Migration of a Client-Server Application to a Cloud Architecture The Case of an E-Banking Application

Martim Bravo
martim.bravo@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

November 2021

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

1. Introduction

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.

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.

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.

2. Background

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 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.

The logical architecture as is visible in Figure 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.

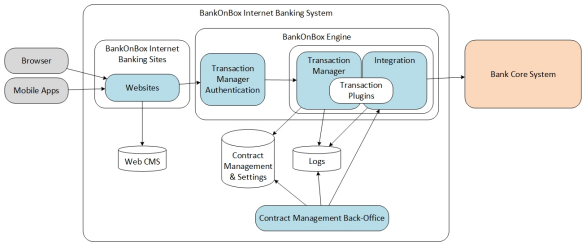

Figure 1: BankOnBox Logical Architecture

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.

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. The code loaded is responsible for the transaction validations and execution.

The transaction manager authentication checks if the request received needs a second level of authentication.

The transaction manager module is the part of the BankOnBox Engine component that makes all the validations to the requests received. When the transaction manager receives a transaction request, it does all the generic validations first and then all the transaction-specific validations. After all the validations are done it passes the request to the integration to be executed.

The integration module is responsible for making the conversion between the application model and core system model, and calling the core system to execute it.

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

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 communication between the websites and the transaction manager authentication, and between the transaction manager authentication and the transaction manager 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.

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

The contract management & settings database contains all the data for an internet banking contract and the functional logs.

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.

All the databases are SQL Server databases.

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 the second level is used for operations.

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, and the access to the web CMS database is done using SQL authentication.

The physical architecture of the BankOnBox application once deployed in the clients is fairly simple as it is possible to see in Figure 2.

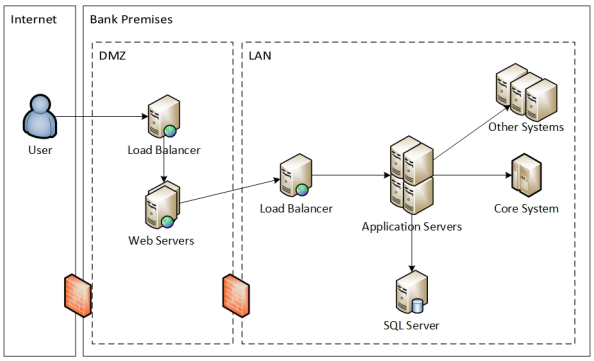

Figure 2: BankOnBox Physical Architecture

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

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 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 [11], [1], and [5], 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 [9], [10], [6], and [2], that, when followed, lead to a better optimized cloud application are:

- Design for self-healing
- Make all things redundant
- Minimize coordination
- Design to scale out
- Design for the operations team
- Use managed services
- Use the best data store for the job
- Design for evolution

2.4. Cloud Migration Strategies

There are many strategies to migrate applications to the cloud, as described in [3], [8], [7], and [4], 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, redeployment, 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 redesigning, 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.

3. Solution

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.

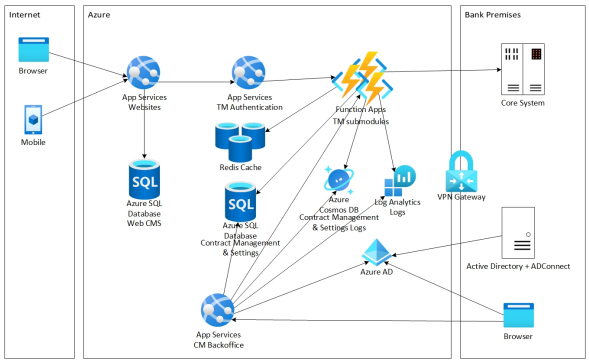

Figure 3: Proposed Architecture

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 ahead. 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 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 application used by the bank's staff is in an App Service.

The authentication of the bank's staff 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 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.

A closer look into the architecture of the transaction manager and integration components is visible in Figure 4.

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

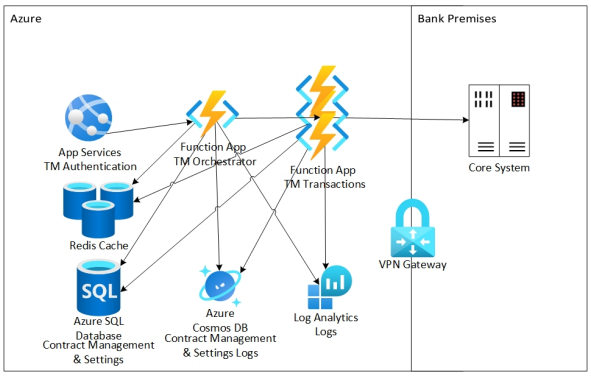

Figure 4: Proposed Architecture Close-up

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.

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 5. It has two VMs in the same virtual network and subnet.

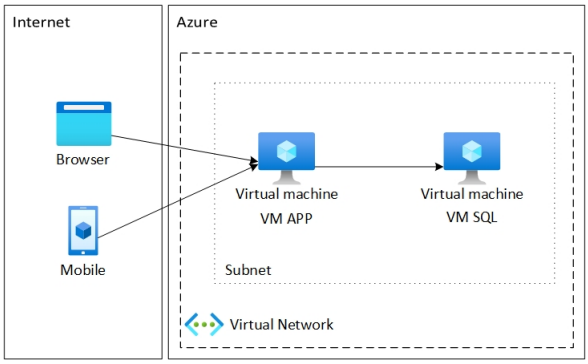

Figure 5: First Iteration Architecture

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.

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 6.

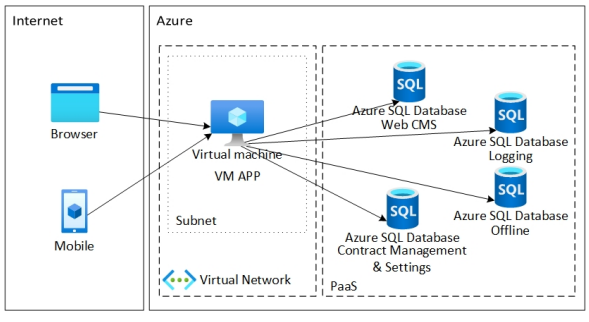

Figure 6: Second Iteration Architecture

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.

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 7.

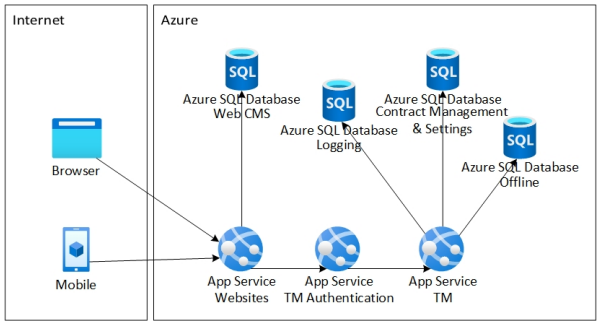

Figure 7: Third Iteration Architecture

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.

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 8.

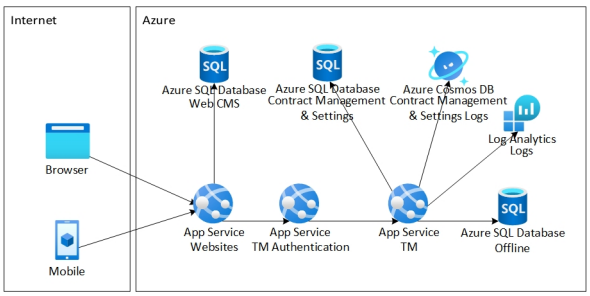

Figure 8: Fourth Iteration Architecture

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.

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

The architecture of this iteration is visible in Figure 9.

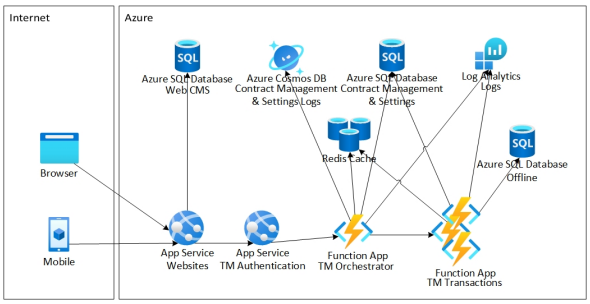

Figure 9: Fifth Iteration Architecture

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.

4. Results

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.

4.1. 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, between 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 his 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, 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, 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, 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, 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, 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 running in 64-bit. The databases are the same as the previous iteration.

4.2. Performance Results

The performance results following the testing methodology are visible in the Tables 1, 2, 3, 4, and 5. The Tables have the minimum, maximum, and average time of a request in milliseconds for the three transactions implemented combined.

For Table 4 a column with the percentage of requests that return an Internal Server Error or a Connection Timeout was added.

For Table 5 a column was added for each transaction with the initial and final number of Function App instances that were processing requests. This

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

column includes the Function Apps of the transactions and of the orchestrator. 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.

# Threads	Min	Max	Avg
1	117	369	227.633
5	112	626	234.373
10	110	6 670	449.957
25	114	1 179	464.237
50	113	2 939	1 518.527
100	117	63 049	3 333.347

Table 1: Lift & Shift Performance Results

# Threads	Min	Max	Avg
1	191	587	340.317
5	135	869	335.743
10	132	641	318.797
25	131	2 005	518.053
50	139	3 914	1 367.650
100	137	8 510	3 778.027

Table 2: Databases Re-Platforming Performance Results

# Threads	Min	Max	Avg
1	189	473	315.767
5	130	548	292.113
10	131	566	293.177
25	130	1 409	452.313
50	135	9 771	1 327.827
100	144	6 018	2862.497

Table 3: Servers Re-Platforming Performance Results

4.3. 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.

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

# Threads	Min	Max	Avg	Err %
1	299	8 340	783.900	0%
5	311	26 587	1 392.157	1%
10	65	38 248	1 427.440	29%
25	64	60 147	3 940.330	54%

Table 4: Databases Re-Factoring Performance Results

# Threads	Min	Max	Avg	# Inst
1	355	1 208	712.067	4
5	290	19 587	1 073.533	4-14
10	282	1 344	506.763	14
25	275	1 787	524.357	13
50	261	2 185	491.697	13-15
100	264	18 974	1 003.443	15-21

Table 5: Server Re-Factoring Performance Results

For the implemented Lift & Shift architecture, 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, 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 ar-

chitecture, 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, 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, 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 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.

4.4. 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.

5. Conclusions

5.1. 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.

5.2. 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.

5.3. 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 10.

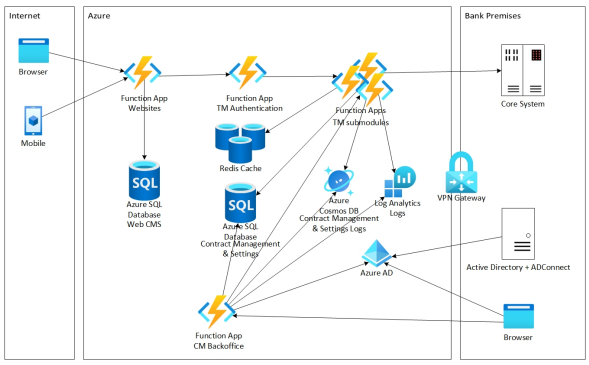

Figure 10: Recommended Architecture

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 Section 3.

References

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

[2] 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.

[3] 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.

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

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

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

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

[8] 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.

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

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

[11] 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.