Telepulse: A Novel Scalable Serverless Framework for Vehicle Telematics using Microsoft Azure Cloud Service

Prabavathy BALASUNDARAM*, **Vishnu K. KRISHNAN, Vrishin VIGNESHWAR , Suraj JAIN**

Department of Computer Science and Engineering Faculty
Sri Sivasubramaniya Nadar (SSN) College of Engineering, Kalavakkam, India

***Corresponding author:** Prabavathy BALASUNDARAM
prabavathyb@ssn.edu.in

Abstract: Fleet management is a series of processes that allows fleet owners to see and manage all information associated with their vehicles. Fleet management usually deals with a high volume of data being received and processed every second. This data can be the engine's fuel value, the current speed of the vehicle or the GPS location to name a few. Collectively, this type of data is called telemetry data. From this telemetry data, the fleet owners can derive timely and meaningful insights such as monitoring vehicle health, location and behaviour of anomalous drivers. These insights are essential for optimal and reliable business decision-making. The fleet management problem has been well-researched over the past decade, and most of the solutions proposed involve one or more types of ELT or ETL pipelines. They are usually purely server-based solutions. The main drawback of most of the solutions is that they increase the investment costs significantly. Therefore, this research aims to propose and build a Telepulse Framework which is cost-efficient, serverless solution for the fleet management problem with high development speed.

Keywords: Vehicle telematics, Serverless Computing, Fleet management, AWS Services, Azure Services.

1. Introduction

Fleet management encompasses the processes involved in allowing organizations or companies to operate fleets of vehicles efficiently and seamlessly. Fleet managers require precise real-time information to gain clear insights into operations and make decisions as necessary. With the growing use of sensors in vehicles, meaningful information such as GPS location, fuel tank capacity, seat belt state, distance driven, vehicle health, and speed can be collected and analyzed in real-time. This method of monitoring the vital information related to the vehicles is known as vehicle telematics. This research work is targeted towards fleet owners and vehicle manufacturers. The requirements met by this solution are essential to the targeted consumers. These include decreasing fuel costs, improving driver safety, increasing productivity, improving vehicle visibility, data-driven vehicle maintenance schedules, and precise payroll management.

Lack of vehicle telematics can lead to increased fuel costs, increased repair costs and worsening driving behaviour. It may also lead to unsafe driving and cost increases due to the requirement of physical inspection. To tackle these issues, vehicle telematics is essential. Therefore, the fleet management problem requires developing a system capable of efficient, scalable, and cost-effective data processing and pipelining. Currently, most vehicle telematics architectures utilise server-based solutions. Carstream research processes and monitors real-time data from millions of cars to maintain vehicle quality and safety. Some serverless paradigms have been studied and benchmarked recently. Various applications that find the usage of serverless computing solutions were discussed. Antreas's proposed architecture uses AWS Serverless functions called *lambda functions* for their pipeline.

This paper aims to propose a cost-effective, scalable, and efficient data pipeline with the help of Azure's serverless service called *Azure Functions*. These functions offer a more flexible solution in the context of a larger real-time workload.

The rest of this paper is organized as follows: Section 2 describes the technologies used with the proposed methodology in detail. Section 3 discusses the experimental results and analysis; Section 4 encompasses the conclusion and possible future scope.

https://doi.org/10.33436/v33i2y202302

2. Related work

Several research papers have been published over the years in the areas of stream processing and fleet management systems.

This work proposed an event-driven Extract, Transform, and Load (ETL) pipeline serverless architecture. The architecture was evaluated for its performance over a range of dataflow tasks of varying frequency, velocity, and payload size. In this work, it has been mentioned that this pipeline must be tested with the other cloud providers as they had some specific bottleneck issues using AWS. This is one of the reasons why the proposed architecture utilises Azure services over AWS. It is also to be noted that they have used a stateless model, that is, they don't persist state information across the function calls (Antreas & Georgios, 2020).

Stateful Function as a Service (FaaS) has been created with the combination of Distributed Shared Objects (DSO) and serverless computing (Barcelona et al., 2019). Current FaaS offerings are stateless functions that are involved in minimal I/O and communication. This paper presents the design and implementation of a stateful FaaS platform that provides familiar Python programming with low-latency mutable state and communication while maintaining the autoscaling benefits of serverless computing. This work does not use DSO, but instead uses a self-hosted VM that exposes an API. These papers discuss two different ways to solve the stateless problem, but clearly, these are not the only ways, but they serve as proof to reaffirm the feasibility of the idea (Sreekanti et al., 2020).

The main bottleneck in business intelligence solutions is the provision of the Extract, Transform, Load (ETL) process in near real-time. Hence, this work proposes Distributed on Demand (DoD)-ETL, a process which is a combination of a data stream pipeline with a distributed, parallel, and technology-independent architecture with in-memory caching and efficient data partitioning. This work has been compared with other stream processing frameworks used to perform near real-time ETL. From the results, it was found that DOD-ETL executes workloads up to 10 times faster. This work has been deployed in a large steelwork as a replacement for its previous ETL solution, enabling near real-time reports previously unavailable (Machado et al., 2019).

This work aims to build a GPS tracker for vehicles. Location data is stored in a cloud database, which can be used for further processing. The tracker is equipped with a web/mobile application for viewing the live results. A unique process runs for every service where it communicates through a well-defined mechanism to serve a business goal. It is called serverless because the specifications of the server are determined by the cloud. Since GPS is enabled on vehicles, it is possible to track the route of the vehicles using Google Maps with the supply of origin and destination. When a vehicle moves outside a specific geographical boundary, it is detected and intimated through e-mail (Anand et al., 2019).

It is to be noted that serverless computing comes with significant risks from the user perspective. Using serverless architecture without proper planning can potentially hinder the smooth and successful development process. It requires rigorous design and planning to ensure controllable costs as deployment scales (Castro et al., 2019).

A good example of FaaS services failing to meet expectations is mostly due to ill-borne design-driven decisions. Therefore, it is very pertinent to realise that shifting to serverless technologies comes with a convenience-over-control trade-off. Thus, one must constantly ensure in every decision made that delegating control will not impede changing business requirements in the present or the future (Hellerstein et al., 2018).

Comparison of platforms and tools for serverless computing were analysed. Various applications that find the usage of serverless computing solutions were discussed. The applications discussed are Chatbot, Information retrieval, File processing, Smart grid, Security, IoT, and Networks. This work describes various functional and non-functional challenges. It includes cost and pricing model, cold start, resource limits, security, scalability, vendor lock-in, fault tolerance, function composition, resource sharing and long-running (Hassan et al., 2021).

Fleet management processes are optimized using Fleet Telematics System (FTS). Therefore, the selection of FTS is driven by transport specifications from the customer side, leading to substantial search costs. However, FTS vary significantly in their design requirements to assist road freight operations. Existing telematics vendors elicit thirty-one Design Requirements (DRs), which are aggregated as nine Requirement Set (RS). Subsequently, forty-two practitioners from five digital road freight service enterprises experienced in using FTS validated the DRs and evaluated their importance with RS following the Analytical Hierarchy Process (AHP) method. The results reveal that DR and RS promoting driver monitoring and IT integration are perceived as more important than items promoting fleet and logistics support (Heinbach et al., 2022).

Edge computing enables low-latency Internet of Things (IoT) applications, by shifting computation from remote data centres to local devices, which are less powerful but closer to the end user's devices. However, this creates the challenge of determining how to best assign clients to edge nodes offering compute capabilities. Previously, two opposing architectures were proposed: centralised resource orchestration and distributed overlay. Uncoordinated access is proposed in this paper, which involves allowing each device to explore multiple opportunities to opportunistically embrace network heterogeneity and load conditions towards diverse edge nodes. This contribution is intended for emerging serverless IoT applications, which do not have a state on the edge nodes executing tasks (Claudio et al., 2020).

3. Proposed Telepulse architecture

The proposed Telepulse architecture shown in Figure 1, comprises five main components, namely: Data-Ingester, Transformer, Processor, Loader, and Front End.

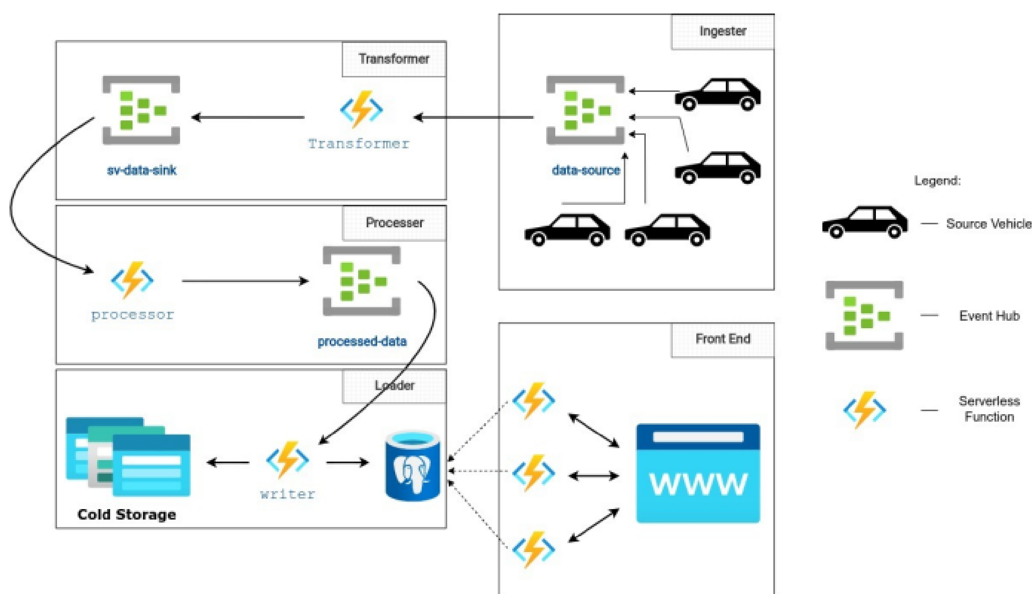

Figure 1. Design of Telepulse Architecture

The system is expected to receive unrestricted, unbounded telemetry data from several vehicles at any time. This means that the system is dealing with real-time data that can suffer from peak and nonpeak hours as in any other real-time data pipeline. The five main modules have their separate responsibilities and are connected from end to end to form the data pipeline. Firstly, in the Data-Ingester module, synthetic data is generated by the mock server to simulate real-time streaming data sources which then go into the data-source event hub. This is then connected to the Transformer module where the input data from various sources of various formats are converted into a corresponding object of a uniform standardized format. This is termed as the <SV> object. These <SV> objects are sent to the Processor module. This is the heart of the pipeline. Here, useful insights are derived from the <SV> object to generate event objects. Then a Pulse object is created containing the data from the <SV> objects along with an event queue which is where the generated event objects

are pushed into. This is then sent to the Loader module. Here, the objects received are written to the PostgreSQL table. The fifth module is the front end which accesses the data written into the PostgreSQL table.

3.1. Data-ingester

This component is responsible for the generation and ingestion of data into the pipeline. The mock server handles the data generation and streaming. It simulates the real-time streaming of data, which generates vehicle telemetry data on the fly. This data is written into the data-source event hub. The data-source event hub acts as the input to the pipeline. Data from different sources are written into different partitions of the event hub. The partition here can be thought of as a commit log which contains an ordered sequence of events.

3.2. Transformer

The transformation of various input data objects into a Standardized Vehicle (SV) data object is handled by the transformer. The data-source event hub triggers Azure Functions on receiving data. The transformer Azure Function transforms the different data objects received from various sources into Standardized Vehicle or <SV> objects. This is necessary because the data coming from different sources might have different time zones and different units of measurement as well. The transformer standardises this and creates <SV> objects. If the architecture were to be scaled up horizontally, the processor can easily handle the increase in load, since all the data objects are in a single format. The transformed <SV> object is then written into the sv-data-sink event hub. This makes the processing of data easier in the further steps.

3.3. Processor

The event hub sv-data-sink on receiving the transformed <SV> data object triggers the processor. It is a core component of the proposed architecture. It generates pulse objects which are a combination of telemetry and event data. The <SV> objects contain hidden information which is used to derive insightful data and events by the processor. The <SV> object contains only telemetry data, whereas pulse objects contain both telemetry data and event objects. On the occurrence of specific events, the processor generates event objects. For example, if the car's speed has suddenly shot up, a speeding event object is created at that particular instance. This speeding event object is then pushed into the event queue of the pulse object. The pulse object generated is then written into the processed-data event hub.

3.4. Loader

The Loader component is delegated with the task of writing the received pulse objects to the PostgreSQL database. This is carried out when the Processed-data event hub receives the pulse object thereby triggering the writer Azure Function. This function writes all the objects created by the processor into a PostgreSQL database and Azure Blob storage. The PostgreSQL database data is used by the front-end. The blob storage acts as cold storage for the entire system. It makes the system more reliable and helps in validation. This data can also be used for analytics in the future.

3.5. Front end

The front-end triggers API Azure functions through REST API calls. This function reads the corresponding data from the PostgreSQL table and sends it to the front-end. The front-end then renders the new changes in the dashboard. The front-end website was developed using Next.js, an open-source web development framework with Material-UI which is a React component library.

The development of fleet management applications with serverless computing involves lower costs, greater scalability, and faster entry into the market. Though these are the advantages of serverless computing, there are certain pitfalls to its utilisation. They are as follows:

- Loss of Control over the Software Stack;
- Security of the entire application;
- Architectural Complexity;
- Lack of Testing;
- Lack of Monitoring.

The following discusses the pitfalls with the methods to resolve them.

In a server-based solution, every functionality, like queues, databases, and authentication systems, will be designed by the developer. Hence, there will be more control over the software stack. However, to retain control, less significant functions alone can be delegated to third-party control. With this way of tuning, it is possible to spend more time and energy building the application, which can have lots of business significance.

Applications may require several functions. If permissions are not properly given for the functions, there will be possibilities for security breaches. To avoid this, all the functions of the application need to be given proper permission, in addition to the transfer of encrypted data for outside communication from the cloud.

The development of applications using serverless computing involves more about how the individual service functions are configured than the complexity of the code in them. This necessitates the program developers to follow a solid architectural pattern to design the application. Usually, architects think of synchronous communication. However, the communications will be asynchronous. It needs to be accommodated in the design.

The very nature of distributed serverless applications leads to challenges in testing the application. In this context, it is also necessary to perform integration testing in addition to unit testing. It is important that this testing be successful only when it is architected properly.

Like testing, it is more challenging to monitor a distributed application. It is best to adapt to advanced tools like *Lambda* and *ELK Stack* to visualise console logs and metrics.

Hence, before the development of an application, it is very important to analyse the critical parts of the business application, the budget required, and the right cloud technology to be used. This ensures the quick development of applications that can enter the market soon.

4. Results and discussion

This section describes the implementation and performance of the proposed system.

4.1. Implementation

The majority of the Telepulse architecture was implemented on the cloud using Azure Cloud Services. The *Azure for students* free credits were used for the implementation of the pipeline. The *mock server* alone was implemented locally. The core components used on Azure Cloud Services include Azure Event Hubs, Azure Functions and PostgreSQL tables.

The data generator is the *mock server* which handles the generation of values such as latitude, longitude and speed of the vehicle. This was carried out using the random function present in the Math library of JavaScript. Then this was run on a local machine to act as a source for the data pipeline.

The other four modules were deployed onto the cloud using Azure Functions. The *transformer component* first identifies the input format and then sends it to the appropriate transformation handler. Here the object goes through a JSON parse to extract the data, after which the <SV> object is created. It then sends it to the correct partition of the destination event hub using the connection string and the partition key.

The *processor component* handles the event generation and creation of the *pulse* object. This is carried out with the help of JSON parsing, after which appropriate operations are carried out based on the events generated. Then the Processor sends the *pulse* objects to the *processed-data* event hub.

The *writer component* is responsible for the writing of the received *pulse* objects into the PostgreSQL table. This is carried out with the use of the *helpers functions* from the pg-promise module. The front end then retrieves the data using API Azure functions through REST API calls. The NextJs framework is used for the website and is built upon the NodeJs framework.

4.2. Performance metrics

The performance of the proposed Telepulse architecture was measured with the performance measures namely, *execution time* and *cost*. These are critical factors that should be considered when migrating to server-less configurations.

Execution time is a very important parameter as it provides insight into the overall performance of the system. The execution time of the end-to-end pipeline along with the execution time of the individual functions namely *Transformer*, *Processor* and *Writer* are recorded. This is carried out to identify potential bottlenecks and help address them.

Cost is another crucial factor which decides the feasibility of the system. The cost of the systems depends on three factors. They are execution duration, memory usage and execution count. The cost of an azure function depends upon which category the three factors come under. For example, if the Azure function falls below 400,000 GB/s of execution and 1,000,000 executions then it will come under the free tier.

4.3. Experimental results

Multiple experiments have been conducted to test the performance of the proposed architecture, based on the above-mentioned performance metrics.

4.3.1. Impact of Telepulse on the individual execution time of Azure function

Objective: To analyse the execution times of each function and reduce the execution times of the functions with higher execution times.

The proposed Telepulse architecture was tested to measure the performance of each individual Azure functions namely, *Transformer, Processor* and *Writer*. Experiments were carried out using three different input load sizes of 1MB, 5MB and 10MB sent in every 2 seconds. Execution time was measured by subtracting the time recorded when the first data object enters the Azure function and the time recorded when the last data object exits the Azure function. This experiment was carried out 10 times and the average execution times of each function were recorded. In carrying out the experiment, Figure 2 is plotted from observations recorded. This experiment is necessary because the execution time of each function affects the overall performance and cost of the system.

While monitoring the execution times of each Azure function, there was a large difference in the total observed execution times between the writer function and both the processor and transformer functions. After investigations, it was found that the cause for this was due to the large number of network calls being made. To reduce this, a bulk update was considered. It allows the writer function to make multiple writes in just one network call while keeping the number of executions the same. With bulk update, the number of network calls or connections being made could be reduced significantly, which could greatly improve the performance of the writer function. After the implementation of bulk update, Figure 3 is plotted from the observations recorded.

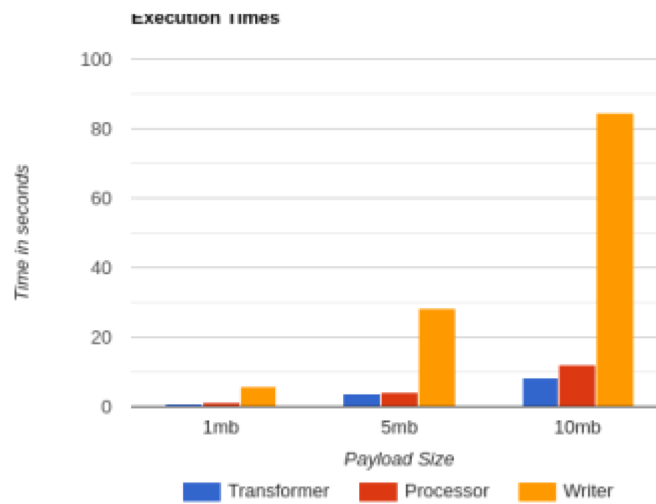

Figure 2. Execution time for various functions of Proposed Telepulse Architecture (without bulk update)

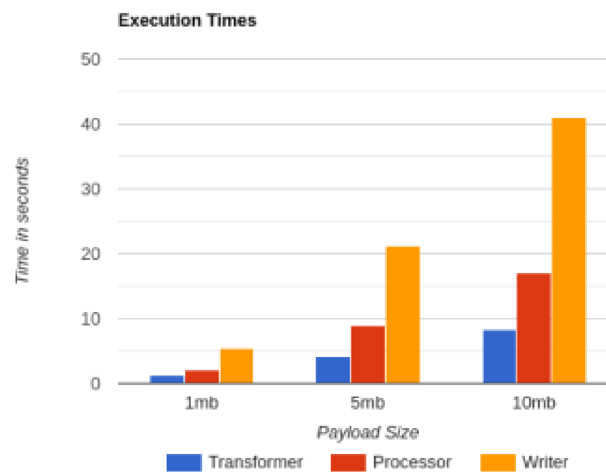

Figure 3. Execution time for various functions of Proposed Telepulse Architecture (with bulk update)

From Figures 2 and 3, it is clear that the *writer function* in the first Figure takes longer to execute than the *writer function* in the second Figure. Before bulk update, the *writer function* would need to make a network call whenever it needs to write an object into the PostgreSQL table. After the implementation of bulk update, the number of network calls that needed to be made was reduced significantly, which saved a large amount of time that used to go into making network calls with the PostgreSQL table.

4.3.2. Impact of Telepulse on end-to-end execution time

Objective: To analyse the performance of Telepulse architecture as a complete data pipeline.

The entire system was tested with 1MB, 5MB, and 10MB payload sizes as input to the data-source event hub sent every 2 seconds. The end-to-end execution time from the data generated by the mock server to the writing of the pulse object to the PostgreSQL table was recorded. This execution time was measured by subtracting the time recorded when the first generated data object is sent to the *data-source* event hub and the time recorded when the last data object is written into the PostgreSQL table. This experiment was carried out 10 times and the average end-to-end execution times were recorded.

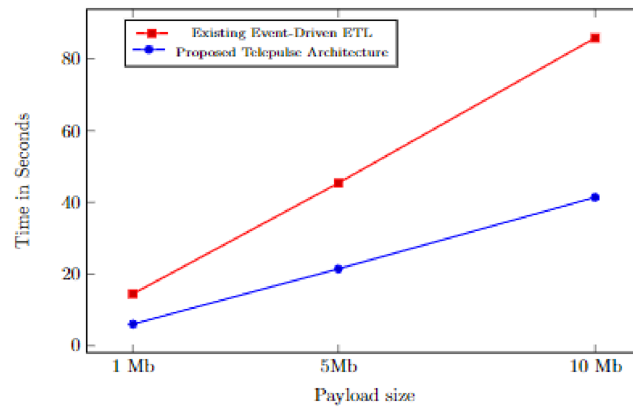

Figure 4. Total execution time of Proposed Telepulse Architecture with Existing Event-driven ETL

Figure 4 illustrates the end-to-end times from the source to the PostgreSQL table for different payload sizes. For all the payload sizes considered, the Azure model which is the Telepulse architecture with Azure model performs significantly better than the existing Event-driven ETL with AWS model (Antreas & Georgios, 2020). The execution time of the Telepulse architecture with Azure model is more than double the time faster when compared to that of the existing Event-driven ETL with AWS model.

4.3.3. Impact of Telepulse on cost

Objective: To conduct cost analysis on the proposed Telepulse architecture.

Estimating the costs of the system is crucial in determining the advantages and disadvantages of the system configuration proposed. The cost for the usage of Telepulse Architecture is calculated for a year.

The process of calculating the cost for an Azure function is as follows:

- Number of times a function is invoked in a month;
- Execution time for an individual function;
- Total execution time for the function, which can be found by multiplying the *count x execution time;*
- Resource consumption for the function in a month.

The example cost calculation for the Transformer Azure function for 10MB data is as follows:

- Number of times the Transformer Azure function is invoked is measured at 46. This is invoked every 2 seconds for 10 hours a day and is calculated to be 46 X 30 X 60 X 10 X 30 - 25 Million;
- Execution time for the function was calculated to be 0.1 seconds;
- Total execution time for the Transformer Azure function is 25 Million x 0.1 seconds = 2.5 Million Sec.;
- Memory usage of Azure function to be under 256 MB was observed from the cloud dashboard. Hence, the resource consumption was found to be 2.5 Million Sec. X 1/4 = 0.625 Mil GB Sec.

Similarly, total resource consumption is calculated for the other two Azure functions and the sum is taken. This sum is then subtracted with the monthly free grant of 0.4GB Sec. This is multiplied by the tariff of 0.000016/GB Sec. Which comes around to $11.2 for a month and $135 for a year.

On average, the price of server-based solutions prices is estimated to be 70-100$ a month, which totals 840-1200$ per year. From Table1, it is evident that the proposed architecture is significantly cheaper compared to the existing server-based architecture.

Table 1 Cost for the Telepulse Architecture with Existing Event-driven ETL model

Cost for Telepulse Architecture with Azure model	Cost for Existing Event-driven ETL with AWS model
$135	840-1200$

5. Conclusion

The Telepulse architecture has been proposed to handle real-time telemetry data. This server-less architecture has been implemented on the Cloud using Azure services such as Azure event hubs and Azure functions. Azure's serverless functions have been used to perform all the data processing, from getting the input to displaying it to the end user. This research has proved that the proposed Telepulse architecture has shown nearly 83% reduction in costs as compared to server-based systems. It has also shown a 51% reduction in the execution time of the *writer* function after bulk update was implemented. In addition, approximately 50% improved total execution time was recorded when compared to similar research done in AWS (Antreas & Georgios, 2020).

This has also demonstrated that using server-less functions speeds up the development, because server-less solutions only require their own intended logic to be up and running, and everything else, such as network calls, upscaling, and security, is handled behind the scenes. This means that there is very little maintenance overhead to manage the architecture. Nevertheless, discussed below are the limitations of the research and ways to overcome the same.

Future work

The Azure functions in the proposed Telepulse architecture support only 20 minutes of maximum running time per function instance. If the app was scaled up to greater than 10 GB of data being processed in batches per second, and if maximum number of function invocations at any instant was limited, then each function invocation may take more than 20 minutes to finish executing fully. If the function dies in between execution after 20 minutes, then the data read is dropped without being processed or written into subsequent event hubs which could result in data loss. To avoid this, function orchestration should be implemented. Azure's offering for what is synonymous to function orchestration is called durable functions.

The front-end polls the back-end API serverless functions periodically. This number is set to 5 secs currently to mimic real-time processing in the front end. However, this is not scalable and leads to wasted network calls. This can be avoided by utilizing live websockets between the front end and the back end. Socket.IO is an event-driven JavaScript library for real-time web applications and is built exactly to solve this problem.

REFERENCES

Anand, S., Johnson, A., Mathikshara, P. & Karthik, R. (2019) Real-time GPS tracking using serverless architecture and ARM processor. In: *Proceedings 2019 11th International Conference on Communication Systems & Networks (COMSNETS), January 07-11, 2019, Bengalore, India.* IEEE. pp. 541-543.

Antreas, P. & Georgios, S. (2020) An Event-Driven Serverless ETL Pipeline on AWS. *Applied Sciences.* 11(1), 1-13. doi:10.3390/app11010191.

Barcelona-Pons, D., Sánchez-Artigas, M., París, G., Sutra, P. & García-López, P. (2019) On the faas track: Building stateful distributed applications with serverless architectures. In: *Proceedings of the 20th International Middleware Conference*. pp. 41–54. doi:10.1145/3361525.3361535.

Castro, P., Ishakian, V., Muthusamy, V. & Slominski, A. (2019) The server is dead, long live the server: Rise of Serverless Computing, Overview of Current State and Future Trends in Research and Industry. [Preprint] https://doi.org/10.48550/arXiv.1906.02888 [Accessed: 7th June 2019].

Claudio, C., Marco, C. & Andrea, P. (2020) Uncoordinated access to serverless computing in MEC systems for IoT. *Computer Networks*. 172, 107184. doi:10.1016/j.comnet.2020.107184.

Hassan, H. B., Barakat, S. A. & Sarhan, Q. I. (2021) Survey on serverless computing. *Journal of Cloud Computing.* 10(29), 1-29. doi:10.1186/s13677-021-00253-7.

Heinbach, C., Kammler, F. & Thomas, O. (2022) Exploring Design Requirements of Fleet Telematics Systems Supporting Road Freight Transportation: A Digital Service Side Perspective. In: *Proceedings of 17th International Conference on Wirtschaftsinformatik (WI) 22, 21-23 February, 2022, Nürnberg*. pp. 1-15. https://www.researchgate.net/publication/358467406_Exploring_Design_Requirements_of_Fleet_Telematics_Systems_Supporting_Road_Freight_Transportation_A_Digital_Service_Side_Perspective.

Hellerstein, J. M., Faleiro, J. M., Gonzalez, J., Schleier-Smith, J., Sreekanti, V., Tumanov, A. & Wu, C. (2019) Serverless Computing: One Step Forward, Two Steps Back. In: *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings, 2019.* doi:10.48550/arXiv.1906.02888.

Machado, G. V., Cunha. Í., Pereira, A. C. M., Oliveira, L. B. (2019) DOD-ETL: distributed on-demand ETL for near real-time business intelligence. *Journal of Internet Services and Applications.* 10 (21), 1-15. doi:10.1186/s13174-019-0121-z.

Sreekanti, V., Wu, C., Lin, X. C., Schleier-Smith, J., Gonzalez, J. E., Hellerstein, J. M., Tumanov, A. (2020) Cloudburst: Stateful functions-as-a-service. In: *Proceedings of the VLDB Endowment*. 13(12), 2438-2452. doi:10.14778/3407790.3407836.

Prabavathy BALASUNDARAM is currently associate professor in the Department of Computer Science and Engineering at Sri Sivasubramaniya Nadar College of Engineering (SSNCE). She has a total of 25 years of experience in teaching. She received her Bachelor's Degree from Thiagarajar College of Engineering from Madurai Kamaraj University, her Master's Degree from NIT Trichy and her Ph.D. from Anna University, Chennai, Tamil Nadu, India. She has 14 years of research experience in guiding UG, PG and Ph.D. students in Cloud computing, Distributed computing, Hadoop Ecosystem, Big Data Analytics and Image processing using Machine Learning and Deep Learning Techniques. She won the Best Teacher Award for the year 2013-2014 and was the second winner in the inaugural Teaching Video Challenge organized by the ACM India iSIGCSE (Special Interest Group on Computing Science Education).

Vishnu K. KRISHNAN is currently pursuing a master's in computer science at the Georgia Institute of Technology. He completed his undergraduate degree in Computer Science and Engineering at SSN College of Engineering. His areas of research interest include Big Data, Data Science, and Machine Learning and has worked on various projects related to them. Currently, he works part-time and studies as a graduate teaching assistant and enjoys mentoring his peers. He aspires to implement and lead projects that deliver complex, large-scale solutions that provide business value and improve people's quality of life.

Vrishin VIGNESHWAR is an aspiring software developer, leader, influencer, and builder currently working at Motorq, a tech startup. He has a proven track record of academic excellence, actively participating in college competitions, online contests and completing certifications. His passions lies in the intersection of tech and building world-class teams, and he has a knack for mentoring and inspiring others.

During his college days, he articipated in over 50 college competitions, completed numerous certifications, and consistently ranked within the top 5% academically. He also played a pivotal role in transforming his college coding club into a world-class task force, accomplishing unprecedented feats with his team within just one year.

Currently, he is known for his super energetic and highly inquisitive nature, coupled with a versatile yet sophisticated skill set in tech. Excluding his professional ventures, he constantly dabbles in various entrepreneurial ventures inside and outside of work, always seeking new challenges and opportunities for growth.

Suraj JAIN is a 1st-year master's student at the University of Massachusetts Amherst, pursuing Computer Science. He completed his undergraduate degree in Computer Science and Engineering at SSN College of Engineering. He is currently working as a graduate teaching assistant, helping students learn algorithms. He is interested in algorithms, machine learning and data science and is also passionate about quantitative trading. He is motivated by the potential for technology to enhance people's lives, and he believes that software development provides a unique opportunity to create tools that can make a tangible difference. He is passionate about exploring new technologies and programming languages, and he is constantly seeking ways to improve his skills and knowledge in the field.