CONTENTS

ABSTRACT 2
INTRODUCTION 3
LITERATURE REVIEW 4
LAMBDA PERFORMANCE PROFILE 5
COMPLEX SYSTEM CASE STUDY 8
OPTIMIZATION STRATEGIES 9
  ROBUST RETRY IMPLEMENTATION 10
  CIRCUIT BREAKERS 10
  BULKHEADS 10
  LANGUAGE-AGNOSTIC APPROACH 10
  WARMING STRATEGY 10
SUMMARY 12
REFERENCES 13

ABSTRACT

In the constantly changing technological landscape the concept of serverless computing in a public Cloud is a relatively new development. Over recent years the serverless abstraction has gained significant traction in the IT industry. Google, Microsoft and AWS all now provide feature equivalent serverless implementations as part of their Cloud-based offerings and solution architects throughout the industry are using serverless as part of mission-critical enterprise systems.

In this paper we examine the performance profile of the serverless ecosystem in a low latency, high availability context, present results on the integral performance of such systems and outline some practical mitigation strategies to optimize serverless architectures. We confine our investigation to one aspect of the AWS implementation of serverless known as AWS Lambda. Our results show there are opportunities to tune the performance characteristics of Lambda-based architectures and we outline considerations such as cold starts and potential latency characteristics created by a combination of factors including external systems and events. We propose a diverse set of strategies, approaches and techniques which, when successfully implemented and deployed, simultaneously play to its strengths, with the ultimate goal of providing a set of design patterns aimed at increasing the applicability of serverless computing to a wider set of problem domains.

This paper was presented at SmartCloud 2018, the 3rd IEEE International Conference on Smart Cloud, September 2018.
INTRODUCTION

Serverless computing is something of a misnomer. Whilst the server is not exposed in any meaningful fashion to the developer, code is obviously being executed on physical hardware in the same way as it has for decades. Where serverless really differs from more traditional approaches, and indeed from early Cloud-based offerings, is that compute power is only provisioned and code only executed when triggered by some external event. This has several advantages - billing only occurs whilst the code is executing, processes can be scaled horizontally almost indefinitely, and the concept of self-contained discreet functionality lends itself well to the trend towards micro and nano-service based system designs[1]. Despite an industry push towards embracing serverless paradigms, academic interest in the topic has been somewhat limited, as evidenced by the relatively small number of journal publications on the topic.

The concepts relevant to serverless computing are not new. Context-free services running within a contained environment have existed for decades. The Tandem Computers Guardian Operating System[2] implemented a message-based architecture which could be horizontally scaled by packaging code, adding more servers and replicating processes across the hardware. J2EE stacks also encapsulate the idea of well-defined services running within a contained environment. The difference between previous iterations of similar technology is execution in a public environment where the developer does not control the infrastructure and is reliant on the Cloud vendor to provide a secure, scalable, reliable and consistent environment.

Serverless computing presents new opportunities to Cloud orientated solution architects and developers. Primarily, it provides a simplified programming model for developing distributed Cloud-based systems, with the infrastructure abstracted away. It is no longer the concern of the developer to manage load balancers, provisioning and resource allocation (although system implementers need to be aware of such things). This reduced focus on operational concerns should allow greater attention to be paid to delivering value, functionality and an ability to adapt rapidly to change. Issues such as deployment, monitoring, quality of service and fault tolerance are moved into the hands of the Cloud provider and still need to be actively considered and managed. Serverless computing is still in its infancy and it is highly likely that as the model matures further, tools will be created to allow developers and architects to create patterns and processes to more fully exploit the advantages of the serverless model.

The approaches, designs and optimizations discussed in this paper are aligned with the best practices prescribed by AWS. When working with AWS Cloud offerings it is vital that developers and architects comply with AWS’ recommendations. Where the authors have made use of specific best practices we tie these approaches back to the source of the original recommendation.

In this paper, we present some of the unique challenges inherent in the performance profile of an AWS-based serverless architecture, specifically systems requiring synchronous interaction, lambda chaining or low latency response. These challenges include:

- Understanding and managing the concept of cold starts - the necessary delay introduced as the underlying infrastructure dynamically provisions new resources at runtime
- Increased latency caused by complex inter-component communication
- Configuration considerations
- The difficulties involved in monitoring dynamically distributed systems to identify bottlenecks when they occur

In addition we outline a case study of a complex serverless system and report a number of practical mitigation strategies aimed at combatting the performance issues inherent in serverless architectures.
Several research groups have begun publishing reports on the current state of serverless.

Adzic et al.\(^{(3)}\) assess the economic and architectural impact of the serverless concept. They report on two real world cases of migrations to AWS Lambdas from monolithic architectures, with particular attention paid to cost savings. Their first case study reports operational cost savings of 66% post migration despite a significant uptick of users and increased functionality. Their second case study reported "an operational cost reduction of greater than 95% for a comparable amount of computational resources." In both instances, this was attributed to billing based on actual utilization and not reserved capacity. Indeed, the research mostly focuses on the cost saving aspects of serverless, but also considers the architectural impact associated with delivering lower cost operational infrastructure. In addition to reduced cost associated with actual utilization billing, their findings also attributed lower costs to distributed request level authorization. They hypothesize that "as the serverless platforms no longer have a gatekeeper server process, using the traditional model where back-end resources implicitly trust servers is not viable." The architectural ramifications of this statement mean that "it's perfectly acceptable, even expected, to allow client applications to directly access resources traditionally considered back-end." Thus, used in conjunction with AWS' authentication and authorization services, fine grain permissions can be applied at the client level - a major change to the traditional client/server model.

The new approach does allow for significant optimization of which services are used and when. The trivial example given is one which would allow a client application to connect directly to AWS' Pinpoint service. The traditional approach would have required a server to handle requests and to push analytics into a database. By connecting directly to this service on an ad-hoc basis they suggest that overhead can be reduced at the expense of having to architect the system with a more serverless mind-set.

An additional high-level study of serverless and its future was recently conducted by Baldini et al.\(^{(4)}\) in their analysis of current trends and open questions in the serverless community. Initially they give a high-level overview of the genesis and evolution of serverless computing, briefly touch on the currently available platforms and summarize new and upcoming technologies.

Their summary of the benefits and drawbacks of serverless are of most interest within the context of this paper. They call out the advantages of no longer needing to manage and provision servers thus freeing up developers to work on business logic and features. Developers need to understand the platform intricacies affecting scalability, performance and fault tolerance. The authors also strongly suggest that taking advantage of the extended ecosystem offered by the various Cloud providers is an attractive draw to developers but seen as a potential risk by the business due to concerns around vendor lock-in and dependencies. The most interesting of these considerations is their identification that the developer is now required to have intimate knowledge of the particular performance profile of a platform and its associated services. Our research aims to provide examples of baseline performance and behavior, alongside real-world examples, such that developers are better equipped to develop software in harmony with serverless architectures.

Academic publications relating to serverless performance tend to focus on its applicability to particular problem domains rather than raw performance data. Ishakian, Muthusamy and Slominski\(^{(5)}\) evaluate the suitability of the serverless computing environment for the inferencing of large neural network models using the MxNet deep learning framework. Regarding scalability, they conclude that "the platform seems to scale with demand particularly for large memory sizes where the latency is typically under an acceptable user expected response time."

With regards to latency they find "warm serverless function executions are within an acceptable latency range" but that in the longer term "serverless platforms will need to support more stateful workloads", "access to GPU" and "a declarative way to define workloads (e.g. the minimum time to keep containers warm)."

Spillner, Mateos and Monge\(^{(6)}\) explore the prospect of serverless as applied to scientific and high performance computing. They monitor the performance of massively parallel tasks including computing Pi, face detection, password cracking and precipitation forecasting. Their research created successful experimental implementations for each of the problem domains and noted the performance versus a locally executed multi-process implementation. Whilst detailing the limitations of functions as a service (FaaS) they do conclude "the true on-demand provisioning and billing of hosted functions makes them attractive for research tasks."

There have not been many significant published works dealing exclusively with the performance of serverless beyond its applicability to particular application domains. This is in part due to the relative immaturity of the platform but other contributing factors include complexity regarding what and when to consider while evaluating performance. Serverless functions on all platforms typically support multiple development languages and multiple entry points into the serverless infrastructure. Precisely what to test and how to make the comparison are very much open to debate. Nevertheless, outside of academically published research there is a wealth of comparative studies available online.
Billock[8] compares performance across the major platforms using roundtrip time as the main assessment metric whilst separating the testing into hot and cold scenarios. This research compares performance against different vendors.

The first issue with all of these studies is that they tend to measure performance from an external viewpoint. Our research hopes to inspect the internals of the serverless ecosystem, specifically in relation to AWS Lambdas for low latency trading, to understand their internal performance profile and thus gain deeper understanding of how to better architect serverless systems.

The second issue with the reported high level overviews of the serverless ecosystem is that they tend to present problems which no accompanying solutions, therefore, the second component of this research is to present practical approaches to resolve or mitigate some of the general issues inherent in serverless architectures.

**LAMBDA PERFORMANCE PROFILE**

This section assesses the performance profile of AWS Lambdas under several conditions. The experiments are isolated to Python 2.7 Lambdas triggered via calls from API Gateway, and by direct Lambda to Lambda calls. The primary goal of these experimental results is to deep dive into the Lambda ecosystem and determine the causes of latency. In addition to presenting the timing results we also demonstrate some of the tools available on the AWS platform for performance profiling distributed systems.

The simplistic system consists of an API Gateway endpoint to handle incoming requests, a Reverse Array Lambda and a Reverse String Lambda. The Reverse Array gateway/Lambda accepts a list of strings which is parsed and passed as individual strings into the Reverse String Lambda. The Reverse String Lambda reverses the string and returns to the Reverse Array Lambda and subsequently back to the calling client. Calls into the Reverse String Lambda are made in parallel. JMeter is used as a test platform to call and monitor responses from API Gateway. In the experimental setup, the Reverse Array endpoint is called from an AWS EC2 instance running in the same region. Five small random strings of equal length are passed with the request on each call meaning that the Reverse String Lambda is called five times for each request which hits API Gateway. The REST endpoint was called 1,000 times at a rate of two requests per second for the duration of the test run meaning that the test run took approximately 500 seconds. The full test run was executed twice with results from the first test run discarded to ensure that there were a sufficient number of warm Lambdas to eliminate the effects of cold Lambdas on the latency results. It should be noted that the string reversal use-case is optimal for serverless systems since it is a small, fast to execute, and incurs no IO-bound operation.

Results from the test run were collected and analyzed using JMeter. Table 1 shows the high-level results of the test run. The average response time, including SSL handshake and wire latency, was 112ms. The maximum of 1,002ms was recorded midway through the test run. At this level of analysis, we are unable to determine the exact cause, but it could be attributed to any of the underlying system components. Further investigation into the back-end latency results should clarify the cause of this spike.

Figure 1 (overleaf) shows the response time across the test run. Values are averaged over ten second windows and as such the maximum response time is not immediately visible. There appears to have been a number of slower than average response times towards the start of the test run. It is unlikely that these are due to cold starting Lambdas since the system was fully warmed prior to the test run - they are mostly likely due to some initial connections being created. This could have been caused by API Gateway initializing, despite having been used during the warming run, however, AWS does not provide data on the start-up profile of the gateway itself.

A trivial system was implemented to assess the performance of the interconnected Cloud components. The simplistic system consists of an API Gateway endpoint to handle incoming requests, a Reverse Array Lambda and a Reverse String Lambda. The Reverse Array gateway/Lambda accepts a list of strings which is parsed and passed as individual strings into the Reverse String Lambda. The Reverse String Lambda reverses the string and returns to the Reverse Array Lambda and subsequently back to the calling client. Calls into the Reverse String Lambda are made in parallel. JMeter is used as a test platform to call and monitor responses from API Gateway. In the experimental setup, the Reverse Array endpoint is called from an AWS EC2 instance running in the same region. Five small random strings of equal length are passed with the request on each call meaning that the

<table>
<thead>
<tr>
<th># Samples</th>
<th>Average</th>
<th>Median</th>
<th>Deviation</th>
<th>95%</th>
<th>99%</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>112ms</td>
<td>96ms</td>
<td>63.45ms</td>
<td>213ms</td>
<td>338ms</td>
<td>46ms</td>
<td>1,002ms</td>
</tr>
</tbody>
</table>

Table 1: JMeter high level performance results

Whilst instructive about the general performance of the system the results do not offer much insight into which components of the system are taking time to execute. However, the chart confirms that the system handles most requests at the average/median value. The variance is predominantly caused by a start-up spike and two spikes starting at 9:38:00 and 9:44:00. The nature of these spikes indicates an initial start-up delay and subsequent contention for resources. The string reversal function takes less than 1ms to execute and the system only calls for 5 strings to be reversed per call. Clearly overhead involving other components is adding significant latency to each call. This internal latency cannot be monitored using the high-level benchmarking figures, however, AWS provides several services for monitoring distributed system performance.
We use a combination of CloudWatch and X-Ray to assess the performance of the internal components of the system. of services for more accurately monitoring the distributed systems performance. We use a combination of CloudWatch and X-Ray to assess the performance of the internal components of the system.

CloudWatch provides an array of metrics and tools to graph, alert and inform the specifics of where time is being spent in an application.

68ms executing the back-end, this suggests that it takes somewhere in the region of 10-15ms to execute a Lambda from API Gateway.

Finally, taking into account the latencies reported for each of the Lambdas on the bottom graphs we note that whilst the Reverse Array Lambda takes approximately 55ms to execute the Reverse String Lambda takes approximately 2.2ms to execute. Given that we are calling the Reverse String Lambda in parallel clearly there is significant overhead in making Lambda to Lambda calls. Two contributing factors for this are likely the time spent to create the connection and, perhaps more importantly, the time taken to validate the Lambda has appropriate permissions to call the second Lambda. As previously discussed one of the advantages of the serverless approach is that fine-grained access controls allow levels of access control not commonly available in traditional architectures. It is in situations such as the Lambda to Lambda calls where we see the disadvantages of such an approach - the additional performance overhead required to validate each request. It is likely that the various Cloud providers are well aware of such issues and are working behind the scenes to introduce caching or other mechanisms to reduce latencies in such situations.

CloudWatch allows a more in-depth analysis than can be achieved using external system monitoring, however it is best suited to monitoring items at the level of individual components e.g. Lambdas and API Gateway. To fully understand the internals of a particular sub-system additional analysis is required. This may take the form of additional logging or, alternatively, via pushing performance

Figure 2 shows a CloudWatch dashboard showing latencies for various system components. Values are again shown as averages within rolling ten second windows. The top graph shows the API Gateway latency and integration latency. Integration latency is the time between when API Gateway relays a request to the back-end and when it receives a response from the back-end. Latency is the time between when API Gateway receives a request from a client and when it returns a response to the client. The latency includes the integration latency and other API Gateway overhead. The lower two graphs in Figure 2 show the average amounts of time spent inside both the Reverse Array Lambda and the Reverse String Lambda.

The Reverse Array API Gateway graph shows a number of interesting things. Firstly, approximately 3ms is spent on each call to process the request using API Gateway (as shown by the difference between the orange and blue lines).

Secondly, it takes approximately 55ms to execute the Reverse Array Lambda when it is called from API Gateway. Given that API Gateway reports spending approximately 68ms executing the back-end, this suggests that it takes somewhere in the region of 10-15ms to execute a Lambda from API Gateway.

Figure 1: JMeter HTTP response times (10 second averages)
metrics into an external system. AWS offers AWS X-Ray to help address this issue. AWS X-Ray tracks requests as they travel through the AWS infrastructure and will report latencies as they pass through Lambdas, DynamoDB, RBS and other supported services. Custom code also allows the developer to track arbitrary code blocks within Lambdas.

To fully understand the performance profile of the experimental platform we carry out a trace analysis of a single call into the Reverse Array function. Figure 3 shows the life cycle of a typical call into the system. The total time spent in the Reverse Array Lambda is approximately 55ms. This aligns well with the average response time for this Lambda and therefore we can assume this is a relatively typical trace. Without any additional configuration, AWS X-Ray shows the total response time for the selected Lambda and then further breaks this down into calls into other Lambdas together with the overhead for initializing the Lambda, before showing the total time it took for the function to execute. Any additional latency, such as that introduced whilst the Lambda was initializing, will also be shown here. Since the Lambda was warm when it was called this is not shown in the trace being examined.

Analyzing the trace from the bottom upwards, we show that the time taken in the actual string reversal function is generally less than or equal to 1ms, and the required time to execute the whole Lambda was on average 6.6ms. This indicates that approximately 5ms is spent on making the call into the Lambda before we hit the actual handler function. Continuing to move up the trace we can see the source of the Lambda invocation in the Reverse Array Lambda. There is a slight delay as the Lambda loops through the five threads that it creates before calling the invocation function and triggering the string reversal. From the context of the Reverse Array Lambda we can see that, on average, it takes 29.56ms to invoke a remote Lambda. Therefore, we can conclude that it takes approximately 20ms to invoke a remote Lambda under optimal conditions. Clearly if a fresh container had to be created then this latency would be much greater.

In general, none of the latencies, when taken individually, pose a particular problem, however, it is easy to see how they can cumulatively cause response times which may require further optimization. The average latency as recorded by JMeter was 112ms whereas the average response time recorded by API Gateway was 72ms so clearly the SSL handshake and network transport overheads contribute a portion of the overall response time. Furthermore, the average response time of the Reverse Array Lambda is approximately 54ms, API Gateway adds an additional 20ms onto the total. Finally making Lambda to Lambda calls introduces another approximately 20ms of latency. Therefore, this architecture adds about 100ms overhead to any transaction. About half this time is consumed by the API Gateway component/routing the Lambda and the other half by the two Lambda functions.

It is clear that attention must be paid to inter-system latencies when architecting Lambda-based systems that cannot tolerate this overhead. It becomes all too easy to develop a system lacking in performance by generating too many interacting parts. Naturally this was a simple system developed to generate performance figures and the overheads involved may be dwarfed by the actual work being carried out. The benefits gained by using a serverless architecture (such as horizontal scalability and cost) could make such overheads a non-issue, but it is exactly these types of concerns that should be addressed when selecting the technology stack and target architecture for any software development project.
This in-depth analysis of the internal and external components of a typical serverless shows where latency is introduced at each stage of the system. We have demonstrated how each additional step introduces more latency into the system even in the trivial experimental setup.

The complex case study presented below discusses a far more complex system and the additional complications that arise once we move away from more trivial architectures. It also gives greater consideration to the issues surrounding cold starts which are far more prevalent in systems with more moving parts, deeper call stacks and carrying out greater workloads on each invocation.

**COMPLEX SYSTEM CASE STUDY**

This section describes and highlights some of the characteristics of working with Lambda as part of complex system implementations. The system under consideration was a real-world e-commerce implementation with numerous customer facing endpoints, a complex mix of legacy and greenfield back-end systems and several business constraints limiting how much of the original system could be replaced as part of a “big-bang” approach to releases.

The challenges faced throughout this real-world development are tightly linked to the performance profile described earlier, notably the Lambda to Lambda latency and the effect of cold starts on the overall perceived performance. The system was entirely implemented using Java Lambdas which are known, along with C#, to have the greatest issues with cold starts due to JVM start time and the need to load many packages during the initialization stage.

Much consideration was given to the suitability of a serverless architecture as a solution to this business problem. However, the combination of a customer-facing website, log processing, IT automation and a desire to eliminate under-used server capacity are widely accepted use cases so the solution was deemed viable.

The functionality of the system incorporated user management systems, shopping baskets, product search, interfaces with legacy delivery booking systems and many ETL and data synchronization subsystems. In general customer facing endpoints were served to a React front-end via calls into API Gateway. Some back-end functionality was serviced by DynamoDB data stores with AWS ElastiCache buffers to reduce data access latency, however, many requests ultimately had to be serviced by calls out to slow legacy systems, traditional relational databases and external search providers. This resulted in Lambdas performing tasks which, when accounting for performance of external systems, would often take several seconds to complete. The system employed a combination of asynchronous and synchronous processes. The experience with the former was entirely positive so therefore this section will be constrained to the
synchronous use cases. Since the system was tightly linked to a web-based user interface, issues with the synchronous processes had the potential to cause a noticeable impact on the user experience.

In production, there were numerous occasions where a large number of Lambdas (1,000+) were instantiated in a short time period (approximately one minute) and as a result a number of customers were impacted by an unresponsive website. Initially this was exacerbated by AWS soft limits which triggered AWS throttling of calls to resources whose limits were being breached. New containers could not be constructed due to Elastic Network Interfaces (ENI) soft limit breaches. Lambdas were throttled due to exceeding the total Lambda pool limit and calls to DynamoDB were also throttled. Increasing soft limits reduced the scope of the problem but obviously did not rectify the underlying issues. Identifying the root causes of these issues was complicated due to the distributed nature of the system.

Investigation of the characteristics revealed that observations fell into a number of categories, but the conclusion is that Lambdas may be best suited to small tasks which complete quickly. High latency calls to error-prone external systems ultimately created a bottleneck. The issue was compounded by the specific retry strategy implementation which was configured to retry too aggressively, causing call cascades which could result in 80% of Lambda to Lambda invocations generating retries. Furthermore, the AWS Lambda to Lambda invoker is, by default, configured to retry three times with an exponential back-off strategy. This resulted in some Lambdas being called long after the original call had completed. Finally, some calls to external services were not cached in instances where they could have been. The majority of these issues were subsequently resolved.

Ultimately it became clear that calls to external services, internal caches and invocation policies are topics requiring close consideration when architecting complex distributed Lambda-based systems.

Outside of some of the configuration characteristics Java Lambda implementations are susceptible to cold start issues. Both the time taken to initialize the JVM and time taken to load a potentially large number of libraries may result in longer initialization times. This performance hit is further increased if the Lambdas are inside a VPC.

In the AWS implementation, if a request hits a cold Lambda, rather than being serviced by the next available Lambda (which may be ready milliseconds after the call was made) the caller must wait for the cold Lambda to start. It is the authors’ opinion that this aspect could be further optimized to reduce the impact of cold Lambdas. Without it, the effects of cold Lambdas must be mitigated by the system implementer.

The following section describes some of the approaches that can be used to address this problem.

Lambdas can be susceptible to external latency which may result in a high number of Lambdas being instantiated should there be a sudden spike in the latency of a third-party service. This is to be expected as the Lambdas will balance the number of customers against the performance of the Lambda. If Lambdas that are taking 100ms to service a set of customers suddenly start taking 500ms to execute then five times the number of Lambdas will be required to service the same set of customers.

Having addressed the configuration and soft limit issues, the only times spikes in Lambda creation were seen was during periods of increased latency in third party systems. This, however, is the Lambda infrastructure scaling as expected and, to some degree, providing protection from the unresponsiveness of an external system. In some instances this was mitigated utilizing queues to connect to external infrastructure although this was only possible where asynchronous behavior was acceptable. At other times, when a synchronous response was required, a queue implementation was not practical and starting additional Lambdas to compensate for those that were busy was the only viable solution.

Clearly there are many benefits to using serverless including scalability, DevOps strategies, cost and reduced management overhead. All of these benefits were realized during this implementation.

The following section introduces a number of design patterns and strategies which were used to mitigate and optimize the points described in this section and to improve the perceived performance of the system from the user’s perspective.

OPTIMIZATION STRATEGIES

In this section, we explore some of the optimization strategies that were implemented. Strategies applied were shown to increase performance, reduce latency and generally stabilize the performance of the system.

Throughout the design and development of the system architecture close attention was paid to the AWS Well Architected Framework\[10]. Two important tenets of the framework are that systems should allow for evolutionary architectures (and be able to take advantage of innovations as a standard practice), and that architecture should be driven using data. The strategies described in this section were implemented as a result of following these best practices - employing optimization strategies informed by data from a production environment and a system amenable to architectural change. In each instance, we tie the proposed optimization strategy to a specific best practice.
ROBUST RETRY IMPLEMENTATION
The most important design pattern employed to overcome the fact that requests would wait for a Lambda to start rather than taking the next free resource, was a robust retry implementation. This approach significantly mitigates the effect of cold starts since a request can abandon its wait for the cold Lambda and simply take the next available free handler. Care must be taken to tune the retry parameters correctly to avoid instantiating unnecessarily large volumes of Lambdas. The retry strategy is of particular importance if the Lambda is making calls to several other Lambdas.

The large number of interconnected components in the system architecture results in call stacks several Lambdas deep. Each link in the chain introduces the potential for cold starts, additional latencies and system errors. This implies that the Lambda call stack should be kept short. Three or fewer Lambdas in a chain is preferable when servicing a time-critical request, though there are circumstances where instantiating an asynchronous process would better serve the user’s need.

For example, in some cases where the invocation is the result of a front-end request, such as from a website, it could be possible to make multiple parallel calls to the back-end in cases where all the data is not required simultaneously. Such a strategy would result in a greater number of calls from the front-end but would also increase the responsiveness of the interface as a whole. This strategy of using parallel, concurrent requests to lazy-load specific components of a website is specifically recommended in the Serverless Applications component of the AWS Well Architected Framework[12].

CIRCUIT BREAKERS
Circuit breakers allow a call to an unresponsive system component to be aborted without needlessly consuming resources trying to repeatedly connect and retry. There will be occasions when components are unresponsive and the system should be able to handle this without cascading failure. It is in situations like this that retry is not beneficial and may well have harmful effects if it ends up spinning up many cold Lambdas. A circuit breaker is required that will identify when a system is in stress and will back off. If this is linked with the front-end it would become possible for the server to issue a 503 HTTP response and the front-end to silently retry after a predetermined back-off.

BULKHEADS
Bulkheads effectively isolate components of the system that display inconsistent latency. These may be Lambdas which take a variable amount of time to complete based on the workload or which interact with external systems with an inconsistent performance profile.

As an example, the case study project initially had a single Lambda which handled customer user data. The architecture was such that customer data was refreshed and cached from an external system during the initial authentication process, however, other Lambdas which needed access to the customer data frequently would call the customer Lambda requesting the cached data. The asymmetrical nature of the performance profile between the refresh and request calls, with the refresh operation suffering significantly higher latency than the request call, could cause refresh calls to unnecessarily divert requests to cold Lambdas.

Implementing bulkheads separating high latency operations from low latency application request flows significantly reduced the probability of a given request being impacted by cold Lambdas. In real terms this required separating the request and refresh functionality into separate Lambdas to prevent high latency in one part of the system adversely affecting another. Whilst not referred to as bulkheads, AWS best practices do state “having smaller functions that perform scoped activities contribute to a more well-architected serverless application”[12] and as such the migration to smaller, more isolated components is in line with published best practices.

LANGUAGE-AGNOSTIC APPROACH
A language-agnostic approach to Lambda development significantly improves latency within some parts of the system.

Teams responsible for Lambda development should use the language best suited to the particular service. Whilst this may reduce code reusability it allows for a reduction in latency in system components that are highly sensitive to Lambda initialization timings. Using either Node.js or Python Lambdas on front-end facing Lambdas reduces latencies since these languages are less susceptible to problems with cold starts and can then offload to Lambdas implemented in other languages in a manner which would not negatively impact the user experience.

As stated as part of the AWS Lambda best practice documentation “the compiled languages (Java and .NET) incur the largest initial start-up cost for a container’s first invocation, but show the best performance for subsequent invocations. The interpreted languages (Node.js and Python) have very fast initial invocation times compared to the compiled languages, but can’t reach the same level of maximum performance[13]. The implication is that latency-sensitive applications or those expecting spiky traffic should use interpreted runtimes where possible. It can be further extrapolated that Lambdas forming part of the same application can use different runtimes depending on the predicted workload for a specific component.

WARMING STRATEGY
A final latency mitigation strategy which can be used in tandem with the approaches outlined above is to use a warming strategy to ensure that an appropriate number of
Lambdas is kept warm at all times. This approach advocates the implementation of a Lambda, executed on a schedule, which makes dummy calls into the other Lambdas in the system such that they are forced to keep warm. This approach presents some optimization requirements in that it must be determined in advance which and how many Lambdas should be kept warm. Whilst this approach somewhat defeats the purpose of a system which should dynamically scale in response to demand it is nevertheless a viable strategy in mitigating the effect of cold Lambdas on overall latency. It should be noted that this approach will not significantly raise the cost of the deployment since calls to ping the relevant Lambdas would only have to be made infrequently, however, it is difficult to predict the Lambda usage requirements at a given time. Dealing with large spikes in demand therefore may be a challenge but it has the potential to reduce issues when used in combination with the other techniques outlined in this section. Technical approaches to warming Lambdas are outlined in numerous places across the internet\textsuperscript{[14,15]} however this strategy is not particularly endorsed by AWS. It may be that future work at the infrastructure and configuration level is a more desirable solution than artificial mechanisms implemented to avoid container destruction.

This section briefly outlines the concepts of retries, circuit breakers, bulkheads, language agnosticism and warming strategies within the context of the serverless framework. It aligns the proposals for the particular design patterns based on issues observed in the complex system case study. All but the simplest of serverless architectures should look to implement these design patterns from project conception. They should be implemented as common libraries or code for the project such that such patterns are implemented robustly and consistently throughout a given system. Furthermore, developers and architects should have a solid understanding of the importance of such principles in order to avoid unexpected and difficult to diagnose latencies as the system reaches a critical mass of users in a production environment.
SUMMARY

We present evidence that the current academic literature surrounding serverless performance and design patterns is somewhat lacking. An attempt to partially redress this situation is made by providing a comprehensive breakdown of Lambda performance, highlighting where optimization strategies can improve performance characteristics and reduce latency.

We analyze the performance of a trivial serverless architecture to show how interconnected components cause, potentially unexpected, latencies to be introduced.

A real-world case study of a complex system is presented, including a description of the issues faced when implementing a traditional e-commerce system using a novel serverless architecture.

Finally, we discuss and expand upon a variety of optimization strategies which were essential in addressing some of the issues faced when deploying complex systems to serverless architectures.

We frame real-world experience against the backdrop of industry best practices and provide design and implementation recommendations. Both the trivial test application and the complex case study indicate why best practices should be adopted and demonstrate the need for both developers and architects to have a firm grasp of infrastructure on which their applications are running, even in an ecosystem where they may not have direct visibility of the underlying hardware configuration.
4. Baldini, I.C., Paul; Chang, Kerry; Fink, Stephen; Ishakian, Vatche; Mitchell, Nick; Muthusamy, Vinod; Rabbah, Rodric; Slominski, Aleksander; Suter, Philippe, Serverless Computing: Current Trends and Open Problems. 2017, IBM Research. 
ABOUT THE AUTHORS

DANIEL J. BARDSELEY
Daniel has been a senior developer with BJSS for 10 years.
He has a diverse, cross vertical skill set which has been applied to numerous consulting engagements with BJSS clients. Daniel has worked on a wide range of technology stacks encompassing open source, Cloud, server and client side projects. Prior to working with BJSS Daniel obtained his PhD from Nottingham University where he studied computer vision with a focus on 3D face recognition.

JOHN HOWARD
John is a Technical Architect based in BJSS’ London Delivery Center.
He has held key technical leadership roles in a range of sectors including Oil Trading and Retail e-commerce, where he is leading the transition from a monolithic e-commerce application to a public Cloud microservices architecture.

LARRY RYAN
Larry is BJSS Inc’s Chief Technology Officer (CTO) with responsibility for the technology direction of BJSS’ Americas operation.
He spends most of his time consulting with BJSS clients across multiple technology areas, including Cyber security, Cloud, Agile development, Big Data, Data Science, and the application of technology in low-latency Trading, Market Surveillance and Market Data Distribution.
ABOUT BJSS

BJSS is the UK’s largest privately-owned I.T. and business consultancy. We work with the world’s largest public and private sector organisations to design, deliver and support large-scale digital transformation.

As pioneers of agile delivery at scale with a 25-year track record, our innovative Enterprise Agile® approach has been recognised by the Queen’s Award for Enterprise and enables organisations of all sizes to maximise the business value of I.T.