Cost-Aware Migration to Functions-as-a-Service Architecture

Adam Stafford, Farshad Ghassemi Toosi and Anila Mjeda

Senior Software Engineer, Independent Researcher
Computer Science Department at Munster Technological University, Cork Campus
Computer Science Department at Munster Technological University, Cork Campus

Abstract

Serverless functions, also known as Functions-as-a-Service (FaaS), provide the capabilities of running functional code without the requirement of provisioning or managing the underlying infrastructure with the potential to significantly reduce overall running costs. FaaS can provide a quick time to market, reduced server management overhead, and with the pay-per-use model of FaaS, billing is based solely on the number of requests, execution time, and memory consumption. Although FaaS is a popular area of cloud computing, there is a lack of industry standards for the migration process of monolithic applications to FaaS architecture. Without this, when opting to migrate to this architecture style there is little or no roadmap to guide with best practices. This may result in functions underperforming and incurring a higher cost than necessary. The migration technique outlined in this paper explores the area of FaaS, proposing a new set of guidelines to rectify these shortcomings. This research aims to find the ideal structure for running serverless functions optimised to reduce memory consumption and running costs. Two experiments are executed, the first analyses the behaviour of serverless functions throughout several refactoring iterations. The second experiment extracts serverless functions from a monolithic application. A migration technique is then created for migrating a monolithic application to FaaS architecture based on these findings.

1. Introduction

A Monolithic Architecture is an architectural pattern in which all functionality of a system is encapsulated into a single application. This architecture style has some advantages which include simple development, easy deployment, and simple debugging compared to distributed architectures [1]. However, as monolithic applications grow, the flaws of the architecture pattern become apparent. A change to any of the layers of a monolithic application requires an entire system retest and deployment. This results in an application that is slow to adapt to change with deployments reducing the uptime of the application [2]. Scaling also becomes an issue as a monolithic application requires the entire application to be replicated despite usually only a subcomponent of the system's functionality requiring scaling [2]. As cloud computing and containerisation grow the monolithic architecture was not suitable for these new advancements.

The problems identified with monolithic applications have been addressed with alternative architecture styles such as Service Oriented Architecture and Microservices [3, 4]. Functions-as-a-Service are often seen as the next logical phase of architecture style. Taibi et al, [4] outline how in some cases FaaS is better suited to newer advancements than Microservices. That said, the area is quite immature in comparison to other architectural patterns such as SOA and Microservices.

The main reason for the adoption of FaaS is to reduce costs [5]. This research aims to contribute to the serverless field and provide structured guidance on how to decompose a monolithic application into a FaaS architecture. This research also aims to provide insight into the behaviour of serverless functions during a refactoring process to reduce memory consumption and costs. In doing so, the aim is to bridge the gap between this architecture and its predecessors SOA and Microservices and ensure that functions reduce running costs and consume less memory.

2. Related Work

Castro et al. [6] identify Serverless computing and Functions-as-a-Service to be the natural progression architecture style surpassing the latest trend of Microservices. Eismann et al. [5] report that users tend to adopt a serverless architecture to reduce their hosting costs.

The granularity of the architecture style that permits users to scale at the function level along with the pay-per-use pricing model of serverless functions are some of the reasons for the architecture style's continued popularity growth. Considering this and the continued growth of

ECSA2021 Companion Volume
adam.ryanstafford@gmail.com (A. Stafford); farshad.toosi@mtu.ie (F. G. Toosi); anila.mjeda@mtu.ie (A. Mjeda)
https://www.linkedin.com/in/adam-ryan-stafford/ (A. Stafford); https://www.linkedin.com/in/farshad-ghassemi-toosi-428a5852/ (F. G. Toosi); https://www.linkedin.com/in/anila-mjeda-32a5064/ (A. Mjeda)
0000-0002-4320-4682 (A. Stafford); 0000-0002-1105-4819 (F. G. Toosi); 0000-0003-1311-6320 (A. Mjeda)

the architecture style, more companies are foreseen to migrate their existing applications to a serverless architecture. However, FaaS architecture lacks the migration techniques that come with other architectural styles such as Microservices and currently there are no established migration techniques which consider the migration of a monolithic application to a FaaS architecture style.

Cost Analysis of FaaS – The main focus of this migration technique is to reduce hosting cost of the serverless functions. Varghese and Buyya [7] compared traditional hosting on virtual machines to serverless hosting. With traditional hosting, the owner pays for the entire time the virtual machine is running, even when the application lies idle. With serverless, the owner is no longer required to pay for computing resources that are not in use (e.g., when a function lies idle).

Eivy [8] warns of the hidden cost implications that need to be fully understood to use FaaS to its full potential and reap the benefits of the service, since the cost benefits heavily depend on the execution behaviour and the volumes of the application workload. Highlighting that contrary to popular belief FaaS is not always cheaper than provisioning alternative infrastructure. If a serverless function has a very high transaction per second rate it may be cheaper to explore alternatives such as virtual machines, container hosting, etc. However, considerations also need to be in place regarding the operation costs of managing the infrastructure provisioned. The migration technique outlined in this paper ensures that the application is optimised to avoid the hidden costs outlined by Eivy [8].

Cost and performance optimisation is a well-researched area of FaaS. Mahajan et al., [9] propose enhancements into serverless cost optimisation by splitting the workload between virtual machine rental and serverless functions using a hybrid system of virtual machines and serverless functions.

Existing Industry Standards & Best Practices for FaaS – Hong et al. [10] propose six design patterns (Periodic Invocation Pattern, Event-Driven Pattern, Data Transformation Pattern, Data Streaming Pattern, State Machine Pattern, and Bundled Pattern) to model a serverless architecture in the cloud based on security services. Rabbah et al. [11] propose a patented technique for dealing with composing serverless functions to build new applications. Their patent describes the use of a processor which determines the primary function of a query sent by the user and identifies if that function is to be broken into subsidiary functions. This use of function chaining gives a FaaS environment the capabilities to host entire backend applications. Bernstein et al. [12] propose a virtual actor model as an ideal platform to build a stream processing system. Using the virtual actor model as opposed to the traditional actor model on assigned servers provides the capabilities of distributing the workload across multiple servers in the cloud.

Bardsley et al. [13] analyse the performance of FaaS functions to create a set of strategies that play to the strengths of the FaaS service. The researchers analysed a single serverless function which was called 1,000 times at a rate of two requests per second for the duration of the test run. However, the focus of the work carried out by Bardsley et al. [13] was on optimising the performance of a serverless environment. Implementing a circuit breaker pattern is one of the strategies proposed.

Nupponen & Taibi [14] identify several good and bad practices when developing a FaaS architecture. The bad practices include overuse of asynchronous calls increasing complexity, functions calling other functions leading to complex debugging and additional costs, and the adoption of too many technologies such as libraries, frameworks, languages resulting in maintenance issues.

Drawbacks of FaaS – A known drawback to serverless functions is the issue of *cold starts* [15]. When a function is deployed to the underlying infrastructure of the cloud service provider, it runs on a container. If a function has not been triggered in some time, these containers can go idle and the function will release any resources. When a serverless function is executed after being idle, a gateway component checks if a container capable of serving the request exists. If no container is available, the gateway allocates a new one and directs the request to the respective machine. This process causes a delay in the request execution time. This delay is commonly known as cold start time [16].

Each cloud service provider restricts the functions to a limited execution time. AWS Lambda offers one of the highest execution times of 15 minutes [17]. This limitation of serverless functions running in the cloud means that any long-running function or algorithm such as machine learning or big data process are not suitable for serverless functions. The execution time also affects the overall cost of running functions. Therefore, it is in the user's best interest to have the functions running for as short time as possible. FaaS pricing model is discussed further in *Section 4*.

3. Research Contribution

Research Questions – This research focuses on three *research questions*:

- How does refactoring several different serverless functions for better performance[1] affect the running costs of the functions?
  This was addressed by preparing several refactoring iterations based on common best practices

[1]Performance optimisation was based on recommended practices identified in this research.

and analysing the behaviour of the functions before and after refactoring.

- What are some of the best practices that serverless functions should adhere to that reduce the running cost of the functions?
  After analysing the functions after each refactoring iteration, the memory usage and execution duration metrics identified whether the refactoring had a positive or negative impact when running in a FaaS environment. Based on this analysis, several best practices were discovered to reduce memory consumption, execution duration and thus cost when hosting serverless functions in a FaaS architecture.
- How can a monolithic application be decomposed into serverless functions which are optimised to reduce running costs in a FaaS architecture?
  Based on the previous findings, the best practices for reducing cost were used in a migration process from Monolithic to FaaS architecture. Implementing the best practices during the migration phase optimised the application for reduced cost while running in a FaaS architecture.

Contribution to Field – This research examines several research breakthroughs in the area of FaaS. These include research into cost optimisation, industry patterns, and migration techniques of alternative architecture styles. We contribute to the existing literature by considering the decomposition from monolithic to a FaaS architecture. Application granularity was also identified as a trend with architecture styles progressing from monolithic to SOA to Microservices and the future of FaaS. The migration technique outlined in this paper proposes a technique for decomposing a monolithic application into serverless functions while reducing memory consumption and execution duration. Therefore, reducing the overall hosting costs consumed by the serverless functions. Considering the gap in the field of study when it comes to migration techniques and best practices, this research bridges the existing software architectural gap for FaaS.

4. Cost Breakdown Analysis

We analysed the cost implications of hosting serveries functions by four major cloud providers (CSP). All four of the major cloud service providers of FaaS provide a pay per execute model [17, 18, 19, 20]. More specifically we analysed the pricing models of AWS Lambda, Azure Functions, Google Functions and IBM Functions. A simple use case was applied to each pricing model: *A function with 128MB of memory, invoked 8 million times per month and running for 300ms each time.*

In a real-world scenario, calculating costs would not be as straightforward as the running time may vary, hence, a real-world scenario is presented in the experiments stage of this paper.

For the sake of brevity, the details of calculations can be seen on Github [2] and the results of the use-case cost analysis are displayed in Table 1. For all the four CSP providers the common factors affecting the running costs were found to be *memory consumption* and *execution duration.* This element of the pay-per-use model is the basis for the migration technique proposed in this research. By improving the performance and reducing memory consumption the serverless functions will run at a reduced cost compared to functions running without the implementation of the migration technique.

Table 1
Use-Case Cost Summary

Cloud Service Provider	Total Cost
AWS Lambda	$6.60
Azure Function	$6.40
Google Function	$8.75
IBM Functions	$5.10

5. Experiments and Evaluation

Azure Functions was the chosen service to host the serverless function throughout this research. Two Function Apps were required, each had a runtime of .Net Core 3.1, specified the consumption plan and hosted the serverless functions. A single VM was provisioned to run the experiments. The purpose of the Azure VM was to host a docker image containing a benchmark application that would generate traffic to the Azure Function endpoints. An Azure SQL Database was used during the experimental phase of this research. Both experiments required a database in which the serverless functions could interact. A single Azure SQL Server was used to host both the database used in the experiments.

Application Insights was used to analyse the performance of the Azure Functions. Data was exported from Application Insights by querying the logs for the relevant data. The analytical data of interest during the experiment was the duration of the requests and the memory consumption of the serverless functions on the host machine.

5.1. Experimental Design

Analysis of Serverless Functions experiment – Analysed the memory consumption of a group of serverless functions. Every iteration, the functions were refac-

[2] https://github.com/adam-stafford1/Cost-Aware-Migration-to-Functions-as-a-Service-Architecture/blob/main/CostBreakdownAnalysis.pdf

tored in an attempt to reduce memory usage, execution duration and ultimately reduce the running costs. Requests were sent to the serverless functions hourly for 24 hours with varying payloads. After three runs, the mean average was calculated for that iteration and the serverless functions were analysed, in terms of memory consumption, execution duration and cost before refactoring and conducting further analysis. The findings were used to develop a set of serverless function best practices. These practices facilitated the development of a migration technique.

Iteration 1 - Baseline – The initial iteration of this experiment produces a baseline to which other iterations could be compared before refactoring.

Iteration 2 - Combining Serverless Functions – The serverless functions DatabaseQuery and DatabaseQuerySingle are combined into a single function. The newly created function executed the appropriate functionality for each of the endpoints based on the parameters provided. The parameters passed to the function specified if the request was to execute the DatabaseQuery or the DatabaseQuerySingle functionality.

Iteration 3 - Asynchronous Functions – In this iteration, the serverless endpoints are refactored to run asynchronously. The purpose of this is to increase the performance of the serverless function by introducing the asynchronous programming model and best practice.

Iteration 4 - Dependency Injection – During this iteration, the serverless functions are refactored to use dependency injection for the creation and disposal of the database context. The database context is registered with the scope lifetime. Once the database context was set up it is injected into each of the required serverless function classes.

Iteration 5 - GET vs POST – For this iteration, two serverless functions are refactored to accept data via a GET request instead of POST. CalculateTotal and Distance are both refactored to accept data via a GET request.

Iteration 6 - Changing the ORM – It is discovered that the out-of-the-box .Net ORM, Entity Framework Core was not the most efficient ORM in terms of performance. An alternative ORM was discovered known as Dapper. Dapper boasted better performance in comparison to Entity Framework Core. Basheleishvili et al. [21] identified the performance improvements of Dapper over the standard Entity Framework. Enetity Framework Core is replaced with Dapper as part of this iteration.

Figure 1 presents an experiment design diagram illustrating the experimental process for the Analysis of Serverless Functions experiment.

Decomposition of Monolithic to Serverless Functions experiment – Analysed a monolithic application and created a decomposition technique to produce a group of serverless functions from the monolith. The decomposed functions are analysed for cost in a FaaS environment before the discovered best practices from the Analysis of Serverless Functions experiment were applied. After applying the best practices the functions are analysed a second time to evaluate the quality of the change and the reduction in price the implementation had on the serverless functions.

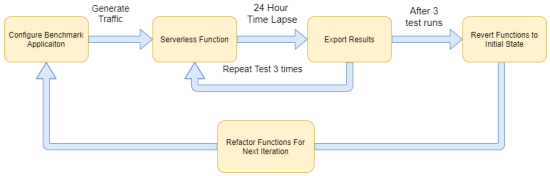


Figure 1: Experiment Design Diagram

The application used for this experiment was a sample application developed by Microsoft, eShopLegacyMVC and publicly available on Github[3]. The application was developed as part of a series of tutorials in which Microsoft guides on best practices and modernising applications and is publicly available. This particular application was developed as a starting point in a tutorial for modernising legacy .net framework applications to the Azure Cloud and Windows Containers and comes with an accompanying PDF document[4].

5.2. Analysis of FaaS Serverless Functions Results

For this experiment, the data provided for the Total Memory usage is presented by the total number of Private Bytes consumed on that given day for all serverless functions. The Total Execution Duration is presented by the total execution time in milliseconds for the given day. The data exported represents the combined sum of all running serverless functions. All cost estimations were based on 8 million requests per month.

Iteration 1 - Baseline – This initial iteration is executed to set a baseline for comparison with future iterations. Memory consumption and execution duration metrics of the serverless functions were exported daily. As these metrics do not show consistent results on each run the average of the three days was calculated and used for comparison.

Iteration 2 - Function Combination – The purpose of this iteration is to analyse the behaviour of the serverless functions when the functionality of two functions was combined into a single function. In combining the

[3]https://github.com/dotnet-architecture/eShopModernizing/
[4]https://aka.ms/liftandshiftwithcontainersebook/

Table 2
Iteration 1 Results

Date of Run	Total Memory Usage	Total Execution Duration
22/02/2021	544,871,743,488	2,472,551.827
18/03/2021	527,623,954,432	2,634,156.957
19/03/2021	567,170,166,784	2,564,192.976
Average	546,555,288,235	2,556,967.25333
Cost Estimation	$36.80	

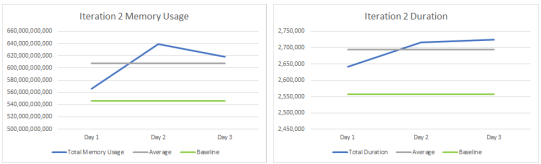
(a) Total Memory Usage (b) Total Execution Duration

Figure 2: Iteration 1 Line Chart

functions, this iteration aimed to eliminate the performance implications of cold starts on a single function. This newly created function handled the network traffic for both of the previous functions.

Table 3
Iteration 2 Results

Date of Run	Total Memory Usage	Total Execution Duration
24/02/2021	565,889,548,288	2,641,953.29
25/03/2021	639,237,783,552	2,716,128.474
28/03/2021	618,846,830,592	2,724,818.97
Average	607,991,387,477	2,694,300.24467
Cost Estimation	$40.00	
Baseline Comparison	+61,436,099,242	+137,332.99134



(a) Total Memory Usage (b) Total Execution Duration

Figure 3: Iteration 2 Line Chart

As shown in Table 3, there is an overall increase in memory consumption and execution duration compared to the baseline iteration. Although this increase was not substantial in the experiment, it would be quite substantial on a larger scale. This enforces what is already a recommended best practice, to implement the single responsibility pattern. Enforcing the single responsibility pattern during the migration process is a recommended best practice during the construction of the migration technique.

Iteration 3 - Asynchronous Functions – It is a recommended best practice for Azure functions to be asynchronous non-blocking calls [22]. When analysing the functions, several were identified which did not adhere to this best practice. Ensuring that the functions ran asynchronous non-blocking calls was the next refactoring iteration of this research. DatabaseQueryForReport, DatabaseQuerySingle, DatabaseWrite, DatabaseQuery, DownloadFile, and GenerateBarcodeImage were all updated to run asynchronously using the async/await operators of the C# language [23].

Table 4
Iteration 3 Results

Date of Run	Total Memory Usage	Total Execution Duration
27/02/2021	565,889,548,288	2,545,098.398
14/03/2021	601,180,618,752	2,716,128.474
30/03/2021	649,217,093,632	3,336,305.686
Average	607,729,938,432	2,865,844.186
Cost Estimation	$40.00	
Baseline Comparison	+61,174,650,197	+308,876.93267

(a) Total Memory Usage (b) Total Execution Duration

Figure 4: Iteration 3 Line Chart

As shown in Table 4, there is an increase in the memory consumption and the execution duration compared to the baseline run of the experiment. After further inspection into the results and researching the best practice, it is discovered that serverless functions were charge for the time that the function waits for the results of an async request [24].

These results demonstrate that enforcing functions to run asynchronously can increase memory consumption and duration. However, the serverless functions produced for this experiment have no external dependencies. Using asynchronous programming in cases where the function does not require a response from an external dependency and can continue to execute calling multiple dependencies asynchronously should have a positive impact on the performance. Although, as demonstrated in this experiment enforcing the synchronous functions to run asynchronously has had a negative impact on performance.

Iteration 4 - Dependency Injection Pattern – Dependency injection is a commonly utilised best practice in which classes are decoupled from their dependencies to provide better modularisation of software [25]. For this iteration, the dependency injection pattern is applied

to the serverless functions to abstract the initialisation of the database context. The goal of this iteration is to remove this initialisation from the serverless functions to reduce the duration in which the functions executed and increase the performance of the functions. The

Table 5
Iteration 4 Results

Date of Run	Total Memory Usage	Total Execution Duration
01/03/2021	578,505,195,520	2,663,114.111
22/03/2021	616,094,019,584	2,645,294.872
23/03/2021	549,142,319,104	2,549,578.342
Average	581,247,178,069	2,619,329.10833
Cost Estimation	$36.80	
Baseline Comparison	+34,691,889,834	+62,361.855

(a) Total Memory Usage (b) Total Execution Duration

Figure 5: Iteration 4 Line Chart

results displayed in Table 5 shows that although the dependency injection pattern increased modularity and testability it has a negative impact on the running costs of the serverless functions. Dependency injection has its own merits, including the testability of functions. It is a known best practice in the industry. However, this research is focused solely on optimising costs. Therefore, this research recommends that before implementing the dependency injection pattern in a FaaS environment, the change should be evaluated to identify the impact on the hosting costs.

Iteration 5 - GET vs POST – In this iteration, the serverless functions are modified to examine the effect of different HTTP methods. Several endpoints were modified from accepting data via a POST request to accepting via a GET request. This change also affected the function code as the different HTTP methods required different parsing functionality. With a GET method, the data is taken straight from the query parameters but required manual parsing of integers or decimals. With the POST method, data in the body of the request is parsed using a stream reader. Therefore, this iteration tested the efficiency of the HTTP methods and the parsing method required with each. Table 6 presents the findings of this iteration.

Table 6 presents the results of the GET vs POST iteration. The iteration produced interesting results. As shown, the average memory consumption has increased. However, the average execution duration has decreased compared with the baseline iteration. It is gathered from these results that GET requests execute faster than POST

Table 6
Iteration 5 Results

Date of Run	Total Memory Usage	Total Execution Duration
04/03/2021	595,949,375,488	2,446,181.996
06/03/2021	610,399,891,456	2,621,606.274
07/03/2021	597,077,692,416	2,509,399.675
Average	601,142,319,787	2,525,729.315
Cost Estimation	$36.80	
Baseline Comparison	+54,587,031,552	-38,463.661

(a) Total Memory Usage (b) Total Execution Duration

Figure 6: Iteration 5 Line Chart

requests. However, POST requests perform better in terms of memory consumption.

Figure 10 presents a comparison of the two refactored function. Since the memory consumption of individual functions is not a supported metric exportable from Azure Application Insights, the function duration was used for comparison. The bar chart displays the duration of the two modified functions from the fifth iteration compared to the baseline iteration. The bar chart in figure 10 highlights the reduction in duration when using GET over POST for these functions. Following this discovery, the migration technique favours GET over POST methods were applicable.

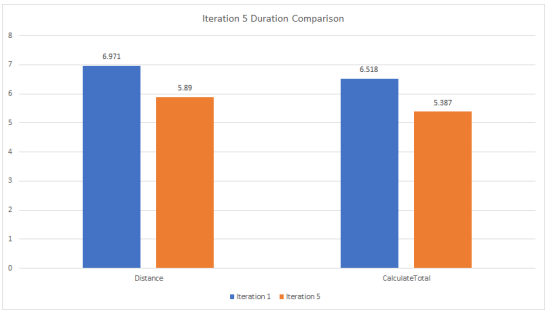


Figure 7: Iteration 5 Duration Comparison

Iteration 6 - Changing the ORM – The Object Relational Model (ORM) is replaced from Entity Framework to Dapper. The Dapper ORM boasted performance improvements compared to the Entity Framework ORM. Several serverless functions are refactored to use the improved ORM and tested to identify how this change affected the performance in a FaaS architecture.

Table 7 shows an overall improvement in performance

compared to the baseline iteration. This is the first iteration to demonstrate a successful improvement in memory consumption and execution duration. Figure 9 presents the duration of the serverless functions which had been refactored to use the Dapper ORM.

The results in Figure 9 show a significant improvement in the duration of the serverless functions as a group and individually. For each of the four functions, the duration dropped by more than half. This is a very clear indicator that this refactoring iteration is successful in improving the performance and reducing the costs. When constructing the migration technique, the ORM was analysed and replace with the Dapper ORM due to its improvements in performance when running in a FaaS environment.

Table 7
Iteration 6 Results

Date of Run	Total Memory Usage	Total Execution Duration
30/03/2021	498,894,196,736	2,234,666.598
31/03/2021	561,415,585,792	2,102,961.566
01/04/2021	580,808,814,592	2,177,295.468
Average	547,039,532,373	2,171,641.21067
Cost Estimation		$33.60
Baseline Comparison	+484,244,138	-385,326.04266

(a) Total Memory Usage (b) Total Execution Duration

Figure 8: Iteration 6 Line Chart

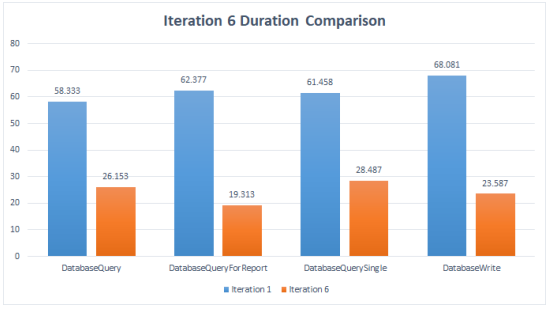


Figure 9: Iteration 6 Duration Comparison

5.3. Discovered Best Practices

The experiment successfully produced several best practices (*Figure 10*) to adhere to when migrating functions to a FaaS environment. Throughout the experiment, it was identified that some of the industry recommended patterns may have benefits in other aspects of software construction however, they increased the memory consumption of the serverless function group and therefore, would increase the overall hosting costs. These results demonstrate unclarity regarding cost implications of CSPs recommended best practices. This experiment is successful in producing best practices based on these two factors. Although some of the best practices may go against industry recommendations, this experiment was focused solely on memory consumption and the execution duration of the serverless function. As discussed in section 4, these two metrics contribute to the costs of running serverless functions across all of the leading CSPs.

Functions should adhere to the single responsibility pattern.

Only run functions asynchronously where necessary, do not enforce synchronous functions to run asynchronously.

Fully evaluate the dependency injection pattern as it could increase hosting costs and energy consumption.

Favour sending data via GET method over POST method where applicable.

Figure 10: Summary of the Discovered Best Practices

5.4. Decomposition of Monolithic to Serverless Functions Results

In this section, the results of the Decomposition of Monolithic to Serverless Functions experiment are presented and evaluated. The functions suitable for migration to a FaaS architecture are identified from a monolithic application and extracted into serverless functions. Once extracted, the functions are deployed to an Azure Function app. The performance of the serverless functions was monitored over three days to estimate the hosting cost. Next, the previously discovered best practices were implemented and the functions redeployed to the Azure Function app where they were monitored for a further three days. The experiment provided the data required to evaluate the effectiveness of the best practices discovered in the Analysis of FaaS Serverless Functions experiment and were used to construct the migration technique.

Function Extraction – The Data Access Layer was identified as suitable for extraction to a FaaS environment during this experiment. The functions were identified by understanding the limitations and suitability of FaaS. Once identified, an Azure Function app was created and all function dependencies were migrated into the new function app. This included the database context used by the functions to manipulate the database as well as any data model object classes. The functions were refactored to run in a FaaS environment, accepting data via RESTful endpoints instead of direct dependencies. Finally, the functions were migrated and deployed to the

Azure Function app.

Setting a Baseline – The serverless functions extracted from the monolithic application are deployed to the Azure Function app. With the functions running in Azure, the benchmark application is deployed to a virtual machine to generate traffic to the endpoints. This experiment demonstrated the cost implications of the migration technique by comparing the memory consumption and execution duration before and after the best practices are implemented. The total memory consumption and total execution duration exported from Azure Application Insights. Table 8 presents the results of the serverless functions before the best practices were implemented. This initial iteration shows the average memory consumption of 531,748,489,899 bytes per day and an average execution duration of 631,612.481 milliseconds per day.

Table 8
Experiment 2 Before Best Practices

Date	Total Memory Usage	Total Execution Duration
28/03/2021	531,556,438,016	642,659.234
29/03/2021	531,175,735,296	615,578.125
30/03/2021	532,513,296,384	636,600.084
Average	531,748,489,899	631,612.481

Applying Best Practices – Table 9 presents the changes made to the application when applying the best practices

Evaluating Best Practices – After the initial execution, the serverless functions were refactored to implement the previously discovered best practices outlined in section 5.3. After refactoring, the functions were redeployed and analysed a second time to demonstrate any performance. Table 10 presents the results of this experiment. Table 10 displays a significant improvement in performance across the three days. The average was compared to the average prior to the best practices being implemented. The results show the number of bytes consumed per day on average by the functions was reduced by 27,784,484,182 bytes or 27.78 GB, and the total execution duration was reduced by 241,638.730333 milliseconds or 241.64 seconds. This simple refactoring experiment identifies the importance of optimisation in a serverless environment. The results demonstrate a significant improvement in memory consumption and execution duration with a simple implementation of several best practices.

Evaluating the Cost Reduction of the Migration Technique – The cost presented in Azure only displayed the overall cost for the monthly billing period and not a detailed breakdown. For that reason, a manual calculation must be performed to present the cost analysis. Additional data was required to calculate the cost. This data exported from Azure Application Insights included the average allocated memory per execution in megabytes and the average duration per execution in seconds. Since the experiments didn't run for a consecutive month, a value of 8 million requests per month was used to calculate the costs. This value was purely for demonstration purposes and didn't impact the differences in costs since it was the same for both the before and after best practices calculations. Table 11 presents the exported data.

Table 9
Best Practices Applied

Best Practice	Changes Applied
Single Responsibility Pattern	Each of the functions extracted performed some form of CRUD operation on the database and each had a single purpose. For that reason, the function had already adhered to the single responsibility pattern.
Async Functions	Each of the extracted functions ran synchronously, the second best-practice identified that running synchronous functions asynchronously had a negative impact on the performance of the functions. Therefore, for the extracted functions, no asynchronous code was introduced.
Dependency Injection	The dependency injection pattern was removed and replaced with the C# using statement to instantiate and dispose of the database context within the function code.
Favour GET over POST	Due to the complex data type being sent in each request, the endpoints could not be changed to use the GET method over POST.
Changing the ORM	The Entity Framework ORM and that was replaced by the Dapper ORM

Table 10
Experiment 2 After Best Practices

Date	Total Memory Usage	Total Execution Duration
22/04/2021	497,534,685,184	392,567.499
23/04/2021	513,865,043,968	416,414.657
26/04/2021	500,492,288,000	360,939.096
Average	503,964,005,717	389,973.750667
Compared to Baseline	-27,784,484,182	-241,638.730333

Table 11
Additional Exported Metrics

	Request Count	Memory Usage	Execution Duration
Before Best Practices	8,000,000	229 MB	0.44208 s
After Best Practices	8,000,000	224 MB	0.25 s

The formula for calculating the cost is given as:

$$TotalCost = (TotalGBs \times \$0.000016) + (Rqst.Count \times \$0.00000020)$$

$$TotalGBs = (Rqst.Count \times Exe.Duration) \times (MemoryUsage \div 1024)$$

Using the data and formulas presented in this section, the costs of hosting the serverless functions could then be calculated. Firstly, the cost of hosting the functions was calculated before the best practices were implemented.

$$TotalGBs = (8000000 \times 0.44208) \times (256 \div 1024) = 884160$$
$$TotalCost = (884160 \times \$0.000016) + (8 \times \$0.20) = \textbf{\$15.75}$$

Next, after the serverless function had been refactored to implement the best practices, the hosting costs were calculated again.

$$TotalGBs = (8000000 \times 0.24) \times (224 \div 1024) = 480000$$
$$TotalCost = (480000 \times \$0.000016) + (8 \times \$0.20) = \textbf{\$9.28}$$

As shown from the calculations, the total charge after implementing the best practices was calculated as $9.28. This shows a reduction in overall running costs of the serverless functions of $6.25. The experiment was conducted on a relatively small scale and reduced the cost by 40%. This hugely significant cost reduction was produced by simply understanding the FaaS environment and how code refactoring can impact the memory consumption and duration of the functions. It also illustrates the importance of optimisation to reduce costs when hosting serverless functions.

Evaluating & Comparing Migration Techniques – To the best of our knowledge, no established migration techniques of monolithic to the FaaS architecture exist in the field of FaaS. In this section, the migration technique outlined in this research is evaluated against similar techniques from alternative architectures. As part of this evaluation, two techniques from the Microservices architecture style have been identified as suitable comparisons to the technique outlined in this research.

Agarwal & Lakshmi [26], identify the shortcomings of some Microservices deployments and focus their research on optimising costs in terms of sizing and scaling of the Microservices. Although with FaaS the scaling is taken care of by the cloud services providers, this research has similarities in terms of the consumption of additional unused resources. Agarwal & Lakshmi propose an algorithm for enabling real-time scaling decisions to reduce the amount of unused resources occupied by the Microservices. In comparison, the Migration of Monolithic Applications to Functions-as-a-Service Architecture for Reduced Hosting Cost technique outlined in this research goes slightly further than the size and scaling and identifies code refactoring techniques which ultimately reduce the amount of resources consumed by the functions.

Similar to the identification of functions from a monolithic application outlined in this research, Mazlami et al. [27] identified a technique for identifying Microservices in a monolithic application. In the five phase migration technique outlined in this research, the serverless functions are identified by simply understanding the application and the FaaS environment. In addition, the monolithic application was categorised in terms of the multiple layers before the extraction was applied. In the research carried out by Mazlami et al., they discuss a much more in-depth technique of identifying the Microservices. The formal model proposed in the research uses a clustering algorithm to generate recommendations for potential microservice candidates throughout the refactoring process. The migration technique proposed in this paper, provides a more hands-on approach and focuses more on the code refactoring and performance optimisation of the migration. However, this comparison opens up future work to identify a formal model for the extraction of functions from a monolithic application.

6. Conclusions

FaaS currently lacks the industry standards of its established architectural predecessors and this research proposes a technique to help bridge this gap.

We propose several best practices which feed into a migration technique. The proposed five-phased technique facilitates the migration of a monolithic architecture to a FaaS environment with a focus on optimizing for reducing hosting costs.

Furthermore, in our experiments we identified several industry standards or recommendations that although have their own merits, had a negative impact on cost. We identified several functions suitable for migration from a monolithic application to the FaaS architecture. When extracting the functions according to the proposed best practices, our experiments point towards significant savings in terms of the overall hosting cost of the serverless functions.

This research opens the area of FaaS in terms of memory consumption and optimisation and provides insights into the behaviour of this architecture. Future work will focus in identifying additional refactoring methods and analyse different types of applications and use cases for serverless functions to refactor and develop this migration technique into a pattern verified across different application types.

The migration technique completed as part of this research encourages the adaptation of FaaS architecture and identifies the cost implications of code refactoring when working working with the architecture style.

References

[1] S. Li, H. Zhang, Z. Jia, Z. Li, C. Zhang, J. Li, Q. Gao, J. Ge, Z. Shan, A dataflow-driven approach to identifying microservices from monolithic applications, The Journal of systems and software 157 (2019) 110380.

[2] M. Abdullah, W. Iqbal, A. Erradi, Unsupervised learning approach for web application auto-decomposition into microservices, The Journal of systems and software 151 (2019) 243–257.

[3] N. Serrano, J. Hernantes, G. Gallardo, Service-oriented architecture and legacy systems, IEEE software 31 (2014) 15–19.

[4] D. Taibi, J. Spillner, K. Wawruch, Serverless computing-where are we now, and where are we heading?, IEEE Software 38 (2021) 25–31. doi:10.1109/MS.2020.3028708.

[5] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, A. Iosup, IT-fakulteten, d. G. Institutionen för data-och informationsteknik, D. of Computer Science, C. S. G. Engineering, G. universitet, G. University, I. Faculty, Serverless applications: Why, when, and how?, IEEE software 38 (2021) 32–39.

[6] P. Castro, V. Ishakian, V. Muthusamy, A. Slominski, The rise of serverless computing, Commun. ACM 62 (2019) 44–54. URL: https://doi-org.cit.idm.oclc.org/10.1145/3368454. doi:10.1145/3368454.

[7] B. Varghese, R. Buyya, Next generation cloud computing: New trends and research directions, Future Generation Computer Systems 79 (2018) 849–861. URL: https://www.sciencedirect.com/science/article/pii/S0167739X17302224. doi:https://doi.org/10.1016/j.future.2017.09.020.

[8] A. Eivy, Be wary of the economics of "serverless" cloud computing, IEEE Cloud Computing 4 (2017) 6–12.

[9] K. Mahajan, D. Figueiredo, V. Misra, D. Rubenstein, Optimal pricing for serverless computing, in: 2019 IEEE Global Communications Conference (GLOBECOM), 2019, pp. 1–6.

[10] S. Hong, A. Srivastava, W. Shambrook, T. Dumitras, Go serverless: Securing cloud via serverless design patterns, in: 10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18), USENIX Association, Boston, MA, 2018. URL: https://www.usenix.org/conference/hotcloud18/presentation/hong.

[11] R. Rabbah, N. M. Mitchell, S. Fink, O. L. Tardieu, Serverless composition of functions into applications, 2021. US Patent 10,896,181.

[12] P. Bernstein, T. Porter, R. Potharaju, S. Venkataraman, W. Wu, Serverless event-stream processing over virtual actors, in: Conference on Innovative Data Systems Research (CIDR), 2019. URL: https://www.microsoft.com/en-us/research/publication/serverless-event-stream-processing-over-virtual-actors/.

[13] D. Bardsley, L. Ryan, J. Howard, Serverless performance and optimization strategies, IEEE, 2018, pp. 19–26.

[14] J. Nupponen, D. Taibi, Serverless: What it is, what to do and what not to do, IEEE, 2020, pp. 49–50.

[15] J. Manner, M. Endreß, T. Heckel, G. Wirtz, Cold start influencing factors in function as a service, in: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), IEEE, 2018, pp. 181–188.

[16] D. Bermbach, A.-S. Karakaya, S. Buchholz, Using application knowledge to reduce cold starts in faas services, in: Proceedings of the 35th Annual ACM Symposium on Applied Computing, SAC '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 134–143. URL: https://doi-org.cit.idm.oclc.org/10.1145/3341105.3373909. doi:10.1145/3341105.3373909.

[17] AWS, Aws lambda, 2021. URL: https://aws.amazon.com/lambda/, [Accessed 21-February-2021].

[18] Microsoft, Azure functions, 2021. URL: https://azure.microsoft.com/en-us/services/functions/, [Accessed 21-February-2021].

[19] Google, Google cloud functions, 2021. URL: https://cloud.google.com/functions, [Accessed 21-February-2021].

[20] IBM, Ibm functions, 2021. URL: https://cloud.ibm.com/functions/, [Accessed 21-February-2021].

[21] D. Zmaranda, L.-L. Pop-Fele, C. Gyorödi, R. Gyorodi, G. Pecherle, Performance comparison of crud methods using net object relational mappers: A case study, International Journal of Advanced Computer Science and Applications 11 (2020).

[22] Microsoft, Best practices for performance and reliability of azure functions, 2021. URL: https://docs.microsoft.com/en-us/azure/azure-functions/functions-best-practices/, [Accessed 17-March-2021].

[23] A. Santhiar, A. Kanade, Static deadlock detection for asynchronous c# programs, volume 128414, ACM, 2017, pp. 292–305.

[24] Microsoft, Estimating consumption plan costs, 2019. URL: https://docs.microsoft.com/en-us/azure/azure-functions/functions-consumption-costs/, [Accessed 01-April-2021].

[25] R. Laigner, M. Kalinowski, L. Carvalho, D. Mendonça, A. Garcia, Towards a catalog of java dependency injection anti-patterns, in: Proceedings of the XXXIII Brazilian Symposium on Software Engineering, SBES 2019, Association for Computing Machinery, New York, NY, USA, 2019, p. 104–113. URL: https://doi-org.cit.idm.oclc.org/10.1145/3350768.3350771. doi:10.1145/3350768.3350771.

[26] P. Agarwal, J. Lakshmi, Cost aware resource sizing and scaling of microservices, in: Proceedings of the 2019 4th International Conference on Cloud Computing and Internet of Things, CCIOT 2019, Association for Computing Machinery, New York, NY, USA, 2019, p. 66–74. URL: https://doi-org.cit.idm.oclc.org/10.1145/3361821.3361823. doi:10.1145/3361821.3361823.

[27] G. Mazlami, J. Cito, P. Leitner, Extraction of microservices from monolithic software architectures, IEEE, 2017, pp. 524–531.