nothing to the target system BAIS.

### 1. SNS Notification with Payload

In its current implementation, in case of errors or failures during the ETL process, subscribers of the SNS topic are informed. However, they just receive a message claiming, that

the SNS topic got a notification. Unfortunately, the AWS SNS service offers limited capabilities to modify such messages [5]. Here, a replacement with AWS SES (Simple E-Mail Service) [2] or the like could be taken into consideration. However, it has to be noted that other services offer different service level agreements and message exchange patterns and this must be considered when exchanging this service is a requirement.

### C. Backup Strategy

Last but not least, especially for the contents in the buckets, there is no backup strategy implemented at the moment. The entire AWS infrastructure can be deployed via Terraform scripts and the current state of the scripts is preserved in an S3 bucket. This can be regarded as a backup for the infrastructure and its components. However, for the buckets themselves and especially their content, there is currently no backup strategy implemented. Here a strategy, that syncs the content on a daily basis would be recommended for the simplest case.

## REFERENCES

- [1] Amazon Web Services, "Amazon Redshift - Cloud Data Warehouse - Amazon Web Services," 2020. [Online]. Available: <https://aws.amazon.com/redshift>
- [2] —, "Amazon Simple Email Service Documentation," 2020. [Online]. Available: <https://docs.aws.amazon.com/ses>
- [3] —, "Amazon Simple Queue Service Documentation," 2020. [Online]. Available: <https://docs.aws.amazon.com/sqs>
- [4] —, "Amazon Simple Storage Service Documentation," 2020. [Online]. Available: <https://docs.aws.amazon.com/s3>
- [5] —, "Amazon SNS message and JSON formats - AWS SNS," 2020. [Online]. Available: <https://docs.aws.amazon.com/sns/latest/dg/sns-message-and-json-formats.html>
- [6] —, "AWS API Gateway Documentation," 2020. [Online]. Available: <https://docs.aws.amazon.com/apigateway>
- [7] —, "AWS Batch Documentation," 2020. [Online]. Available: <https://docs.aws.amazon.com/batch>
- [8] —, "AWS CloudWatch Documentation," 2020. [Online]. Available: <https://docs.aws.amazon.com/cloudwatch>
- [9] —, "AWS Code Pipeline Documentation," 2020. [Online]. Available: <https://docs.aws.amazon.com/codepipeline>
- [10] —, "AWS Lambda Documentation," 2020. [Online]. Available: <https://docs.aws.amazon.com/lambda>
- [11] —, "AWS Step Functions Documentation," 2020. [Online]. Available: <https://docs.aws.amazon.com/step-functions>
- [12] —, "Bucket Restrictions and Limitations - Amazon Simple Storage Service," 2020. [Online]. Available: <https://docs.aws.amazon.com/AmazonS3/latest/dev/BucketRestrictions.html>
- [13] —, "Using versioning - AWS S3," 2020. [Online]. Available: <https://docs.aws.amazon.com/AmazonS3/latest/dev/Versioning.html>
- [14] Apache Maven, "Maven - Guide to Working with Multiple Modules," 2020. [Online]. Available: <https://maven.apache.org/guides/mini/guide-multiple-modules.html>
- [15] BSM GmbH, "BAIS Webpage," 2020. [Online]. Available: <https://www.bsmgmbh.de/>
- [16] Google, "Google Cloud Platform Documentation," 2020. [Online]. Available: <https://cloud.google.com/docs>
- [17] HashiCorp, "Terraform by HashiCorp," 2020. [Online]. Available: <https://www.terraform.io/>
- [18] R. Kimball and J. Caserta, *The data warehouse ETL toolkit : practical techniques for extracting, cleaning, conforming, and delivering data*. Wiley, 2004.
- [19] Mambu GmbH, "Mambu Webpage," 2020. [Online]. Available: <https://www.mambu.com/>
- [20] R. C. Martin, *Clean architecture: a craftsman's guide to software structure and design*. New York, USA: Prentice Hall.
- [21] Matillion Ltd., "ETL Software Transform Your Cloud Data Warehouse," 2020. [Online]. Available: <https://www.matillion.com/>

- [22] <sup>1</sup> Microsoft, "Azure Functions Documentation," 2020. [Online]. Available: <https://azure.microsoft.com/en-us/services/functions/documentation>
- [23] <sup>1</sup> M. T. Minella, *The Definitive Guide to Spring Batch : Modern Finite Batch Processing in the Cloud*, 2nd ed. Chicago, IL, USA: Apress, 2019.
- [24] S. Newman, *Monolith to microservices : evolutionary patterns to transform your monolith*. Sebastopol, CA, USA: O'Reilly.
- [25] —, *Building microservices*. Sebastopol, CA, USA: O'Reilly, 2015.
- [26] Oracle, "Maven Version Numbers," 2020. [Online]. Available: [https://docs.oracle.com/middleware/1212/core/MAVEN/maven\\_version.htm](https://docs.oracle.com/middleware/1212/core/MAVEN/maven_version.htm)
- [27] <sup>1</sup> —, "Oracle NetSuite Webpage," 2020. [Online]. Available: <https://www.netsuite.com/portal/home.shtml>
- [28] <sup>1</sup> PASS Consulting Group, "PASS Core Banking System Webpage," 2020. [Online]. Available: <https://www.pass-consulting.com/en/industries/banking/core-banking-system>
- [29] <sup>1</sup> Snowflake Inc., "Your Cloud Data Platform," 2020. [Online]. Available: <https://www.snowflake.com/>

89%

SIMILARITY INDEX

PRIMARY SOURCES

1

Jens Kohler. "A Serverless FaaS-Architecture: Experiences from an Implementation in the Core Banking Domain", Institute of Electrical and Electronics Engineers (IEEE), 2021  
Crossref Posted Content

4439 words — 88%

2

eprints.hsr.ch  
Internet

17 words — < 1%

3

Daniel Bub, Laura Hartmann, Zdravko Bozakov, Steffen Wendzel. "Towards Passive Identification of Aged Android Devices in the Home Network", EICC 2022: Proccedings of the European Interdisciplinary Cybersecurity Conference, 2022  
Crossref

9 words — < 1%