Received February 18, 2020, accepted March 25, 2020, date of publication April 2, 2020, date of current version April 24, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2985282

Enabling Serverless Deployment of Large-Scale AI Workloads

ANGELOS CHRISTIDIS[1], SOTIRIS MOSCHOYIANNIS[1], (Member, IEEE), CHING-HSIEN HSU[2,3,4], (Senior Member, IEEE), AND ROY DAVIES[5]
[1]Department of Computer Science, University of Surrey, Guildford GU2 7XH, U.K.
[2]School of Mathematics and Big Data, Foshan University, Foshan 528000, China
[3]Department of Computer Science and Information Engineering, Asia University, Taichung 41354, Taiwan
[4]Department of Medical Research, China Medical University Hospital, China Medical University, Taichung 40402, Taiwan
[5]EMU Analytics, London WC1N 2LG, U.K.

Corresponding author: Sotiris Moschoyiannis (s.moschoyiannis@surrey.ac.uk)

This work was supported in part by the EIT Digital IVZW through the Real-Time Flow Project under Grant 18387–SGA201, in part by the EPSRC IAA Project AGELink under Grant EP/R511791/1, and in part by the National Natural Science Foundation of China under Grant 61872084.

ABSTRACT We propose a set of optimization techniques for transforming a generic AI codebase so that it can be successfully deployed to a restricted serverless environment, without compromising capability or performance. These involve (1) slimming the libraries and frameworks (e.g., pytorch) used, down to pieces pertaining to the solution; (2) dynamically loading pre-trained AI/ML models into local temporary storage, during serverless function invocation; (3) using separate frameworks for training and inference, with ONNX model formatting; and, (4) performance-oriented tuning for data storage and lookup. The techniques are illustrated via worked examples that have been deployed live on geospatial data from the transportation domain. This draws upon a real-world case study in intelligent transportation looking at on-demand, real-time predictions of flows of train movements across the UK rail network. Evaluation of the proposed techniques shows the response time, for varying volumes of queries involving prediction, to remain almost constant (at 50 ms), even as the database scales up to the 250M entries. The query response time is important in this context as the target is predicting train delays. It is even more important in a serverless environment due to the stringent constraints on serverless functions' runtime before timeout. The similarities of a serverless environment to other resource constrained environments (e.g., IoT, telecoms) means the techniques can be applied to a range of use cases.

INDEX TERMS Intelligent transportation, predicting train delays, AWS, functions as-a-service, Lambda, NoSQL, serverless, resource-constrained, serverless codebase optimization, rail traffic big data.

I. INTRODUCTION

Standard architectures for deploying AI workloads currently mirror typical client-server architectures with the AI models and data sitting on the server side and requests coming from the client side. This kind of architecture has been shown to: not scale well, as larger volumes of requests will require multiple servers; infer overheads for ensuring all servers are synchronized; compromise reliability, by introducing failure points; and, make load balancing even more challenging. Therefore, deployed AI platforms typically ship with bulky system architectures which present bottlenecks and a high risk of failure.

Serverless architectures in comparison 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 at real-time, resulting in cost savings for all parties involved, and less strain on the developers. Such benefits drive the shift we are witnessing nowadays from traditional architectures to 'microservices' or serverless based solutions.

Serverless architectures abstract away most of the complexities and bottlenecks mentioned previously that are

The associate editor coordinating the review of this manuscript and approving it for publication was Sabah Mohammed.

typically associated with high volume, production level workloads. These architectures place focus on solving the problem at hand and developing a high quality application rather than getting lost in the realm of architectural constraints. Code is bundled in a deployment package, which on demand, runs in an encapsulated but stateless computing container. The containers are inflated dynamically and triggered by events (e.g. APIs) which may last for as little as one invocation. Following invocation(s), all resources and dependencies are 'destroyed', reinforcing the concept of a stateless architecture and consequent billing model. However, the growing complexity of a codebase, and specifically AI workloads, could render the application incompatible with serverless deployment.

In order to deliver the promised 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 intended to perform low level automated tasks such as scheduled data transfer from one database to another, or 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.

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 on intelligent transport in digital cities. The RTF project focuses on novel techniques for monitoring and predicting the flows of people and goods across transport networks in an urban environment, and is collaborative effort between the Ferrovial,[1] Amey,[2] Emu Analytics[3] and the University of Surrey. 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. However, given the pairing of traditional serverless constraints with an increasingly complex RTF codebase, this would not work 'out-of-the-box'. In our case, the codebase involves machine learning and deep reinforcement learning models [5] for prediction and control that would not originally support serverless deployment due to factors such as size and runtime.

Serverless deployments can mitigate pitfalls of current deployments of heavy AI workloads and provide a cost-effective, automatically scalable (up or down) and elastic real-time on-demand AI solutions. 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. This gradually led us to the techniques derived in the research reported in this paper.

In this paper, we propose a suite of optimisation techniques which can be applied to any AI codebase and transform it into a package which is ready for serverless deployment. More specifically, our contribution lies with presenting and evaluating a number of different techniques, including:

1) Reducing the footprint of the AI (python) libraries & frameworks used in the codebase
2) Dynamic loading and injection of the AI models into temporary runtime memory
3) A 2-Step Framework ML Process: Training and Inference, with ONNX formatted models
4) Improving the handling of data lookup and storage, through innovative partitioning and indexing techniques.

Our 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.

We note that this paper builds and extends the work that appeared in the IEEE Conference on Service Oriented Computing and Applications (SOCA) 2019 [6]. More specifically, the present paper includes a more elaborate treatment of the data handling aspects (lookup and storage), reports on additional experiments and associated evaluation, takes a closer look at related work and the more general context of infrastructures for serving AI workloads in the cloud, although this piece of work is fairly novel.

The remainder of this paper is structured as follows. Section II presents the key techniques (points 1) - 4) mentioned earlier) which comprise the main contribution of the paper. Section III reports on evaluation via experiments in the context of a real case study involving a number of experiments on the lifecycle of predictions on train delays. Section IV outlines the live deployment of the serverless AI in a real-time location analytics platform, as part of an integrated Intelligent Transportation solution. Section V reports on related work. Section VI contains some concluding remarks.

[1]http://www.ferovial.com
[2]http://www.amey.co.uk
[3]http://www.emu-analytics.com

II. OPTIMIZATION TECHNIQUES AS SOLUTIONS TO SERVERLESS CONSTRAINTS

A. MINIMIZING / SLIMMING' PYTHON LIBRARIES AND FRAMEWORKS AND AUTOMATING THE PROCESS

As noted in [7], 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 [7], 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.

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.

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

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

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.

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 overcoming 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 Real-Time Flow project use cases: RNNs, CNNs, LSTM, LCS [8] (including XCS [9], UCS, and XCSI [10]) models that are associated with a specific use case may exceed a couple of hundred MB themselves [11]. 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 [12] 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).

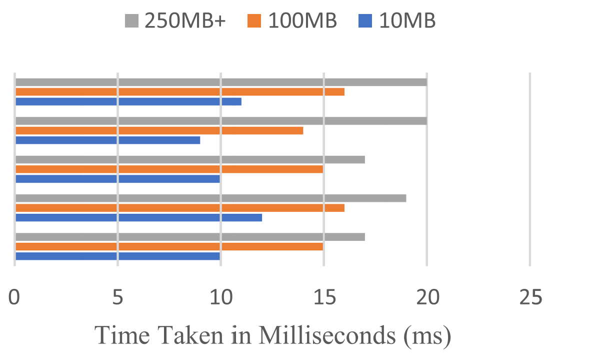


FIGURE 2. **Time to load & extract varying model sizes from cloud storage to local '/tmp' directory of a serverless AWS Lambda function.**

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) [13]. Typically, developers of AI systems will prefer to use a complex library for the training stage of the SDLC. Such an example is pytorch [7], which offers great dynamic graphing capabilities as well as other training performance boosters [7] 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 1 - 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. The 2-step framework process is denoted through the 1st and 3rd boxes in the figure. The intermediate 2nd box, simply provides a description for how models are passed between the two frameworks.

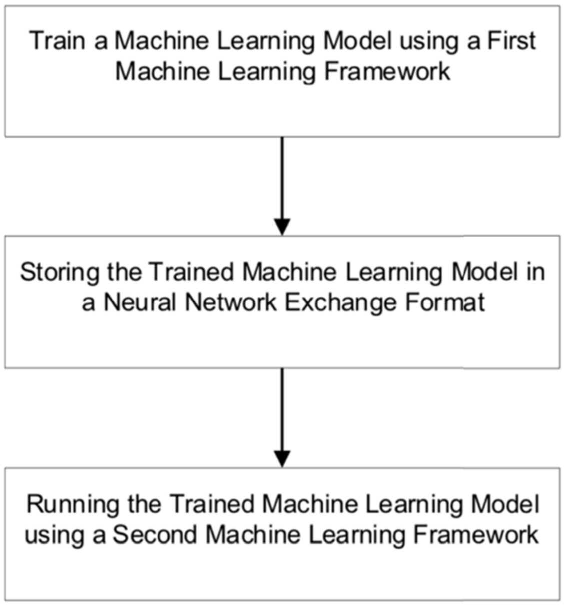

Train a Machine Learning Model using a First Machine Learning Framework
Storing the Trained Machine Learning Model in a Neural Network Exchange Format
Running the Trained Machine Learning Model using a Second Machine Learning Framework

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 [14]. 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 [15], [16]. 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 [17] – 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 [18] 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 1/2 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) [19].

III. 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.

Looking further and solely into step b), we turn back to the legacy SQL-based system used originally in the RTF project for a head-to-head comparison against the optimized NoSQL solution derived in Part 4 of Section 3 (Implementation). The tests that follow aim to provide a finer look into the exact performance gains behind transitioning over to, and thus pairing a NoSQL based setup with our serverless AI deployment.

The format of the 'Big Data' in the RTF use case closely resembles the example tables shown previously. After data is appended/collated from other sources, the database consists of about 20 columns; with 250M entries which are cross-referenced upon serving real-time predictions. Using the original SQL based database system with Microsoft SQL Server Management Studio (SSMS), we test the query times in increments. A reading is taken each time the database grows by 25M rows; a reading being the time taken for the SQL query to execute to completion. The 'query' in this case would be a request for the set of all occurrences of a certain 'train_id'. Note that the tests were performed on the University of Surrey's central SQL Server, which is considered to be a high spec server.

Figure 4 shows how query time changes (for both SQL and NoSQL) as database size scales massively up to 250M rows total. It is obvious how the query time increases linearly for the legacy SQL system as the database size increases. Setting aside the linear growth, even at the first reading – where database size was restricted to just 25M rows i.e. $\frac{1}{10}$ total size – we find the query time to be 22 seconds. This would not work in favor of 'best-practice' real time system design and would not only deteriorate the whole predictive system's performance, but also ruin the experience for the intended end user. The long query times are only part of the prediction process, which would cause serverless functions to 'hang' and induce lag. In most cases, this is due to the fact that whole table scans are required in order to retrieve all occurrences of a single entity (by 'train_id). It is also worth pointing out that given a large enough dataset, theoretically the queries would take so long that the serverless function maximum lifetime runtime (mentioned previously) would be exceeded – causing the functions to fail completely.

In the case of our optimized NoSQL solution, we observe that the query time does not change, rather follows a constant behavioural trend as the database scales up to the 250M entries. Queries constantly run in sub-second time intervals ($\sim$50ms) due to the setup described in Section 3D, regardless of database growth. The partitions are set up and sorted in such a way that whole table scans are not required. The NoSQL solution is extremely horizontally scalable; meaning that different nodes/storage entities can hold different parts of the data. In the case of the legacy SQL solution, we observed a monolithic, vertically scalable solution; meaning that the only the specifications of the single machine entity holding the whole database can be boosted to increase performance. The limitations of hardware do come into play here and as we have already mentioned the tests were performed on an already high spec SQL Server instance.

The second chart included in Figure 5 showcases a series of 10 requests sent to the deployed system via RESTful HTTP requests [20], performed 4 different times. Each stage

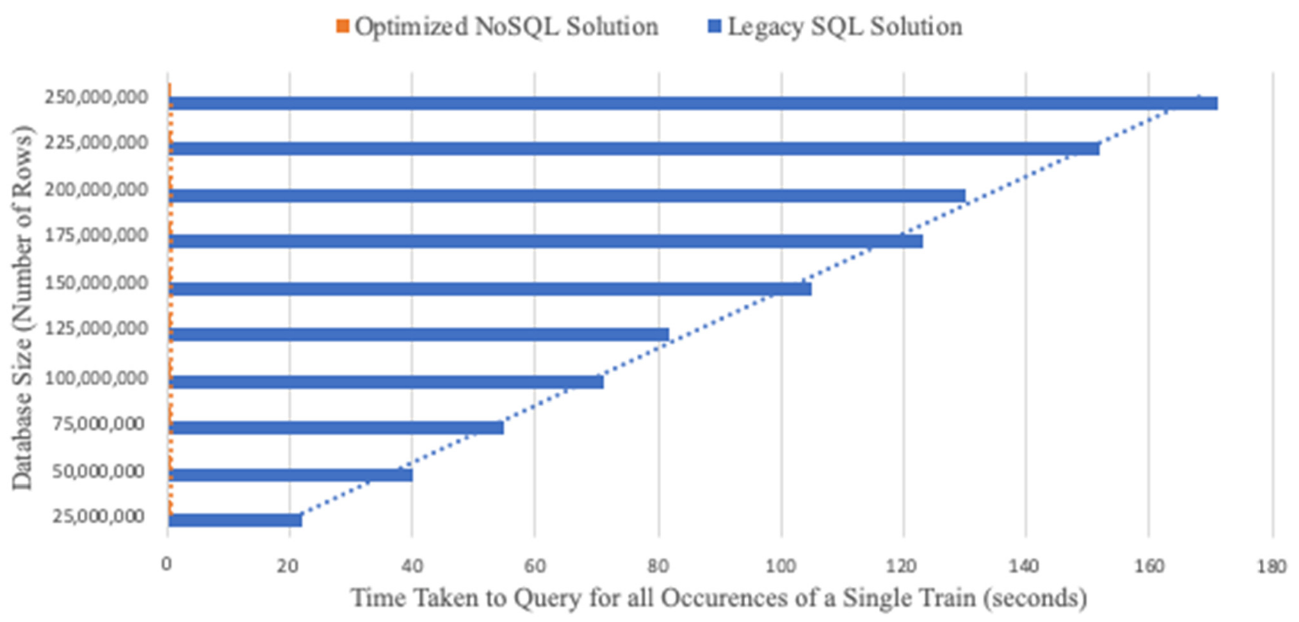


FIGURE 4. Legacy SQL vs optimised NoSQL query times.

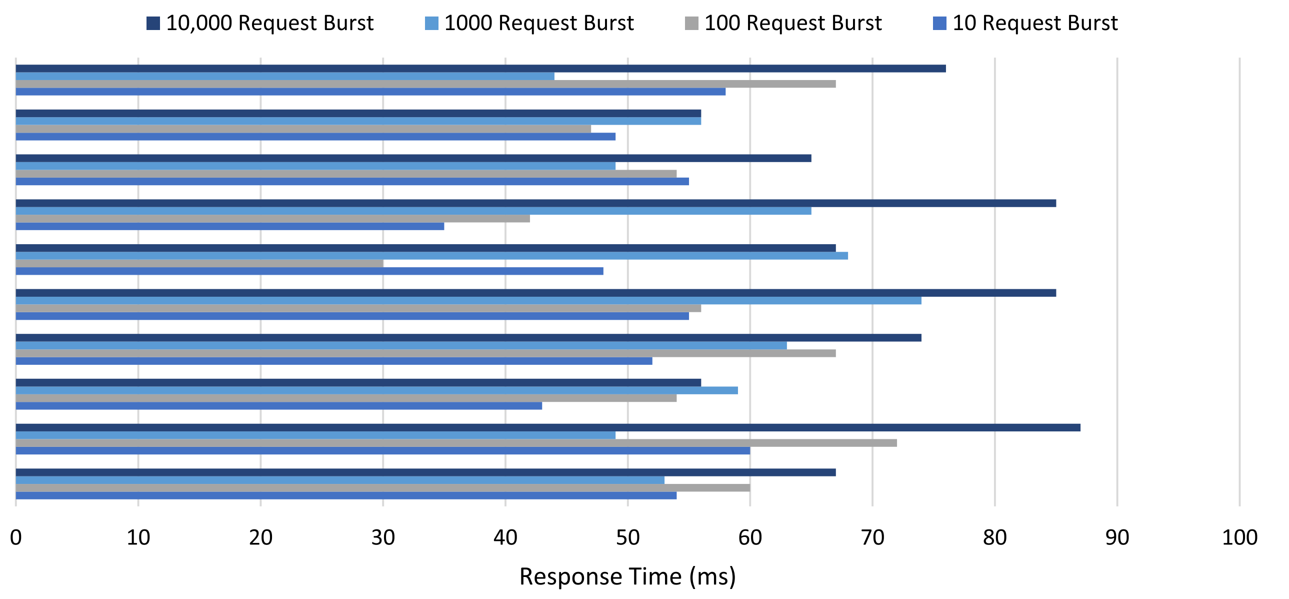


FIGURE 5. Response times of deployed system from 4 different stages/categories of burst requests.

denotes a different ‘state’ of load on the system; in the first stage we test response time when the system was subject to only 10 requests. We scale this up to 100, 1000 and finally 10,000 parallel requests to the system to prove that there is no degrading in the performance of the deployed live system. There is no ‘flinch’, namely a minor reaction which ensures response times remain constant across the board.

We observe a consistent response time, evidently with no negative impact on the whole system’s performance from any of the optimization techniques. The lifecycle of the request goes through all steps a) to e) mentioned in the beginning of the Evaluation section.

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

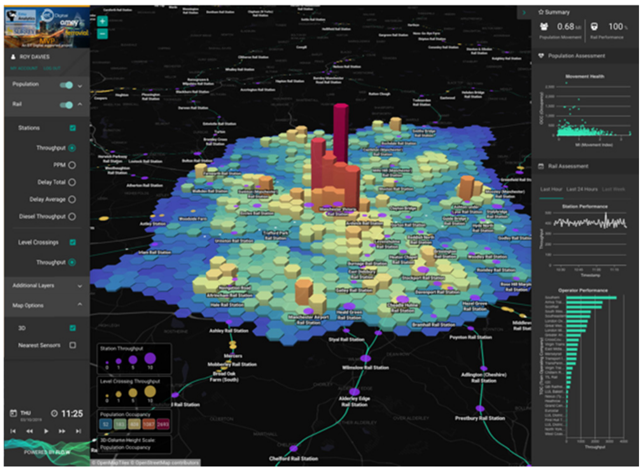


FIGURE 6. EMU analytics Flo.w^{TM} application visualising population movements alongside real-time U.K. train movements and metrics for the RTF project.

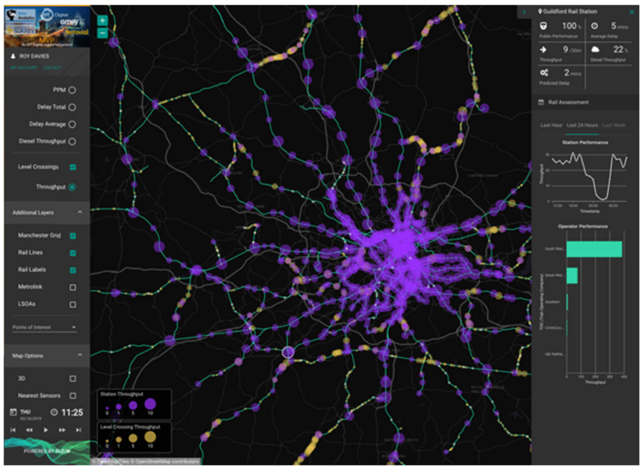


FIGURE 7. University of Surrey serverless AI/ML architecture and model delivering on-demand predicted train service delays into the Flo.w^{TM} platform.

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 AND 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], [21] 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.

IV. CASE STUDY–THE REAL TIME FLOW (RTF) PROJECT

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^{TM} solution.

Flo.w^{TM} 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.).

The Flo.w^{TM} platform itself is also based on a micro-service architecture. For real-time analytics It contains an enhanced Kafka based ingestion and processing service that can run its own analytical micro-services or call out to external ones. Similarly, the visualisation layer is capable of calling external services when on-demand user interface actions are initiated. The Flo.w^{TM} user-interface (UI) is delivered within an internet browser and is designed to be hyper-performant. To deliver this performance it places a requirement on both its own and external supporting services to be highly optimised and efficient at all levels of scale.

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^{TM}. 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^{TM} 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 platform aims to commercialize train delay predictions by targeting Train Operating Companies (TOCs) directly. Case studies by the UK's technology and innovation centre for Intelligent Mobility (Catapult Transport Systems), have shown funding/investments in public transport; in order to shift towards a 'less car dependent' population [22]. There is a strongly rising trend in the demand of rail travel - demand has more than doubled since 1994 [23]. TOCs face the constant challenge of keeping their rail services running smoothly in unison with this rising demand. Unfortunately, passenger satisfaction levels are dropping [23] and TOCs are facing fines in excess of £100M annually because of compensation paid whenever their trains are delayed. This highlights the importance of such a flow prediction platform developed in RTF, as TOCs can actively monitor their network performance into the future and instantly put contingency plans in place to minimize disruption. These plans will help minimize delay times and not only drive operating costs down, but also increase customer satisfaction levels. Furthermore, in some cases TOCs may choose to reroute/cancel some services to prioritize others. This factor, together with the fact that contingency planning will avoid the need to run extra rail journeys (to make up for delays), means that the total number of rail journeys a TOC will need to run to achieve the same commute plan will also drop.

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^{TM} 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.

V. RELATED WORK

Whilst there have been previous attempts at deploying AI workloads to AWS Lambda [24], the work is performed within the 'comfort zone' of the platform. To the best of our knowledge, there are no examples of workloads which proved to be incompatible so that constraints of the serverless environment had to be breached. Implementations such as the one described in [24] fall short of the complexity that would make the associated codebase incompatible with serverless deployment.

To enforce the above, we refer to the 'limitations of serverless computing' mentioned in [25]. As this piece of research mentions, there is a direct relationship between the high level of granularity of serverless computing with the decreasing compatibility of libraries and frameworks with the serverless computing deployment model. Code aside, as libraries themselves increase in complexity, their usage and inclusion starts to be almost impossible in a serverless package due to severe file size restrictions associated with deployment packages. This is a factor which becomes even more of a problem when the codebase itself also begins to grow in complexity. With this, there is even less overhead for the amount of space allocated to libraries/frameworks; rendering large footprint versions incompatible with this deployment strategy.

Furthermore, the study mentions the limits of deploying a 'single solution' to a 'single serverless function' due to imposed constraints of serverless environments as mentioned above. For example, it is argued that sometimes developers are forced to go 'too granular'; splitting one piece of work into two – because of restrictions. This factor causes problems with monitoring of code, debugging, multiple authentication or even further complications around code refactoring. This contradicts the serverless aim of simplifying the architecture and deployment as much as possible. Instead, the solution is almost 'forced' in some cases to deteriorate (due to vendor platform restrictions).

Although discussed further in our future work section, it is worth making a mention of attempts to connect GPU processing to serverless environments. Especially in the AI case, this would speed up execution times even more as serverless would be 'GPU enabled' [26]. The current serverless platform offerings from cloud vendors do not support GPU processing within the encapsulated serverless environments. To date, the primary factor has been cost of GPU resources, but the high runtime of GPU related activities. The second factor would not pair with a serverless environment, as one of the restrictions – as already mentioned – is a limited maximum runtime allowance for functions. However, there is further research to be done in this topic as:

- AI inference does not take as long as the training stage. We show how inference is possible to achieve in current serverless standards with our optimizations. Therefore, since exceeding runtime is not a problem during inference, GPU addition should be considered.
- The current GPU enabled serverless developments are not 'truly' 100% in line with serverless standards; as some of the modular blocks which host the GPUs still follow an 'always enabled' pattern.

Since current vendor options do not include a GPU enabled version of serverless, there have been attempts at creating parallelization through a custom formula of marriage between serverless function 'workers' at runtime [27]. The concept involves a central master which spawns other workers (serverless functions) which work in parallel to complete a large-scale optimization problem. Efficiency is almost doubled in this case when compared to a traditional setup; but again, it must be noted that the solution is not 100% true to the serverless name. As in the previous case, we find that the 'central master node' is actually a server. Including this in the architecture – which unfortunately at this stage of technology is necessary - voids the integrity of labelling this a fully serverless solution.

Rising codebase complexity could be related to reasons such as large shipped AI model size, or even high-volume library usage, which also contributes to mounting codebase size. For instance, there is a trend in developer behaviour to roll back to a traditional server-based architecture once a codebase becomes too complex for serverless deployment.

Research reported in [28], [29] has analysed the ideal architecture for a microservices / serverless environment, focusing on the turning point 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 [30], [31] 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.

VI. CONCLUSION AND FUTURE WORK

We have presented and detailed a set of serverless code optimization techniques that can be used to transform production AI workloads on big data so that they can be deployed in a serverless architecture. The approach has been illustrated in the context of monitoring and predicting flows of train movements, at real-time (Real-Time Flow project). The AI workload in this case is involved Machine Learning (RNNs, DRL) models executing on large data set of scheduled vs actual departure and arrival times, per station, for each train service across the whole of the UK rail network.

In previous work, we have been concerned with aspects of specification and verification [32] in service-oriented environments [33], which picked up from work on long-running transactions [34] and a RESTful architecture [35], [36] for resources in complex digital ecosystems [37], [38]. The application of rule-based machine learning [39] to complex networks [40], in combination with identifying overlapping parts of commuter journeys [41] has been successful in making accurate personalized recommendations to stranded passengers for their onward journeys [10].

However, the data sets we are dealing with here are magnitude larger – 6 years of train movements across the whole of the UK rail network. Additionally, the Intelligent Transportation solution here comes with stringent requirements on serving the requests for prediction at real time.

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^{TM} 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], [42].

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 [43]. Additionally, another possibility includes investigating the development of compression algorithms [44] to reduce the footprint of predictive models even further.

With the introduction of newly released specialist AI-accelerator hardware [45], 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 [46], 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," 2017, *arXiv:1708.08028*. [Online]. Available: http://arxiv.org/abs/1708.08028

[2] I. Baldini *et al.*, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*. Singapore: Springer, 2017, pp. 1–20.

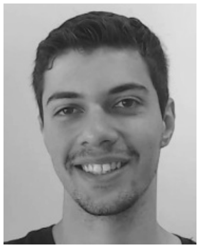
[3] *AWS Lambda Developer Guide, AWS Lambda Limits*. Accessed: Feb. 10, 2020. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html

[4] A. Jangda, D. Pinckney, Y. Brun, and A. Guha, "Formal foundations of serverless computing," in *Proc. ACM Program. Lang.*, vol. 3, 2019.

[5] G. Papagiannis and S. Moschoyiannis, "Learning to control random Boolean networks: A deep reinforcement learning approach," in *Complex Networks, SCI*, vol. 881. Cham, Switzerland: Springer, 2019, pp. 721–734.

[6] A. Christidis, R. Davies, and S. Moschoyiannis, "Serving machine learning workloads in resource constrained environments: A serverless deployment example," in *Proc. IEEE 12th Int. Conf. Service-Oriented Comput. Appl. (SOCA)*, 2019, pp. 55–63.

[7] A. Paszke *et al.*, "Automatic differentiation in PyTorch," in *Proc. 31st Conf. Neural Inf. Process. Syst. (NIPS), Workshop Autodiff Decision*, Long Beach, CA, USA, 2017.
[8] R. J. Urbanowicz and W. Browne, *An Introduction to Learning Classifier Systems*. Berlin, Germany: Springer, 2017.
[9] S. Moschoyiannis and V. Shcherbinin, "Fine tuning run parameter values in rule-based machine learning," in *Proc. 13th RuleML Challenge RuleML+RR (CEUR-WS)*, vol. 2438, 2019. Accessed: Apr. 8, 2020. [Online]. Available: http://ceurws.org/Vol-2438/paper9.pdf
[10] M. R. Karlsen and S. Moschoyiannis, "Learning condition-action rules for personalised journey recommendations," in *Proc. RuleML+RR*, LNCS 11092. Cham, Switzerland: Springer, 2018.
[11] W. Gao *et al.*, "Data motifs: A lens towards fully understanding big data and AI workloads," in *Proc. 27th Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, 2018, pp. 1–14.
[12] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *Proc. 14th USENIX Conf. Netw. Syst. Design Implement. (NSDI)*, 2017, pp. 613–627.
[13] 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," in *Proc. Emerg. Deep Learn. Accel. (EDLA) Workshop HiPEAC*, 2019.
[14] *Caffe2 Documentation, Integrating Caffe2 on iOS/Android*. Accessed: Feb. 12, 2020. [Online]. Available: https://caffe2.ai/docs/mobileintegration.html
[15] *Open Neural Network Exchange |(ONNX) Documentation*. Accessed: Feb. 12, 2020. [Online]. Available: https://onnx.ai
[16] X. Cai, P. Zhou, S. Ding, G. Chen, and W. Zhang, "Sionnx: Automatic unit test generator for ONNX conformance," Jun. 2019, *arXiv:1906.05676*. [Online]. Available: http://arxiv.org/abs/1906.05676
[17] AWS. *Amazon DynamoDB Online Documentation*. Accessed: Feb. 12, 2020. [Online]. Available: https://aws.amazon.com/dynamodb/
[18] N. Matthew and R. Stones, *Beginning Linux Programming*, 4th ed. Hoboken, NJ, USA: Wiley, 2008.
[19] 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 Things J.*, vol. 4, no. 5, pp. 1125–1142, Oct. 2017.
[20] *Amazon API Gateway REST API Reference*. Accessed: Feb. 12, 2020. [Online]. Available: https://docs.aws.amazon.com/apigateway//making-http-requests/
[21] 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 *Proc. IEEE Int. Conf. Ind. Internet (ICII)*, 2018, pp. 179–180.
[22] *Traveller Needs and UK Capability Study—'Supporting the Realisation of Intelligent Mobility in the UK'*, Catapult Transp. Syst., U.K. Dept. Transp., U.K. Dept. Bus. Innov. Skills, InnovateUK, London, U.K., Oct. 2015.
[23] *Rail Factsheet*, Dept. Transp., London, U.K., Dec. 2019, p. 6.
[24] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving deep learning models in a serverless platform," in *Proc. ACM*, New York, NY, USA, 2018, pp. 257–262.
[25] M. Sewak and S. Singh, "Winning in the era of serverless computing and function as a service," in *Proc. 3rd Int. Conf. Converg. Technol. (I2CT)*, 2018, pp. 1–5.
[26] J. Kim, T. J. Jun, D. Kang, D. Kim, and D. Kim, "GPU enabled serverless computing framework," in *Proc. 26th Euromicro Int. Conf. Parallel, Distrib. Network-Based Process. (PDP)*, 2018, pp. 533–540.
[27] A. Aytekin and M. Johansson, "Exploiting serverless runtimes for large-scale optimization," in *Proc. IEEE 12th Int. Conf. Cloud Comput. (CLOUD)*, 2019, pp. 499–501.
[28] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. Workshops (ICDCSW)*, 2017, pp. 405–410.
[29] D. Crankshaw *et al.*, "The missing piece in complex analytics: Low latency, scalable model management and serving with velox," Sep. 2014, *arXiv:1409.3809*. [Online]. Available: http://arxiv.org/abs/1409.3809
[30] J. Bhogal and I. Choksi, "Handling big data using NoSQL," in *Proc. IEEE 29th Int. Conf. Adv. Inf. Netw. Appl. Workshops*, 2015, pp. 393–398.
[31] Y. Li and S. Manoharan, "A performance comparison of SQL and NoSQL databases," in *Proc. IEEE Pacific Rim Conf. Commun., Comput. Signal Process. (PACRIM)*, 2013, pp. 15–19.
[32] S. Moschoyiannis and P. J. Krause, "True concurrency ion long-running transactions for digital ecosystems," *Fundamenta Informaticae*, vol. 138, no. 4, pp. 483–514, 2015.
[33] M. P. Papazoglou *et al.*, "Service-oriented computing roadmap," in *Proc. Dagshtul Seminar Service-Oriented Comput. (SOC)*, 2006, pp. 1–29.
[34] A. R. Razavi, S. K. Moschoyiannis, and P. J. Krause, "Concurrency control and recovery management for open e-business transactions," in *Proc. Commun. Process Archit. (CPA)*. Amsterdam, The Netherlands: IOS Press, 2007, pp. 267–285.
[35] A. Razavi, A. Marinos, S. Moschoyiannis, and P. Krause, "RESTful transactions supported by the isolation theorems," in *Proc. Int. Conf. Web Eng. (ICWE)*, LNCS 5648. Berlin, Germany: Springer, 2009, pp. 394–409.
[36] A. Marinos, S. Moschoyiannis, and P. Krause, "Towards a RESTful infrastructure for digital ecosystems," in *Proc. ACM Conf. Manage. Emergent Digit. Ecosyst. (MEDES)*, 2009, pp. 340–344.
[37] A. Kobusińska and C.-H. Hsu, "Towards increasing reliability of clouds environments with RESTful Web services," *Future Gener. Comput. Syst.*, vol. 87, pp. 502–513, 2018.
[38] A. R. Razavi, S. K. Moschoyiannis, and P. J. Krause, "A scale-free business network for digital ecosystems," in *Proc. IEEE Int. Conf. Digit. Ecosyst. Technol. (DEST)*, 2008, pp. 241–246.
[39] M. R. Karlsen and S. Moschoyiannis, "Evolution of control with learning classifier systems," *Appl. Netw. Sci.*, vol. 3, no. 1, p. 30, 2018.
[40] S. Moschoyiannis *et al.*, "A Web-based tool for identifying strategic intervention points in complex systems," in *Proc. Games Synth. Complex Syst. (CASSTING ETAPS)*, EPTCS 220. Amsterdam, The Netherlands: Elsevier, 2016.
[41] M. R. Karlsen and S. Moschoyiannis, "Customer segmentation of wireless trajectory data," Dept. Comput. Sci., Univ. Surrey, Guildford, U.K., Tech. Rep. 03-12-2019, 2019. Accessed: Feb. 12, 2020. [Online]. Available: http://arxiv.org/abs/1906.08874
[42] G. Adzic and R. Chatley, "Serverless computing: Economic and architectural impact," in *Proc. 11th Joint Meeting Found. Softw. Eng. (ESEC/FSE)*, 2017, pp. 884–889.
[43] *AWS Lambda Layers, AWS Documentation 2020*. Accessed: Feb. 12, 2020. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/configuration-layers.html
[44] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "A survey of model compression and acceleration for deep neural networks," *IEEE Signal Process. Mag.*, pp. 126–136, Jan. 2018.
[45] Y. Yu, Y. Li, S. Che, N. K. Jha, and W. Zhang, "Software-defined design space exploration for an efficient AI accelerator architecture," Mar. 2019, *arXiv:1903.07676*. [Online]. Available: http://arxiv.org/abs/1903.07676
[46] 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. Red Hook, NY, USA: Curran Associates, 2018, pp. 7265–7275.



ANGELOS CHRISTIDIS has been working as a research software developer on the projects funded by the European Institute of Innovation and Technology (EIT) and U.K. research councils on implementing modern AI techniques across real-time scalable platforms. His experience in pairing serverless technologies with AI techniques has proved crucial in pushing these projects into their next stages; all covered by APIs across cloud services (AWS). His experience in planning and executing cutting edge software through a range of activities, including reaching finalist and winning stages at various hackathons, such as Facebook, Bank of Cyprus, and others, as well as through IEEE publications. His other activities include entrepreneurship and innovations camps.

SOTIRIS MOSCHOYIANNIS (Member, IEEE) is a Senior Lecturer in complex systems with the University of Surrey. He has led several U.K. and EU funded research projects. In the most recent project, Real-Time Flow with partners Ferrovial, Amey, and EMU Analytics, funded by EIT Digital, he leads the development of AI models to enhance the predictive capability, including rule-based machine learning, evolutionary computation, and deep reinforcement learning, on large datasets (train movements across the U.K. railway). He has published over 50 peer-reviewed publications, including four book chapters, and has received six best paper awards at international journals and peer-reviewed conferences. He also led the OJPA Project funded by Innovate U.K., where he developed a rule-based AI engine that is currently used by Moses Mobility Ltd., to offer shared taxi rides to stranded commuters in London. His research is on the application of computational techniques and mathematical methods to the analysis of complex networks.

Dr. Moschoyiannis serves on the IEEE Technical Committee on Industrial Informatics and is a member of the Executive Committee of the IEEE Technical Committee on Cloud Computing (TCCLD). He also serves on the Programme Committee for a number of peer-reviewed conferences, including the Complex Networks and the IEEE Conference on Service-Oriented Computing and Applications (IEEE SOCA), and he is also a Co-Chair of the RuleML+RR 14th International Rule Challenge. He is a regular Reviewer of esteemed journals such as the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING AND METHODOLOGY), the IEEE TRANSACTIONS ON NEURAL NETWORKS AND LEARNING SYSTEMS, IEEE ACCESS, the *IEEE SOFTWARE*, *Array* (Elsevier), and *Applied Network Science* (Springer).

CHING-HSIEN HSU (ROBERT) (Senior Member, IEEE) is a Chair Professor and the Dean of the College of Information and Electrical Engineering, Asia University, Taiwan. He has published 200 articles in top journals such as the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEM, the IEEE TRANSACTIONS ON SERVICES COMPUTING, ACM TOMM, the IEEE TRANSACTIONS ON CLOUD COMPUTING, the IEEE TRANSACTIONS ON EMERGING TOPICS IN COMPUTING, the IEEE SYSTEMS JOURNAL, and the *IEEE Network*, top conference proceedings, and book chapters in his research areas. He has been acting as an author/coauthor or an editor/coeditor of ten books from Elsevier, Springer, IGI Global, World Scientific, and McGraw-Hill. His research interests include high performance computing, cloud computing, parallel and distributed systems, big data analytics, and ubiquitous/pervasive computing and intelligence.

Prof. Hsu is a Fellow of the IET (IEE). Since 2008, he has been serving on the Executive Committee, IEEE Technical Committee of Scalable Computing, the IEEE Special Technical Committee on Cloud Computing, and the Taiwan Association of Cloud Computing. He was awarded talent awards six times from the Ministry of Science and Technology and the Ministry of Education, and distinguished awardd nine times for excellence in research from Chung Hua University, Taiwan. He is also the Vice Chair of the IEEE Technical Committee on Cloud Computing (TCCLD) and the IEEE Technical Committee on Scalable Computing (TCSC). He is the Editor-in-Chief of the *International Journal of Grid and High Performance Computing* and the *International Journal of Big Data Intelligence*, and serving on the Editorial Board of a number of prestigious journals, including the IEEE TRANSACTIONS ON SERVICE COMPUTING, the IEEE TRANSACTIONS ON CLOUD COMPUTING, the *International Journal of Communication Systems*, the *International Journal of Computational Science*, and AUTOSOFT Journal.

ROY DAVIES worked extensively with the mobile telecommunication business sector across a wide range of IT systems with a broad base of experience in the management and applications of data science and analytics, technical IT architecture, and system design and development. He has over 20 years of experience, including IT system development and support (across full lifecycle), IT product management, business and technical strategy, people and team management, business analysis, programme and project management, cost management, and external supplier liaison. He has flexible, enthusiastic, and pragmatic approach with the ability to understand, analyse, resolve, and explain both business and technology problems at all levels. His specialties are data science, analytics, big data, IT system development and support (across full lifecycle), enterprise and system architecture, network and service management, IT product management, business and technical strategy, people and team management, business analysis, programme and project management, cost management, and external supplier liaison.