TÉCNICO LISBOA

Migration of a Client-Server Application to a Cloud Architecture

The Case of an E-Banking Application

Martim Duarte Honório da Silva Bravo

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor(s): Prof. José Manuel da Costa Alves Marques

Examination Committee

Chairperson: Prof. Mário Jorge Costa Gaspar da Silva
Supervisor: Prof. José Manuel da Costa Alves Marques
Member of the Committee: Prof. Sérgio Luís Proença Duarte Guerreiro

November 2021

Dedicated to my family

Acknowledgments

I would like to thank Professor José Alves Marques for the opportunity to work on this thesis.

I would like to thank Rui Rasteiro, Helena Oliveira, Filipe Dias, and Pedro Samorrinha for their technical support and help.

I would like to thank my family for all their support, with special thanks to my parents for financially supporting my studies.

I would also like to acknowledge Francisco Delgado, Gonçalo Adolfo, and Marcelo Branco for helping keep me sane during this last year.

Finally, I would like to thank you, the reader, for taking the time to read my thesis.

Resumo

As empresas estão cada vez mais a migrar as suas aplicações para a cloud, seja para tirar vantagens do que a cloud tem para oferecer ou simplesmente pelos custos serem menores.

Esta tese está inserida num projeto maior onde uma empresa está a analisar a migração de uma aplicação cliente-servidor que vende a bancos para uma arquitetura cloud-native. Devido a parcerias que a empresa tem, o provedor de serviços cloud usado na tese é o Azure da Microsoft.

Os objetivos desta tese são: a investigação de várias arquiteturas para aplicações na cloud e estratégias de migração; propor uma arquitetura e uma estratégia de migração; desenvolver uma prova de conceito onde uma amostra da aplicação corre na cloud.

Uma visão geral da aplicação a migrar é feita de forma a contextualizar o que a aplicação faz e como está atualmente arquitetada.

Na sequência da investigação, são identificadas as diferentes categorias de serviços cloud, exemplos de arquiteturas cloud-native, princípios a seguir quando se desenha uma arquitetura cloud-native, estratégias de migração para a cloud, e serviços de interesse que o Azure possui.

Uma arquitetura é proposta seguindo os princípios identificados e tirando ideias dos exemplos de arquiteturas cloud-native encontradas, tendo como base a arquitetura atual da aplicação. Uma estratégia de migração com várias arquiteturas intermédias até se chegar à proposta é definida. É feita a descrição da implementação de cada iteração.

Na análise dos custos e da performance de cada iteração, chega-se à conclusão de que a arquitetura proposta é a mais promissora.

Palavras-chave: Cloud, Arquitetura, Migração, E-Banking, Azure

Abstract

Enterprises are increasingly migrating their applications to the cloud, whether to take advantage of what the cloud has to offer or simply because of the lower costs.

This thesis is part of a larger project where a company is analyzing the migration of a client-server application that it sells to banks to a cloud-native architecture. Due to the partnerships that the company has, the cloud service provider used in the thesis is Microsoft's Azure.

The objectives of this thesis are: the investigation of various architectures for cloud applications and migration strategies; propose an architecture and a migration strategy; develop a proof-of-concept where a sample of the application runs in the cloud.

An overview of the application to be migrated is given in order to contextualize what the application does and how it is currently architected.

Following the investigation, it is identified the different categories of cloud services, examples of cloud-native architectures, principles to follow when designing a cloud-native architecture, migration strategies to the cloud, and services of interest that Azure has.

An architecture is proposed based on the current application architecture, and following the identified principles and taking ideas from the examples of cloud-native architectures found. A migration strategy with several intermediate architectures until reaching the proposal is defined. The implementation of each iteration is described.

In the analysis of the costs and performance of each iteration, it is concluded that the proposed architecture is the most promising.

Keywords: Cloud, Architecture, Migration, E-Banking, Azure

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Nowadays enterprises are increasingly migrating their applications to the cloud. This migration occurs because of a need to modernize the applications and because the cloud offers advantages that the on-premises facilities do not. Some of the advantages that the cloud brings are cost savings, ease of security, business continuity, and monitoring.

In terms of cost savings, as it is not necessary to make upfront investments in hardware, which reduces the risk involved, there is no need to dimension the solution to the peak of use because the pay-per-use model and the ease of scalability allows to dimension the application to the normal use and only scale when there are more users, that may end up paying for the scaling.

The security that is not specific to a solution, like distributed denial of service (DDoS) attack protection and data protection, is easy to activate and in many services is already activated by default, so there is no need to implement it by hand which sometimes may lead to problems down the line, as security is not always trivial to implement, leaving only the solution-specific security to be implemented.

Business continuity is one of the big advantages of the cloud as data backups and data recovery is easy to activate and have control over. Disaster recovery is also easier to implement and, because of so, there is no need to have facilities in other geographic locations for this purpose. The high availability of the cloud services backed up by the Service Level Agreements (SLAs) is one of the strong suits of the cloud.

The monitoring of performance, costs, availability, and security is built-in to the cloud services and there is no need to spend money and time in building a solution for that purpose.

1.2 Topic Overview

This thesis is framed in a larger project in which the focus is the migration of a client-server application to a cloud-native architecture. The application in question, BankOnBox, is an e-banking application developed by Link Consulting. The application will be migrated to Microsoft's cloud service Azure. Azure

was chosen as the cloud service to migrate the application to because the current version of BankOnBox is built on Microsoft technologies and the company is a Microsoft Partner.

1.3 Objectives and Deliverables

Being part of the other project, the objectives and deliverables attributed to this thesis are:

- Investigate cloud architectures and migration strategies;
- Propose a target cloud architecture and a migration approach;
- Develop a proof-of-concept, where a sample of the application will be migrated to that architecture.

1.4 Thesis Outline

In the Background chapter, Chapter 2, an overview of the BankOnBox application is given, along with the different types of services that cloud providers make available, principles to follow when designing an application's cloud architecture, migration strategies, examples of cloud architectures, and an overview of some Azure services.

In the Solution chapter, Chapter 3, the architecture proposed to be implemented is described as well as the migration strategy used to implement it.

In the Implementation chapter, Chapter 4, all the steps taken to implement the architecture are explained, the problems encountered in the implementation, and how they were solved.

In the Results chapter, Chapter 5, an overview of some functionalities of Azure services used to debug the implementation is given. The process used to obtain the results is described. The pricing for the multiple iterations of the architecture is given. The results are shown and analyzed.

Chapter 2

Background

In this section, the current version of BankOnBox will be explained in detail, as well as the different types of cloud computing services, some principles of cloud architecture design, some migration strategies, some reference cloud architectures, and some of Azure's services.

2.1 BankOnBox

BankOnBox is an online banking platform developed by Link Consulting and sold to financial institutions. It is a robust and highly versatile platform to deliver consistent online services across several digital channels like mobile, Internet, chat, SMS, among others. It currently supports the home banking and mobile banking sites of approximately 20 customers in Europe and Africa.

The BankOnBox application was developed with the intent of bridging the gap between the technologies, the bank's needs, and the customer's needs to captivate and retain customers in a highly competitive environment. The BankOnBox application allows the bank customers to perform the traditional functionalities of home banking like consult account balances, make payments, make transfers, among others. In the current state of the application, it is deployed on-premises on the facilities of the clients.

In this section, the architecture of the application will be explained. The BankOnBox was already the subject of work in another thesis [1] and as such was also explained there.

2.1.1 Logical Architecture

The logical architecture as is visible in Figure 2.1 is composed of two main components: the BankOnBox Internet Banking Sites and the BankOnBox Engine. Other components that are part of the logical architecture are: the browsers and mobile apps that communicate with the BankOnBox Internet Banking Sites via HTTPS; the three databases, one used by the BankOnBox Internet Banking Sites to load its pages, one used for contract management and settings, and the other for logs; the back-office application that the bank's staff uses; and the bank core system. The communication between the BankOnBox Internet Banking Sites and the BankOnBox Engine is made using SOAP requests and responses.

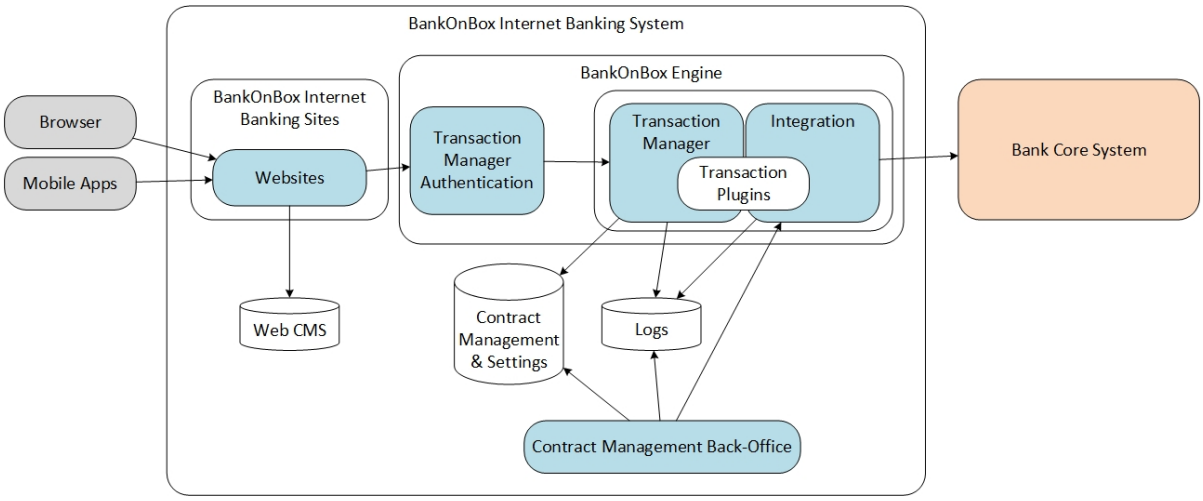

Figure 2.1: BankOnBox Logical Architecture

The BankOnBox Engine is composed of three modules, the transaction manager authentication, the transaction manager, and the BankOnBox integration. The BankOnBox Engine was designed with a modular architecture and plugin-based extensibility in mind since the BankOnBox application is sold to different banks that may need different operations exposed in the BankOnBox Internet Banking Sites and that have different interfaces to their core systems.

2.1.2 Technologies

The BankOnBox Internet Banking Sites component was built on ASP.NET 4 and is running on Microsoft's Internet Information Services. The web pages are currently being migrated to use AngularJS by a team at Link.

The BankOnBox Engine component was built on .NET 4 and is also running on Microsoft's Internet Information Services.

All the databases are SQL Server databases running on on-premises machines.

The back-office application was built on Silverlight and is currently being migrated to AngularJS and .NET Core.

2.1.3 BankOnBox Internet Banking Sites

The BankOnBox Internet Banking Sites component uses dynamically generated web pages that it loads from its database.

The BankOnBox Internet Banking Sites communicate with the transaction manager authentication module of the BankOnBox Engine component. The communication is done using Windows Communication Foundation (WCF) messages, that implement the WS-Security standard, and are sent over HTTP. The protection of the messages is done at the message level using WS-Security.

2.1.4 BankOnBox Engine

The BankOnBox Engine component, as said previously, is composed of the transaction manager authentication, the transaction manager, and the integration modules.

To allow the addition of new transactions, without changing the source code, the BankOnBox Engine uses a plugin. The plugin used is Microsoft Unity Framework that does dependency injection. The plugin loads by reflection all the classes that implement the transaction interface to a catalog, this allows that only the code for the new transactions needs to be added without changing the existing code.

Every request that the BankOnBox Engine receives and every answer it sends is logged to the contract management & settings database. Access to the databases is done using Entity Framework 4.

2.1.5 Transaction Manager Authentication

The transaction manager authentication is the BankOnBox Engine's module that deals with the communication from the BankOnBox Internet Banking Sites.

This module checks if the request received needs a second level of authentication.

If a second level of authentication is needed, this module sends a response to the BankOnBox Internet Banking Sites component saying that and forwards the request to the transaction manager module that puts the request on hold until it receives the second level authentication.

If a second level of authentication is not needed, then this module forwards the request to the transaction manager modules and its response to the BankOnBox Internet Banking Sites component.

The communication with the transaction manager module is also done using Windows Communication Foundation (WCF) messages and WS-Security.

2.1.6 Transaction Manager

The transaction manager module is the part of the BankOnBox Engine component that makes all the validations to the requests received.

The transaction manager has only one endpoint, this also allows to easily change the transactions offered as there is no need to add or remove endpoints in the BankOnBox Engine and also add or remove them in the code of the BankOnBox Internet Banking Sites.

The transactions can be grouped into two categories: queries and operations. Transactions of the category queries only read information from the contract management database and the bank core system, and an example of a query transaction is the request to check how much money is in one account. The transactions of the category operation involve writes, usually, money transfers.

Each transaction type has its specific code to distinguish them since there is only one endpoint. Each transaction also has its specific parameters.

When the transaction manager receives a transaction request, it does all the generic validations first, like, for example, checking if there is a session active if the user has permission to execute the transaction, and after that makes all the transaction-specific validations. After all the validations are done it passes the request to the integration to be executed.

The classes defined for the transactions in the transaction manager are responsible for defining what are the specific parameters that the transaction needs and the specific validations.

The transaction manager uses a state machine built with Windows Workflow Foundation (WWF) to make all the validations.

2.1.7 Integration

The integration module is the part of the BankOnBox Engine component that deals with the execution of the transactions and with the communication with the bank core system.

This is the part of the overall solution that is specific to each of the banks that BankOnBox is sold to, as each bank may have a different interface to access its core system, being that a web API or an IBM iSeries with direct program calls, message queues or ODBC database access.

The integration is responsible for making the conversion between the application model and core system model, so that the transaction can be executed, and calling the core system to execute it.

The classes defined for the transactions in the integration are responsible for defining the logic of the transaction execution.

2.1.8 Contract Management & Settings Database

The contract management & settings database has tables for users, contracts, and configurations, and it is in this database that all the data for an internet banking contract is stored. It is also in this database that the user is connected to the accounts that he is allowed to access and operate.

An internet banking contract regulates the access of a customer to their banking information via the digital channel. The contract has information like the digital channel credentials, the accounts the customer has access to in the digital channel, the types of operations they can execute in each account, the limits of the allowed transactions (transaction maximum values, dailyy maximum values, among others), additional authentication factors required by operation, among others. For corporate contracts, it also includes who approves transactions, and how many approvals are required for a transaction to go through.

The transactions of the category operations can have automatic or manual processing. The manual processing operations require a member of the back-office staff to process them, and as such, they are stored in this database.

This database is also used to store logs, although the logs stored in this database are only functional ones. These logs are stored in two tables, one named “TransactionLogs” and the other named “Operations”.

The table “TransactionLogs” contains all the requests and responses received and sent by the BankOnBox Engine. This is useful for the bank to collect statistics like how many transactions are being done per second, how many money transfers, how many account balance consults, between many others.

The table “Operations” also contains the requests and responses received and sent by the BankOnBox Engine, but only the ones regarding money transferring and logins or logouts, this is because some enterprise contracts require multiple users to authorize a money transfer. This table is where all the pending operations to be executed later are stored.

2.1.9 Logs Database

The logs database is only used to store application logs like application errors, connectivity errors, and all the requests and responses sent and received to the bank core system.

2.1.10 Security

In terms of functional security, the BankOnBox application offers two levels of authentication and multi-approval of operations in business contracts.

The first level of authentication is used for queries and is a username and password combination. The second level is used for operations and it can be a confirmation key, a coordinate card, or a one-time password (OTP) that can be an SMS or a token.

In terms of non-functional security, the authentication of back-office staff is done with Microsoft Active Directory; the passwords and OTPs are stored in the database after being hashed with a salt and ciphered; in the production environment, the access to the contract management & settings and logs databases is done using Windows authentication, where the transaction manager runs as a Windows user and the SQL Server only allows connections for reads and writes by that user, the access to the web CMS database by the BankOnBox Internet Banking Sites component is done using SQL authentication.

2.1.11 Physical Architecture

The physical architecture of the BankOnBox application once deployed in the clients is fairly simple as it is possible to see in Figure 2.2.

The physical architecture is composed of three zones: the Internet, the demilitarized zone (DMZ), and the local area network (LAN). The Internet and the LAN have between them the DMZ. Between the Internet and the DMZ, and between the DMZ and the LAN there are firewalls. The DMZ contains the web servers, in front of the web servers exists a load balancer. The LAN contains the application servers that also have a load balancer in front of them, the SQL servers, the banking core, and other systems that the application servers might communicate with, like, for example, the domain controller.

2.2 Types of Cloud Computing Services

The different cloud computing services offered by the cloud providers can be divided into three main groups: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS). The division into these three groups is done by what are the parts of the service stack that are

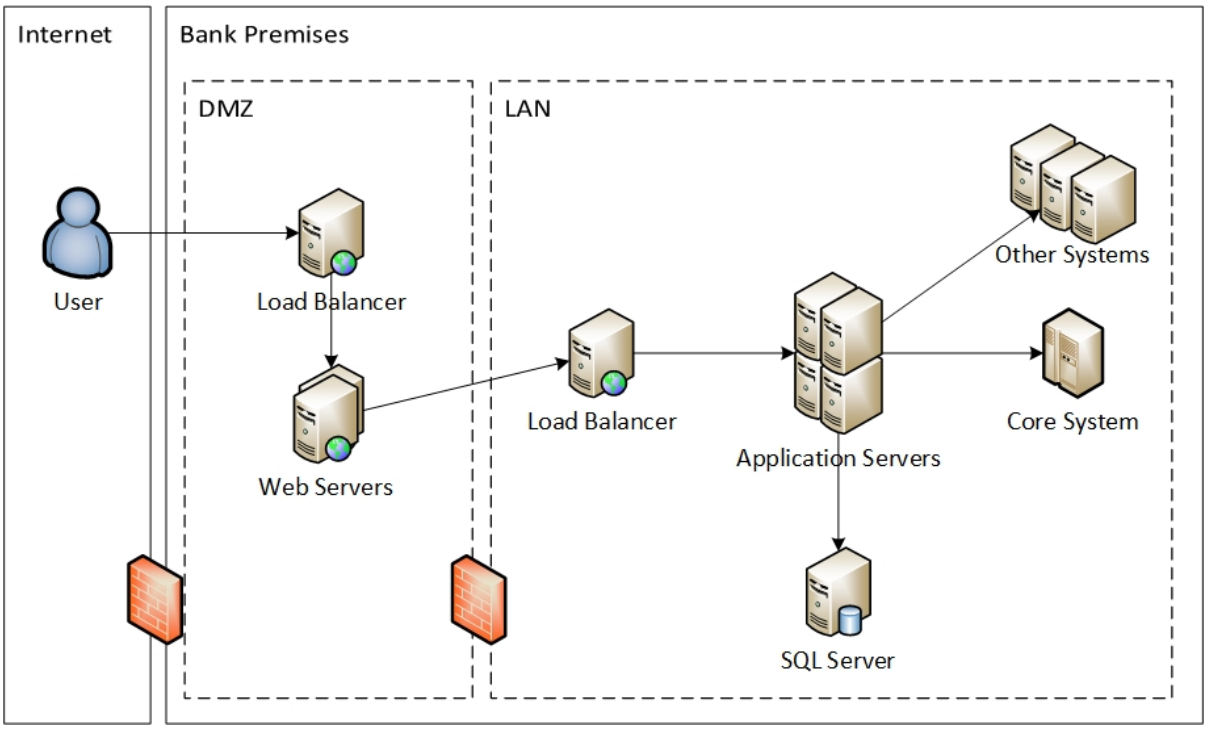

Figure 2.2: BankOnBox Physical Architecture

managed by the provider and what are the parts that are managed by the cloud user. On an application running on-premises, the whole stack would be managed by the person or organization that owns it.

Cloud computing, as said in [2], [3], and [4], has many definitions, but the most commonly accepted is the one by the National Institute of Standards and Technology (NIST). The definition of cloud computing by NIST is as follows: “Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.”.

The stack that composes the applications is, from top to bottom, as follows:

- Interface;
- Application;
- Data;
- Runtime;
- Middleware;
- Operating System;
- Virtualization;
- Servers;
- Storage;

- Networking.

On an IaaS service, the part of the stack managed by the cloud customer is from the Operating System upwards, while the cloud provider manages from the Virtualization downwards. An example of an IaaS service is a VM hosting service.

On a PaaS service, the part of the stack managed by the cloud user is from the Data upwards, while the cloud provider manages from the Runtime downwards. On a PaaS service, the cloud customer only has control over his code, he does not have access to the machine where that code runs, and as such, one example of a PaaS service is a website hosting service.

On a SaaS service, the cloud customer only has access to the Interface used to interact with the application. The cloud provider controls the application, the software, and the hardware it runs on. Some examples of SaaS services are Microsoft Office 365, Google Apps, and Dropbox.

In terms of the administration continuum, from the point of view of the cloud customer, the groups of services, from higher administration to lower administration, are ranked as follows: IaaS, PaaS, SaaS.

2.3 Principles of Cloud Architecture

Some cloud application architecture principles, as defined in [5], [6], [7], and [8], that, when followed, lead to a better optimized cloud application are:

Design for self-healing In distributed systems failures occur. Therefore, there is a need to design the application to be self-healing when that happens. This requires an approach that detects failures, responds to failures gracefully, and logs and monitors failures to give operational insight.

Make all things redundant Build redundancy into the application to avoid having single points of failure. A resilient application routes around failure. To do that there is a need to identify critical paths of the application and check if there is redundancy at each point of the path.

Minimize coordination Minimize coordination between application services to achieve scalability. Most cloud applications consist of multiple application services like front-ends, databases, business processes, reporting and analysis, among others. To achieve scalability and reliability these services should run on multiple instances.

Design to scale out Design the application so that it can scale horizontally. One of the big advantages of the cloud is the ability to use as much capacity as needed, scaling out when load increases, and scaling in when the extra capacity is no longer needed. The application should be design in such a way that allows it to scale horizontally, adding or removing new instances as demand requires.

Design for operations Design the application so that the operations team has the tools needed. The cloud changed the role of the operations team as they are no longer responsible for the management of the hardware and infrastructure that hosts the application. However, the operations team is still important in running a successful cloud application. Some of the important functions of the operations team include deployment, monitoring, escalation, incident response, and security auditing. For that to be possible, robust logging and tracing are important, and as such, the operations team should be involved in the designing and planning to ensure that the application gives them all the data and insight that they need.

Use managed services When possible use PaaS rather than IaaS. Managed services are easier to configure and administer as there is no need to provision VMs, set up virtual networks, manage patches and updates, and all the other overhead associated with running software on a VM.

Use the best data store for the job Pick the storage technology that best fits the data and how it will be used. Relational databases are very good at providing ACID guarantees for transactions over relational data, but they come with costs. These costs include queries that may require expensive joins, data that must be normalized and conform to a predefined schema, and lock contention that may impact performance. In a large solution, it is probable that a relational database will not fill all the needs, and as such alternatives like keyvalue stores, document databases, and graph databases may be a better fit.

Design for evolution An evolutionary design is key for continuous innovation. All successful applications change over time, whether it be to fix bugs, add new features, bring in new technologies, or make existing systems more scalable and resilient. If all the parts of an application are tightly coupled, it becomes very hard to introduce changes into the system as a change in one part of the application may break another part. This problem does not only occur in monolithic applications as an application decomposed into services may still exhibit the sort of tight coupling that leaves the system rigid and brittle.

2.4 Cloud Migration Strategies

There are many strategies to migrate applications to the cloud, as described in [9], [10], [11], and [12], most of the strategies can be grouped into four categories: re-hosting, re-platforming, re-purchasing, and re-architecting.

Re-hosting, also known as lift & shift, re-deployment, and copy & paste, is when the migration to the cloud is done by only migrating the physical servers and VMs to the cloud as-is, without any changes to the code. This strategy has the advantage of being the fastest one to be done and the cheapest in terms of the cost of the migration process.

Re-platforming, also known as re-packaging, is similar to re-hosting, with the difference being that instead of migrating the existing application to an IaaS it is migrated to PaaS with minimal code changes.

This strategy has the advantage of lowering the costs of running and managing the application with the cost of the migration itself not being too high.

Re-purchasing is moving to a different product, most commonly moving to a commercially available off-the-shelf (COTS) or SaaS product. This strategy has the advantage of buying a product that is already built and only needs adaptations instead of commissioning, building, or re-building a product from the ground up.

Re-architecting, also know as re-factoring and re-designing, is when the architecture of the application is re-imagined using cloud-native features to better optimize it to the cloud platform and scalability. This strategy has the advantage of cost-effectively meeting scalability requirements and allows the addition of new features more easily.

2.5 Reference Cloud Architectures

In [7], Microsoft offers some reference cloud architectures to common application architecture styles.

2.5.1 N-Tier Architecture Style

N-tier is a traditional architecture for enterprise applications that divides an application into logical layers and physical tiers.

Layers are a way to separate responsibilities and manage dependencies. Each layer performs a specific logical function, such as presentation, business logic, and data access. A layer can only call layers that sit below it.

Tiers are physically separated, running on separate machines. A tier can call another tier directly or by using asynchronous messaging, such as a message queue. It is not required for each layer to be hosted in its own tier. One tier can host multiple layers. Physically separating the tiers improves scalability and resiliency but also adds latency from the additional network communication.

N-tier applications have an architecture that resembles the one in Figure 2.3.

Some of the best practices when using this architecture are: use autoscaling to handle changes in load; use asynchronous messaging to decouple tiers; cache semi-static data; configure database tier for high availability; place a web application firewall (WAF) between the front-end and the Internet; place each tier in its own subnet, and use subnets as a security boundary; restrict access to the data tier, by allowing requests only from the middle tier(s).

An example of an n-tier architecture on virtual machines is visible in Figure 2.4.

In this architecture, each tier consists of two or more VMs with scaling capability. Having multiple VMs provides resilience in case one VM fails. Load balancers are used to distribute requests across the VMs in a tier.

Each tier is placed inside its own subnet. This makes it easy to apply network security rules and route tables to individual tiers. Network security rules restrict access to each tier. For example, the database tier only allows access from the business tier.

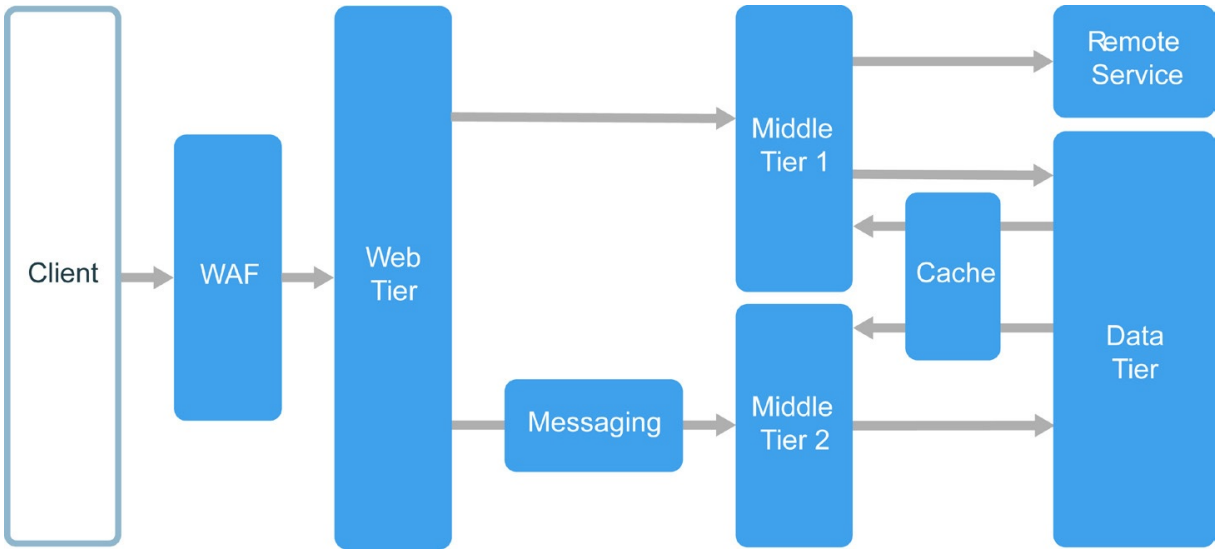

Figure 2.3: N-Tier Architecture

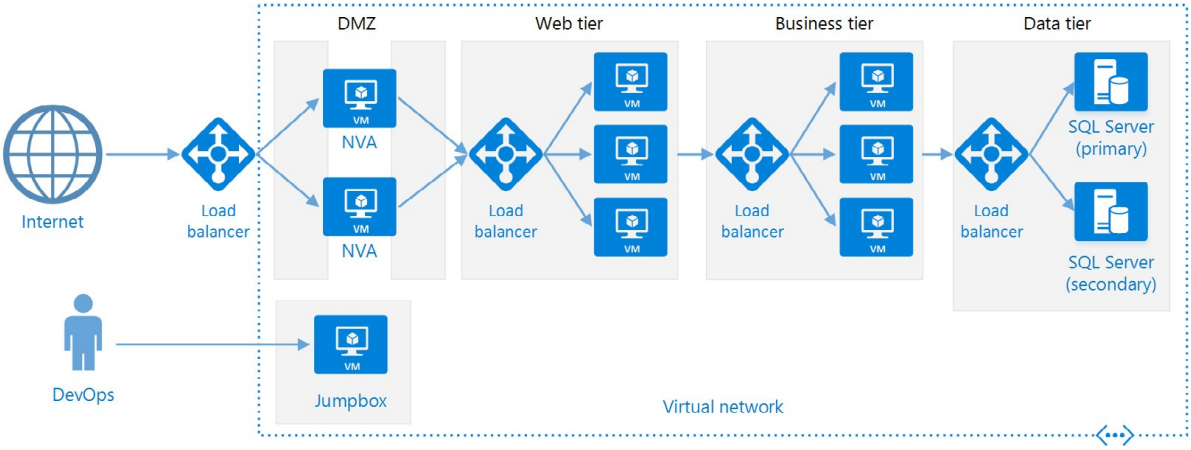

Figure 2.4: N-Tier Architecture on VMs

The web and business tiers are stateless. Any VM can handle any request for that tier. The data tier should consist of a replicated database.

2.5.2 Web-Queue-Worker Architecture Style

In the Web-Queue-Worker style, the application has a web front-end that handles HTTP requests and a back-end worker that performs CPU-intensive tasks or long-running operations and where the front-end communicates to the worker through an asynchronous message queue. Web-Queue-Worker applications have an architecture that resembles the one in Figure 2.5.

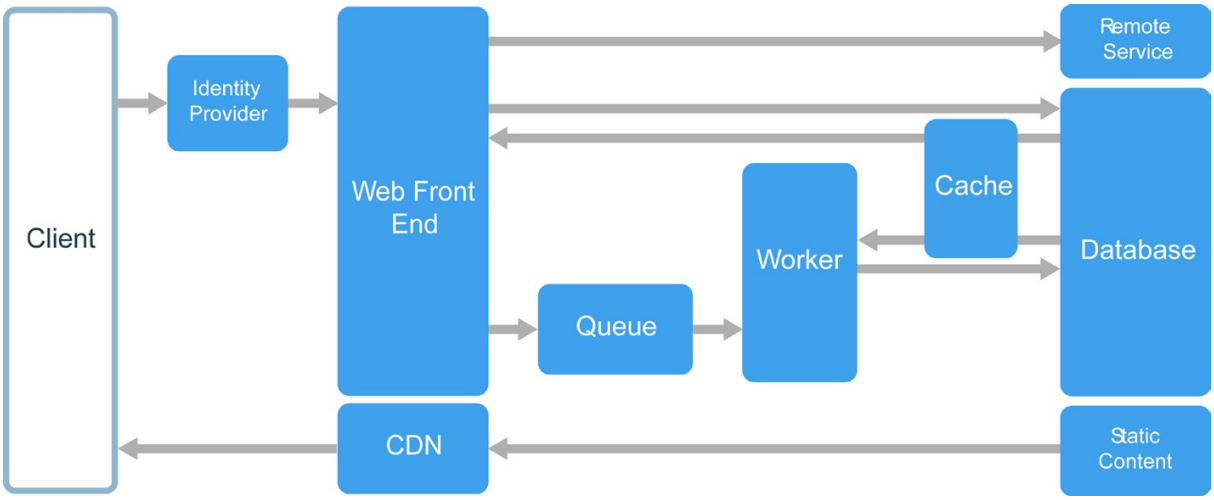

Figure 2.5: Web-Queue-Worker Architecture

Some of the best practices when using this architecture are: use polyglot persistence when appropriate; use autoscaling to handle changes in load; cache semi-static data; use data partitioning.

An example of a web-queue-worker architecture on Azure App Services is visible in Figure 2.6.

In this architecture, the front-end is implemented as an Azure App Service Web App, and the worker is implemented as a Function App. The web app is associated with an App Service Plan that provides the instances and the scaling policy. The web worker is serverless.

The message queue used can be either an Azure Service Bus or an Azure Queue Storage.

A Redis Cache is used to store session state and other data that needs low latency access.

A CDN is used to cache static content such as images, CSS, or HTML.

For storage, the storage technology that best fits the needs of the application is used. Multiple storage technologies can be used.

2.5.3 Microservices Architecture Style

A Microservices architecture style application is composed of many small and independent services where each service implements a single business capability and where the services are loosely coupled and communicate synchronously through API contracts or asynchronously through message queues. Microservices applications have an architecture that resembles the one in Figure 2.7.

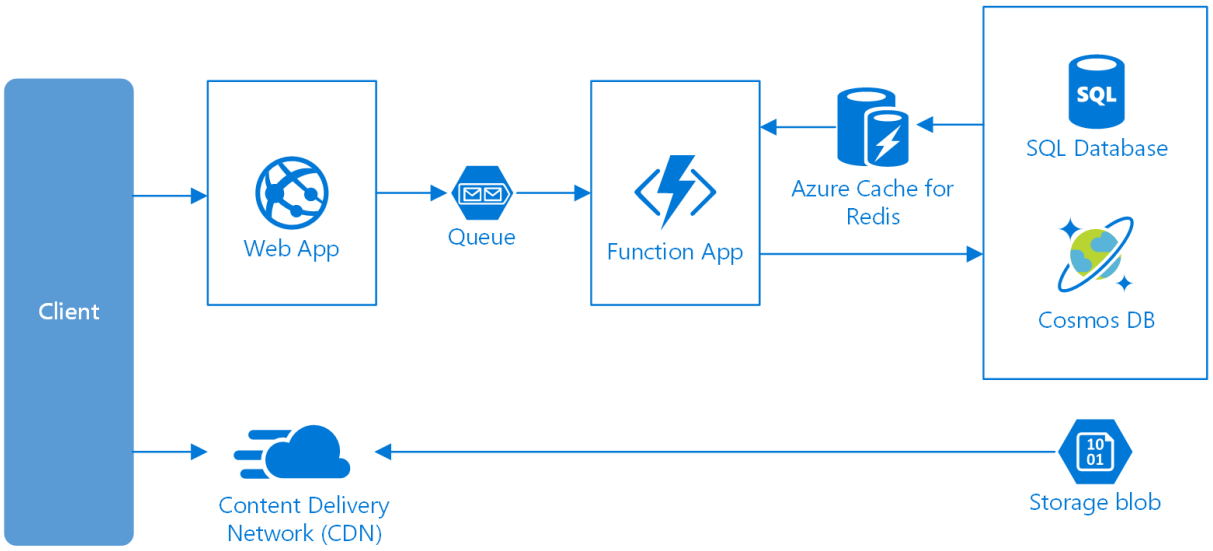

Figure 2.6: Web-Queue-Worker Architecture on Azure App Services

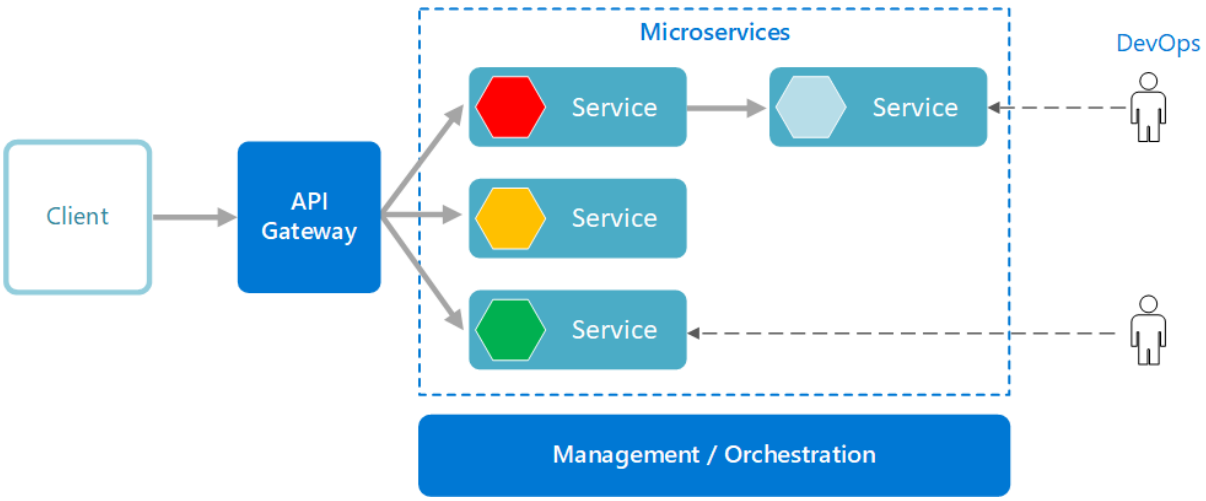

Figure 2.7: Microservices Architecture

Some of the best practices when using this architecture are: model services around the business domain; data storage should be private to the service that owns the data; services communicate through well-design APIs; avoid coupling between services; keep domain knowledge out of the gateway; services should have loose coupling and high functional cohesion.

An example of a microservices architecture on Azure Container Service is visible in Figure 2.8.

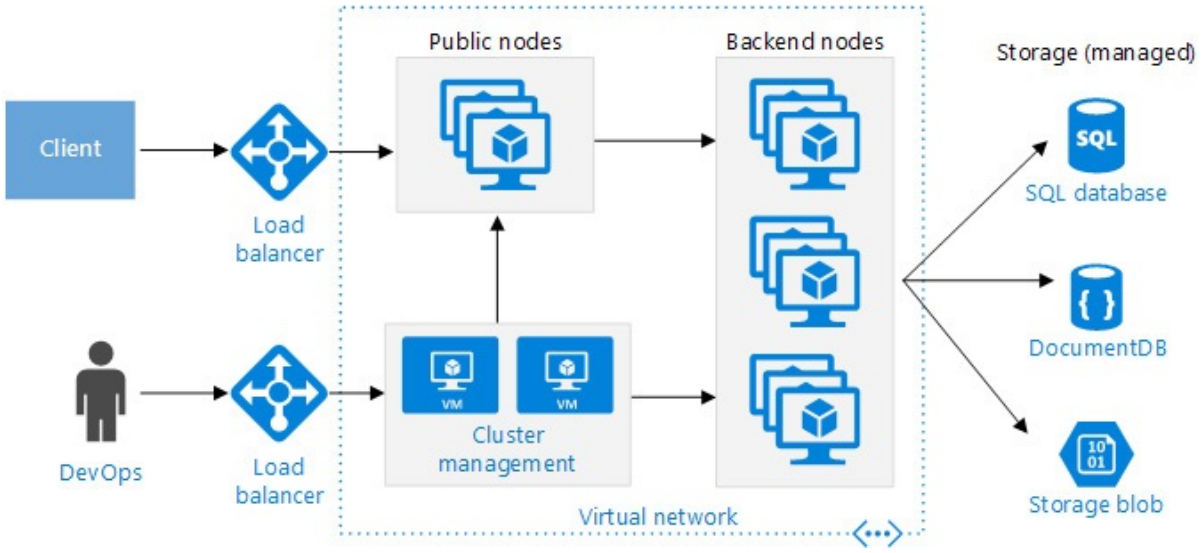

Figure 2.8: Microservices Architecture on Azure Container Service

In this architecture, the public nodes are reachable through a public-facing load balancer. The API gateway is hosted on these nodes.

The back-end nodes run services that clients reach via the API gateway. These nodes do not receive Internet traffic directly. The back-end nodes might include more than one pool of VMs, each with a different hardware profile.

The management VMs run the master nodes for the container orchestrator.

The public nodes, back-end nodes, and management VMs are placed in separate subnets within the same virtual network.

An externally facing load balancer sits in front of the public nodes. It distributes internet requests to the public nodes. Another load balancer is placed in front of the management VMs to allow secure shell traffic to the management VMs using NAT rules.

2.6 Azure Services

Some Azure services of interest for this thesis are as follows.

2.6.1 Virtual Machines

The virtual machines (VMs) service[13] is an IaaS. The VMs can have as the operating system Linux distros or Windows Server.

VMs are typically chosen when more control over the computing environment is needed when compared to what other choices offer.

An Azure VM gives the flexibility of virtualization without having to buy and maintain the physical hardware that runs it. However, it is still needed to maintain the VM by performing tasks, such as configuring, patching, and installing the software that runs on it.

The size of a VM is determined by factors such as processing power, memory, and storage capacity. Azure offers a wide variety of sizes to support many types of workloads.

Azure charges an hourly price based on the VM's size and operating system. For partial hours, Azure charges only the minutes used. Storage is priced and charged separately.

VMs use virtual hard disks (VHDs) to store their OS and data. The disk size and performance tier (Standard or Premium) can be specified, and Azure creates and manages the disk. VHDs are also used for the images chosen from to install an OS.

Azure provides a marketplace for images. Images have different operating systems, versions, and software installed.

Azure has a Service Level Agreement of 99.9% for VMs when the VM is deployed with premium storage for all disks.

All resources created in Azure are distributed across multiple geographical regions around the world. Usually, the region is called location when creating a VM. For a VM, the location specifies where the virtual hard disks are stored and VM resources are located.

2.6.2 App Services

The App Service[14] is PaaS. The App Service is an HTTP-based service for hosting web applications, REST APIs, and mobile back-ends. It supports many languages, be it .NET, .NET Core, Java, Ruby, Node.js, PHP, or Python. Applications run and scale with ease on both Windows and Linux-based environments. The implementation for .NET is based on IIS for Windows.

App Service adds advantages of the cloud such as security, load balancing, autoscaling, and automated management to the applications running on it. App Service also has DevOps capabilities, such as continuous deployment from Azure DevOps, GitHub, Docker Hub, and other sources. Other capabilities are package management, staging environments, custom domain, and TLS/SSL certificates.

With App Service, the pricing is determined by the Azure compute resources used. The compute resources used are determined by the App Service plan that the application runs on.

Some of the key features of App Service are:

- Multiple languages and frameworks - App Service has first-class support for ASP.NET, ASP.NET Core, Java, Ruby, Node.js, PHP, or Python. It can also run PowerShell and other scripts or executables as background services.

- Managed production environment - App Service automatically patches and maintains the OS and language frameworks.

- Containerization and Docker - App Service can run dockerized apps and host a custom Windows or Linux container. It can also run multi-container apps with Docker Compose.
- DevOps optimization - App Service supports continuous integration and deployment with Azure DevOps, GitHub, BitBucket, Docker Hub, or Azure Container Registry. The updates can be promoted through test and staging environments. Apps can be managed by using Azure PowerShell or the cross-platform command-line interface (CLI).
- Global-scale with high availability - Apps can scale up or out manually or automatically. Apps can be hosted anywhere in Microsoft's global datacenter infrastructure, and the App Service SLA promises high availability.
- Connections to SaaS platforms and on-premises data - It is possible to choose from more than 50 connectors for enterprise systems (such as SAP), SaaS services (such as Salesforce), and internet services (such as Facebook). Access to on-premises data can be done using Hybrid Connections and Azure Virtual Networks.
- Security and compliance - App Service is ISO, SOC, and PCI compliant. Users' authentication can be done with Azure Active Directory, Google, Facebook, Twitter, or Microsoft accounts.
- Application templates - Extensive list of application templates in the Azure Marketplace, such as WordPress, Joomla, and Drupal.
- Visual Studio and Visual Studio Code integration - Dedicated tools in Visual Studio and Visual Studio Code streamline the work of creating, deploying, and debugging.
- API and mobile features - App Service provides turn-key CORS support for RESTful API scenarios, and simplifies mobile app scenarios by enabling authentication, offline data sync, push notifications, and more.

An App Service plan[15] defines a set of compute resources for a web app to run. These compute resources are analogous to the server farm in conventional web hosting. One or more apps can be configured to run on the same computing resources.

When creating an App Service plan in a certain region, a set of compute resources is created for that plan in that region. The apps put into the App Service plan run on these compute resources as defined by the App Service plan. Each App Service plan defines the OS, the region, the number of VM instances, the size of the VM instances, and the pricing tier.

2.6.3 Function Apps

Azure Functions[16] is a PaaS serverless solution that allows to write less code, maintain less infrastructure, and save on costs.

Systems are often built to react to a series of critical events. Whether it is building a web API, responding to database changes, processing IoT data streams, or even managing message queues, every application needs a way to run some code as these events occur.

To meet this need, Azure Functions provide “compute on-demand” in two significant ways:

1. Azure Functions allow the implementation of system logic into readily available blocks of code. These blocks are called “functions”. Different functions can run anytime it is needed to respond to critical events.

2. As requests increase, Azure Functions meets the demand with as many resources and function instances as necessary, but only while needed. As requests fall, any extra resources and application instances drop off automatically.

Providing compute resources on-demand is the essence of serverless computing in Azure Functions.

Common scenarios for Azure Functions are:

- Build a web API - Implement an endpoint for the web application using the HTTP trigger.
- Process file uploads - Run code when a file is uploaded or changed in blob storage.
- Build a serverless workflow - Chain a series of functions together using durable functions.
- Respond to database changes - Run custom logic when a document is created or updated in Cosmos DB.
- Run scheduled tasks - Execute code on pre-defined timed intervals.
- Create reliable message queue systems - Process message queues using Queue Storage, Service Bus, or Event Hubs.
- Analyze IoT data streams - Collect and process data from IoT devices.
- Process data in real-time - Use Functions and SignalR to respond to data in the moment.

Functions Apps can be developed in C#, Java, JavaScript, PowerShell, Python, or using a custom handler to use virtually any other language.

When creating a function app, a hosting plan for the app has to be chosen[17]. There are three basic hosting plans available for Azure Functions: Consumption plan, Premium plan, and Dedicated App Service plan. All hosting plans are generally available on both Linux and Windows virtual machines.

The chosen hosting plan dictates the following behaviors:

- How the function app is scaled.
- The resources available to each function app instance.
- Support for advanced functionality, such as Azure Virtual Network connectivity.

The benefits of the Consumption plan are that it scales automatically and only charges for compute resources when the functions are running. On the Consumption plan, instances of the Functions host are dynamically added and removed based on the number of incoming events.

The benefits of the Premium plan are that it automatically scales based on demand using pre-warmed workers which run applications with no delay after being idle, it runs on more powerful instances, and it connects to virtual networks.

The benefit of the Dedicated App Service plan is that it runs the functions within an App Service plan at regular App Service plan rates. It is best for long-running scenarios where Durable Functions cannot be used.

The operating system where the functions run influences the language that can be used. Linux is the only supported OS for the Python runtime stack. Windows is the only supported OS for the PowerShell runtime stack. Linux is the only supported operating system for Docker containers and it is only supported in the Premium and Dedicated App Service plans.

In the Consumption plan, apps may scale to zero when idle, meaning some requests may have additional latency at startup. The Consumption plan does have some optimizations to help decrease cold start time, including pulling from pre-warmed placeholder functions that already have the function host and language processes running.

In the Premium plan, perpetually warm instances are used to avoid any cold start.

In the Dedicated plan, the Functions host can run continuously, which means that cold start is not really an issue.

The service limits for each plan are visible in Table 2.1.

Regardless of the function app timeout setting, 230 seconds is the maximum amount of time that an HTTP triggered function can take to respond to a request. This is because of the default idle timeout of Azure Load Balancer.

In the Consumption plan, the billing is based on the number of executions, execution time, and memory used.

In the Premium plan, the billing is based on the number of core seconds and memory used across needed and pre-warmed instances. At least one instance per plan must be kept warm at all times. This plan provides the most predictable pricing.

In the Dedicated plan, the billing for function apps is the same as it would be for other App Service resources.

2.6.4 Azure SQL

Azure offers three SQL Server services[19][20], two platform-as-a-service (PaaS) and one infrastructure-as-a-service (IaaS). The two PaaS implementations are Azure SQL Database and Azure SQL Managed Instance, and the IaaS implementation is SQL Server on Azure VMs.

The order of all these services from the point of the cloud migration effort, re-host to re-build, and the administration continuum, higher administration to lower administration, is the following:

1. SQL Server on Azure VMs (IaaS);
2. Azure SQL Managed Instance (PaaS);
3. Azure SQL Database (PaaS).

Resource	Consumption plan	Premium plan	Dedicated plan
Default timeout duration (min)	5	30	30
Max timeout duration (min)	10	unbounded	unbounded
Max outbound connections (per instance)	600 active (1200 total)	unbounded	unbounded
Max request size (MB)	100	100	100
Max query string length	4096	4096	4096
Max request URL length	8192	8192	8192
ACU[1] per instance	100	210-840	100-840
Max memory (GB per instance)	1.5	3.5-14	1.75-14
Max instance count	200	100	varies by SKU
Function apps per plan	100	100	unbounded
App Service plans	100 per region	100 per resource group	100 per resource group
Storage	5 TB	250 GB	50-1000 GB
Custom domains per app	500	500	500
Custom domain SSL support	unbounded SNI SSL connection included	unbounded SNI SSL and 1 IP SSL connections included	unbounded SNI SSL and 1 IP SSL connections included

[1] Azure Compute Unit (ACU)[18] provides a way of comparing CPU performance across Azure SKUs. ACU is currently standardized on a Small (Standard_A1) VM being 100.

Table 2.1: Plans' Service Limits Comparison

SQL Server on Azure VMs

The SQL Server on Azure VMs is a normal VM with the only difference being that it uses a pre-built image with the SQL Server installed. This implementation is best for migrating workloads that require 100 percent SQL Server compatibility and OS-level access.

Some scenarios where this solution makes more sense are: re-host of rich SQL apps to the current SQL Server version; migrate a single or a few applications to the cloud; re-host of sunset applications, that is, applications that are being phased-out or terminated.

Some of the benefits of this implementation are: 100 percent SQL Server compatibility; full control of the operating system and SQL Server level; hybrid high availability and disaster recovery; SQL Server Reporting Services (SSRS), SQL Server Analysis Services (SSAS), and SQL Server Integration Services (SSIS) support.

Azure SQL Managed Instance

The Azure SQL Managed Instance is almost the same as having the full SQL Server experience, with the only difference being that it runs on a managed system. This implementation is best for modernizing existing apps.

Some scenarios where this solution makes more sense is when there is a need to modernize and migrate existing SQL applications to the new SQL Server version with minimal code changes.

Some of the benefits of this implementation are: rich, instance-centric programming model; fully managed with no patching or maintenance required; virtual network integration; AI-driven performance and security.

Azure SQL Database

The Azure SQL Database is the most cloud-optimized service of these three and the least like having SQL Server running on an on-premises machine. It does not offer all the compatibility that Azure SQL Managed Instance offers for migrating, since some functions, statements, triggers, views, stored procedures are not supported, and, because of that, some re-build is needed, but it is a cheaper option and the one with the least administration needed.

Some scenarios where this solution makes more sense is when building cloud applications from the ground up on the current version of SQL Server.

Some of the benefits of this implementation are: simplicity and flexibility of SLA-backed deployments and scale; AI-driven performance and security; hyperscale storage capabilities and available serverless compute; fully managed, no patching or maintenance required.

2.6.5 Cosmos DB

Azure Cosmos DB[21] is a PaaS NoSQL database for modern app development. It offers single-digit millisecond response times, automatic and instant scalability, guaranteed speed at any scale.

Business continuity is assured with SLA-backed availability and enterprise-grade security.

App development is faster and more productive thanks to turn-key multi-region data distribution anywhere in the world, open-source APIs, and SDKs for popular languages.

As a fully managed service, Azure Cosmos DB takes database administration off the user's hands with automatic management, updates, and patching.

Cosmos DB handles capacity management with cost-effective serverless and automatic scaling options that respond to application needs to match capacity with demand.

Cosmos DB's free tier offers the first 1000 RU/s[1] and 25 GB of storage for free.

Cosmos DB has guaranteed speed at any scale through SLA-backed speed and throughput, fast global access, and instant elasticity. Real-time access with fast read and write latencies globally, and throughput and consistency are all backed by SLAs. Multi-region writes and data distribution to any

[1] A request unit (RU) is the measure of throughput in Cosmos DB. A 1 RU throughput corresponds to the throughput of the GET of a 1 kB document

Azure region with the click of a button. Independently and elastically scaling storage and throughput across any Azure region, even during unpredictable traffic bursts, for unlimited scale worldwide.

Simplified application development with open-source APIs, multiple SDKs, schemaless data, and no-ETL analytics over operational data. Deeply integrated with key Azure services used in modern cloud-native app development including Azure Functions, IoT Hub, Azure Kubernetes Service, App Service, and more. Multiple database APIs including Core (SQL) API, API for MongoDB, Cassandra API, Gremlin API, and Table API. Running no-ETL analytics over the near-real-time operational data stored in Cosmos DB with Azure Synapse Analytics. Cosmos DB's schemaless service automatically indexes all the data, regardless of the data model, to deliver blazing-fast queries.

Mission-critical ready with guaranteed business continuity, 99.999% availability, and enterprise-level security for every application. Cosmos DB offers a comprehensive suite of SLAs including industry-leading availability worldwide. Ease of distributing data to any Azure region with automatic data replication. Enterprise-grade encryption-at-rest with self-managed keys. Azure role-based access control keeps the data safe and offers fine-tuned control.

Fully managed and cost-effective due to end-to-end database management, with serverless and automatic scaling matching the application and TCO needs. Fully-managed database service means automatic, no-touch, maintenance, patching, and updates, saving time and money. The serverless model offers spiky workloads automatic and responsive service to manage traffic bursts on demand.

Solutions that benefit from Azure Cosmos DB include any web, mobile, gaming, and IoT application that needs to handle massive amounts of data, reads, and writes at a global scale with near-real response times for a variety of data.

2.6.6 Application Insights

Application Insights[22] is a SaaS feature of Azure Monitor. It is an extensible Application Performance Management (APM) service for developers and DevOps professionals.

Application Insights is used to monitor live applications. It will automatically detect performance anomalies and includes powerful analytics tools to help diagnose issues and understand what users actually do with the app. It is designed to help continuously improve performance and usability.

It works for apps on a wide variety of platforms including .NET, Node.js, Java, and Python hosted on-premises, hybrid, or any public cloud.

It integrates with DevOps processes and has connection points to a variety of development tools.

Application Insights work by installing a small instrumentation package (SDK) in the application or by enabling it using the Application Insights Agent when supported. The instrumentation monitors the app and directs the telemetry data to an Azure Application Insights Resource using a unique GUID referred to as an Instrumentation Key.

Application Insights can instrument not only the BankOnBox Engine application, but also any background components, and the JavaScript in the web pages themselves. The application and its components can run anywhere, it does not have to be hosted in Azure.

The impact introduced by Application Insights in the application is small. Tracking calls are non-blocking and are batched and sent in a separate thread.

Application Insights monitors:

- Request rates, response times, and failure rates - Helping find out which pages are most popular, at what times of day, and where the users are. Which pages perform best. If response times and failure rates go high when there are more requests.
- Dependency rates, response times, and failure rates - Helping find out whether external services are slowing the app.
- Exceptions - Both server and browser exceptions.
- Pageviews and load performance - reported by the users' browsers.
- AJAX calls from web pages.
- User and session counts.
- Performance counters from Windows and Linux server machines - Such as CPU, memory, and network usage.
- Host diagnostics from Docker or Azure.
- Diagnostic trace logs from the app - To correlate trace events with requests.
- Custom events and metrics.

2.6.7 Log Analytics

Log Analytics[23] is a SaaS tool in the Azure portal used to edit and run log queries with data in the Azure Monitor Logs.

It allows writing a simple query that returns a set of records and then use features of Log Analytics to sort, filter, and analyze them. Or to write a more advanced query to perform statistical analysis and visualize the results in a chart to identify a particular trend. Log Analytics uses a custom query language called Kusto.

Whether it is to work with the results of the queries interactively or to use them with other Azure Monitor features such as log query alerts or workbooks, Log Analytics is the tool used to write and test them.

Chapter 3

Solution

In this chapter, the architecture proposed to be implemented is explained as well as the strategy to migrate and implement the architecture.

3.1 Architecture

The current architecture of BankOnBox when simply ported to the cloud is not optimized for production environments since it has multiple components of the logical architecture running in the same resources and does not follow the principles of cloud architecture listed in Chapter 2.3.

A more appropriate architecture is proposed in Figure 3.1.

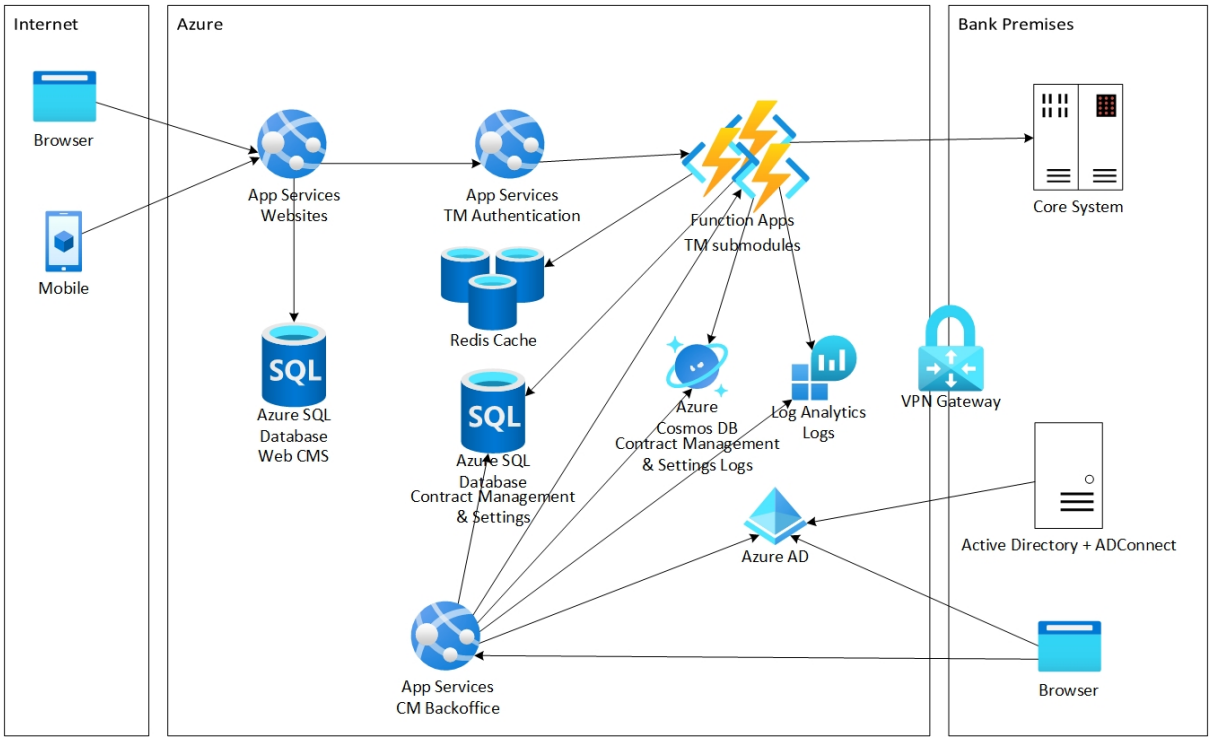

Figure 3.1: Proposed Architecture

All current IIS sites get dedicated services. The BankOnBox website and the transaction manager authentication are each their own App Service. The App Service was chosen because it is the Azure equivalent to Windows' IIS. The site composed of the transaction manager and integration components is divided into Function Apps. This division is further explained in Chapter 3.1.1. The Function App service was chosen because it is a serverless service and this is the component that would benefit the most from the scalability it offers. Both the App Service and Function Apps have built-in auto-scalers and load balancers, following the principle of cloud architecture that says to design to scale out.

Each database also gets its dedicated service. The BankOnBox website's database and the contract management & settings database are both Azure SQL Databases. This service was chosen from the ones listed in Chapter 2.6.4 due to being the most cloud-optimized, and the cheapest of the managed database services. The less compatibility that it offers when compared to a version of SQL Server running on-premises is not relevant in this case since the service has built-in alternatives.

An Azure Cache for Redis is used to cache the responses of database queries. This is used instead of an in-memory cache because the same cached response can be accessed by multiple instances.

The database used for applicational logging is replaced by the Log Analytics service. This service allows easy querying and visualization of information about the logs which makes it an ideal solution.

The functional logs present in the contract management & settings database are moved to Cosmos DB. Using Cosmos DB for storing logs allows the structure of these to change over time if needed without changing the database schema. Another advantage is that the SQL databases are kept small since the majority of the storage is occupied by logs.

The decisions of what service to use to store functional and applicational logs follow the principle of cloud architecture that says to use the best data store for the job.

The contract management back-office application used by the bank's staff is in an App Service.

The authentication of the bank's staff in the back-office application is done using the Azure Active Directory (Azure AD). The Azure AD is configured to replicate the on-premises Active Directory. For that to be possible, it is necessary to have an on-premises machine running Azure AD Connect.

To have an added layer of security and the Azure services that communicate with the core system to be viewed as if they were on-premises, the communications between Azure services and bank facilities are all done through a VPN connection. The alternative would be to have an ExpressRoute connection, but that requires a dedicated connection from the bank facilities to Azure, and it is a more expensive solution.

Due to not existing a core system that can be used in this thesis, some components of the architecture will not be implemented.

A core system database that is normally used when BankOnBox is in an offline state, to for example make a backup, will be put on the cloud to simulate the core system. This means that transactions that involve transfers of money cannot be implemented, only transactions to consult information.

The contract management back-office application used by the bank's staff will also not be implemented since an Active Directory to be replicated is necessary. The fact that the main focus is on the architecture and performance of the transaction manager and integration components, and that this

management application is being migrated to other technologies make this less of a priority.

3.1.1 Transaction Manager Architecture

A closer look into the architecture of the transaction manager and integration components is visible in Figure 3.2.

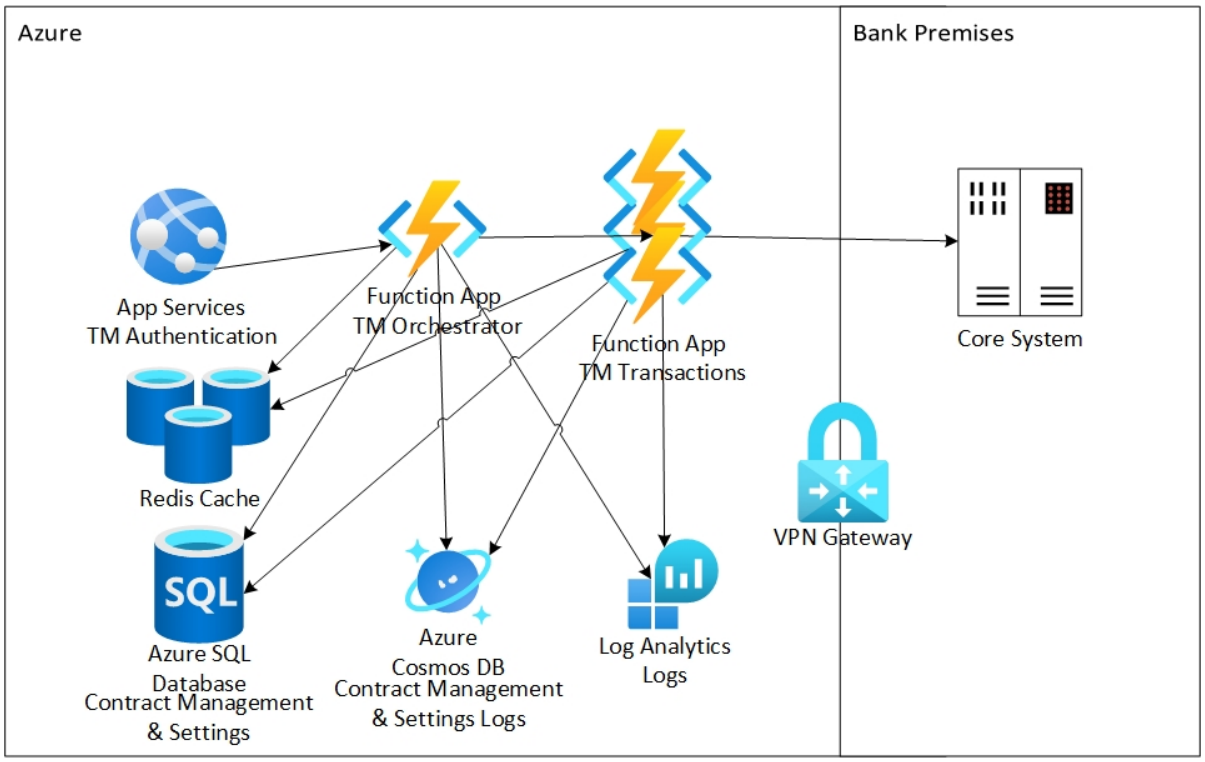

Figure 3.2: Proposed Architecture Close-up

Currently, the transaction manager and integration components are part of a bigger component that is responsible for all the execution of a transaction. The transaction manager is responsible for the orchestration of the code execution, the validations, and the functional logging. The integration is responsible for the communication with the core system and the execution of the transaction-specific code.

The architecture proposed separates these two components into multiple Function Apps. An orchestrator Function App and one Function App per transaction type.

The orchestrator Function App is responsible for, as the name implies, orchestrating the code execution, it is also responsible for the functional logging and the generic validations that all transactions require.

The transactions' Function Apps are responsible for all the logic that is transaction-specific, being validation or execution. They are also responsible for communicating with the core system.

In the current architecture, the logic for each transaction type is added to the BankOnBox application via plugins.

To maintain the plugin logic, all transaction Function Apps share a base URL, where the only part that changes is the transaction type, and have one endpoint for the validation of the parameters, one for

the transaction validations, and one for the execution.

The sequence of actions when a request arrives at the orchestrator is as follows:

1. The orchestrator loads the settings for the transaction type received. This also validates that the transaction type does exist.
2. The orchestrator checks the application status, whether it is online or offline. This influences which transactions can be executed.
3. The orchestrator logs the request. This log generates the id used to track the transaction.
4. The orchestrator does the generic validations.
5. The orchestrator makes a request to the validate parameters endpoint of the transaction Function App.
6. The transaction Function App checks if the parameters are valid.
7. The orchestrator makes a request to the transaction validation endpoint of the transaction Function App.
8. The transaction Function App checks transaction-specific validations.
9. The orchestrator based on the transaction type settings and the application status checks if the transaction can be executed or if it has to be stored for later execution.
10. If the execution is possible, the orchestrator makes a request to the execution endpoint of the transaction Function App.
11. The transaction Function App executes the transaction logic.
12. The orchestrator logs the response.

This sequence of actions means that for each request that the orchestrator receives, it makes three requests to the transactions Function Apps.

This division into orchestrator and transactions allows for transactions that typically receive more requests to scale out differently from transactions that receive less. This division also allows having different transactions implemented with different technologies and in different locations more easily.

3.2 Migration Strategy

Since this thesis deals with an application that already exists and not with something new, before arriving at the architecture proposed to be implemented, the overall solution will go through multiple intermediary architectures. These architectures allow increasing the solution's complexity gradually, to measure the performance of each iteration and compare the results.

The multiple iterations are as follows:

1. Lift & Shift;
2. Databases Re-Platforming;
3. Servers Re-Platforming;
4. Databases Re-Factoring;
5. Transaction Manager Re-Factoring.

3.2.1 Lift & Shift (First Iteration)

The goal for this first iteration is to take what currently exists and put it on the cloud to serve as a baseline to which to compare all other iterations.

To achieve this it is necessary to have the websites, the web service, and the databases running on virtual machines in the same virtual network in the cloud. It is also necessary to have a service mocking the OTP for second-level authentication.

Since there is no core system, the offline database also needs to be running in the cloud.

The fact that there is no bank AD means that an Active Directory domain controller serving also as the DNS server for the virtual network is needed. This is because the databases use Windows authentication to authenticate the connections from the websites and the web service.

The architecture of this iteration is visible in Figure 3.3. It has two VMs in the same virtual network and subnet.

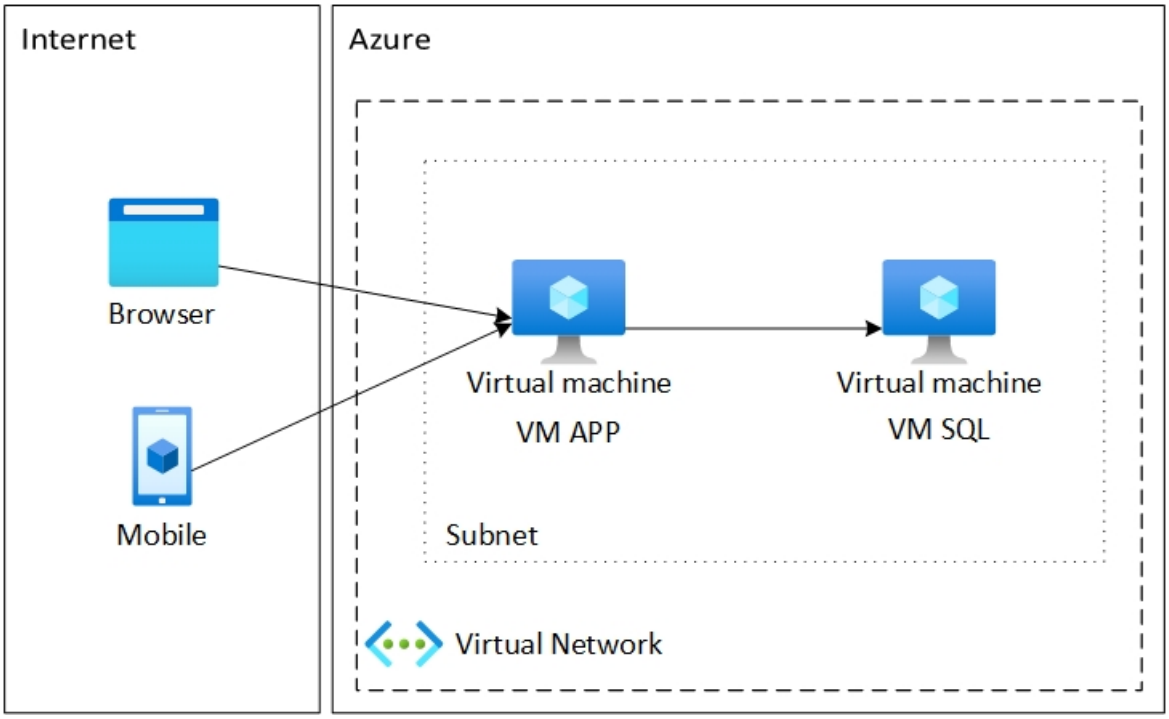

Figure 3.3: First Iteration Architecture

The VM APP contains both the websites and the web service servers. This VM is also the domain

controller for the Active Directory. This machine has to have a static private IP address to be configured as the DNS server for the virtual network.

The VM SQL contains all the databases required by the application, in addition to the offline database used to mock the core system.

3.2.2 Database Re-Platforming (Second Iteration)

The goal of this iteration is to move all databases from the VM SQL to Azure SQL Databases. This means less administration is needed by moving to a PaaS service.

The architecture of this iteration is visible in Figure 3.4.

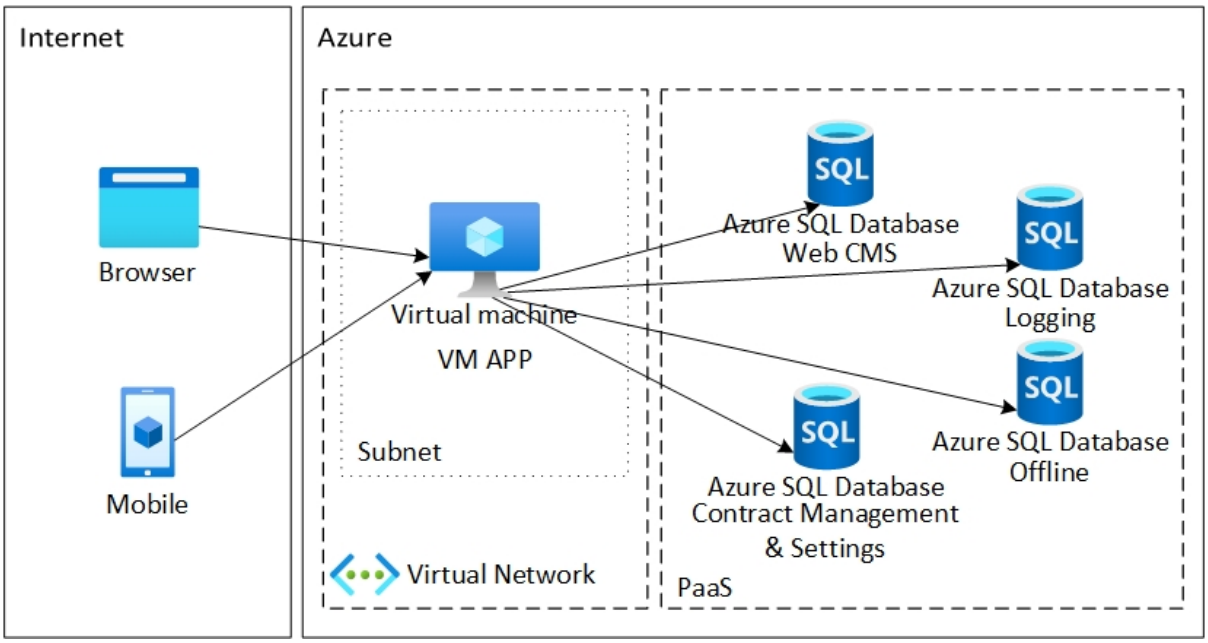

Figure 3.4: Second Iteration Architecture

The architecture is similar to the previous iteration in the fact that the VM APP is still running all the same services.

The databases are all Azure SQL Databases. The authentication of the connections is changed from Windows authentication to SQL Server login.

3.2.3 Server Re-Platforming (Third Iteration)

The goal of this iteration is to move the sites running in the VM APP to multiple App Services. This means less administration is needed by moving to PaaS services.

The architecture of this iteration is visible in Figure 3.5.

In this architecture, the websites, the OTP mock, the TM authenticator, and the TM have each their own App Service. The TM is comprised of the transaction manager and integration modules and is not yet divided into sub-modules.

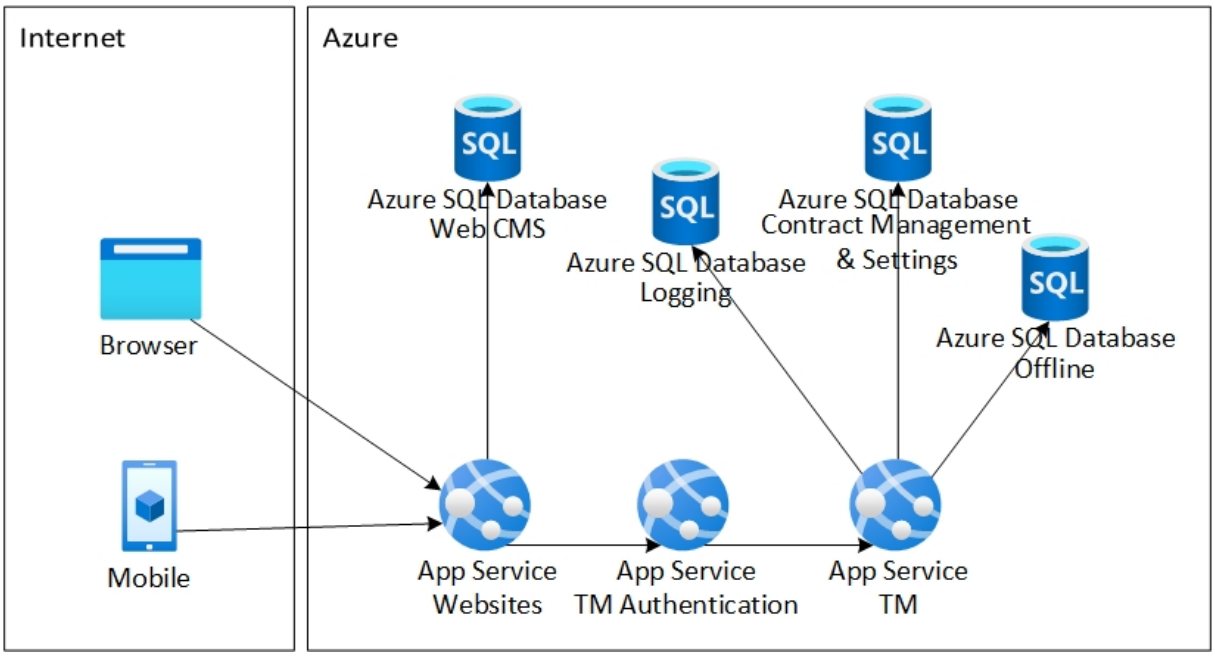

Figure 3.5: Third Iteration Architecture

3.2.4 Database Re-Factoring (Fourth Iteration)

The goal of this iteration is to stop using SQL databases for logging and instead use more appropriate solutions. This is the first iteration that needs the BankOnBox code to be re-factored.

The architecture of this iteration is visible in Figure 3.6.

In this architecture, the functional logs are moved from the contract management & settings Azure SQL database to a Cosmos DB database. The applicational logs database is replaced by the Log Analytics service.

3.2.5 Transaction Manager Re-Factoring (Fifth Iteration)

The goal of this iteration is to divide the transaction manager and integration modules into sub-modules, in other to better divide the load and allow an architecture that scales better.

The architecture of this iteration is visible in Figure 3.7.

In this architecture, a Function App is added for the orchestrator and a Function App per each type of transaction implemented. An Azure Cache for Redis is introduced to store the response of queries to the database in a way that makes it accessible to multiple Function App instances at the same time.

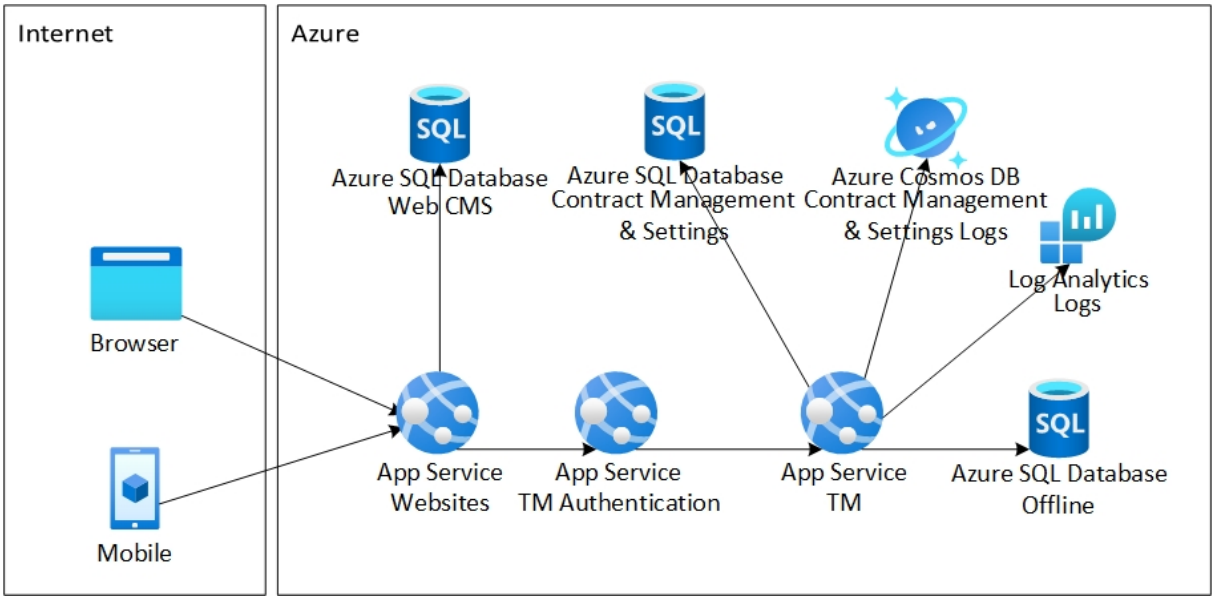

Figure 3.6: Fourth Iteration Architecture

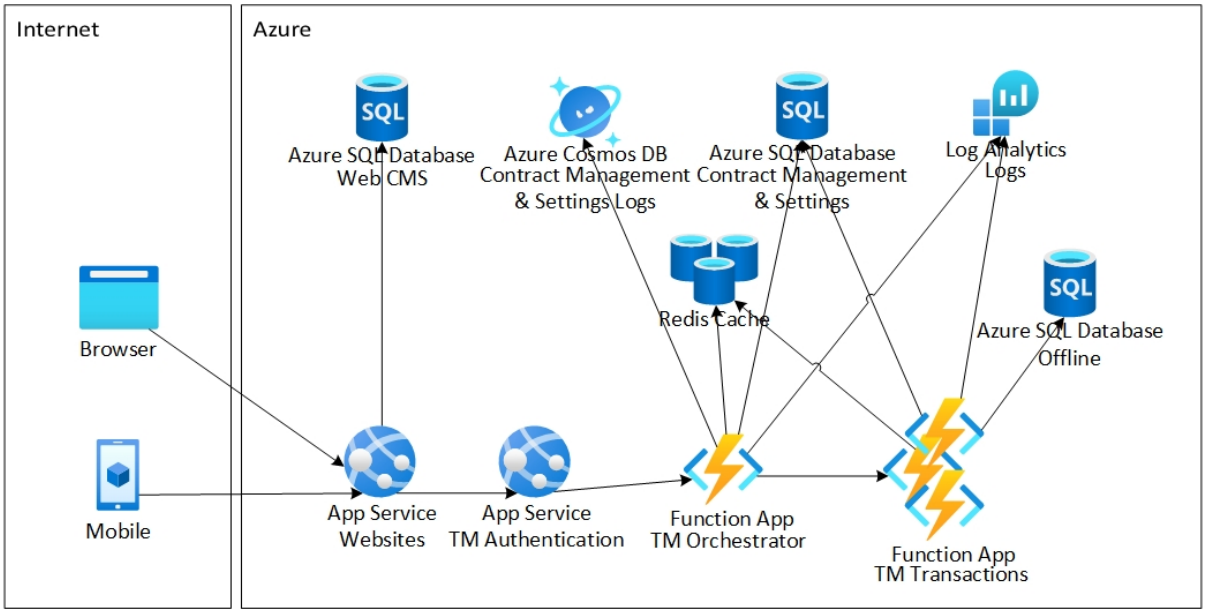

Figure 3.7: Fifth Iteration Architecture

Chapter 4

Implementation

In this chapter, the implementation process of the various iterations is explained as well as some problems found and how they were solved.

As it has been said before, the focus of the implementation was on the users' side of the architecture, with a special focus on the transaction manager and integration component.

Before any migration, an effort was made to try and upgrade the version of the .NET Framework being used. However, the tool used to assess the changes needed to upgrade the version reported that much of the code would need to be changed to be compatible with current versions; because of that, and due to the intention of having the various iterations, that upgrade was deemed not worth it.

After implementing each iteration, their performance was tested and the results were analyzed to try to find possible performance bottlenecks, and validate the scalability where applicable.

4.1 Lift & Shift (First Iteration)

To try and achieve the goals for this iteration, previously referred on Chapter 3.2.1, it was necessary to do the following.

In the Azure portal, it was necessary to create two VM resources, with the second one in the same subnet as the first one. The VM APP is the VM with the BankOnBox servers and the VM SQL is the VM with the databases. Both VMs use the same Windows Server image.

In the VM APP, it was necessary to install the Active Directory Domain Services (ADDS) and the Internet Information Services (IIS). It was also necessary to copy all the published code of the BankOnBox components to the VM.

After installing the ADDS it was necessary to promote this VM to a domain controller, doing that also installs a DNS server in the machine. To promote the VM to a domain controller it was necessary to add a new domain forest and a domain name. With the machine as a domain controller, all the virtual machines in the virtual network needed to use it for DNS lookups. For that to be possible, the private IP of the machine needed to be changed to a static one in Azure, and the DNS server of the virtual network changed to that static IP instead of Azure's default one.

After installing the IIS it was necessary to create two IIS sites one for the websites component and the other for the web service component. Since the transaction manager uses integrated security to access its databases, a new user was created in the domain and a new application pool was created in the IIS that runs with the identity of that user.

The site created for the web service runs on port 5000 with HTTP connections and is configured for test purposes to allow connections from any IP, on a production environment only the websites' component would be able to access the web service. Because of this, a new rule was created in the network security group of this VM in Azure. Three applications were added to the created site, one for the transaction manager authentication, one that is an OTP mock for second level authentication, and one for the transaction manager and integration. The applications on this site were configured to use the application pool running with the identity of the created user.

The site created for the websites runs on port 80 with HTTP. HTTPS was not configured because there is no certificate issued for this prototype. Two applications were added to this site, one for the bank's business clients and the other one for the private customers. The applications were configured to use the default IIS application pool.

Every application added to either of the sites points to its corresponding code in the file system. After the installation of the VM SQL, it was necessary to change the connection strings in the configuration files to point to the databases in that VM. For the websites component applications, it was also necessary to change configuration files to point to the web service running on the local machine.

In the VM SQL was necessary to install the SQL Server 2019, and add this machine to the domain that the other one is the controller of.

After installing the SQL Server, the schemas and the data of the databases were migrated to the VM from their on-premises counterparts. The SQL user used by the websites and the domain user used by the transaction manager were added to the database server logins, with only the domain user having sysadmin privileges.

All the goals of this iteration were achieved and as so the architecture implemented after this iteration is the same as the one in Chapter 3.2.1, and is visible in Figure 4.1.

4.2 Databases Re-Platforming (Second Iteration)

To try and achieve the goal for this iteration, previously referred on Chapter 3.2.2, to use one of the SQL Server services that Azure offers instead of having a virtual machine with the databases, the following was done.

The tool Data Migration Assistant (DMA)[1] from Microsoft was used to assess the readiness of the databases to be migrated to the Azure SQL Database service.

The tool reported that some features being used in the databases on-premises were not supported in Azure SQL Database, but it did not find any migration blockers. A closer inspection of the reported unsupported features showed that this was a non-problem as the feature, SQL Server Agent jobs, has

[1] https://docs.microsoft.com/en-us/sql/dma

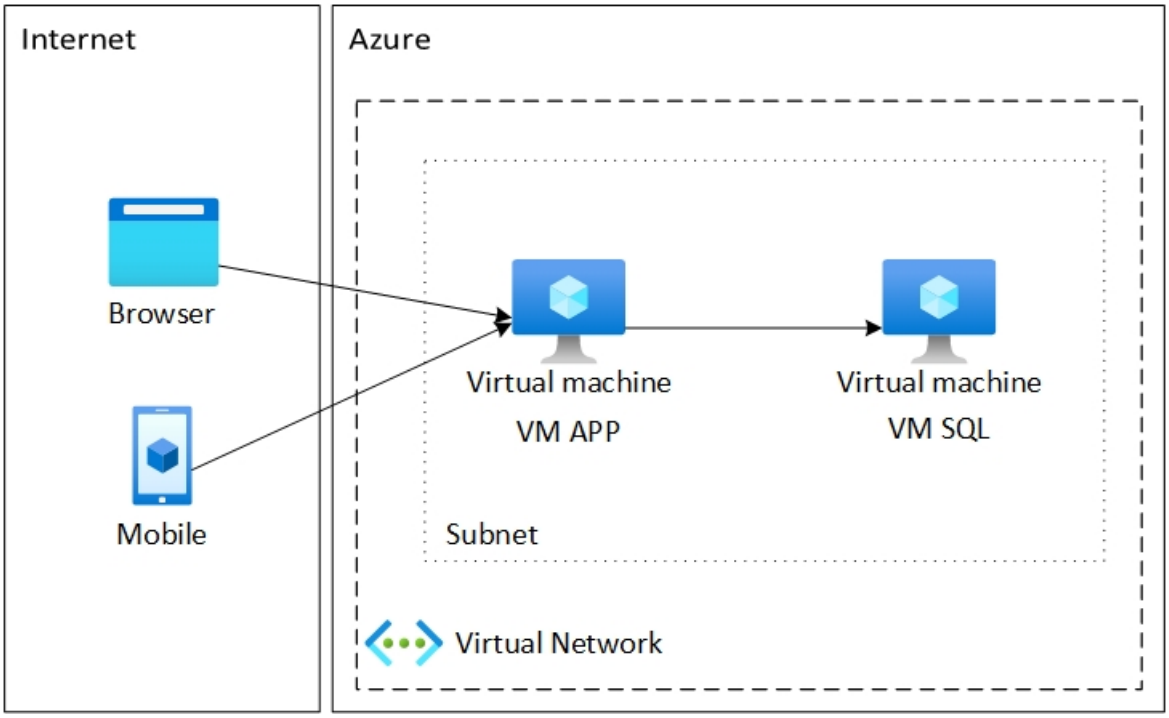

Figure 4.1: First Iteration Implemented Architecture

a counterpart in Azure SQL Database, elastic jobs, and that the only jobs that were relevant to the databases in question were the backup job and the shrink job that are automatically taken care of by the Azure platform.

After the assessment, it was time to create the databases in Azure. An SQL Server resource was created in Azure and inside that SQL server, the databases used by the websites component, the web service component, and the database used for when the BankOnBox is in offline state, used in this case also as a mock of the bank's core system, were created.

To migrate the databases' schemas and data the DMA tool was also used. To be possible to migrate, it was necessary to add a rule to the firewall of the SQL server in Azure to allow connections from the machine running the tool.

For the services that run on the VM APP to use the Azure SQL databases, it was necessary to change the connection strings in the configuration files. After doing that, one problem was encountered, the VM did not have access to the databases, because the SQL server resource's firewall did not allow connections from the machine's IP. To solve that problem three solutions were found.

The first solution was to change the public IP of the VM to a static one and create a rule in the firewall of the SQL server resource allowing connections from that IP like it was done for the migration process. The big problem with this solution, and the reason why it was not chosen, is that, even though both the VM and the server are in Azure, the connection would go through the public Internet.

The second solution was to add a private endpoint to the virtual network where the VM is located, which would allow the SQL server resource to be seen as being in that virtual network. This solution has the added benefit of allowing on-premises machines that are connected to the virtual network via a VPN connection to access the databases. This solution was not chosen because, in the context of this

thesis, the added benefit was not relevant. In a real environment, this solution would make more sense as it would allow the banks to have direct access to the databases and build their custom data exploring solutions if needed. This solution also incurs more costs as it is a paid feature inside the virtual network service.

The third solution was to add a virtual network service endpoint to Microsoft.SQL in the virtual network subnet and then allow in the firewall of the SQL server resource connections from that virtual network subnet. This solution was chosen because the connections do not go through the public Internet and because this solution does not incur additional costs.

In the migration of the databases, it is not possible to migrate the certificate and the symmetric key used in the on-premises databases to cipher some of the columns. The way that the key and certificate were created on-premises does not allow to re-create them. Because of this, a new certificate and symmetric key were created in the Azure database.

Since there is a new certificate and a new symmetric key, the data migrated that was ciphered with the original ones would not be readable. Two solutions were found to solve this problem. The first solution, and the best option in a real-world environment, would be to decipher the data before sending it over from the on-premises database and cipher with the new key in the cloud database. The second solution would be to reset the users' passwords in the cloud database after the migration. The second solution was chosen because it was faster to reset the necessary passwords than to develop a SQL script to decipher all the passwords on-premises and cipher them when in the cloud.

After this process, the goal of this iteration was achieved with the implemented architecture, visible in Figure 4.2, being the same as the one in Chapter 3.2.2.

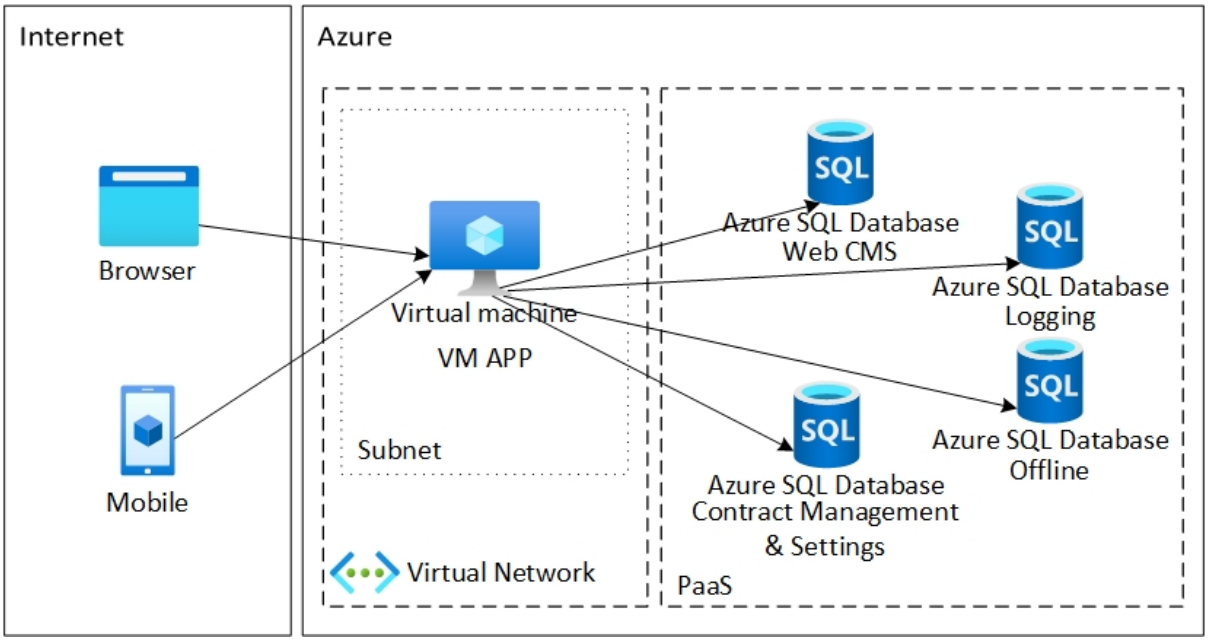

Figure 4.2: Second Iteration Implemented Architecture

4.3 Servers Re-Platforming (Third Iteration)

To try to achieve the goal previously referred on Chapter 3.2.3 the following was done.

Four App Service resources were created, one for the websites, one for the TM Authentication, one for the OTP mock, and one for the transaction manager and integration. All of the App Services were created within the same App Service Plan, and all with ASP.NET V4.8 as the runtime stack and Windows as the operating system. The App Services were all created in the same App Service Plan because the price paid is per App Service Plan and not per App Service.

Once an App Service is created there are three ways of deploying the code, continuous deployment via a git repository, manual deployment using OneDrive or Dropbox, or via FTPS. Since all the code was already published and in the VM APP, the deployment for all App Services was done via FTPS.

After all the code was deployed, the configurations were changed for each App Service so that they all pointed to the versions running in App Services and the databases running in Azure SQL Database.

When testing this iteration, the website and the transaction manager servers seemed to be working fine, but when trying to log in via the website an error would occur, although no error occurred when trying to login via a SOAP request directly to the transaction manager server. The problem was in the TM Authentication, as the technology used and the way that it is implemented is not compatible with the Azure App Service due to necessary permissions that are not given and are not possible to change.

Because of this incompatibility, and since, it made no sense to try to solve this due to the website component and TM Authentication being currently moved to a different technology by a team at LINK, the configurations on the website in the VM APP were changed to point to the App Service containing the transaction manager and the databases in the Azure SQL Database.

The goal of this iteration was not completely achieved, since only one App Service is being used, the transaction manager and integration one. This is the most important one as it is the one where the performance is tested. The other App Services were nice to have but are not essential for the thesis.

The implemented architecture for this iteration is visible in Figure 4.3.

4.4 Databases Re-Factoring (Fourth Iteration)

To try and achieve the goal for this iteration, previously referred on Chapter 3.2.4, to stop using a SQL database and use more adequate services for logs, the following was done.

4.4.1 Functional Logs

For the functional logs, the contract management & settings database was duplicated in Azure and the tables BackOfficeLogEntries, OperationAuthorizations, Operations, OperationsLogs, and TransactionLogs were removed as they were suited to be moved to NoSQL databases.

In Azure, a Cosmos DB account was created with a database containing those five tables as containers. Each container has as its partition key an attribute that makes it so that all related items are always in the same logical partition, and to ensure that there is a good distribution of values.

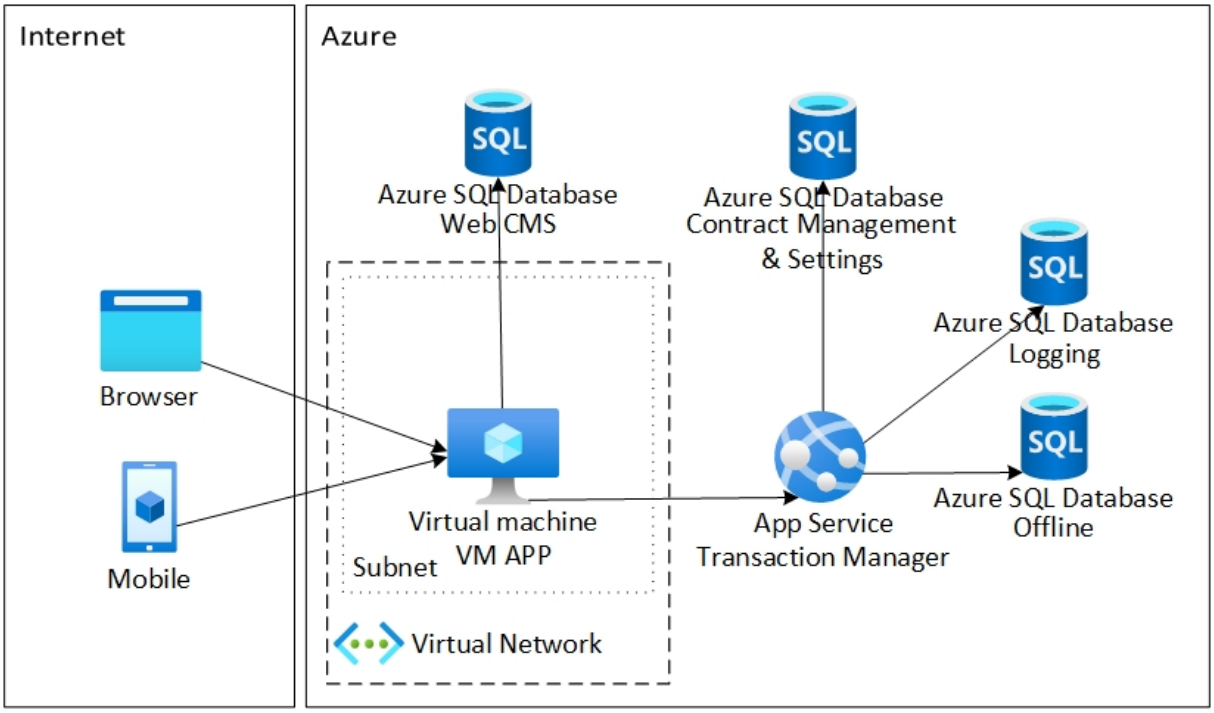

Figure 4.3: Third Iteration Implemented Architecture

Due to the fact of the ids automatically generated by Cosmos DB being GUIDs instead of integers, a table was added to the contract management and settings database, containing a mapping between an auto-generated integer key and the Cosmos id. It was necessary to add this table since it is easier for people to work with integers than with GUIDs, and the product support and bank IT staff will need to work with these logs.

To update the SQL tables in the source code, it was only necessary to use the “update model from database” functionality of Entity Framework 4. The model for the Cosmos DB had to be done by hand since this version of Entity Framework does not support Cosmos and the Cosmos library used does also not have that functionality. The library used for Cosmos was version 2 of the official Microsoft SQL API library, this version is one version behind the current one, two versions behind the preview, and will be supported until August 31st, 2024. This version was chosen because it was the most recent with compatibility with the .NET Framework version used.

Most of the examples in the documentation of the Cosmos DB used async requests, however, to change just some database requests was not feasible to change almost all the source code to async methods, so the sync options available were used. These options, however, came with a problem, they made as many trips to the database as required to fetch the entire result-set. This caused errors due to too many RU/s being used per request as the database grew because the query was being processed in the client instead of in the database.

The Application Insights resource associated with the App Service resource, described in the Chapters 2.6.6 and 5.1, was used to help diagnose that a simple request to query a Cosmos table by id was taking more than the 400 RU/s allowed in the free version of a container. That simple query usually

takes around 22.5 RU/s. When the RU/s are surpassed Cosmos responds with 429 Too Busy.

The alternative was to use the async methods synchronously. Meaning it would be necessary to block the thread waiting for the response. Since none of the documentation's synchronous examples used asynchronous methods, a bit of trial and error was necessary. The solution was to create the query object as a DocumentQuery (this object only has asynchronous methods), and while the query says it has more results and no result has come yet, ask for the next result and actively wait for the response. Doing this only while the query has more results like it is done in the asynchronous examples, does not work because the query can say that it has more results, the result had already arrived and the last result is empty.

This problem does not exist in the next iteration since all the application code is re-factored.

4.4.2 Application Logs

For the applicational logs, the library being used by the current version of the BankOnBox was removed and an alternative that integrated with Azure Log Analytics was searched.

In Azure, all that was necessary to do was to create a Log Analytics workspace resource.

No library was found that allowed to do the logging to the Log Analytics workspace the way that it was being done. All the libraries that were found either did not support the .NET Framework version or did not allow custom logs. To solve the problem, the Log Analytics' HTTP API was used, allowing to make custom logs.

To avoid the logging blocking the execution of a transaction, it was implemented using the ”fire and forget” pattern. In this pattern, a new thread is created, in this case, a task, to send a message that the response is not of interest. In the implemented code the task created is not awaited and its code retries the sending of the request after a constantly increasing amount of time in case of the response being a 500 Internal Server Error, a 503 Service Unavailable, or a 429 Too Busy until it is successful.

All the code developed in this iteration was published to the App Service directly from Visual Studio.

The goal of this iteration was achieved and the implemented architecture is visible in Figure 4.4, this architecture differs from the one in Chapter 3.2.4 due to the previous iteration.

4.5 Transaction Manager Re-Factoring (Fifth Iteration)

To try and achieve the goal for this iteration, previously referred on Chapter 3.2.5, to re-factor the transaction manager and integration modules into sub-modules to an architecture that scales better, the following was done.

Only three transactions were re-factored into this architecture, namely: login; user's homepage, which shows their assets and liabilities; and the viewing of the user's default account movements. No other transactions were implemented due to the unavailability of a core system to allow the support of transactions that involved money transfers.

In Azure, four Function App resources were created, one for the orchestrator and the rest for the

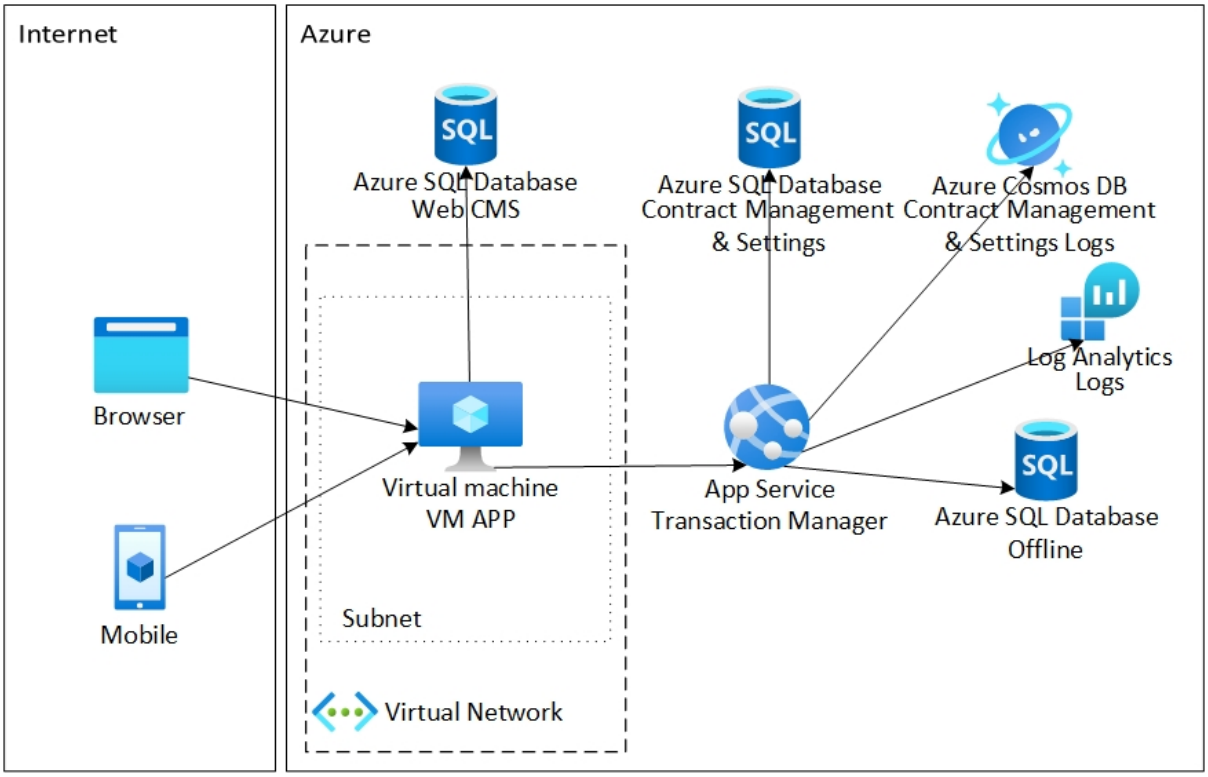

Figure 4.4: Fourth Iteration Implemented Architecture

transactions. All the Function Apps are code publish, have a .NET runtime stack with version Core 3.1, are running Windows as the operating system, and have the consumption serverless plan. All the code was published directly from Visual Studio.

All the functions in the Function Apps have HTTP request triggers. Since these triggers do not have built-in support for SOAP requests, in this iteration the requests were changed to REST, this means that this version does not support the websites running in the VM APP and as such this version of the architecture does not have a GUI.

The code was all developed following the suggestions from Microsoft for .NET Core servers [2] and Function Apps specifically [3]. This means primarily using dependency injection and asynchronous programming.

Objects in the .NET Core dependency injection can have one of three lifetimes:

- Singleton, where the objects are instantiated the first time they are requested, further requests return the same instance, and have the lifetime of the program;
- Scoped, where they are instantiated once per client request (connection), they have the same lifetime as the request, and the same instance is used during that lifetime;
- Transient, a new instance is created every time that is requested.

An injected object cannot depend on another that has a shorter lifetime.

[2] https://docs.microsoft.com/en-us/aspnet/core/performance/performance-best-practices?view=aspnetcore-3.1
[3] https://docs.microsoft.com/en-us/azure/azure-functions/performance-reliability

Since pretty much all classes in BankOnBox were static, they are injected as singletons, meaning that only one instance is created per application lifetime, saving time. The Entity Framework Core contexts, used to query the databases, and which many of these classes depend upon, have by default a scoped lifetime, this could not be as singletons cannot depend on scopeds, but changing the lifetimes of the contexts to singleton introduces a concurrency problem as the context does not support multiple opened result sets at the same time. To solve this problem, while continuing to use dependency injection a factory of contexts need to be injected with a new context instance created every time that it is needed. All the classes that have triggers are by definition scoped.

The code written for the orchestrator was based on the Windows Workflow currently used. The current Windows Workflow was not used due to Function Apps not supporting either Windows Workflow Foundation or an equivalent solution. The rest of the code was adapted from the existing one while trying to modernize it wherever possible since a lot of the code is old, with some of it being essentially Java code from a previous version of BankOnBox that was adapted to .NET.

All the requests made in the Functions Apps, being either to other Function Apps, to Azure SQL Database, or to Cosmos DB are done using async requests. This allows the Function Apps to process other requests while waiting for the responses. This theoretically allows for better throughput of requests.

The version of the Entity Framework Core used does not have a GUI, and as such, the process to generate the database model in a database-first context like this one is necessary to use the Scaffold-DbContext command in a NuGet Package Manager Console in a project that compiles, after installing a compatible version of the Entity Framework Core library.

This version of Entity Framework Core supports Cosmos DB with SQL API, however, only the version for .NET 5 supports ETags, that are being used in the code to verify in some situations if a document has not already been updated. Because of this incompatibility, the latest stable version of the Microsoft library for Cosmos DB was used.

The model for the Cosmos DB tables had to be implemented manually since the library does not have that functionality.

All the logging in the re-factored code uses the logging library built into .NET Core that already integrates with App Insights and with Log Analytics.

The orchestrator Function App at the moment is accessible by any machine on the Internet. The Function Apps for the transactions need an authorization token with every request. This received token is the same for every transaction Function App and needs to be manually added in the Azure resource as an App Host key. The orchestrator Function App currently has this token in its configurations. The orchestrator knows how to call the correct transaction and action (for example, do the transaction validations) by changing two variable parts of a base URL shared by the transaction Function Apps. All the requests to Function Apps use HTTPS.

This iteration unlike the other ones does not use caching since the Redis Cache was not implemented due to time constraints. This, however, needs to be implemented in future work, as no caching adds latency to many of the requests.

The re-factoring part of this iteration was achieved, while the better scaling part needs testing to

confirm. The implemented architecture is visible in Figure 4.5, this architecture differs from the one in Chapter 3.2.5 because the SOAP requests were changed to REST, and the Redis Cache was not implemented.

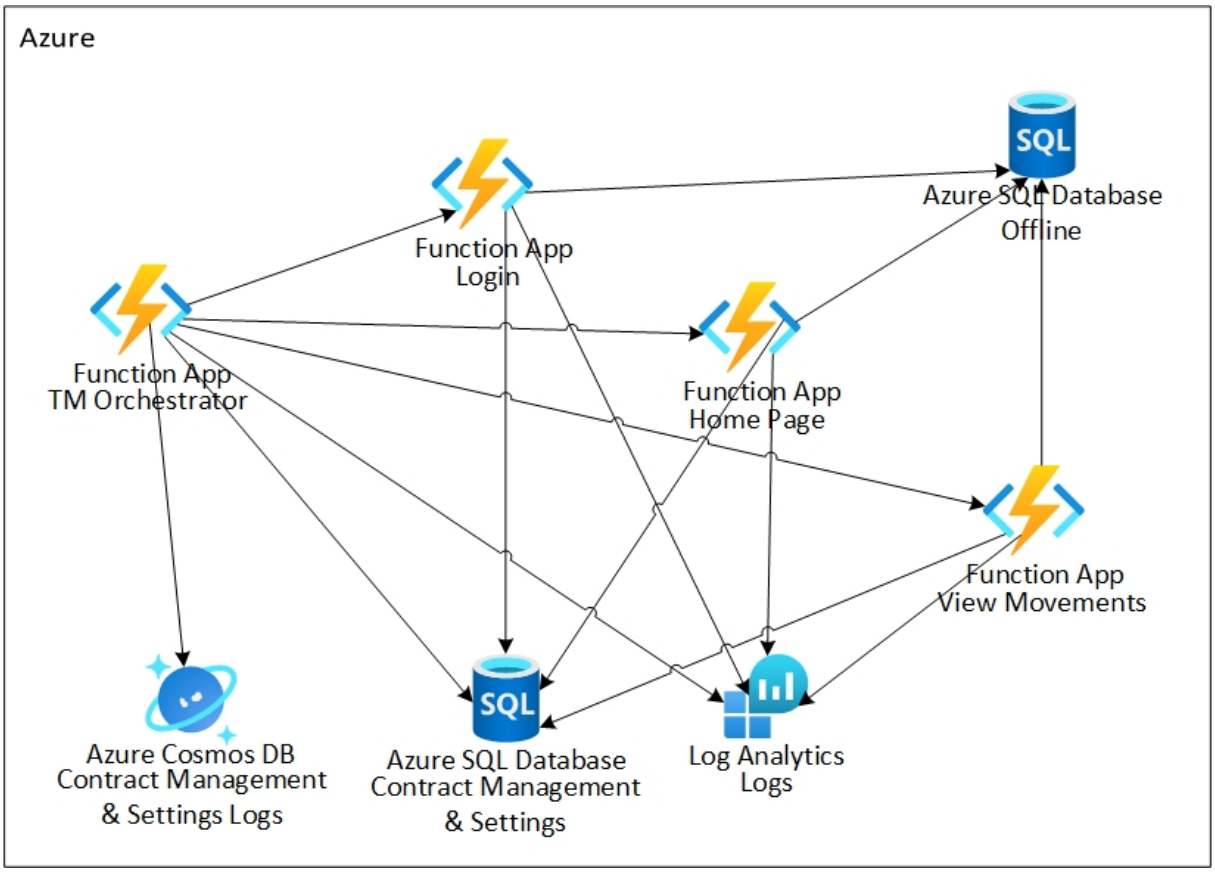

Figure 4.5: Fifth Iteration Implemented Architecture

Chapter 5

Results

In this chapter performance and price for each iteration are listed, an overview of diagnostic tools used is given, and an analysis of the results is done.

In a small bank, with 30 000 registered users, that uses the BankOnBox application there are in a regular day: a total of around 76 700 requests; approximately 3 300 daily users; the average time that takes the BankOnBox application to process the requests, including calls to the core system, is between 50 and 300 milliseconds. If we take that this data is from a 10-hour window that gives an average of around 130 requests per minute and from that average, we assume there are around 50 users on average in a minute.

5.1 Performance Diagnose Tools

During the development and testing of the iterations, Application Insights associated with the resources, and features of the resources themselves were used to collect more information and debug performance.

The Applications Insights have very useful and interesting functionalities to dive deep into the performance of the resources, the ones used were Application Map, Performance, Live Metrics, and Logs.

The Application Map[24] allows having a view of the various external dependencies that a resource has, the number of calls it made to that dependency, how much time it took on average, how many different instances ran during that period of time, among others. An example is visible in Figure 5.1, this is the Application Map for the Orchestrator Function App for a time period of 30 days. This example shows the calls to Cosmos DB and to the transactions' Function Apps. By clicking on one of the arrows it is possible to see the slowest calls to that dependency by endpoint, the top failing status codes with the number of requests that resulted in that status code. These metrics can be further explored by clicking on a button that leads to the Performance pane with the data already filtered to the data that we want to see.

The Performance allows to analyze and filter the request's performance for a select time period. For all the data it is possible to see the average time for a point time and to see the distribution of all the

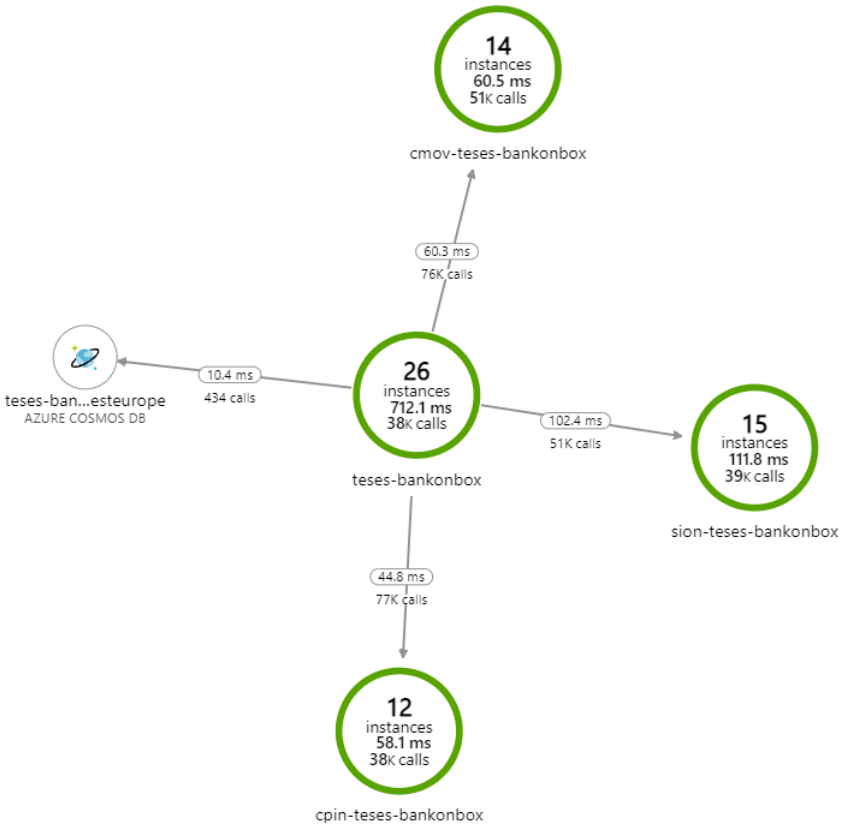

Figure 5.1: Application Insight's Application Map Example

requests' time in a histogram that also gives where the 50th, 95th, and 99th percentiles are located. The data can be filtered by endpoints of the resource that is associated with the Application Insights, by external dependency request, or by endpoint or method of the external dependency.

The Live Metrics[25] allow viewing a stream of metrics in real-time. The metrics are divided into five sections: incoming requests, outgoing requests, overall health, sample telemetry, and servers. The incoming requests section has line graphs for the request rate, request duration, and request failure rate. The outgoing requests section has line graphs for the dependency call rate, dependency call duration, and dependency call failure rate. The overall health section has line graphs for the committed memory, the total CPU percentage, and the exception rate. The sample telemetry section has a list with all the logs arriving at Azure related to the resource associated with the Application Insights. The servers section has all the same data as the incoming requests, outgoing requests, and overall health sections but it is separated by running instances, and the information is not in line graphs.

The Logs allows the creation of specific queries and graphs based on all the logs collected by Azure on the resource. All other functionalities of Application Insights are built using the information in these logs.

Aside from the Application Insights, a functionality of the App Services was used to debug the performance of these. The functionality is called "Collect .Net Profiler Trace" and it collects a profiling trace to identify the root cause of the app's downtime and slowness by identifying the methods in the code where the app spends the most time. This functionality is located in the "Diagnose and solve problems"

pane under “Diagnostic Tools”.

5.2 Testing Methodology

The performance of each iteration of the architecture was tested by performing a load test using a tool called SoapUI[1]. The SoapUI tool allows the creation of multiple threads that make the same set of requests to an endpoint during a given time period or a given number of runs with a random delay between the sets of requests. The tool, while making the requests, collects statistics like minimum, maximum, and the average time that the requests took, the number of requests made, how many returned errors, among others. The times collected by this tool are not equivalent to the ones that serve as a reference, as these are the total times the request took, including the time to arrive at the server and back, and not the time that the application took to process them.

To test the iteration’s performance a given number of threads make a sequence of three requests twenty times. The order of requests is login, user’s home page, and transactions consult. The number of threads used is 1, 5, 10, 25, 50, and 100 to simulate different amounts of load. A thread can be viewed as a user doing the necessary steps a user would have to do to consult their last transaction. Between each run of the thread, there is a random wait time between 0 and 3 seconds. The requests in all iterations are done directly to the transaction manager and integration component of the architecture. Before each test, ten requests of each type of transaction are done to guarantee that the architecture’s components are warm.

For the implemented Lift & Shift architecture (Chapter 4.1), the VMs are of the Standard D4s v3 SKU (4 vCPUs, 16 GiB RAM, 6400 maximum IOPS) and are the equivalent to what is suggested to the banks that use BankOnBox.

For the implemented Databases Re-Platforming architecture (Chapter 4.2), the VM is of the Standard D4s v4 SKU (4 vCPUs, 16 GiB RAM, 6400 maximum IOPS), the Contract Management & Settings and the Offline databases are Azure SQL Database General Purpose Serverless Gen5 with 10 vCores, and the logging database is Azure SQL Database Basic with 5 DTUs.

For the implemented Servers Re-Platforming architecture (Chapter 4.3), the App Service is of the Premium P2V3 SKU (4 vCPUs, 16 GiB RAM, 195 minimum ACU/vCPU) running the code in 64-bit, with always-on turned on, and ARR affinity turned off. The SKU of the App Service is equivalent to the VMs’ SKUs of the previous iterations. The databases are of the same SKUs as the previous iteration.

For the implemented Databases Re-Factoring architecture (Chapter 4.4), the App Service is the same as the previous iteration, the Contract Management & Settings and Offline databases are still Azure SQL Database General Purpose Serverless Gen5 with 10 vCores, and the Cosmos DB is autoscale provisioned throughput between 400 and 4000 RU/s.

For the implemented Transaction Manager Re-Factoring (Chapter 4.5), the Function Apps are of the Consumption Plan type and are configured with a maximum scale-out limit of 200 instances, each instance can process up to 100 requests in concurrency with 100 more in the queue, and the code is

[1] https://www.soapui.org/

running in 64-bit. The databases are the same as the previous iteration.

5.3 Performance Results

The performance results following the testing methodology are visible in the Tables 5.1, 5.2, 5.3, 5.4, and 5.5. The Tables have the minimum, maximum, and average time of a request in milliseconds for each of the three transactions implemented. The transactions are represented in the Tables by their transaction code, login is SION, the user homepage is CPIN, and view movements is CMOV.

For Table 5.4 a column with the percentage of requests that return an Internal Server Error or a Connection Timeout was added.

For Table 5.5 a column was added for each transaction with the initial and final number of Function App instances that were processing requests. This column was also added for the orchestrator's Function App. The values for the number of instances were obtained using the Live Metrics capability of the Application Insights associated with each Function App.

For the first three iterations, the requests per second for 1 thread are around 1, that is, 60 per minute; for 5 threads around 6 requests per second or 360 per minute; for 10 threads they are around 11 per second or 660 per minute. For the first two iterations for 25 threads the requests per second are around 20 per second, 1 200 per minute, and around 23 per second, 1 380 per minute, for 50 and 100 threads. For the third iteration, the requests per second for 25 threads are 25, 1 500 per minute; for 50 and 100 threads the requests per second are around 30, 1 800 per minute. For the fifth iteration, the requests per second for 1, 5, 10, 25, and 50 threads are very similar to the third iteration, but for 100 threads there are around 67 requests per second, 4 020 per minute.

	SION			CPIN			CMOV		
# Threads	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
1	265	342	285.45	247	369	269.60	117	160	127.85
5	257	386	293.47	246	626	279.26	112	217	130.39
10	258	6 670	740.50	243	2 465	396.17	110	3 708	213.20
25	267	1 179	600.50	247	1 053	531.14	114	688	261.07
50	261	2 939	1 626.53	246	2 591	1 585.56	113	2 264	1 343.49
100	269	63 049	4 277.02	251	8 065	3 190.00	117	6 431	2 533.02

Table 5.1: Lift & Shift Performance Results

	SION			CPIN			CMOV		
# Threads	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
1	382	587	426.30	332	474	379.55	191	289	215.10
5	332	869	420.66	289	687	374.10	135	462	212.47
10	331	592	397.38	299	641	377.55	132	317	181.46
25	340	1 975	651.77	312	2 005	595.97	131	1 470	306.42
50	351	3 914	1 849.20	313	3 221	1 365.47	139	2 066	888.28
100	1 351	8 510	3 885.58	415	6 566	3 886.47	137	6 151	3 562.03

Table 5.2: Databases Re-Platforming Performance Results

	SION			CPIN			CMOV		
# Threads	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
1	373	473	402.30	318	425	344.50	189	212	200.50
5	316	548	368.31	269	459	332.46	130	252	175.57
10	319	566	371.66	267	489	333.89	131	397	173.98
25	318	1 409	560.01	278	1 081	510.78	130	728	286.15
50	327	9 771	1 543.60	300	9 270	1 352.13	135	9 444	1 087.75
100	451	6 018	3 251.77	323	5 557	2 877.91	144	4 004	2 457.81

Table 5.3: Servers Re-Platforming Performance Results

	SION			CPIN			CMOV			TOTAL
# Threads	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Error %
1	589	8 340	1 324.60	354	3 103	629.80	299	1 299	397.30	0
5	446	26 587	2 861.77	366	17 770	902.50	311	2 333	412.20	1
10	65	38 248	2 378.95	66	31 415	1 193.33	70	10 275	710.04	29
25	64	60 057	6 051.32	74	60 106	2 680.97	102	60 147	3 088.70	54
50										
100										

Table 5.4: Databases Re-Factoring Performance Results

	SION				CPIN				CMOV				ORCH
# Threads	Min	Max	Avg	# Inst	Min	Max	Avg	# Inst	Min	Max	Avg	# Inst	# Inst
1	462	1 208	700.20	1	355	1 012	665.60	1	491	992	770.40	1	1
5	406	17 169	1 075.09	1-4	290	16 349	1 049.73	1-4	441	19 587	1 095.78	1	1-5
10	375	1 035	550.65	4	282	1 109	385.25	4	416	1 344	584.39	1	5
25	326	1 328	522.89	4	275	1 675	408.95	4	398	1 787	641.23	1	4
50	337	1 595	501.65	4	261	2 185	388.36	4	381	1 730	585.08	1	4-6
100	329	1 751	531.49	4	264	15 887	438.41	4	391	18 974	2 040.43	1-5	6-8

Table 5.5: Server Re-Factoring Performance Results

5.4 Pricing

Azure Pricing Calculator[2] was used to calculate the pricing for each iteration. All the prices presented are the monthly costs and only represent the resources necessary to run the transaction manager and integration components. All these prices in a real-world situation would be lower because the enterprise customer of Azure would have a contract with Microsoft that would give them discounted prices.

For the implemented Lift & Shift architecture (Chapter 4.1), the price for the VM APP with the D4s v3 SKU running Windows with the OS License Included located in West Europe is 261,02€, the price for the VM SQL with the same SKU and the Windows and SQL Server Standard licenses included is 515,29€, the total price is 776,31€. In alternative using the Azure Hybrid Benefit, where on-premises licenses can be used in the cloud, the price for the VM APP is 147,75€, the VM SQL price is 400,22€ for a total of 547,97€.

For the implemented Databases Re-Platforming architecture (Chapter 4.2), the price for the VM APP is the same at 261,02€ with license included and 147,75€ with Azure Hybrid Benefit; the Contract Management & Settings and Offline database in West Europe as Single Database Geo-Redundant backup storage General Purpose Serverless Gen5 with local redundancy, 1 maximum vCore, 0,5 minimum vCores, 32 GB of storage and a 0,5 CPU vCores and 2,02 GB used during a period of 892 800 seconds per month are 84,51€ each; the Logging database in West Europe as Single Database Geo-Redundant backup storage DTU Basic with 5 DTU and 2 GB of storage is 4,13€; the total is 434,17€ with the license included and 320,90€ with Azure Hybrid Benefit. The maximum vCores for the serverless databases were set to 1 because during the testing the maximum ever used was 0,5 vCores. The 892800 seconds come from assuming that during a day the databases spend one-third of the day with CPU being used.

For the implemented Servers Re-Platforming architecture (Chapter 4.3), the price of the App Service in West Europe with Windows, Tier Premium V3, P2V3 SKU for a full month up is 422,74€; the databases are the same price as the premium versions, the Contract Management & Setting and Offline are 84,51€ each, and the Logging database is 4,13€; the total is 589,30€. An alternative, although not tested and depending on the necessities of the client, would be to use multiple instances of a cheaper SKU with a load balancer.

For the implemented Databases Re-Factoring architecture (Chapter 4.4), the App Service has the same price as in the previous version at 422,74€; the Contract Management & Settings and Offline databases with the only difference being 8 GB of storage instead of 32 GB are 81,74€ each; the Cosmos DB with Autoscale, Single Region Write, maximum 4000 RU/s, an average of 15% RU/s utilization, and 32 GB of storage is 44,32€; the total is 624,95€.

For the implemented Transaction Manager Re-Factoring architecture (Chapter 4.5), the price of the Function Apps in West Europe, Consumption Plan, 256 MB per execution, an average of 500 milliseconds per execution, and 9 436 400 executions per month is 12,23€; the prices of the databases are the same, 81,74€ for each SQL one, and 44,32€ for the Cosmos DB; the total price is 220,03€. The number of execution per month comes from the daily number of requests, 76 700, times 31 days, times 4, for

[2] https://azure.microsoft.com/en-us/pricing/calculator/

each orchestrator execution there are three transaction executions. To avoid cold starts from Function App inactivity, a Function App Premium Plan, that allows multiple Function Apps to be deployed in the same plan, and has more performance than a Consumption Plan, could be used. One instance of the Premium Plan with an EP1 instance pre-warmed the whole month, plus an additional 2 instances for 80 hours per month (4 hours per 20 days a month), would cost 161,14€, raising the total price to 368,94€ (this option was not tested).

Recapitulating, the estimated monthly costs of running the transaction manager and integration components and the databases for each iteration are:

- Lift & Shift - 776,31€ or 547,97€
- Databases Re-Platforming - 434,17€ or 320,90€
- Servers Re-Platforming - 589,30€
- Databases Re-Factoring - 623,95€
- Transaction Manager Re-Factoring - 220,03€

These prices do not include the resources necessary to run the current BankOnBox websites and back-office application, and the resources to communicate with the core system. The monthly costs of these resources are approximately the same for each iteration.

5.5 Results Analysis

The results of the Lift & Shift iteration are the baseline to which the other results are to be compared. In this iteration, the performance is stable for 1 and 5 threads but it starts to decrease thereafter. The performance for 10 and 25 threads is still in the acceptable range but is worse. For 50 and 100 threads the performance enters in the unacceptable range, with a single request taking multiple seconds. The interval of threads that best represent the small bank described at the beginning of the chapter would be between 5 and 25 threads.

The results for the Databases Re-Platforming, the performance results are very similar to the first iteration, but on average slightly worse due to the latency introduced by the databases no longer being on the same subnet of the servers. These results being so close to the first iteration show, however, that the bottleneck is located in the servers and not in the databases. This iteration comparing to the first one has the advantage of being cheaper.

The results for the Servers Re-Platforming were expected to be very similar to the second iteration since the App Service SKU is the equivalent to the VM SKU used. This is mostly true, but this iteration has sightly better performance for 50 and 100 threads, although outside of the interval of threads that best represent the small bank. This version also has the disadvantage of costing more than the second iteration.

The results for the Databases Re-Factoring are very bad from the beginning with the requests from 1 thread taking multiple seconds. This performance problem is due to the synchronous implementation of the Cosmos DB calls and makes this implementation as-is unviable.

The results for the Transaction Manager Re-Factoring with a low number of threads are worse than in the first three iterations, but still within an acceptable amount of time. This added time is due to this version not caching the responses from the database requests. This iteration, unlike the others, maintains the request times with the increasing of the load. This version, however, has occasional requests that take upwards of 15 seconds due to the cold starts of new instances that are being created to attend to the increasing load. This version also has the advantage of being much cheaper than the others.

Chapter 6

Conclusions

6.1 Achievements

The objectives and deliverables of this thesis were achieved, with a functional proof-of-concept developed following the proposed migration strategy.

The final version of the proof-of-concept takes better advantage of what the cloud has to offer when compared to the Lift & Shift of the current version as it performs better with the increasing loads and is overall a cheaper solution.

The work developed in this thesis gives a better insight in terms of performance, costs, and architecture to what could be the basis of a future version of the architecture of BankOnBox to the people in charge of making these decisions.

6.2 Future Work

Following the end of this thesis, it is necessary to implement all other transactions that were not subject to re-factoring and deploy in Azure the new versions of the websites and back-office application components when finished. After this is implemented, it will be possible to measure the overall costs and performance of the BackOnBox application running in Azure.

To fully implement the proposed architecture, it will also be necessary to implement the Redis Cache that was not implemented due to time constraints.

Other future work includes the analysis of Azure services that could add value to an architecture that has the one implemented as its basis. Services like Azure AD Authentication for second-factor authentication, Azure Synapse Analytics for reporting on the data in the various databases, among others.

It could also be of interest the analysis of solutions using other services that were not tested like Azure Kubernetes Service and solutions that use other configurations of the databases.

6.3 Recommended Architecture

With all the work developed in this thesis and all the information gathered from the performance results and pricing, the recommended architecture is the one in Figure 6.1.

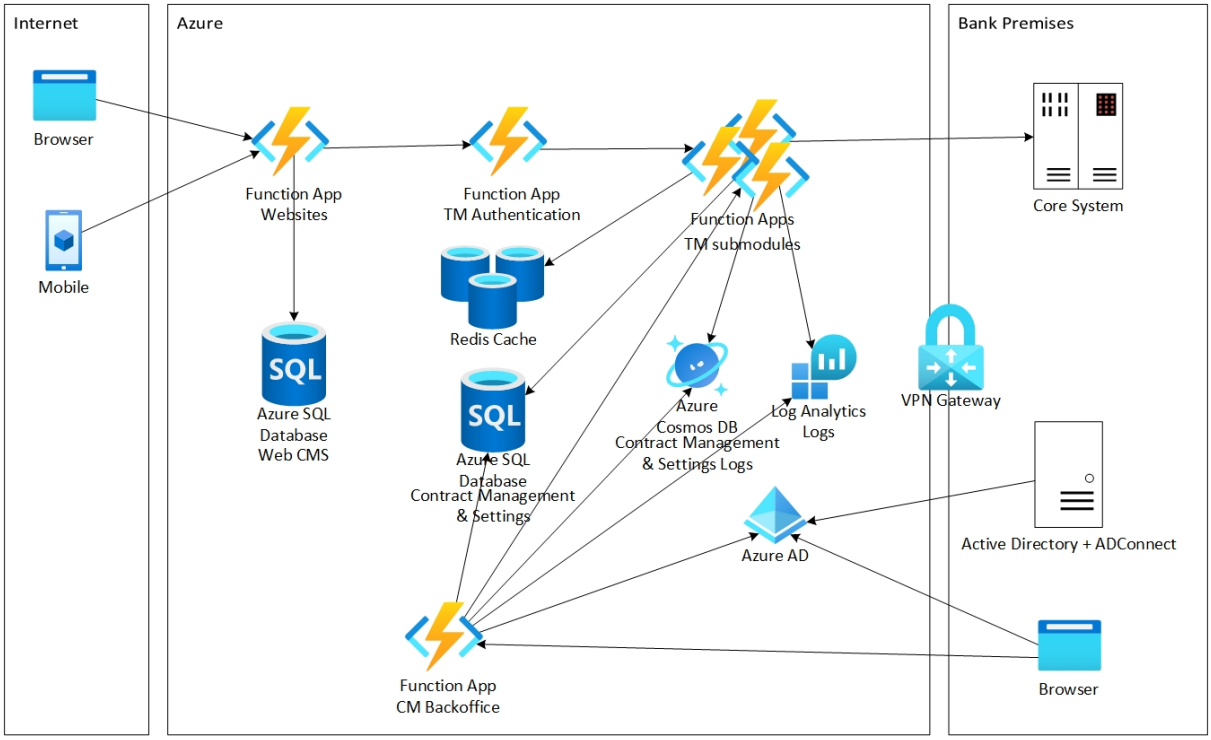

Figure 6.1: Recommended Architecture

In this architecture, Function Apps are recommended instead of App Services due to the pricing and the better scalability offered. The Function Apps are recommended for the new versions of the websites and back-office application that are being developed at Link.

If there is a chance of long periods of inactivity, the usage of the Premium Plan for the Function Apps is recommended. If this is not a problem, or if the cold starts from inactivity can be absorbed, the Consumption Plan is recommended.

The rest of the architecture is the same as the one initially proposed in Chapter 3.

Bibliography

[1] J. Santos. Secure model of web apis development in the cloud. Master's thesis, Instituto Superior Técnico, 2018.

[2] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.

[3] R. Buyya, J. Broberg, and A. M. Goscinski. *Cloud computing: Principles and paradigms*, volume 87. John Wiley & Sons, 2010.

[4] Z. Mahmood. Cloud computing for enterprise architectures: concepts, principles and approaches. In *Cloud computing for Enterprise architectures*, pages 3–19. Springer, 2011.

[5] C. Pahl, P. Jamshidi, and O. Zimmermann. Architectural principles for cloud software. *ACM Transactions on Internet Technology (TOIT)*, 18(2):1–23, 2018.

[6] J. Varia. Architecting for the cloud: Best practices. *Amazon Web Services*, 1:1–21, 2010.

[7] Microsoft. *Cloud Application Architecture Guide*. 2017.

[8] T. Grey. 5 principles for cloud-native architecture - what it is and how to master it. `https://cloud.google.com/blog/products/application-development/5-principles-for-cloud-native-architecture-what-it-is-and-how-to-master-it`, 2019. Accessed: 2021-10-25.

[9] P. Jamshidi, C. Pahl, S. Chinenyeze, and X. Liu. Cloud migration patterns: a multi-cloud service architecture perspective. In *Service-Oriented Computing-ICSOC 2014 Workshops*, pages 6–19. Springer, 2015.

[10] S. Orban. 6 strategies for migrating applications to the cloud. `https://aws.amazon.com/blogs/enterprise-strategy/6-strategies-for-migrating-applications-to-the-cloud/`, 2016. Accessed: 2021-10-25.

[11] Microsoft. *Cloud Migration Simplified*. 2020.

[12] S. Kehrer and W. Blochinger. A survey on cloud migration strategies for high performance computing. 2019.

[13] Microsoft. Overview of windows vms in azure. https://docs.microsoft.com/en-us/azure/virtual-machines/windows/overview, 2021. Accessed: 2021-10-25.

[14] Microsoft. App service overview. https://docs.microsoft.com/en-us/azure/app-service/overview, 2021. Accessed: 2021-10-25.

[15] Microsoft. Azure app service plan overview. https://docs.microsoft.com/en-us/azure/app-service/overview-hosting-plans, 2021. Accessed: 2021-10-25.

[16] Microsoft. Introduction to azure functions. https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview, 2021. Accessed: 2021-10-25.

[17] Microsoft. Azure functions hosting options. https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale, 2021. Accessed: 2021-10-25.

[18] Microsoft. Azure compute unit (acu). https://docs.microsoft.com/en-us/azure/virtual-machines/acu, 2021. Accessed: 2021-10-25.

[19] Microsoft. What is azure sql? https://docs.microsoft.com/en-us/azure/azure-sql/azure-sql-iaas-vs-paas-what-is-overview, 2021. Accessed: 2021-10-25.

[20] Microsoft. Get to know azure sql. Technical report, Microsoft, 2021.

[21] Microsoft. Introduction to azure cosmos db. https://docs.microsoft.com/en-us/azure/cosmos-db/introduction, 2021. Accessed: 2021-10-25.

[22] Microsoft. What is application insights? https://docs.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview, 2021. Accessed: 2021-10-25.

[23] Microsoft. Overview of log analytics in azure monitor. https://docs.microsoft.com/en-us/azure/azure-monitor/logs/log-analytics-overview, 2021. Accessed: 2021-10-25.

[24] Microsoft. Application map in azure application insights. https://docs.microsoft.com/en-us/azure/azure-monitor/app/app-map, 2021. Accessed: 2021-10-25.

[25] Microsoft. Diagnose with live metrics stream. https://docs.microsoft.com/en-us/azure/azure-monitor/app/live-stream, 2021. Accessed: 2021-10-25.