LUT University

Strategies and Challenges in Cloud-to-Cloud Migration Using Infrastructure as Code

Ketonen Teemu, Smolander Kari

This is a Author's accepted manuscript (AAM) version of a publication

published by Springer, Cham

in Product-Focused Software Process Improvement. Industry-, Workshop-, and Doctoral Symposium Papers (PROFES 2024). Lecture Notes in Computer Science

DOI: 10.1007/978-3-031-78392-0_1

Copyright of the original publication:

© 2025 The Author(s), under exclusive license to Springer Nature Switzerland AG

Please cite the publication as follows:

Ketonen, T., Smolander, K. (2025). Strategies and Challenges in Cloud-to-Cloud Migration Using Infrastructure as Code. In: Pfahl, D., Gonzalez Huerta, J., Klünder, J., Anwar, H. (eds) Product-Focused Software Process Improvement. Industry-, Workshop-, and Doctoral Symposium Papers. PROFES 2024. Lecture Notes in Computer Science, vol 15453. Springer, Cham. https://doi.org/10.1007/978-3-031-78392-0_1

This is a parallel published version of an original publication. This version can differ from the original published article.

Strategies and Challenges in Cloud-to-Cloud Migration Using Infrastructure as Code

Teemu Ketonen[1] and Kari Smolander[2]

[1] Evenli, 00220 Helsinki, Finland, teemu@evenli.com
[2] LUT University, 53851 Lappeenranta, Finland, kari.smolander@lut.fi

Abstract. As the landscape of cloud computing continues to transform at a rapid pace, organizations seek to optimize operational costs, enhance system flexibility, and adjust to changing market demands. This paper reports a case study within a Finnish startup operating in self-service mobile payment systems. It provides a detailed analysis of migrating an application architecture from DigitalOcean to AWS (Amazon Web Services) using the AWS Cloud Development Kit (CDK), providing insights into the strategies, challenges, and outcomes of the process. The key findings suggest that while cloud migration involves considerable technical and operational challenges, the utilization of IaC (Infrastructure as Code) can significantly mitigate these obstacles, enhancing the speed, reliability, and security of the migration process. This research contributes to the limited literature on cloud-to-cloud migration, highlighting the effective use of IaC to facilitate transitions between cloud environments.

Keywords: cloud migration, cloud-to-cloud migration, infrastructure-as-code

1 Introduction

The cloud computing market is showing significant growth, projected to grow at a compound annual growth rate (CAGR) of 19.6% from 2021 to 2027. According to Gartner's 2024 forecast, the market valuation is expected to reach 1 trillion euros [1]. This rapid expansion highlights the fundamental shift in how businesses are leveraging cloud computing to boost operational efficiency, scalability, and cost-effectiveness. However, as market demands and dynamics change, organizations are required to continuously refine their IT strategies. This often involves re-evaluating the used cloud service providers to ensure alignment with current requirements for cost, performance, and features.

Consequently, cloud-to-cloud migration, involving the transfer of applications, data, and services between cloud platforms, has become an important concept. This process presents numerous challenges, including the need to maintain system integrity and minimize operational disruptions. Additionally, it demands careful consideration of technology choices, strategic planning, and the implementation of best practices to guarantee smooth and secure transitions.

Previous studies have extensively explored individual aspects of cloud migration, such as security concerns, downtime minimization, and cost efficiency. However, there

is a noticeable gap in comprehensive empirical research focused specifically on cloud-to-cloud migrations [2]. Studies have predominantly concentrated on migration from on-premises infrastructures to the cloud, rather than migrations between cloud platforms. This leaves a critical knowledge gap regarding the unique challenges and opportunities that arise during cloud-to-cloud transitions, such as interoperability issues, data loss prevention, and the maintenance of service availability during the migration process. Furthermore, as highlighted by Kolb et al. [2] more case studies are required in the cloud-to-cloud migration area. Linthicum [3] also supports this claim by mentioning that only a few cloud-to-cloud transitions have happened. We tried to find more recent studies on cloud-to-cloud transitions, but found only a single master thesis [4].

Additionally, Infrastructure as Code (IaC) has become an emerging approach that facilitates the management and deployment of computing infrastructure through codified files. This approach significantly reduces the likelihood of errors compared to traditional manual configuration methods. IaC is particularly advantageous in automating processes, minimizing errors, and enhancing reproducibility, which are essential for efficiently managing complex or large-scale environments. Despite its potential benefits, the specific challenges and strategies involved in implementing cloud-to-cloud migrations utilizing IaC remain underexplored. This research aims to address this gap by investigating these challenges and evaluating the impact of IaC on the cloud-to-cloud migration process.

The goal of this research is to conduct a detailed analysis of cloud-to-cloud application migration, with a specific emphasis on the utilization of Infrastructure as Code to streamline this process. The study aims to contribute valuable insights to the academic field and provide practical insights for organizations planning cloud-to-cloud migration initiatives. The study sets out to identify the key challenges associated with cloud-to-cloud application migration, to evaluate the effectiveness of IaC in addressing these challenges. The research will use a case study methodology to answer the following research questions:

RQ1: What are the key challenges, strategies, and outcomes involved in migrating an application architecture between cloud providers?

RQ2: What are the benefits and challenges of using IaC in the migration process?

Certain limitations have been established to focus the scope and maintain clarity throughout the investigation of cloud-to-cloud migration using IaC. First, it is important to note that this study does not aim to compare or evaluate different cloud architectures or analyse the decision-making processes behind choosing to switch cloud providers. Such evaluations are beyond the scope of this paper and involve a broader set of strategic considerations that are not addressed here. Additionally, the variability in cloud architectures and the specific configurations of services pose a significant limitation. The findings and strategies discussed may not be universally applicable to all cloud environments due to differences in infrastructure and service setups. Each cloud provider offers unique features and configurations, which could affect the transferability of the proposed solutions.

Furthermore, this research does not address the lock-in effect often associated with cloud providers. The lock-in effect can influence the ease or difficulty of migration processes, as proprietary technologies and platform-specific services, such as the IaC implementation with AWS CDK in this case, can create dependencies that complicate migration efforts. This aspect is not considered within the scope of this paper, which could affect the applicability of the strategies in environments where lock-in is a significant factor.

2 Background

2.1 Cloud migration and cloud-to-cloud migration

The benefits of cloud computing drive organizations to move their legacy applications running on-premises infrastructure to the cloud. This process of moving applications or any parts thereof to the cloud is called cloud migration [5]. The key drivers for the migration include decreased starting and operating costs, efficient resource utilization, scalability, and easier maintainability [5, 6].

Cloud migration has been studied extensively in the literature from many perspectives. Jamshidi et al. [5] conducted a systematic literature review on previous cloud migration research and found articles reporting their experiences, lessons learned, and best practices. The existing research highlights cloud migration as a complex process that necessitates thorough planning and the evaluation of multiple factors to guarantee a secure and smooth transition to the cloud [7].

While existing research predominantly addresses the migration of servers from on-premises to cloud environments, the critical aspect of migrating applications between different cloud service providers remains significantly underexplored [2]. Linthicum [3] also supports this claim by mentioning that only a few cloud-to-cloud transitions have happened. Furthermore, Yussupov et al. [8] emphasize the limited research available on cloud-to-cloud migration, particularly within the serverless computing context. Additionally, there is no joint agreement in the literature for the definition of cloud-to-cloud migration. Hussein et al. [9] define the concept of cloud-to-cloud migration as moving infrastructure or applications from one cloud provider to another. Kolb et al. [2] also appears to take the same stance in their research paper. In contrast, Narantuya et al. [10] describe cloud-to-cloud migration as moving virtual machines from one cloud provider to another, which closely relates to the concept of live migration. This paper uses the definition outlined by Hussein et al. [9].

2.2 Infrastructure-as-Code (IaC)

IaC is a core technique in DevOps and cloud computing, where infrastructure is managed and provisioned through code [11]. This replaces the error-prone manual configuration by system administrators with a structured and codified approach [12]. IaC enables DevOps teams to collaboratively automate the management, monitoring, and provisioning of resources in a cloud environment. This approach increases efficiency, reduces manual errors, and enables faster deployment cycle of applications. IaC treats

infrastructure as if it were software. Thus, organizations can ensure consistency in environments, streamline development and deployment processes, and achieve more scalable, manageable, and replicable infrastructure setups [13].

IaC can be implemented using various languages and tools, each with its own strengths and use cases. Popular choices include declarative languages, such as YAML, which is known for its simplicity and readability. Furthermore, programming languages like Python and TypeScript can also be used to define infrastructure for IaC, offering greater flexibility, modularity, and control [11]. According to a study by Rahman et al. [13], Chef and Puppet were found to be most frequently used IaC tools. Some other popular examples include Terraform, which supports IaC for various cloud providers and on-premises infrastructure [14]. In addition, AWS provides their own tool called CloudFormation, which is written with YAML or JSON to create and manage resources on their platform. Furthermore, AWS offers the Cloud Development Kit (CDK), which builds on CloudFormation's capabilities, allowing infrastructure to be defined in programming languages like Python or TypeScript before being deployed to AWS's cloud environment [15].

3 Research process

The research adopts an exploratory case study methodology, focusing on a single case of cloud-to-cloud migration. This method is chosen due to its strength in providing in-depth and detailed insights into “contemporary and complex phenomena within their real-life contexts” as described by Yin [16]. The case study method allows for an exploration of the cloud migration process in its natural setting, enhancing the understanding of the dynamic processes involved.

The data is collected through direct observations made by the first author, who actively participated in the migration project. We acknowledge the limitations of the case study approach. Generalizability of findings may be limited due to the focus on a single case, impacting external validity [17]. However, the research will aim to contribute to the broader understanding of cloud migration between providers by providing a rich and detailed case study with in-depth insights.

3.1 Case Organization

The case organization is a self-service mobile payment platform start-up located in Finland. The company was founded in 2023 and currently has 8 employees. Since its foundation, the company has released solutions for retail and restaurant industries. In the retail sector, it introduces a user-friendly Scan & Go model, enabling customers to effortlessly scan and purchase items using their own phones, thereby improving the shopping experience with its mobile application. Additionally, the company has released a similar service for student restaurants. In Finland, higher education students are required to show their student card to be eligible for a student discounted price. The case company has released an innovative solution within their mobile application, designed specifically for students. This feature enables users to seamlessly link their student

status to the app. Once connected, students gain the ability to purchase meals at a discounted rate. At checkout, customers present a QR code from their receipt to a payment terminal at the cashier. This streamlined process not only facilitates savings for students but also enhances the overall dining experience with convenience and efficiency.

The first author of this paper worked as the Lead Developer in the case company with 9 years of experience. To streamline their development process, the development team employed the Kanban methodology, ensuring efficient workflow and continuous improvement. The team had no previous experience with AWS environment nor IaC technologies.

The initial environment for the application was running on a single DigitalOcean droplet, which is a virtual machine operating on Ubuntu version 20.04. The lifecycle of the application was managed with Docker Compose. It enabled efficient management of services, networks, and volumes through a single YAML file, allowing for the creation and initiation of services with one command. Within the application's architecture, each service was encapsulated within its own Docker container. The architecture consisted of a backend, PostgreSQL database, Redis cache, a reverse proxy, web dashboard interface, a mobile application, and a database management service.

Overall, the existing architecture was not too complex, which made it easy to manage and migrate to another cloud platform. The selected cloud provider for the migration was AWS, which is one of the most popular cloud computing platforms. However, there are also a few pain points, which are identified with the current implementation that should be also resolved during the migration process. These challenges are shown in Table 1.

Table 1. Issues in the current environment

#	Issue
P1	Updates produce downtime while the services are restarting
P2	No automated provision of version updates
P3	Manual management and control over the infrastructure is prone to errors
P4	No load balancing or scalability available
P5	Configuration drift between production and development environments
P6	Secrets are managed in environment specific files, which are hard to manage and error-prone
P7	Security configuration and management are insufficient
P8	Monitoring is difficult
P9	Checking service logs is cumbersome since they must be accessed through SSH and terminal

When migrating the whole existing architecture to AWS, the challenges must be considered when planning the overall architecture. Also, the infrastructure cannot be moved as is, but requires careful consideration in the architecture to tackle these challenges. Additionally, the implementation of the infrastructure will leverage IaC to ensure efficiency and consistency. Furthermore, compliance with GDPR must be carefully integrated and maintained throughout the migration to ensure adherence to data protection regulations.

4 Planning and Implementing the Migration

4.1 Migration Strategy Selection

For the migration to AWS, two primary strategies were evaluated: **replatforming** and **refactoring**. Replatforming means selective improvement of the application through the integration of cloud-specific features without changing its fundamental architecture. This focuses on optimizing performance and maintenance by utilizing cloud capabilities, such as auto-scaling, managed storage and databases, and monitoring [3]. This approach aims to leverage the immediate benefits of the cloud's scalability and managed services without requiring a fundamental overhaul of the existing application infrastructure.

In contrast, the refactoring strategy would involve making the application architecture cloud-native. Cloud-native means that the application is designed for the cloud from the start, leveraging its scalability and resilience [18]. Although this approach offers potential long-term operational and performance benefits, it requires a significant commitment of resources, time, and expertise to re-engineer the application from the ground up [3].

The transition towards a cloud-native architecture, despite its potential benefits, was not favorable due to the extensive effort required. Consequently, the replatform strategy was chosen to proceed with. It is both cost-efficient and does not require major development work. Cloud features will be utilized as necessary to address the identified challenges, focusing on streamlining and simplifying management and maintenance tasks. While it was also technically feasible to migrate the infrastructure to AWS with minimal adjustments, this path was not preferred since it would not solve the previously listed challenges.

4.2 Selection of AWS Services and Resources

The service mapping is shown in **Fig. 1**. The Redis instance was replaced with AWS's fully managed Amazon ElastiCache service for Redis. Furthermore, the bare docker container running PostgreSQL is replaced with an RDS instance, which is AWS managed relational database service. These two services can be thus replaced with a drop-in replacement without changes needed in other parts of the application except changing the connection values to reflect the new services.

The previous NGINX reverse proxy was completely replaced with AWS's cloud offerings. The dashboard web application and backend service both require a domain name for easy access over the internet. For this purpose, Amazon Route53 was used to provide a reliable way to provide domain name system (DNS) routing for these services. Route 53 redirect traffic to the appropriate services, ensuring smooth integration with the broader suite of AWS services.

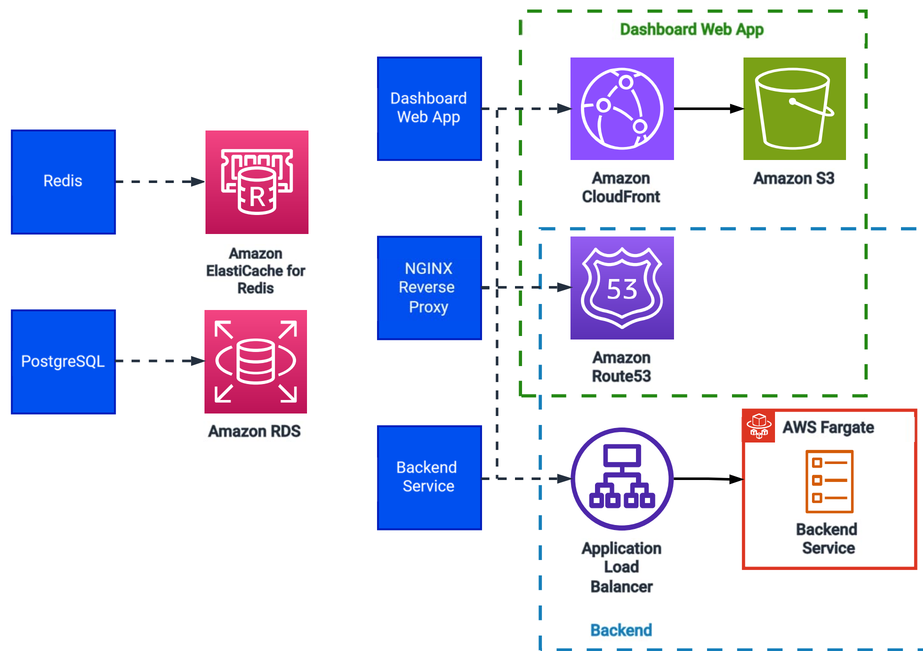

Fig. 1. Mapping of cloud services to new platform.

The dashboard web application utilizes the AWS CloudFront content delivery network (CDN), which comprises a network of servers distributed across various locations. These servers cache content in proximity to end users, allowing content to be delivered from servers closest to users. Furthermore, the dashboard application content is placed on Amazon Simple Storage Service (S3), which is a scalable data storage service. The build artifact of the Flutter application is uploaded to S3 and served over the CloudFront service to end-users.

The backend service was placed in Amazon Fargate, which is a serverless compute engine to run containerized applications. Fargate enables the deployment of containers without having to manage the underlying infrastructure (e.g., updates and configuration). Furthermore, Fargate can be easily configured to scale without much configuration. It also operates on pay-per-use model, where charges are incurred only for the containers actively utilized [19]. Fargate was selected over Amazon ECS for the easier and simplified management, offering solutions to both issues **P1** and **P4**.

Fargate directly integrates with AWS managed Application Load Balancer (ALB), which forwards the traffic between containers and handles load balancing. Together these services can be configured to automatically scale and manage revisions of new deployments without service downtime. A load balancer distributes incoming traffic across multiple targets, such as containers, ensuring high availability and scalability for applications. Additionally, Amazon Elastic Container Registry (ECR) is used to upload the Docker container images and handle versioning.

The overall planned architecture in AWS is visualized in **Fig. 2**. The Fargate service is placed inside of a private subnet with egress, meaning that it can access the internet,

which is required to call third-party APIs like contacting the payment service. This requires using a Network Address Translation (NAT) service, which allows the service to contact outside network while it is placed in a private network. Additionally, RDS and ElastiCache will be placed in private network, meaning that they cannot be accessed from outside of the virtual private network, which provides additional security.

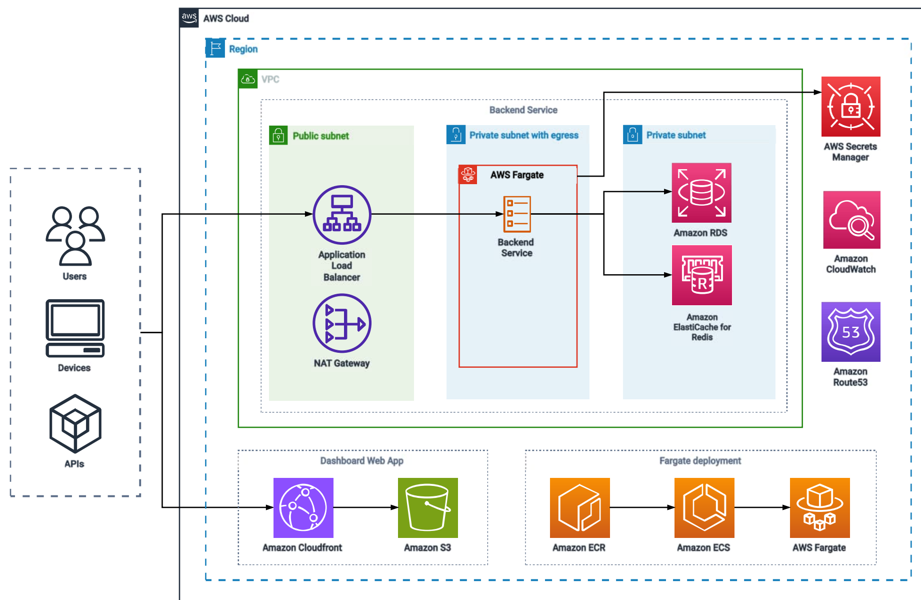

Fig. 2. Planned architecture in AWS.

AWS also provides a monitoring and observability service named Amazon CloudWatch. CloudWatch tracks the health and performance of cloud resources deployed on AWS. It collects metrics on how resources are performing (e.g., CPU and memory usage) and generated logs to facilitate troubleshooting and identifying errors. CloudWatch can be easily accessed in AWS dashboard where metrics can be visualized for usability. For example, performance monitoring is provided for RDS, ElastiCache, and Fargate services. Consequently, the adoption of CloudWatch addresses both issues **P8** and **P9**.

The management and protection of environment secrets and configurations are centralized within AWS Secrets Manager. These secrets can be things such as database credentials, API keys, OAuth tokens. This strategic approach directly addresses concern **P6** by safeguarding sensitive data through enhanced security practices and streamlined secret management processes.

Overall, the integration of AWS managed resources with built-in security features, private networking, data encryption, and Secrets Manager collectively increases security measures, directly addressing the mentioned **P7** issue. Furthermore, the infrastructure will be strategically deployed in the Stockholm region (codenamed eu-north-1),

aligning with GDPR requirements. This location is also chosen for its geographical proximity to Finland, where the case organization operates, to optimize performance.

4.3 IaC Utilization

The described infrastructure was fully constructed using IaC to automate both management and deployment of the infrastructure through code. Among the various IaC tools available, such as AWS CloudFormation, AWS Cloud Development Kit (CDK), Terraform, Ansible, Chef, and Pulumi, CDK was selected for the implementation. This decision was influenced by CDK's compatibility with TypeScript, allowing seamless integration with the existing codebase also written in TypeScript. Although opting for CDK introduces a potential dependency on AWS, posing a risk of vendor lock-in, this risk is considered manageable for the project.

Given the fundamental role of IaC in DevOps practices, the development of CI/CD pipelines was also planned. This strategy directly tackles the challenges identified as **P2** and **P3**, ensuring a streamlined process for continuous integration and delivery. In theory this should avoid infrastructure drift between development and production environments as identified by [20], addressing the concern raised in **P5**.

One pipeline was developed per each application: backend and dashboard. Both applications are hosted on GitHub each in their own repository. The pipelines were configured using GitHub Actions, which allows automating the build, test, and deployment pipeline directly within a GitHub repository. For the development environments the pipeline is triggered when new commits are pushed to the repository. However, for added stability, the same setup for the production environment is only triggered when a new release is created.

When the pipeline is triggered, the process begins with the execution of continuous integration tasks. This includes building the application and then conducting unit testing to verify the functionality of the code. If this step is successful, the infrastructure will be deployed to AWS. CDK automatically checks if there are any changes in the code and proceeds accordingly with the infrastructure provision. In the backend deployment, this setup continues to handle database migrations, which are deployed to reflect changes in database schema. A docker image is then built and pushed to Amazon ECR for versioning. Following this, the new container image is deployed to ECS which starts a new container deployment. Upon the successful completion of this deployment, the previous deployment is terminated. This seamless transition ensures that there is no service interruption when the deployments are exchanged. This process is visualized as an example in **Fig. 3**.

These pipelines lead to significant improvements in managing infrastructure changes and service updates, aligning with DevOps principles. They promote an automated and efficient process towards continuous deployment. As a result, both backend service and dashboard web application can receive timely updates and maintenance. Additionally, the IaC implementation presents an agile environment, which can quickly adapt to changes.

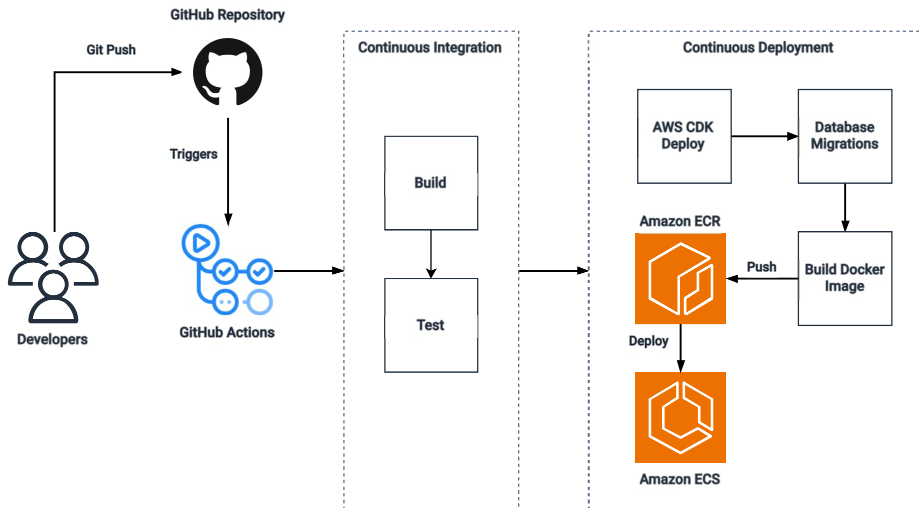

Fig. 3. DevOps pipeline for the Backend service

4.4 Migration Process

The migration process spanned a duration of 15 days, during which the infrastructure code was organized across two distinct repositories, dedicated to the backend and the dashboard respectively. The migration required some code modifications in the backend service to enable horizontal scaling provided by Amazon Fargate. Furthermore, WebSocket connections needed to be reconstructed to support a distributed workload among multiple container instances.

Implementing IaC enabled the development team to efficiently construct their infrastructure within a new cloud environment, significantly accelerating development and progress. Defining the infrastructure through IaC was found to be more intuitive and easier to navigate than the traditional graphical user interface (GUI) in AWS, due to its declarative nature. Consequently, this approach facilitated a quicker learning curve.

IaC enabled fast deployments of new infrastructure configuration and resources together with the DevOps pipelines. These increased the reliability of deployments as they did not require manual effort, which is prone to errors. However, the feedback loop for infrastructure deployments was found to be quite long. It took on average 15 minutes to deploy the whole application infrastructure. And since problems could only occur in the last stack, this could cause an unnecessary long feedback duration to just deploy a small change.

The process of migrating to AWS proved to be relatively straightforward. Most of the services seamlessly transitioned to AWS-managed services as drop-in replacements. With little changes, the infrastructure was able to take a lot of advantage of the features available in cloud, such as security, scalability, and ease of management. However, tailoring the backend service to support scaling required additional effort, highlighting the complexity of ensuring scalability in cloud-based architectures. Also,

circular dependencies caused some troubles in the migration. The error messages in CDK were helpful in pointing out to the right direction to solve these problems.

The migration performed well against the initial plan. Only some changes were necessitated by placing the database in private network, introducing a bastion host as an intermediary access point. Furthermore, the problems found in the previous implementation were properly addressed in the new architecture, facilitating new ways of working. The benefits and challenges of the migration are summarized in Figure **Fig. 4**.

Benefits	Challenges
• Easier monitoring and observability • Improved security posture • Prevent infrastructure drift • Automated deployments • Reduced human error • Easier adaptation with IaC compared to GUI • Code as documentation	• Circular dependencies • New cloud environment and IaC tool learning curve • IaC feedback loop • Technical complexity

Fig. 4. Benefits and challenges of the migration

5 Discussion

The infrastructure migration to AWS was successfully completed according to the plan, underscoring the critical importance of careful and strategic planning. This process involved thoughtful decisions to tackle the technical problems present in the existing architecture, particularly through a detailed strategy for selecting and integrating AWS services as replacements for the old systems. However, the migration process was also characterized by a series of challenges that required strategic decisions for successful completion. One of the primary challenges encountered was the need to modify the backend service to enable horizontal scaling. These kind of portability issues are a key challenge in cloud migration as mentioned by Kumar & Kumar Garg [21] and Linthicum [3].

Strategically, the use of IaC emerged as a key factor for managing the migration complexity. However, the outcomes of the adoption of IaC were multifaceted. On one hand, the migration achieved its technical objectives, with the new cloud infrastructure supporting scalable, distributed applications and leveraging AWS managed services for enhanced security and management. On the other hand, the process presented the extensive learning associated with adopting a new cloud platform and tools, namely AWS and CDK.

The effective utilization of IaC with AWS CDK played a key role in automating and managing the infrastructure during the migration. IaC allowed to share infrastructure between development and production environments, thus preventing infrastructure drift, which supports the findings of Morris [20]. The same code was used to deploy the two environments, although with different configurations. The rapid provisioning

of infrastructure with DevOps pipelines allowed for continuous deployment, which led to less risk of human errors in deployment of new infrastructure and reduced the complexity and time taken to conduct updates. This also facilitated the testing process, as small changes could be made without significant effort.

Building infrastructure in a completely new cloud platform for the development team using IaC allowed for quick development and progression compared to accomplishing the same the traditional way. This infrastructure creation process was also facilitated by code examples, which were easier to follow through than tutorials explaining how to navigate the AWS dashboard. This resulted in a reduction in complexity, making it easier to develop and manage infrastructure using IaC, thereby supporting the claims by Morris [20]. Furthermore, in alignment with the findings presented by Werner et al. [22], this study also found that the infrastructure code also functioned as documentation due to its declarative nature. IaC also allowed the validation of the infrastructure through the inspection of the code, which Morris [20] also mentions.

However, the adoption of IaC was not without its challenges. The feedback loop for infrastructure deployments remained a bottleneck due to the time-intensive nature of deploying the entire application infrastructure. However, the division of the infrastructure into logical, smaller stacks enabled faster feedback loops and facilitated the migration and testing process. Furthermore, the occurrence of circular dependency errors highlighted the complexities of managing interdependent resources through IaC, requiring advanced troubleshooting and configuration strategies. These observations also align with the challenges presented by Guerriero et al. [12].

The transition to a new cloud platform and toolset introduced significant challenges due to the development team's unfamiliarity with these technologies. To overcome this skill gap, efforts on learning and training were deemed important, as also noted by Guerriero et al. [12] and Shuaib et al. [23]. The transition required extensive individual learning efforts by going through AWS resources and other related material. Additionally, the complexity of the migration introduced various technical challenges, which was described as one of the core challenges in cloud migration by Staevsky and Gaftandzhieva [24]. However, comprehensive, and strategic planning played a key role in pre-emptively addressing these issues, effectively mitigating the potential for unexpected problems.

It is also notable that while the cloud application is relatively easy to transform to other cloud service provider, the IaC implementation caused a direct portability issue, effectively leading to a vendor lock-in. The CDK tool has been developed by AWS to only function in their ecosystem. Consequently, if the architecture were to be deployed elsewhere, a similar effort would be needed to perform the migration. Guerriero et al. [12] and Kumar & Kumar Garg [21] have also raised similar concerns related to vendor lock-in. However, the overall portability effort required to move to the new platform was not deemed significant as most resources could be moved to AWS with a drop-in replacement.

Rahman [25] raised concerns that the adoption of IaC increases the chance of forgetting about access control measures and firewall rules during the infrastructure setup. However, in this case, where CDK was being utilized, the results were contradictory. This is due to the inherent nature of CDK to apply the strictest rules by default, meaning

that access needs to be explicitly stated. This security-focused design principle is considered throughout the framework. Consequently, security was not considered a significant challenge, but instead the focus was on verifying the security functionality through code inspection and by applying the best practices.

The findings suggest that while cloud migration involves considerable technical and operational challenges, strategic utilization of IaC can significantly mitigate these obstacles, enhancing the speed, reliability, and security of the migration process. However, the effectiveness of such strategies is dependent upon the development team's familiarity with the cloud platform and IaC tools, as well as their capability to adapt to new technologies. Furthermore, the use of IaC can initially pose more challenges but can be beneficial in the long run. These findings highlight the need for comprehensive training and resources with necessary knowledge to successfully conduct the migration.

This study contributes to the limited body of knowledge on cloud-to-cloud migrations. The migration project, through its challenges and successes, provides valuable insights into the dynamics of cloud-to-cloud migration and the instrumental role of IaC. These insights offer guidance for organizations navigate their cloud migration process while also illustrating the benefits of leveraging IaC for infrastructure management.

6 Conclusions and Lessons Learned

This paper explored the strategies, challenges, and outcomes involved in migrating an application architecture between different cloud providers. Furthermore, the impact of leveraging IaC in the migration process was evaluated. The study was executed as a case study within a small Finnish startup operating in self-service mobile payment systems, offering a detailed examination of the impacts and experiences.

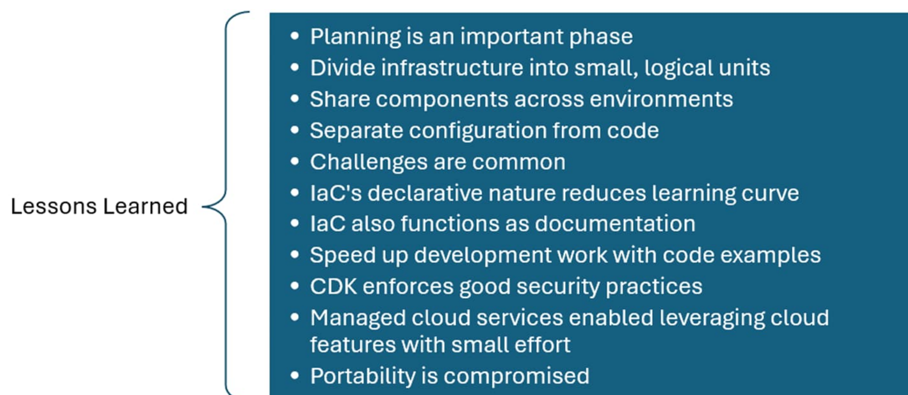

Fig. 5. Lessons learned in the migration process.

Many insights were learned from the migration process, which are helpful for practitioners when planning similar transitions (see **Fig. 5**). We will mention the most

essential ones here. First, the importance of planning cannot be understated. This sets the path for the rest of the migration process, ensuring its successful execution. Furthermore, thorough planning helps to discover possible problems and risks associated with the migration process. For example, the absence of fail-over considerations could have led to significant service disruptions. Additionally, leveraging the Cloud-RMM migration framework [5] facilitated a comprehensive assessment of migration objectives and tasks, ensuring a structured and strategic approach to the transition process. This framework helps in identifying key areas for consideration and sets a clear roadmap for successful migration execution.

It is also important to note that not all challenges can be avoided; they are bound to happen. For example, circular dependency errors are a common occurrence when infrastructure is divided into smaller parts referencing each other. What is important though, is the knowledge that these errors can happen and how to solve them. In addition, best practices should be followed to minimize the number of challenges that happen during the development and maintenance phases. These will also reduce the total ownership cost in the long run. In this aspect, the division of infrastructure into small, logical stacks is important. As previously mentioned, resources within stacks are deployed collectively as a single unit. This means that if a change is made within a stack, the whole stack needs to be redeployed. For example, services like a database are persistent and should not require changes often. This makes it a good choice to separate it in its own stack or to one with the same kind of requirements.

This approach also helps in making the infrastructure deployment break as fast as possible, thus reducing the provisioning time. Deploying the entire infrastructure as a single stack could lead to prolonged feedback loops, where errors might only occur after deploying the last resource, resulting in significant delays. However, dividing the infrastructure into stacks allows for independent deployment. Consequently, if errors occur, only the affected stacks require redeployment after implementing necessary fixes, optimizing efficiency.

Infrastructure components should be shared across different environments while retaining the ability to configure them independently. This means that, for example, both production and development environment use the same infrastructure, but with differing configurations. For instance, this allows for dedicating less expensive computing resources to the development environment, optimizing resource utilization. Additionally, the same principle should be also applied when dealing with sensitive information, such as secrets and API keys.

It was also noted that IaC has a lower learning curve than the traditional way of declaring infrastructure via a website interface. This is due to the declarative nature of IaC, which makes it easy to understand for developers by just glancing through the code. This means that the code should be self-documenting, and the infrastructure should be easily verifiable by reviewing the code. Furthermore, given this characteristic, it is highly recommended to follow documentation and utilize the existing code examples to speed up the development work.

Another key lesson related to placing the database inside a private network. During the planning phase, it was not noticed that this would also affect the management of the database, as it could not be accessed publicly. This required the introduction of a bastion

host, that acted as an intermediary to access the database. Additionally, security considerations were important throughout the migration process. CDK simplified this task by enforcing the strictest access controls by default, facilitating the seamless application of essential security practices.

Lastly, switching to managed cloud services allowed to capitalize on cloud features with minimal effort. Some code modifications were required to adjust to the new cloud computing platform. This enhanced monitoring and observability allowed to pre-emptively fix potential problems. However, this aspect of building for a specific cloud provider compromises portability. Moreover, the utilization of IaC intensifies this issue, leading to a significant risk of vendor lock-in.

References

1. Gartner: Forecast: Public Cloud Services, Worldwide, 2021-2027, 2Q3 Update. (2024).
2. Kolb, S., Lenhard, J., Wirtz, G.: Application Migration Effort in the Cloud - The Case of Cloud Platforms. (2015). https://doi.org/10.1109/CLOUD.2015.16.
3. Linthicum, D.S.: Cloud-Native Applications and Cloud Migration: The Good, the Bad, and the Points between. IEEE Cloud Computing. 4, 12–14 (2017). https://doi.org/10.1109/MCC.2017.4250932.
4. Paajanen, R.: Framework for Seamless Cloud-to-Cloud Service Migration, (2024).
5. Jamshidi, P., Ahmad, A., Pahl, C.: Cloud Migration Research: A Systematic Review. IEEE Trans. Cloud Comput. 1, 142–157 (2013). https://doi.org/10.1109/TCC.2013.10.
6. Rai, R., Sahoo, G., Mehfuz, S.: Exploring the factors influencing the cloud computing adoption: a systematic study on cloud migration. SpringerPlus. 4, 197 (2015). https://doi.org/10.1186/s40064-015-0962-2.
7. Gholami, M.F., Daneshgar, F., Low, G., Beydoun, G.: Cloud migration process—A survey, evaluation framework, and open challenges. Journal of Systems and Software. 120, 31–69 (2016). https://doi.org/10.1016/j.jss.2016.06.068.
8. Yussupov, V., Breitenbücher, U., Leymann, F., Müller, C.: Facing the Unplanned Migration of Serverless Applications: A Study on Portability Problems, Solutions, and Dead Ends. In: Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing. pp. 273–283. Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3344341.3368813.
9. Hussein, N.I., Hashem, M., Li, Z.: Security Migration Requirements: From Legacy System to Cloud and from Cloud to Cloud. Presented at the 2nd International Symposium on Computer, Communication, Control and Automation April (2013). https://doi.org/10.2991/3ca-13.2013.62.
10. Narantuya, J., Zang, H., Lim, H.: Service-Aware Cloud-to-Cloud Migration of Multiple Virtual Machines. IEEE Access. 6, 76663–76672 (2018). https://doi.org/10.1109/ACCESS.2018.2882651.
11. Sokolowski, D.: Infrastructure as code for dynamic deployments. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1775–1779. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3540250.3558912.

12. Guerriero, Mi., Garriga, M., Tamburri, D.A., Palomba, F.: Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry. In: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 580–589. IEEE, Cleveland, OH, USA (2019). https://doi.org/10.1109/ICSME.2019.00092.
13. Rahman, A., Mahdavi-Hezaveh, R., Williams, L.: Where Are The Gaps? A Systematic Mapping Study of Infrastructure as Code Research. Information and Software Technology. 108, 65–77 (2019). https://doi.org/10.1016/j.infsof.2018.12.004.
14. HashiCorp: Terraform by HashiCorp, https://www.terraform.io/, last accessed 2024/03/17.
15. AWS: AWS Cloud Development FAQs, https://aws.amazon.com/cdk/faqs/, last accessed 2024/03/17.
16. Yin, R.K.: Case Study Research: Design and Methods. SAGE (2009).
17. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empir Software Eng. 14, 131–164 (2009). https://doi.org/10.1007/s10664-008-9102-8.
18. Kratzke, N., Quint, P.-C.: Understanding Cloud-native Applications after 10 Years of Cloud Computing - A Systematic Mapping Study. Journal of Systems and Software. 126, 1–16 (2017). https://doi.org/10.1016/j.jss.2017.01.001.
19. AWS: Serverless Compute - AWS Fargate - AWS, https://aws.amazon.com/fargate/, last accessed 2024/04/02.
20. Morris, K.: Infrastructure As Code: Managing Servers in the Cloud. Oreilly & Associates Inc, Beijing (2016).
21. Kumar, V., Kumar Garg, K.: Migration of Services to the Cloud Environment: Challenges and Best Practices. IJCA. 55, 1–6 (2012). https://doi.org/10.5120/8716-7105.
22. Werner, C., Li, Z.S., Lowlind, D., Elazhary, O., Ernst, N., Damian, D.: Continuously Managing NFRs: Opportunities and Challenges in Practice. IIEEE Trans. Software Eng. 48, 2629–2642 (2022). https://doi.org/10.1109/TSE.2021.3066330.
23. Shuaib, M., Samad, A., Alam, S., Siddiqui, S.: Why Adopting Cloud Is Still a Challenge?—A Review on Issues and Challenges for Cloud Migration in Organizations. Presented at the January 1 (2019). https://doi.org/10.1007/978-981-13-5934-7_35.
24. Staevsky, N., Gaftandzhieva, S.: Cloud Migration: Identifying the Sources of Potential Technical Challenges and Issues. International Journal of Advanced Computer Science and Applications (IJACSA). 14, (2023). https://doi.org/10.14569/IJACSA.2023.0141204.
25. Rahman, A.: Anti-Patterns in Infrastructure as Code. In: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST). pp. 434–435. IEEE, Vasteras (2018). https://doi.org/10.1109/ICST.2018.00057.