Serving Machine Learning Workloads in Resource Constrained Environments: a Serverless Deployment Example

Angelos Christidis
a.christidis@surrey.ac.uk
Dept. of Computer Science
University of Surrey
GU2 7XH, UK

Roy Davies
roy.davies@emu-analytics.com
Emu Analytics
London
WC1N 2LG, UK

Sotiris Moschoyiannis
s.moschoyiannis@surrey.ac.uk
Dept. of Computer Science
University of Surrey
GU2 7XH, UK

Abstract **-- Deployed AI platforms typically ship with bulky system architectures which present bottlenecks and a high risk of failure. A serverless deployment can mitigate these factors and provide a cost-effective, automatically scalable (up or down) and elastic real-time on-demand AI solution. However, deploying high complexity production workloads into serverless environments is far from trivial, e.g., due to factors such as minimal allowance for physical codebase size, low amount of runtime memory, lack of GPU support and a maximum runtime before termination via timeout. In this paper we propose a set of optimization techniques and show how these transform a codebase which was previously incompatible with a serverless deployment into one that can be successfully deployed in a serverless environment; without compromising capability or performance. The techniques are illustrated via worked examples that have been deployed live on rail data and real-time predictions on train movements on the UK rail network. The similarities of a serverless environment to other resource constrained environments (IoT, Mobile) means the techniques can be applied to a range of use cases.**

Index Terms: Serverless, FaaS, AI, Machine Learning, Optimization, AWS Lamda

I. Introduction & Motivation

Serverless architectures have introduced an array of benefits to companies as well as developers working on real-time, cloud-based software solutions [1]. Benefits include a much easier development pipeline with codebases abstracted away from architectural complexities. Therefore, serverless platforms are automatically scalable to demand in real-time, resulting in cost savings for all parties involved and less strain on the developers.

This research was partly funded by EIT Digital IVZW under the Real-Time Flow project, activity 18387--SGA201, and partly by the EPSRC IAA project AGELink (EP/R511791/1).

Such benefits drive the shift we are witnessing nowadays from traditional architectures to 'microservice' / serverless based solutions.

In order to deliver these benefits, serverless platforms – such as AWS Lambda – come with certain constraints on developers with regard to *what* and *how* it can be done [2]. Limits are placed on physical codebase deployment package size (up to 250MB for AWS Lambda [3]), maximum amount of RAM allocated, as well as maximum lifetime before the running code instance is abruptly interrupted. These constraints have defined the AWS Lambda Serverless environment as a resource constrained platform [4], which is usually found to perform low level automated tasks such as scheduled data transfer from one database to another; or even performing some simple post-processing when a new item enters a storage medium. In addition, the absence currently of GPU support has also turned developers away from using serverless platforms for AI production workloads.

In this study, we present optimisation techniques. which once applied to a 'serverless-incompatible' AI codebase, can prepare a package which is ready for serverless deployment. We present a number of different techniques, including:

A) 'Minimizing / Slimming' python libraries / frameworks and automating the process.

B) Dynamically loading pre-trained machine learning models from permanent cloud storage into local temporary storage, during serverless function invocation.

C) A 2-Step Framework Process - Utilizing 'Framework A' for Training and 'Framework B' for Inference – with an ONNX formatted model in between each framework.

D) Improving the handling of data lookup and storage.

In previous work, we have been concerned with aspects of specification and verification [5] in service-oriented environments [6], which picked up from work on long-running transactions [7] and a RESTful architecture [8], [9], [10] for resources in complex digital ecosystems [11], [12].

The main motivation for looking at AI in resource constrained environments and carrying out the study reported in this paper was the development of a real-time predictive AI system as part of the *Real-Time Flow* (RTF) research project at the University of Surrey. With partners Emu Analytics, Ferrovial and Amey. RTF focuses on novel techniques for monitoring and predicting the flows of people and goods across transport networks in an urban environment. The system currently developed deploys a suite of AI models for predicting train delays across the UK rail network. Use cases vary in the sense that sometimes, predictions are at a large scale (e.g., concerning simultaneous train movements across the whole of the UK rail network), while at other times they concern a single train. In addition to this scaling up and down aspect, there are days/times where predictions might not be requested at all, while at other days/times prediction requests would come in every second. Finally, the solution should be as cost-effective as possible for all project partners involved. The use cases of our techniques in the RTF Project are described further in the evaluation section (Section IV).

The above factors drove the investigation towards a serverless development strategy. Given the pairing of traditional serverless constraints with our complex RTF codebase however, this would not work 'out-of-the-box'. This gradually led us to the techniques derived in the research reported in this paper.

We propose and detail a set of techniques that allow the serverless serving of AI models with their associated codebases, that would otherwise be incompatible with a resource constrained serverless deployment. In our case, this includes machine learning and deep reinforcement learning systems, that would not originally support serverless deployment; due to factors such as size and runtime. Treatment focuses on the key techniques that allow one to transform and successfully deploy such a system, using AWS Lambda functions in the context of the RTF project. There are more test cases and some other minor optimisations can also be performed.

The remainder of this paper is structured as follows. Section II reports on related work. Section III presents the key techniques (A-D mentioned earlier) which comprise the main contribution of the paper. Section IV reports on evaluation and Section V contains some concluding remarks.

II. Related Work

Whilst there have been previous attempts at deploying AI workloads to AWS Lambda [13], the work is performed within the 'comfort' zone of the platform. We have not seen examples where constraints of serverless workloads are breached; and the incompatibility factor that is associated. Implementations - such as the one in [13] - fail to reach a codebase complexity that would be incompatible with serverless deployment. Rising codebase complexity could be related to reasons such as large shipped AI model size, or even high-volume library usage - which contributes to massive codebase size. For instance, there is a trend in developer behavior to roll back to a traditional server-based system once a codebase becomes too complex for serverless deployment.

Research reported in [14], [15] has analysed the ideal architecture for a microservices / serverless environment and when the associated constraints on deployment package size, limited RAM allocation, restricted lifetime before termination of running code would fire, causing a degradation of the performance of an implementation. However, no implementation strategy has been given on how one can go about overcoming these constraints. For these reasons, we thought it appropriate to base our research on solutions which aim to fill the gaps mentioned above, thus proving that complex workloads can be adapted to handle serverless deployment.

Additionally, we pair research on modern NoSQL data storage mediums [16], [17] with our techniques for optimising AI workloads – since such workloads are usually associated with heterogenous datasets. At the same time, we build on top of this through partitioning and indexing techniques which make use of data representation transformation.

III. Implementing Optimization Techniques as Solutions to Serverless Constraints

A. 'Minimizing / Slimming' python libraries & frameworks and automating the process

As noted in [13], we observe that similar to other restricted execution environments, a serverless architecture will always be constrained by the total physical space (in MB) occupied by the codebase in question. Especially when dealing with an AI or Deep Learning codebase, we find that complex libraries and requirements do not work in our favor. These libraries quickly consume the minimal package size allowance; thus, immediately blocking the possibility of code

execution / deployment in a restricted serverless environment.

The first optimisation technique of our multi-step process is to 'minify' the libraries involved. Since python libraries typically ship with a plethora of functionality - pytorch for example [18], it is only natural that they are associated with a huge file size. Since serverless functions are split in such a way whereby each handles a small task, it is easy to see how we can begin to isolate sections of a python library into each individual function's environment as needed. If a serverless function is making use of 1% of a library, we can perform a few operations to discard the other 99% of the library in a robust manner – thus saving massively on file size which decreases the final deployment package size.

The key word is robustly – blindly deleting library files would be catastrophic, since many times even the smallest file could be referenced somewhere and used by our code. We must make sure to constantly test our code during the minimization process. More importantly, upon pushing an update to our code, this entire process must be performed again from scratch - starting once again with the full library package and working our way down to a minified version.

The process involves monitoring *read/write* operations to library files during main code execution. We begin by initializing read/write monitors on library directories using OS ready monitoring tools. The next step involves running the production ready code while these monitors are active. Monitor outputs will produce lists of files that are 'used' by the code during its execution. We define 'used' to be a set of files that have been accessed either by a read or *write* operation by the source code.

We then proceed to safely discard any unused files; safely in the sense that the code is re-tested after every deletion to ensure it still executes successfully without throwing any errors. In the case where a 'sensitive' file is deleted (i.e. a file that was necessary and causes exceptions/errors by being removed), a reference to this file is noted and the deletion process begins again after restoring an original, unmodified copy of the library – this time without discarding the discovered sensitive file.

Note that in order to ensure a robust codebase, this process should be set to run on every update pushed to a production workload. Upon source code change, the steps denoted in Algorithm 1 (see Figure 1) should be re-run as a 'fresh pass' – since updates could reflect a change in library usage requirements.

We denote the retrieval process of any essential library file (x) which contributes to the 'slimmed' library package as:

$$x \in \{Rf, Wf, Sf\} \setminus \{Tf\}$$

where:

- Rf contains any files accessed by a read operation of the main code,
- Wf contains any files accessed by a write operation of the main code,
- Sf contains any 'sensitive' files that may cause the main code to fail execution.
- Tf contains pre-packaged library test files.

Finally, in some cases, we observe that the codebase is more complex – i.e. it does not just load a model but also performs some other tasks, such as - creating multidimensional tensors or sending data to other devices.

Algorithm 1 Python Library Minimization - Automated Algorithm

```
1:  sensitive_files ← list to track files which should NOT be deleted
2:
3:  original_library ← the filepath of the original library
4:  slimmed_library ← a copy of original_library to be modified
5:
6:  for library root directory and sub-directories do
7:      read_monitor ← initialize file-system read monitor
8:      write_monitor ← initialize file-system write monitor
9:  end for
10:
11: start read_monitor and write_monitor
12:
13: monitoring ← true
14: accessed_files ← list to track file read(s)/write(s)
15: process_id ← process id of line 7
16:
17: while monitoring do
18:     run python code
19:
20:     for file_read, in read_monitor do
21:         append path of file_read to accessed_files
        end for
22:
23:     for file_write, in write_monitor do
24:         append path of file_read to accessed_files
        end for
25:
26:     monitoring ← false
27:
28: kill process_id
29:
30: for file_accessed in accessed_files do
31:     if file_accessed is not in sensitive_files then
32:         delete file_accessed from slimmed_library
33:
34:     run python code
35:
36:     if code still runs without errors then
37:         continue
38:     else if code runs with errors then
39:         append file_accessed to sensitive_files
40:         recursive call to line 4
    end for
41:
42: strip header information from slimmed_library
43: strip symbolic link data from slimmed_library
44: strip debug information from slimmed_library
45: strip test files from slimmed_library
46:
47: if code still runs without errors then
48:     original_library ← slimmed_library
49: else if code runs with errors then
50:     skip culprit line(s) from 42-45 on next run
51:     recursive call to line 4
52:
```

Figure 1 - Pseudocode for Minimization Automation

Here, we may have less of a 'deletion margin' since our code will be using many more files from the involved python library. In this case, we may remove any symbolic links from pre-compiled binaries - greatly reducing total payload size without impacting performance or code execution. Our use case on the RTF Project has seen arbitrary executables be reduced from 400MB all the way down to 80MB. This helps in overcomeing deployment package restrictions.

B. Dynamically loading pre-trained machine learning models from cloud storage into local temporary storage

Depending on the problem at hand, the pre-trained AI models (e.g., machine learning models for prediction, such as those used in the RTF use case: RNNs, CNNs, LSTM, LCS [19] (XCS, UCS [20] XCSI [21], etc.) that are associated with a specific use case may exceed a couple hundred MB themselves [22]. In an environment where total deployment package size is restricted to just 250MB, it is impossible to dedicate a large chunk of this to just models. In the cases of small models - less than 10MB for example - it is sufficient to ship the models inside the deployment package locally. The latter is a practice which we have seen in serverless use cases depicted in [23] but for scalability purposes, it is quite evident that a different methodology for packaging models is required.

This is why we turn to a solution which involves dynamically loading large models at runtime. In the case of AWS Lambda, we study how we can use the '/tmp' or 'temporary' directory given to us with each instance of an encapsulated serverless function [3].

All AWS Lambda serverless functions have a non-persistent '/tmp' directory that allows for up to 512MB of storage [3]. Typically, this directory is used for items created by the code which must undergo some processing before being returned to the user. For example, imagine the code generates an image – which should be colored grayscale before being returned. In order to perform this post generation processing, the image must first be stored somewhere so that the code can then go on to re-load the image and perform the processing (gray-scaling). This use case serves as a textbook example as to when a developer would utilize the '/tmp' directory:

- Generation of the image followed by saving it to the '/tmp' non-persistent, temporary directory
- The image is loaded from the '/tmp' directory
- Processing is performed on the image
- Image is returned following function execution
- Non-persistent '/tmp' means all traces are removed on function lifetime end

Instead of using this directory for artefacts generated by our code, we present a new use case which loads a pre-packaged (ML/DL or otherwise) model from a remote location into this local '/tmp' directory. The steps for loading a model dynamically via cloud storage (AWS S3 for example in our reference implementation) instead of from a local deployment package include:

- Compress the ML model (.zip)
- Store in persistent cloud storage (e.g., AWS S3); same geographical region as function environment
- On function invocation, before any code is run, bring over the model
- Uncompress the model and store it into the local /tmp directory
- Load the model into the allocated RAM and query it as needed

One may reasonably assume that these steps could add latency to the invocation of the serverless functions. This is not the case however as we can choose to store models in storage which lies in the same geographical region as that which serves our serverless functions. The chart in Figure 2 shows a series of 5 tests which aimed to measure the time taken to perform the above steps within an encapsulated serverless environment on models of three different footprints (10MB, 100MB and ~250MB).

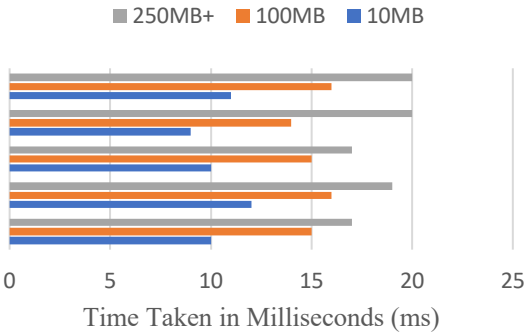

Figure 2 - Time to Load & Extract Varying Model Sizes from Cloud Storage to Local '/tmp' Directory of a Serverless AWS Lambda Function

C. 2-Step Framework Process - Utilizing 'Framework A' for Training and 'Framework B' for Inference

The next optimization involves the utilization of different machine learning frameworks at different stages of the AI software development lifecycle (SDLC) [24]. Typically, developers of AI systems will prefer to use a complex library for the training stage of the SDLC. Such an example is pytorch [18], which offers great dynamic graphing capabilities as well as other training performance boosters [18] which work in a developer's favour. Following the training stage however and entering the production inference / prediction / deployment stage of the SDLC, such functionalities are generally not required. The model is already defined and trained. The framework simply needs to load the model and predict an output given some vector inputs.

Given this reduction of requirements, we start to see another opportunity for reducing the overhead of the predictive framework. In this case however, as opposed to Optimization A - which involved slimming the originally used ML framework ('Framework A') – we move away from 'Framework A' and completely swap it out for 'Framework B'.

The chart shown in Figure 3 provides a high-level overview of this process.

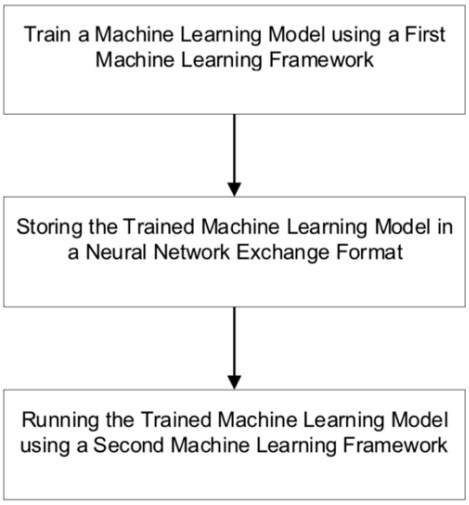

Figure 3 - High-Level Overview of 2-Step Framework Utilization Process

An important factor to consider is the selection strategy of the second machine learning framework. During this selection we must take into consideration the restrictions of the deployment platform – in the serverless case, this includes size constraints, memory constraints and a CPU only environment. Remember that there is no GPU attached to AWS Lambda. As this closely mirrors mobile device environments, a good selection is the 'caffe2' library which is 'optimized for mobile integrations, flexibility… and running models on lower powered devices' as postulated in [25]. Through this, we therefore introduce a great reduction to the prediction framework footprint in relation to the main restriction aspects that ship with such environments during the inference stage.

Another consideration is the format which models are stored in during their distribution between the two different frameworks. This is another crucial step as altering model formats could always have an impact on performance. Rather than converting between versions solely interpreted by each framework separately, we turn to the 'universal' format language of neural networks – ONNX [26] [27]. The ONNX format allows for framework interoperability – as models can be stored directly into ONNX after training by framework A; and loaded directly from ONNX for inference by framework B.

Additionally, by utilizing the Open Neural Network Exchange format (ONNX) which is an open format supported by most – if not all – ML/ Deep Learning frameworks, we minimize the possibility of performance degradation as described previously.

In summary, the process here involves: *a)* using a complex framework for training (whose usage would not suffice in a restricted environment for inference), *b)* exporting the trained model to an open format, *c)* using a much simpler and resource optimized framework for loading and computation of the open format model during inference.

D. (AWS Ecosystem specific) - Improving data lookup speeds for dealing with maximum function lifetime restrictions

Another optimization which we considered for our use case on the RTF Project is to work with the data itself which is used to serve predictions. The system serving predictions on this project (train delay predictions) would first need to lookup relevant data from a database of over 250M rows – in order to dynamically construct the multi-dimensional input vectors which are passed to the predictive models.

This step originally introduced bottlenecks in our production environment. We found that functions tend to 'hang' and induce latency when waiting for the SQL data lookup to complete. Furthermore, another pitfall here is the maximum lifetime allowance of an AWS Lambda serverless function. Although this has recently been extended to 15 minutes [3], it is not ideal for the data lookup stage to churn so much of this lifetime. Lifetime aside, performance pitfalls and execution delays all compromise the 'real-time' and 'on-demand' aspect that such services should offer.

This led the investigation in the direction of storing the data in a modern NoSQL format, utilizing the AWS DynamoDB platform [28] – thus decreasing the data lookup time by several orders of magnitude. It is once again important to note however that failure to effectively use this technique could result in no change or even worse performance when compared to the traditional methods.

Effective NoSQL usage revolves around smart partitioning and sorting of the NoSQL database. We must make sure to choose a partition key which will ensure that *read/write* loads are spread evenly across partitions. This will prevent throttling, bottlenecks and hot/cold partitions during up-scale. In the case of the RTF data, it made sense to use 'train_id' as the partitioning field. Partitioning around this column as the key is ideal since this key is '*unique enough'*. Each train is always assigned its own unique ID for data entries; but a single train can still have multiple row entries in the database (in which case we will see a repetition of the ID/partitioning field).

The most effective optimization however comes with using another field from each entry as a sorting key for each partition of the NoSQL database. An example entry (with other columns removed for simplicity) follows:

train_id	*timestamp*
AAA123	2019/01/01 11:35:55 AM
BBB123	2019/01/01 12:20:00 AM
AAA123	2019/01/01 11:55:55 AM

Each entry includes a timestamp column, which in its plain format does not offer any data storage advantage. Converting these timestamps to UNIX time [29] however allows for their utilization as a sort key. UNIX/Epoch time is described as 'the number of seconds that have elapsed since January 1, 1970 (midnight UTC/GMT)'. Essentially, this is a simple mathematical formula which is used to derive a simple 'number' format field from the complex timestamp field. Using the UNIX form, we observe naturally fully sorted partitions; as more recent times carry a bigger UNIX/Epoch time value.

Converting the above example to UNIX time yields the following results:

train_id	*timestamp*
AAA123	1546342555
AAA123	1546343755
BBB123	1546345200

This factor introduced the biggest performance boost in our data lookup methodologies – we are able to perform lookups through the 250M+ row database in under ½ a second consistently. In turn, this keeps our serverless functions well away from the 'timeout risk' – i.e., in cases where the maximum lifetime would otherwise have been approached/passed. It is not difficult to see how this methodology could be applied to other chronological and IoT device-based data (sensor data for example) [30].

IV. Evaluation

We have demonstrated and shown worked examples of the 4 main optimization techniques developed for working with complex AI workloads in a restricted environment. We used example use-cases from the *Real Time Flow* (RTF) project to illustrate the key ideas behind the techniques. In RTF, one of the objectives was to predict the delay on a trainline at any given time.

Using the aforementioned techniques and optimizations, the lifecycle of the predictor system involves a) loading the appropriate predictive model – from a pool of multiple models, b) querying relative data from a NoSQL database of 250M+ rows, c) pre-processing the query data, d) preparing the multi-dimensional input vectors for the predictive model, and e) running and returning the prediction from the model. For testing purposes, the serverless system has been attached to an API Gateway on both ends - for invocation and for returning predictions back to the user.

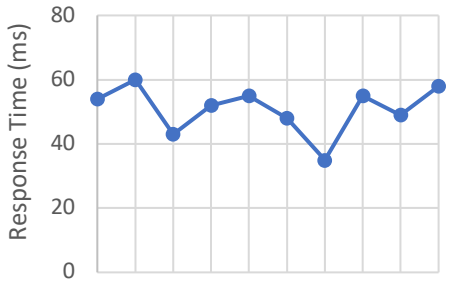

Figure 4 - Response Times of Full Prediction Request in Milliseconds (ms)

The above chart showcases a series of 10 requests sent to the deployed system via RESTful HTTP requests [31]. We observe a consistent response time, evidently with no negative impact on performance from any of the optimization techniques.

A. *Library/Framework Slimming*

The 'minimization' technique proves to be a key step in transforming a codebase incompatible with a serverless deployment into one which is. In our testing and deployed system, we have slimmed the pytorch library from 467MB down to 98.6MB using this technique. With this result, we are technically able to deploy (as we are under the 250MB limit) without applying any other optimizations.

As stated previously however, since the deletion margin varies greatly by the actions performed by the code on a case by case basis, the other recommended methodologies should also be taken into consideration. The automation strategy presented makes it easy to run this technique automatically with an ever-changing and on-going development codebase.

B. *Dynamic model loading from cloud storage; instead of shipping models locally in the deployment package*

Through the multiple tests shown in Figure 1, this method has proven to be robust and should always be used when serving any type of model in a serverless environment. Since there are no trade-offs and performance is consistent, using the '/tmp' directory is a great way to abstract model size and footprint away from the serverless deployment package – in turn allowing for the development of a more complex source code base. Space that would have otherwise been taken up by models can now even be used for the packaging of additional frameworks.

C. *Dual Framework Development with ONNX in between; Complex Framework for training & development, Simple Framework for deployment and inference.*

In the case where the predictive system source code is extremely simple and does not even perform some pre-processing – solely prediction – we have demonstrated how a mobile framework can be deployed to serve predictions. A key step during this procedure is to utilize the ONNX format as the 'middle-man' when passing models through different frameworks. This ensures robustness and has no effects on predictive performance during the inference stage; as we have seen in our test/re-test situations. The granularity and complexity of development is maintained - by utilizing powerful frameworks during the training stage.

D. *Working on Improving the Handling of Data Lookup/Storage Methodologies.*

This step is very specific to use cases which handle chronological data. In the serverless world however, there is definitely a plethora of connected devices and systems [2], [32] which make use of such timestamped data – IoT sensors is an example. Given a system which handles data in a manner similar to what we have shown, predictions could be served in real time even when some complex pre-processing is in place. In the RTF project use cases, we have seen that data querying and processing times take 500-600x less when compared to traditional data handling / storage techniques.

Real Time Flow (RTF) Project Case Study

As part of the RTF project, the previously described Serverless AI architecture, principles and optimization techniques are being actively utilized to deliver the benefits highlighted in Section I. The Real-Time visual analytics software used to create the user interface for the RTF project is Emu Analytics' Flo.w™ solution.

Flo.w™ is an innovative, cloud based geo-spatial analytics and visualization platform that is designed to ingest, analyze and visualize high volume, fast moving data of the type typically delivered from telemetry, IOT sensors and networks. In the RTF Project it is ingesting, analyzing and visualizing the movements and metrics of the whole UK rail network in real-time. In the same interface it provides the ability to traverse backwards in time over historic information and patterns. Other time-series data including population movements (derived from mobile phone movements) is also ingested into the platform alongside several other contextual datasets, including railway infrastructure (lines, stations, level crossings, etc.).

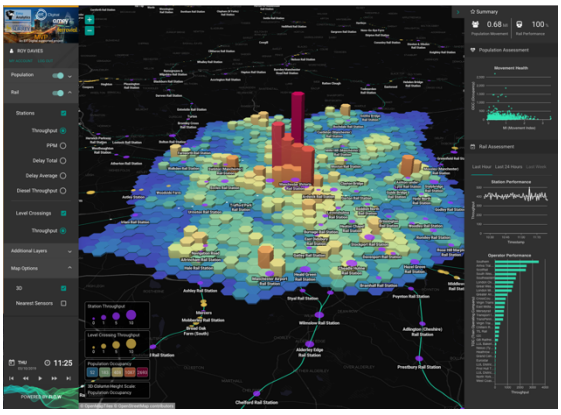

Figure 5 - Emu Analytics Flo.w ™ Application visualising population movements alongside Real-Time UK train movements and metrics for the RTF Project

The predictive AI system, using techniques described in this paper (developed by the University of Surrey), is ideally suited for highly effective and efficient integration into platforms such as Flo.w™. The architectural complexities and overheads of running an AI workload are abstracted away from the visualization platform with a simple parameterized call being all that is required to request predictions. The suite of developed AI models ensure that predictions can be requested at the relevant point within the Flo.w™ platform (i.e., within the analytical processing pipeline or on user-initiated clicks) at both the scope and volume required (i.e., single station or multiple stations).

The optimization techniques employed within the serverless architecture ensure that the response times for delivering the predictions are compatible with the requirements of the Flo.w™ platform to be hyper-performant in delivering real-time, actionable insights to the end user. Finally, the Serverless approach ensures that the predictive AI solution can scale up and down as required by different use cases and deployments at an optimized cost-effectiveness that is based on demand and not on physical infrastructure.

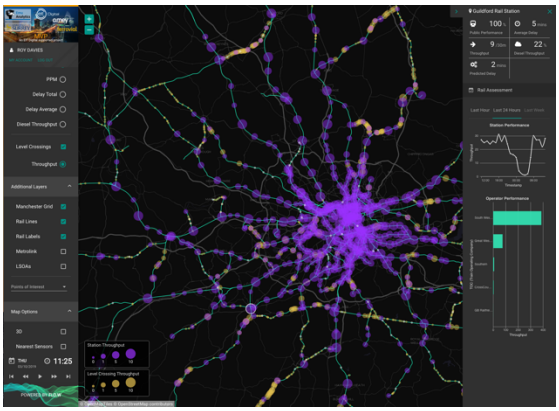

Figure 6 - University of Surrey Serverless AI/ML Architecture and Model delivering on-demand predicted train service delays into the Flo.w ™ platform

V. Conclusion & Future Work

Overcoming the constraints typically surrounding a resource constrained environment is a time-consuming and risky task. Attention must be constantly noted to performance in relation to all the trade-offs being made to accommodate the restrictive environment. In this study we have shown techniques which can be used to accommodate a complex AI workload into a resource-constrained serverless environment. Due to the similarity of such an environment with mobile/IoT devices, it is easy to see how the learnings can be transferred over to other use cases.

In addition, we have proven that the techniques can work together in harmony to deliver an industry level deployed serverless solution. This has been demonstrated by the integration into Emu Analytics Flo.w™ geo-spatial analytics platform. The benefits of shipping serverless over traditional architectures include massive cost savings, robust scalability both ways as well as an easier development pipeline [1], [33].

As Cloud Providers work to update serverless platforms, the next steps include research into how such deployments could be made even easier through the likes of functionalities such as AWS Lambda Layers [34]. Additionally, another possibility includes investigating the development of compression algorithms [35] to reduce the footprint of predictive models even further.

With the introduction of newly released specialist AI-accelerator hardware [36], an implementation which includes such specialized hardware could improve the performance of the underlying system even further. In such a case, it would be ideal to then investigate how such a microservice architecture can apply to the training stage of the SDLC as well. We are seeing an ever increasing demand for faster training times [37], so with such work applied to the training stage, a serverless batch training solution may be possible.

References

[1] G. C. Fox, V. Ishakian, V. Muthusamy, and A. Slominski, 'Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research', arXiv:1708.08028 [cs], 2017.

[2] I. Baldini et al., 'Serverless Computing: Current Trends and Open Problems', arXiv:1706.03178 [cs], Jun. 2017.

[3]docs.aws.amazon.com/lambda/latest/dg/limits.html Last accessed: 10 October 2019

[4] A. Jangda, D. Pinckney, Y. Brun, and A. Guha, 'Formal Foundations of Serverless Computing', arXiv:1902.05870 [cs], Feb. 2019.

[5] S. Moschoyiannis, L. Maglaras, N. A Manaf, 'Trace-based Verification of Rule-Based Service Choreographies', *IEEE Int'l Conf. on Service Oriented Computing and Applications* (IEEE SOCA 2018), 2018

[6] M. P. Papazoglou, P. Traverso, S. Dustdar, *et al*, 'Service-Oriented Computing Roadmap' Dagshtul Seminar Proc. 05462, Service-Oriented Computing (SOC) pp. 1-29, 2006

[7] A. Razavi. S. Moschoyiannis, P. Krause 'A Coordination Model for Distributed Transactions in Digital Ecosystems', *IEEE Int'l Conf. on Digital*

Ecosystems and Technologies (IEEE DEST 2007), 2007.
[8] A Razavi, A. Marinos, S. Moschoyiannis, P. Krause, 'RESTful Transactions supported by the Isolation Theorems', *Int'l Conf. on Web Engineering* (ICWE 2009), LNCS 5648, pp. 394-409, Springer, 2009.
[9] A. Marinos, S. Moschoyiannis, P. Krause, 'Towards a RESTful Infrastructure for Digital Ecosystems', *ACM Conf. on Management of Emergent Digital Ecosystems* (*MEDES 2009*), ACM SIGAPP, pp.340-344, 2009.
[10] A. Kobusinska and C.-H. Hsu, 'Towards increasing reliability of clouds environments with RESTful web services', *Future Generation of Computer Systems*, vol. 87, pp. 502–513, 2018.
[11] P. Krause, A. Razavi, S. Moschoyiannis, and A. Marinos, 'Stability and Complexity in Digital Ecosystems', *IEEE Int'l Conf. on Digital Ecosystems and Technologies* (IEEE DEST 2009), 2009
[12] S. Moschoyiannis, N. Elia, A. Penn *et al*, 'A web-based tool for identifying strategic intervention points in complex systems', *Games for the Synthesis of Complex Systems (*CASSTING'16 @ ETAPS 2016), EPTCS 220, pp.39-52, 2016.
[13] V. Ishakian, V. Muthusamy, and A. Slominski, 'Serving Deep Learning Models in a Serverless Platform', 2018, pp. 257–262.
[14] G. McGrath and P. R. Brenner, 'Serverless Computing: Design, Implementation, and Performance', in 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW), 2017, pp. 405–410.
[15] D. Crankshaw et al., 'The Missing Piece in Complex Analytics: Low Latency, Scalable Model Management and Serving with Velox', arXiv:1409.3809 [cs], Sep. 2014.
[16] J. Bhogal and I. Choksi, 'Handling Big Data Using NoSQL', in 2015 IEEE 29th International Conference on Advanced Information Networking and Applications Workshops, 2015, pp. 393–398.
[17] Y. Li and S. Manoharan, 'A performance comparison of SQL and NoSQL databases', in 2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM), 2013, pp. 15–19.
[18] A. Paszke et al., 'Automatic differentiation in PyTorch', 2017.
[19] R. J. Urbanowicz and S. W. Wilson, 'Intro to Learning Classifier Systems', Springer, Berlin, 2017
[20] S. Moschoyiannis and V. Shcherbinin, 'Fine tuning run parameter values in rule-based machine learning', 13th RuleML Challenge @ RuleML+RR 2019, CEUR-WS, vol 2438, 2019
[21] M. R. Karlsen and S. Moschoyiannis, 'Learning condition-action rules for personalised journey recommendations' RuleML+RR 2018, LNCS 11092, pp. 293-301, Springer, 2018
[22] W. Gao et al., 'Data Motifs: A Lens Towards Fully Understanding Big Data and AI Workloads', arXiv:1808.08512 [cs], Aug. 2018.
[23] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system." in NSDI, 2017, pp. 613–627.
[24] M. de Prado, J. Su, R. Dahyot, R. Saeed, L. Keller, and N. Vallez, 'AI Pipeline - bringing AI to you. End-to-end integration of data, algorithms and deployment tools', arXiv:1901.05049 [cs, eess, stat], Jan. 2019.
[25] Mar 2018, https://caffe2.ai/docs/mobile-integration.html
[26] 2019, https://onnx.ai
[27] X. Cai, P. Zhou, S. Ding, G. Chen, and W. Zhang, 'Sionnx: Automatic Unit Test Generator for ONNX Conformance', arXiv:1906.05676 [cs], Jun. 2019.
[28] 2019, https://aws.amazon.com/dynamodb/
[29] L. P. Quoc, 'Wrox - Beginning Linux Programming 4th Edition (2008)'.
[30] J. Lin, W. Yu, N. Zhang, X. Yang, H. Zhang, and W. Zhao, 'A Survey on Internet of Things: Architecture, Enabling Technologies, Security and Privacy, and Applications', IEEE Internet of Things Journal, vol. 4, no. 5, pp. 1125–1142, Oct. 2017.
[31] 2019, https://docs.aws.amazon.com/apigateway/api-reference/making-http-requests/
[32] E. Al-Masri, I. Diabate, R. Jain, M. H. L. Lam, and S. R. Nathala, 'A Serverless IoT Architecture for Smart Waste Management Systems', in 2018 IEEE International Conference on Industrial Internet (ICII), 2018, pp. 179–180.
[33] G. Adzic and R. Chatley, 'Serverless computing: economic and architectural impact', 2017, pp. 884–889.
[34] 2019, https://docs.aws.amazon.com/lambda/latest/dg/configuration-layers.html
[35] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, 'A Survey of Model Compression and Acceleration for Deep Neural Networks', arXiv:1710.09282 [cs], Oct. 2017.
[36] Y. Yu, Y. Li, S. Che, N. K. Jha, and W. Zhang, 'Software-Defined Design Space Exploration for an Efficient AI Accelerator Architecture', arXiv:1903.07676 [cs], Mar. 2019.
[37] T. B. Johnson and C. Guestrin, 'Training Deep Models Faster with Robust, Approximate Importance Sampling', in Advances in Neural Information Processing Systems 31, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., 2018, pp. 7265–7275