National College of Ireland

Latency Assessment on Inclusion of a FEC Orchestrator Capable of Invoking Serverless Functions

MSc Research Project
MSc in Cloud Computing

David Tynan
Student ID: 20153104

School of Computing
National College of Ireland

Supervisor: Shivani Jaswal

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: David Tynan

Student ID: 20153104

Programme: MSc in Cloud Computing **Year:** 2022

Module: MSc Research Project

Supervisor: Shivani Jaswal

Submission Due Date: 15/08/2022

Project Title: Latency Assessment on Inclusion of a FEC Orchestrator Capable of Invoking Serverless Functions

Word Count: 5863 **Page Count:** 17

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: David Tynan

Date: 15/08/2022

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Latency Assessment on Inclusion of a FEC Orchestrator Capable of Invoking Serverless Functions

David Tynan
20153104

Abstract

Fog and Edge Computing (FEC) is a computing paradigm in which traditionally cloud provisioned resources are moved to distributed fog node devices in closer proximity to user equipment (UE) devices. Whilst UE fog node resource usage can result in lower latencies due to a reduction of required network hopping communications, resources are more limited and less available than cloud provisioned resources. Competition, amongst multiple UE devices, for simultaneous use of a single fog nodes resources could potentially eliminate latency benefits associated with the reduction in network hopping communications. Cases may exist in which UE devices incur less latencies for requests which use cloud provisioned resources, than waiting for local fog node resource availability. In acknowledgement to occurrences in which requests vary in criticality, this paper proposes implementation of an FEC environment orchestrator component capable of request classification and the ability to invoke serverless functions for non-time critical classified requests. Amazon Web Services (AWS) Lambda functions have been integrated, for execution of non-time critical classified requests, allowing only time critical classified requests to have abilities for fog node resource usage. Results indicate that, when implemented using the methodology addressed within this specific paper, serverless function integration is detrimental to latency performance.

1 Introduction

Cloud computing provides the ability to provision and use resources that can be more powerful than found on local UE devices, enabling users to avail of increased capabilities than if restricted by local device resource limitations. Whilst the availability of more powerful resources is beneficial for providing increased capabilities, cloud computing resource requests have longer communication distances than local device resource usage requests. Communications between UE devices and cloud provisioned resources require network hopping, incurring increased latencies than sole usage of local UE resources. Network hopping latencies may be acceptable for some use cases but would be detrimental for time critical requests requiring close to real-time communications.

FEC is a computing paradigm developed to address communication latencies, between UE devices and cloud provisioned resources, by moving traditionally cloud provisioned resources to distributed fog nodes in closer proximities of UE devices [1]. FEC environments are comprised of UE devices which communicate with resource provisioning fog nodes

capable of allocating local resources for UE functionality. Compared to cloud provisioned resources, fog node resource usage reduces the amount of required network hops for communications, but resources are more constrained which could cause resource availability issues. Competition for simultaneous use of a single fog nodes resources amongst multiple UE device requests may potentially eliminate latency benefits associated with FEC, creating cases in which cloud provisioned resource usage may incur less latency than waiting for fog node resource availability. Potential FEC resource availability latencies may be acceptable for some use cases but be detrimental for time-critical requests.

Through the implementation of a FEC environment simulation, with the context of a hospital monitoring its patients using heart rate, blood pressure, oxygen saturation and respiratory rate sensors, this paper aims to assess if inclusion of an environment orchestrator component, capable of invoking serverless functions for requests it identifies as non-time critical, can reduce environment latency. Serverless functions provide users the ability to provision and assign cloud resources for the runtime duration of an invoked function. Use of serverless functions for non-time critical requests may increase fog node resource availability for time critical requests, with potential environment latency reductions.

Serverless functions require dependencies, specific to each, to be initialised within purpose-built environments for successful execution. The process of building serverless function environments can be classified as either a cold or warm start. Cold starts relate to the build of completely new environments and are more time consuming than warm starts, which reuse suitable environments stored in memory from prior builds [2]. To minimise serverless function environment build latencies, the implemented simulation integrates serverless function auto scaling policies to build environments prior to their use.

The European Telecommunications Standards Institute (ETSI) Multi Access Edge Computing (MEC) framework, a framework for the efficient running of UE consumable applications in a multi-access network, already includes an MEC Orchestrator (MEO) component which differs in functionality to the proposed FEC orchestrator. MEO components are responsible for information monitoring of environment fog nodes, resources, services and topologies and can select specific fog nodes for UE requested application hosting [3], however, they do not classify request priorities or invoke serverless functions.

The major contribution of this research is the implementation of a Java developed FEC environment simulation consisting of sensors, actuators and data processing fog nodes which host orchestrator application modules capable of either invoking AWS Lambda functions or using local node resources, determined by request priority classification type. All Lambda functions have been implemented with the ability to publish sensor data to AWS Simple Notification Service (SNS) topics to which environment endpoints are subscribed.

Limitations caused by the unavailability of a physical device topology have required assumptions to be made for fog device uplink bandwidth, downlink bandwidth, busy power consumption, idle power consumption and associated cost usage specification values. Use of simulated fog devices has also resulted in the inability to subscribe simulated devices to AWS Simple Notification Service (SNS) topics, responsible for publishing messages relating to non-critical classified sensor transmit data to device endpoints, preventing SNS message delivery status metrics to be established and aggregated to the local environment usage execution time. Uncertainties in actual application resource consumption demands has

required specification assumptions to be applied to application module RAM and CPU requirements, in addition to module communications, which could potentially result in scenarios which consist of specification assumptions which are realistically inaccurate.

This research project has been undertaken to provide an answer to the research question of:

Can FEC environment latency be reduced upon inclusion of an orchestrator component capable of invoking serverless functions for non-time critical identified requests?

The rest of this paper is organised as follows: Section 2 addresses prior related work. Section 3 outlines the undertaken research, design and implementation methodology. Section 4 provides an evaluation of results identified by this paper. Section 5 provides a conclusion and outlines any future work that can be taken.

2 Related Work

Prior related work concerning MEC architecture, MEC environment request classification, MEO innovation, MEC environment lightweight virtualisation, serverless MEC integration, serverless function cold start time reduction and data transfer amongst serverless functions is addressed throughout this section.

As outlined in the ETSI MEC architecture [3], a user application lifecycle management (LCM) proxy component is the MEC environment entry point for UE device requests and determines the pre-existence of suitable environments, containing required dependencies, for successful execution of each received request. If a suitable environment does not exist, the LCM proxy instructs an operations support system (OSS) component to build a suitable environment which is deployed on resources assigned by the MEO. The MEO also ensures identification and inclusion of request required dependencies for environment builds. If the LCM proxy identifies a suitable pre-existing environment, for successful execution of a request, it establishes direct communication with the MEO which locates and directs the request to the environment, bypassing the OSS. Component responsibilities for MEC environment request lifecycles are outlined in the ETSI MEC architecture but the notion of varying request criticalities is not addressed.

M^2EC [4] is an orchestration solution in which MEC resource users are classified as either basic, premium or gold, determined by user service level agreements. User capabilities and delay guarantees differ for each classification type. A broker component, capable of determining user classification types, is integrated into the MEC environment either by adjusting existing environment components or as a separate broker component. The broker is used to provide loose delay guarantees to basic users, stringent delay guarantees to premium users and low delay guarantees to gold users. Gold users also have application deployment privileges, host resource specification capabilities and the ability to communicate directly with the MEO. Results obtained in M^2EC usage identified a 40 percent reduction in MEC host resource utilization, however, it is limited to using only user origin for request classifications.

Proposed in [5] is a serverless edge computing methodology in which MEC hosts run local serverless frameworks, consisting of serverless function execution resources and an on-demand application activation platform. Self-adjusting MEO tables, containing mappings for UE and function identifiers to application service endpoints, are integrated to allow available application listings to be provided to UE devices. Upon receiving a requested listing of available applications, UE devices select a listed application, creating a context containing an identifier and uniform resource identifier for application access. Analysis results, on this methodology's integration, outline differences between static, centralized and distributed serverless function host assignments, but exclusion of the methodology is not considered to form a basis of comparison. Serverless functions are stored amongst MEC hosts, which could result in latencies relating to host memory issues, however, use of cloud provisioned serverless functions are not considered for non-time critical requests to alleviate host memory usage.

Proposed in [6] is a latency aware proximity zone defining quality of service MEC environment enhancement, as a means of reducing latencies in an applications consumption of services residing in locations outside its host device. The MEO asynchronously gathers and classifies proximity measurements between host devices, enabling host groupings, determined by host service consumption latencies, which are stored in MEO tables to provide statistical performance information metrics. Application migration to a host in closer proximity of the requested service can be recommended by the MEO upon identification of detrimental service consumption latencies. Analysis on the inclusion of latency aware proximity zones identified reduced latencies and host processing times, when compared with their exclusion, however, integration is only concerned with local environment services without investigating the potential of also integrating serverless functionalities.

Investigated in [7] is whether virtual machine (VM), container and unikernel lightweight virtualisation technologies can be exploited to benefit FEC application environment builds. An acknowledgement is established that use of virtualisation for real-time or mission critical tasks, demanding low and predictable latencies, may not be suitable due to additional delays and resource utilization associated with it. Docker container usage is identified as resulting in less resource consumption than VM/unikernel use which require non-negligible hyper virtualisation overheads of operation system installation and virtual hardware device/driver emulation. Whilst use of various lightweight virtualisation technologies is assessed in FEC, serverless functionality integration usage is not investigated.

Defuse [2] is a dependency guided function scheduler developed to reduce serverless function cold start times by identifying dependent subsequent functions and building suitable environments for their execution prior to their invoke. Function dependencies are classified as either strong or weak, with strong dependencies relating to predictable functions, identified through frequent pattern mining and weak dependencies relating to unpredictable functions, identified through use of positive point-wise mutual information values relative to the invoke of subsequent strong functions. An initial mining stage establishes a graph of function dependencies, determined by invocation histories, which is used to generate dependency sets prior to a scheduling stage at which all functions within a set are invoked. Analysis of Defuse identified a 35 percent reduction in cold start rates and a 22 percent memory usage reduction in its use.

Function Fusion [8] is a solution developed to reduce cold start times for parallel functions, through use of a mediator function responsible for encapsulating and coordinating nested functions. It was developed in response to the identification that nested sequential parallel functions can still increase workflow response times, even with cold start reductions associated with subsequent nested functions. Analysis on use of Function Fusion identified decreased workflow response times for both cold and warm starts, however, use cases outside the scope of artificial intelligence are not considered.

In response to an identification of runtime start-up (RTS) and application initialization serverless function phases having the most impact on function start-up times, a prototype is developed in [9] to reduce cold start times through use of the Checkpoint/Restore in Userspace (CRIU) Linux tool. CRIU is used to capture snapshots of previously executed function processes, with the purpose of allowing their restoration for newly invoked functions. Deployment, invocation, scaling, information gathering and metric retrieval of serverless function instances is conducted through use of application programming interfaces (APIs) accessible via a gateway API. Prototype analysis identified 0ms RTS times with improvements of up to 71 percent for serverless function runtime environment build and loading times in its use. Functions containing more complex and larger amounts of code were identified as achieving greater speed up times using the prototype.

Comparisons are made in [10] between commercial data flow tool (DFT), object storage service based and MQTT based serverless data pipelines, containing serverless function pipeline tasks invoked upon movement of pipeline data, in a FEC context. Commercial DFT usage consisted of Apache Minifi located in close proximity to UE devices to receive and pre-process their data prior to transferring it to an Apache NiFi service, responsible for data processing and flow management. Object storage service pipeline usage consisted of serverless function triggering by both gateway nodes and MiniIO storage bucket events. MQTT pipeline usage consisted of serverless function triggering upon message publishing into specific MQTT topics. Comparison results identified commercial DFT usage performed best for large bandwidth demanding applications, object storage service usage performed best for bandwidth and compute intensive applications and MQTT usage was best for non-bandwidth sensitive compute intensive applications.

Fog Function [11], a data centric fog function programming model and context-driven orchestration runtime system, was proposed as a solution to data management issues amongst separate serverless functions. The orchestration system leverages data contexts relating to either UE device data or functions, system contexts relating to fog node resource availability and usage contexts relating to how functions should be executed. Function arguments can accept priority attributes to determine the most suitable resources to use. All fog nodes share and communicate with a common management component, containing an orchestrator and centralized system data indexer, to form a context management layer enabling generation of a global system view of system components and functions. The orchestrator subscribes to each function registered to the system data indexer, allowing access of all fog nodes to the function. Use of Fog Function identified a 30 percent reduction in service latency when compared with its exclusion.

3 Research, Design & Implementation Methodology

The Eclipse Integrated Development Environment (IDE) was used to develop a Java Apache Maven based FEC environment simulation consisting of sensors, actuators, Raspberry Pi devices and application modules with AWS software development kit (SDK) integration for cloud communication. As illustrated in figure 1, the simulations system architecture consists of both FEC and serverless environments.

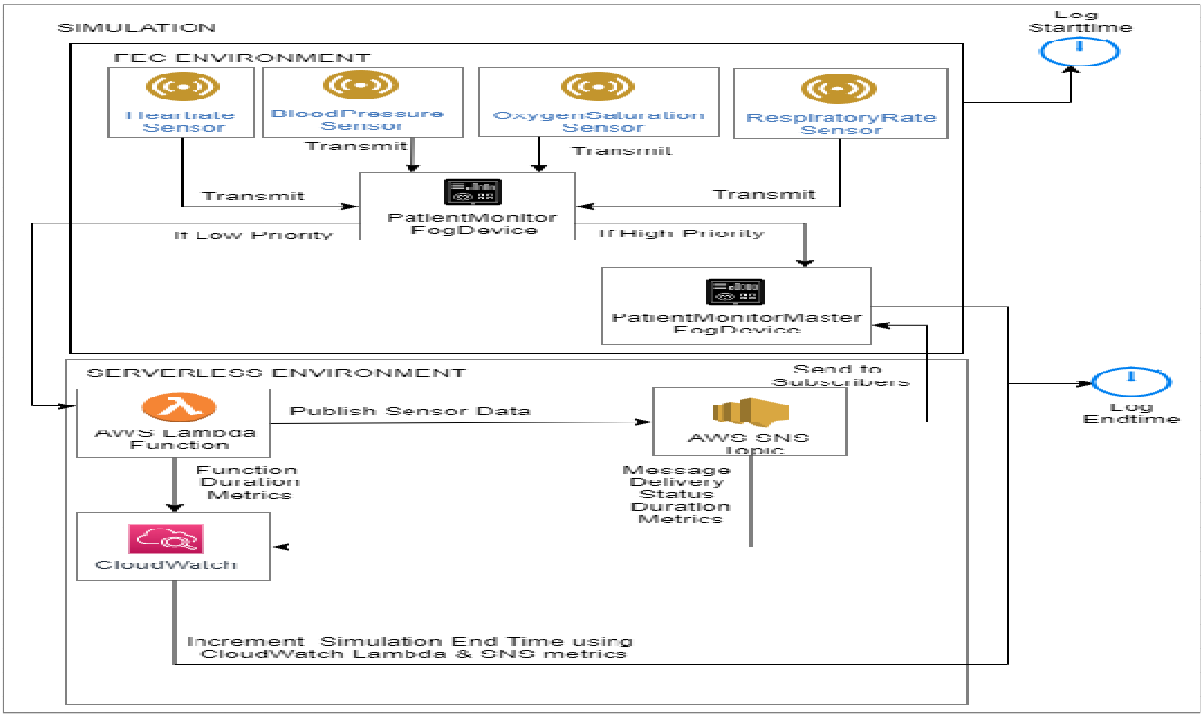

Figure 1: System Architecture

Physical, logical and management components from the iFogSim2 Toolkit, a toolkit providing FEC environment resource management technique modelling and simulation capabilities [12], have been integrated and customised for this projects source code. To provide transparency, in use of the projects serverless environment resources, the AWS serverless application model (SAM) framework has been integrated. SAM integration has enabled local environment development of AWS lambda function code with resource provisioning requirements declared in a YAML formatted template text file [13], located in the project resources directory. In addition to providing AWS CloudFormation details of resources to provision, the YAML file also includes function auto scale provisioning instructions, allowing a function usage tracking scaling policy to invoke concurrent function executions when applicable. An AWS SDK Maven bill of materials (BOM) module is included as a project dependency listed in the project object model (POM) file, allowing inclusion of specified AWS service SDK dependencies only, rather than the entire AWS SDK.

The device topology, illustrated in Figure 2, provides a small-scale device hierarchy level interactivity overview, in which there are three hospital wings with each wing containing two patients, however, evaluations have been conducted for simulations consisting of five wings, with each containing ten patients, and a single wing containing ten patients. Whilst cloud and

proxy server devices have been implemented using device specification values provided by the default iFogSim2 source code [14], patient monitor and patient monitor master devices have been implemented with RAM and CPU clock speed specification values found in Raspberry Pi3 Model B+ devices [15]. All simulated fog devices, sensors and actuators are instantiated and stored in array lists, responsible for storage of specific component types, with each component having an identifier reference to other relative components for communication.

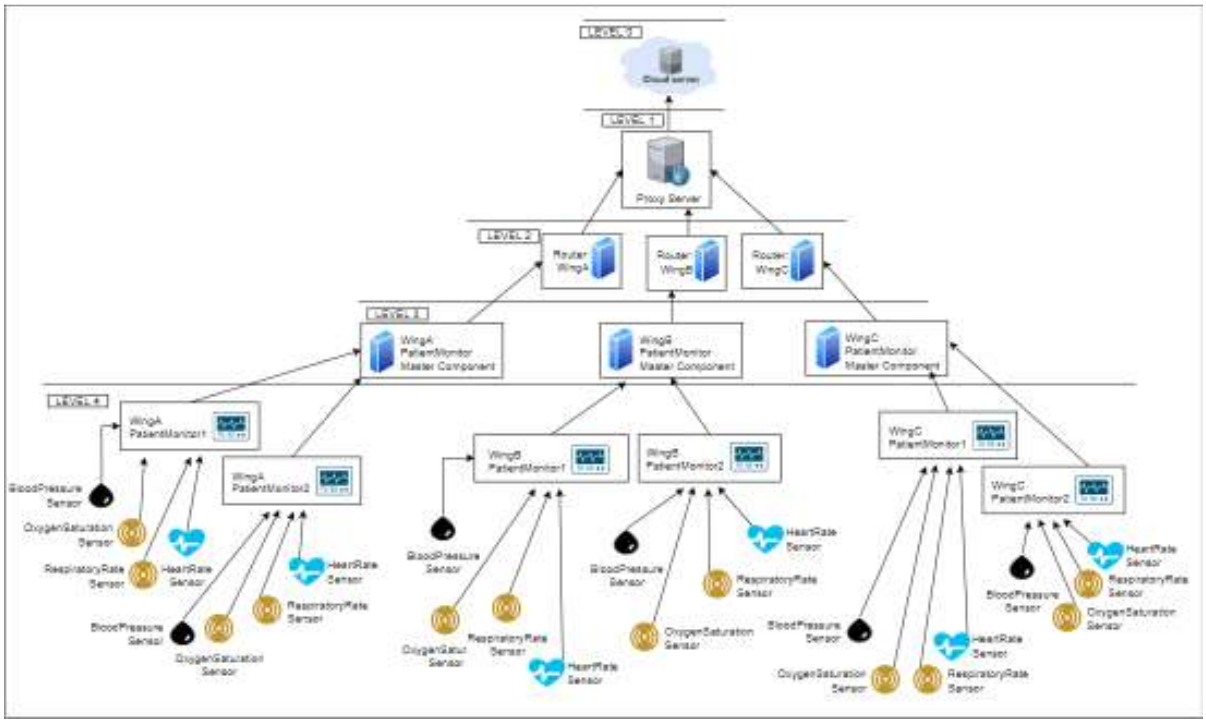

Figure 2: Device Topology

Each patient monitor device is assigned to an individual patient and receives values transmitted by heart rate, oxygen saturation levels, respiratory rate and blood pressure monitoring sensors. Patient monitor master devices are responsible for collection and display of all patient sensor information generated within their assigned wing. The simulation event flow is outlined below:

I. A patient's heart rate, blood pressure, oxygen saturation and respiratory rate sensors transmit their independent values to a patient monitor fog device specifically assigned to them.
II. The patient monitor fog device assesses each received sensor value and assigns a priority value determined by the sensor value.
III. If the priority value is classified as high priority, the patient monitor fog device sends data, relative to the sensor, to the patient master fog device, responsible for collecting sensor data from all patients within a hospital wing to be viewed on a display actuator device, and the event flow finishes at this stage with the duration of time between the sensor transmit and actuator display logged. If the priority value is classified as low priority the serverless environment is utilised, as detailed in the following stages.

IV. An AWS lambda function is invoked which takes in the sensor data as an input.

V. The AWS lambda function publishes the sensor data, as a message with identifiable attributes, to an AWS SNS topic.

VI. After execution, the AWS Lambda functions metrics are sent to Amazon CloudWatch.

VII. All patient monitor fog devices subscribed to the SNS topic to which the message is published receive sensor data correlating to subscription filters relative to their specific hospital wing.

VIII. The SNS message delivery time metric value is sent to CloudWatch.

IX. A CloudWatch function is invoked to retrieve metric statistics for CloudWatch received duration data for all Lambda executions and SNS delivery times from the simulations start time to its end time.

X. Lambda duration metrics and SNS delivery times are summed and aggregated to the local resource usage execution time from stage III, finishing the event flow.

With the aid of a sequence diagram, figure 3 illustrates events which occur as a sensor emits its registered value.

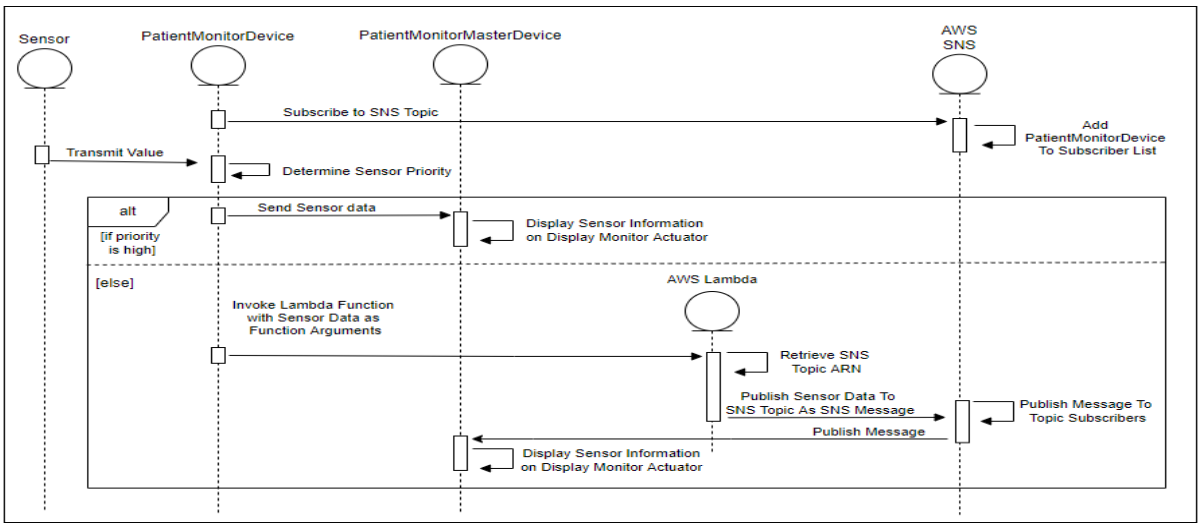

Figure 3: Sequence Diagram

iFogSim2 source code has been configured to overcome its limitations in transmitting actual sensor values, executing actions determined by sensor values and lack of serverless function integration. A class diagram, for the implemented project, is illustrated in figure 4 with an assumption of each Java class also having undocumented getter and setter methods for their declared variables.

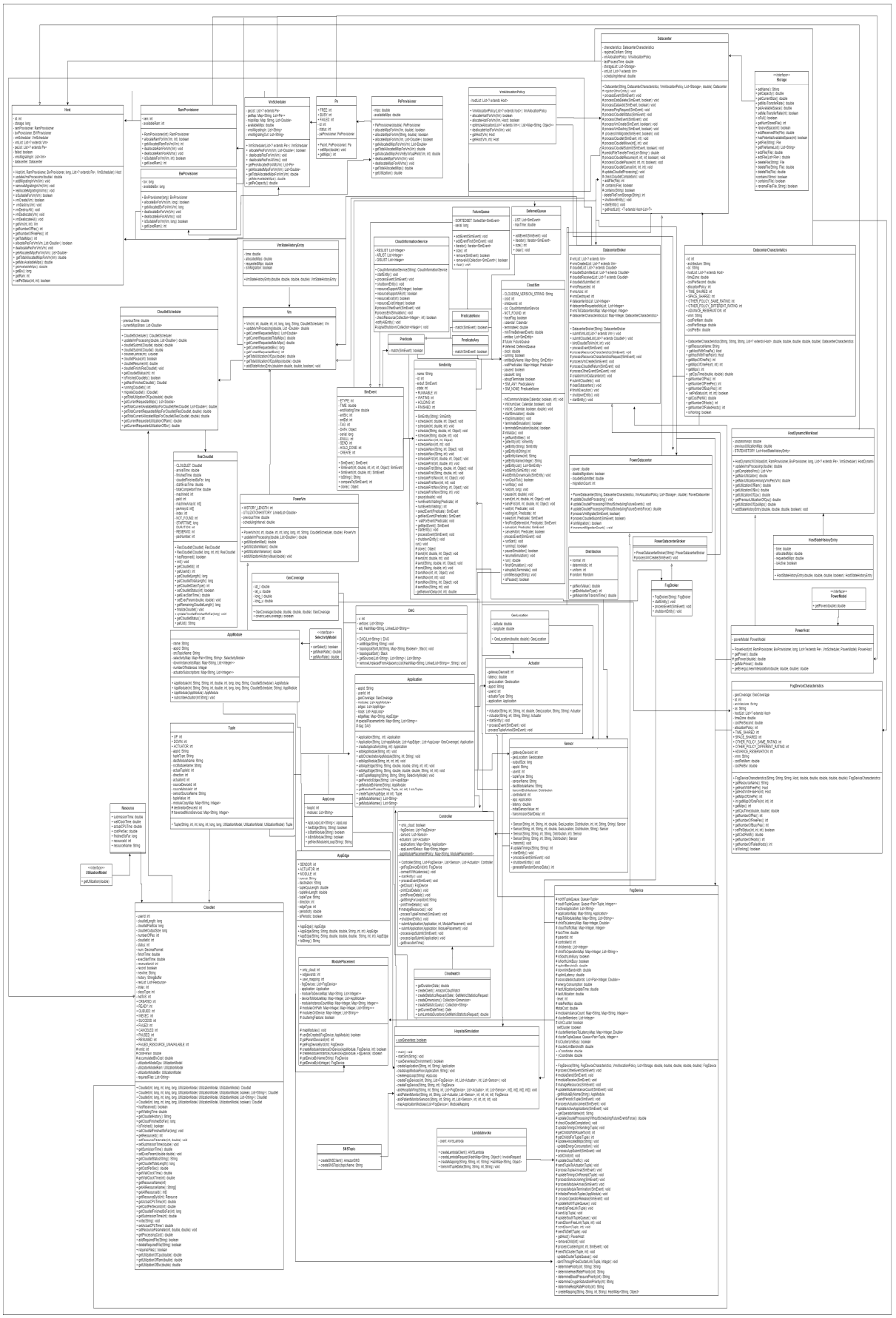

Figure 4: Class Diagram

To enable sensors to emit actual values for processing, the default iFogSim2 Sensor, Tuple and Application classes have been configured. An initialSensorValue integer variable has been declared in the Sensor class and is assigned a value upon instantiation of sensor objects. Sensor objects are instantiated by use of class constructors which accept array index values as constructor initialSensorValue parameter values. The arrays from which index values are retrieved have been hardcoded to allow multiple simulation executions to consist of identical initial sensor values, enhancing comparability abilities amongst different simulation executions. The transmit method of the default Sensor class has been configured to allow a randomly generated integer value to be emitted by the sensor. To prevent drastic fluctuation of a sensors emitted values, the randomly generated integer has a threshold boundary of either five above or below the initialSensorValue set upon sensor instantiation.

Tuple instances are the default iFogSim2 representation of input and outputs amongst simulation components. A tupleValue integer variable has been declared in the Tuple class to allow each tuple instance to contain a sensor value which is assigned a value upon each tuple instantiation. Application modules process tuple instances and the getResultantTuples method of the Application class has been configured to allow application module output tuples have their tupleValue variable value set to match their tuple input. To align with FEC simulation implementations outlined in [16], all application modules have been instantiated with a RAM specification value of ten. Figure 5 illustrates the data flow amongst the simulation's sensors, application modules and display actuator.

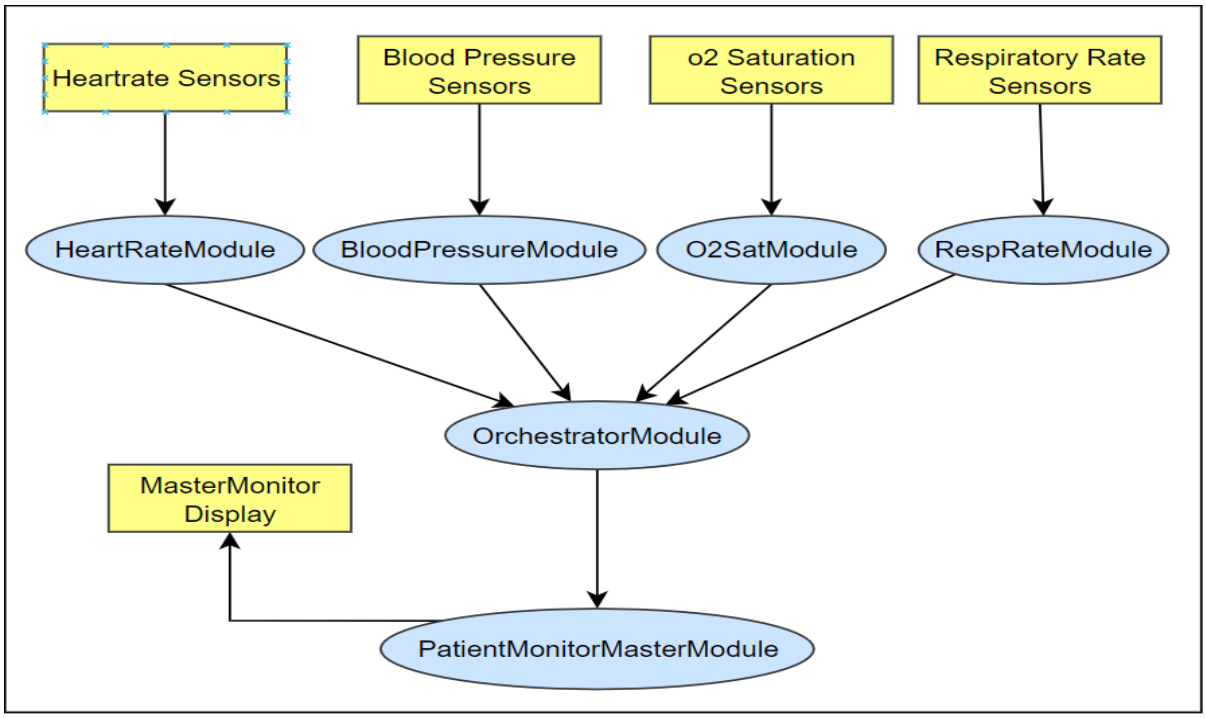

Figure 5: Application Module Data Flow

Tuples transmitted by sensors are received by patient monitor fog devices which host heart rate, blood pressure, o2 saturation and respiratory rate application modules responsible for receiving sensor values from their respective sensor types and transmitting them to orchestrator application modules. Orchestrator application modules are also hosted on patient

monitor fog devices and are responsible for priority classification of received sensor values. The default iFogSim2 AppModule class has been configured by declaring a snsTopicName string variable and inclusion of an additional instance constructor, specifically for orchestrator module instantiations, which accepts a parameter value for assigning a value to the snsTopicName variable.

The executeTuple method of the default FogDevice class has been configured to enable orchestrator modules to classify received sensor value priorities and determine subsequent executable actions, based on classification types. Sensor value priorities are identified by invoking a determinePriority method which accepts arguments for tuple type and sensor value. The determinePriority method returns a sensor priority based on the type of sensor which emitted its transmitted value. Priority classification types can either be p1 or p2, with p1 relating to high priority and p2 relating to low priority. Upon p1 classification, the executeTuple method instantiates a tuple, containing the sensor transmit data, which is sent directly to a patient monitor master module hosted on patient monitor master fog devices.

P2 classifications indicate cloud resource use latencies are acceptable and the simulations serverless environment is utilised with the invocation of a transmitTupleData method which accepts arguments for the sensor source, tuple type and sensor value of the classification subject, in addition to the orchestrator modules snsTopicName variable value. There is a return statement directly after the transmitTupleData invoke statement to ensure execution time increments are not appended to the FEC resource usage execution time. The transmitTupleData method is encapsulated in a LambdaInvoke class. The transmitTupleData invokes nested functions responsible for instantiating a client for AWS Lambda service communication, creating a HashMap consisting of key value pairs related to its arguments and instantiating a Lambda InvokeRequest instance consisting of a JSON formatted version of the HashMap, with an identifier of the Lambda function to invoke which is then executed.

As SAM was used to develop the Lambda function, its source code is viewable in the TransmitTuple class of the project resources directory. The Lambda function invokes nested functions responsible for instantiating a client for AWS SNS communication, retrieving the SNS topic ARN using the snsTopicName contained in its HashMap argument value, instantiating a SNS PublishRequest instance consisting of the sensor value to publish and message attributes relative to the emitting sensor, which is then published to the SNS topic. Executed Lambda function metrics are transferred to Amazon CloudWatch.

The simulation loops through multiple iterations of sensor transmit values travelling amongst system components to reach their endpoint destination of the patient monitor master device, assigned for their residing hospital wing, using the methodology determined by their priority classifications. The printTimeDetails method of the default iFogSim2 Controller class has been configured to enable serverless function latencies to be considered in simulation execution time calculations, when applicable, upon inclusion of a getExecutionTime method.

The getExecutionTime method determines if serverless functions have been integrated or not by invoking a method which returns a Boolean value representing the presence of serverless function integration. If serverless integration is identified as false, the default iFogSim2 methodology of execution time calculation, by subtracting the simulation start time from the time upon which the getExecutionTime method has been invoked, is used. If

serverless integration is identified as true, a getDuration method, responsible for retrieving Lambda metrics from CloudWatch, is invoked. The getDuration method consists of nested functions responsible for instantiating a client for CloudWatch communication, instantiating a CloudWatch GetMetricStatisticsRequest instance with details for retrieving and summing all duration metrics for the Lambda functions responsible for publishing sensor information to SNS topics between the simulation start and end times. The summed duration of all Lambda function durations is then aggregated to the default iFogSim2 calculated execution time. Execution times are measured in milliseconds.

4 Evaluation

Throughout execution of multiple simulations of identical specification, it was observed that the calculated execution time amongst each varied. Unsuccessful remediation attempts, to allow simulations of identical specification to result in identical execution times, were actioned by integrating Docker, AWS EC2 and multithreading approaches. Docker was used to containerize the simulation source code but errors, related to lack of memory, were encountered when running containers containing simulations of the same scale which were successfully executing within the Eclipse IDE. An AWS EC2 instance was provisioned and configured to enable Apache Maven capabilities for simulation source code obtained from its GitHub repository, but errors related to lack of memory resulted in the killing of the application process prior to simulation completion. Use of multithreading techniques, to allow multiple simultaneous running simulation executions, were considered but integration was unsuccessful due to null errors relating to array lists implemented for storage of environment components.

Running multiple simulation executions, of identical specification, in the Eclipse IDE with average execution time calculation was considered a capable methodology for obtaining system latency metrics. Case studies were conducted, in which varying amounts of sensors were instantiated with high priority classified initial transmit values and resulting execution times rounded down to their nearest whole number. It was initially planned to run each case study ten times for both serverless integration inclusion and exclusion but due to the length of time required to calculate execution times, each case study was only executed twice for serverless integration and twice without its integration. Patients in all case studies have four sensors each - heartrate, blood pressure, oxygen saturation and respiratory rate sensors

4.1 Case Study One

Case study one consisted of running simulations in which there are five hospital wings with each wing containing ten patients. 33 percent of the sensors are instantiated with high priority classified values. Table 1 outlines results for both sole FEC resource usage and serverless function integration execution times.

Table 1: Case Study One Results

Simulation	Run 1	Run 2	Average
FEC Only	618055	604572	611313

W/ Serverless	20443142	17749457	19096299

An average execution time increase of 3023.82 percent is observable in case study one, indicating that serverless integration is detrimental to system latency.

4.2 Case Study Two

Case study two consisted of running simulations in which there are five hospital wings with each wing containing ten patients. 67 percent of the sensors are instantiated with high priority classified values. Table 2 outlines results for both sole FEC resource usage and serverless function integration execution times.

Table 2: Case Study Two Results

Simulation	Run 1	Run 2	Average
FEC Only	601298	547015	574156
W/ Serverless	21786033	21503399	21644716

An average execution time increase of 3669.83 percent is observable in case study two, indicating that serverless integration is detrimental to system latency.

4.3 Case Study Three

Case study three consisted of running simulations in which there is a single hospital wing containing ten patients. 35 percent of the sensors are instantiated with high priority classified values. Table 3 outlines results for both sole FEC resource usage and serverless function integration execution times.

Table 3: Case Study Three Results

Simulation	Run 1	Run 2	Average
FEC Only	18174	15194	33368
W/ Serverless	8315945	3338651	5827298

An average execution time increase of 17363.73 percent is observable in case study three, indicating that serverless integration is detrimental to system latency.

4.4 Case Study Four

Case study four consisted of simulations in which there is a single hospital wing containing ten patients. 65 percent of the sensors are instantiated with high priority classified values. Table 4 outlines results for both sole FEC resource usage and serverless function integration execution times.

Table 4: Case Study Four Results

Simulation	Run 1	Run 2	Average
FEC Only	17343	17940	17641
W/ Serverless	3699700	3743620	3721660

An average execution time increase of 20996.65 percent is observable in case study three, indicating that serverless integration is detrimental to system latency.

4.5 Discussion

Throughout simulation executions, FEC environment execution times were monitored prior to the addition of serverless function execution times and it was observed that, upon serverless integration, FEC environment execution times would drastically increase. Figure 6a portrays results for a simulation which has no serverless integration and shows an execution time of 19439. As portrayed in figure 6b, results from a simulation, consisting of identical specifications but with serverless integration, show an execution time of 1222362 just for the FEC environment resource use, contradicting the assumption that reduced local environment resource usage would result in reduced execution times for FEC environment resource use. To further investigate this contradiction, another simulation consisting of identical specifications but with innovation of a return statement instead of the Lambda function trigger statement, was executed and as portrayed in figure 6c, resulted in a reduced execution time of 11515 than sole FEC resource usage, indicating that the serverless integration aspect of the system is not working according to its intended design.

Figure 6a: FEC Resource Only

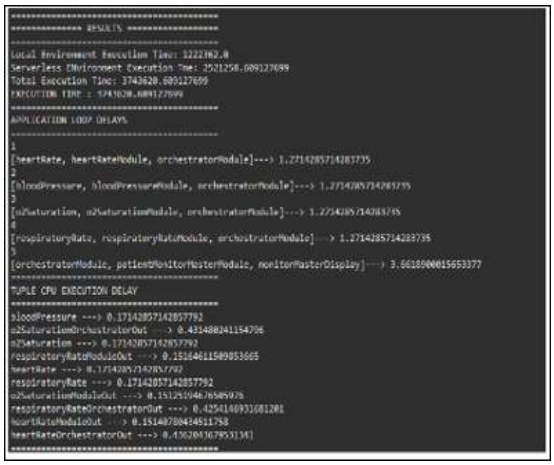

Figure 6b: W/ Serverless

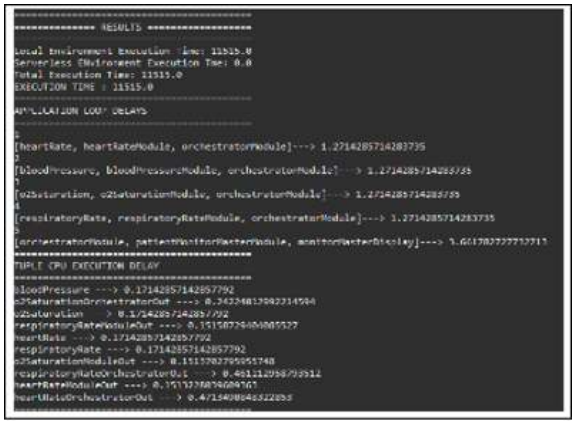

Figure 6c: W/ Serverless FEC Resource Only

The possibility that the system was waiting for successful execution of single Lambda functions before invoking subsequent functions was considered but, as portrayed in figure 7, AWS CloudWatch metrics indicate that multiple Lambda functions are running concurrently. Also considered, was the possibility that durations required for SNS topic message publishing were detrimental for execution times, but simulations were executed with the commands to publish to the SNS topic, within the Lambda function source code, commented out and making them unreadable during function executions, however, this approach indicated that the message publishing was not the root cause of the issue. Also considered, was the possibility that the Lambda functions RAM allocations of 512MB was not efficient enough to enable function executions in quick succession time. Difficulties with obtaining requested relevant AWS account permissions, to allow Lambda and SNS communications, in a timely manner have resulted in too short a timeframe to further investigate and possibly rectify this issue.

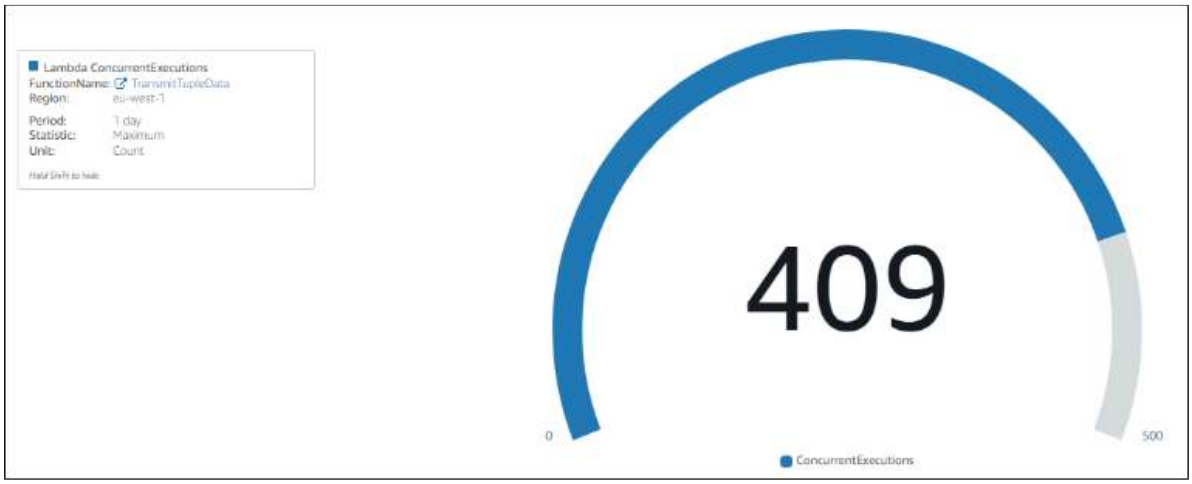

Figure 7: Maximum Concurrent Lambda Function Execution Metric

5 Conclusion and Future Work

This research aimed to assess if an orchestrator component capable of invoking serverless functions for non-time critical identified requests could reduce FEC environment latency. Implementation of a FEC environment simulation, integrated with serverless functionalities with the context of a hospital monitoring its patients using heart rate, blood pressure, oxygen saturation and respiratory rate sensors was established as an objective. A simulation has been implemented in which sensor transmitted values are classified based on priority with subsequent actions determined by classification types. Serverless functions are invoked for use by requests classified as low priority whilst local environment resource usage is reserved for high priority classified requests. AWS Lambda functions publish low priority sensor data to an SNS topic responsible for publishing received messages to subscribed endpoints. AWS CloudWatch has been used to retrieve and sum all Lambda function durations from the start of a simulation execution to its end.

Simulation results indicate that FEC environment latency does not reduce with the integration of serverless functions for low priority classified sensor values, however, limitations caused by uncertainties of resource consumption specifications for fog device

hosted application modules and their communications may have resulted in application module specifications which do not reflect realistic specifications. Future work could be undertaken in which application modules are instantiated using realistic resource consumption specifications with realistic communication protocols between each. Future work could also be undertaken in which cloud resource usage differs than implemented in this simulation. Use of a cloud hosted database, such as DynamoDB, could be investigated to allow endpoints to retrieve low priority classified sensor data instead of the SNS publish/subscribe model implemented for this simulation. Various Lambda function configurations, such as different scaling policies and RAM allocations could also be investigated and finely tuned to determine if they can reduce execution times.

References

[1] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30-39, Jan. 2017. doi: 10.1109/MC.2017.9

[2] J. Shen, T. Yang, Y. Su, Y. Zhou, and M. R. Lyu, "Defuse: A dependency-guided function scheduler to mitigate cold starts on FaaS platforms," in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS),* DC, USA, July 7-10, 2021, pp. 194-204. doi: 10.1109/ICDCS51616.2021.00027

[3] ETSI, "Multi-access edge computing (MEC); Framework and reference architecture," RGS/MEC-0003v221Arch, Dec. 2020.

[4] L. Zanzi, F. Giust, and V. Sciancalepore, "M^{2}EC: A multi-tenant resource orchestration in multi-access edge computing systems", in *2018 IEEE Wireless Communications and Networking Conference (WCNC),* Barcelona, Spain, April 15-18, 2018, pp. 1-6. doi: 10.1109/WCNC.2018.8377292

[5] C. Cicconetti, M. Conti, A. Passarella, and D. Sabella, "Toward distributed computing environments with serverless solutions in edge systems," *IEEE Communications Magazine,* vol. 58, no. 3, pp. 40-46, Mar. 2020. doi: 10.1109/MCOM.001.1900498

[6] M.C. Filippou, D. Sabella, and V. Riccobene, "Flexible MEC service consumption through edge host zoning in 5G networks," in *2019 IEEE Wireless Communications and Networking Conference Workshop (WCNCW)*, Marrakech, Morocco, April 15-18, 2019, pp. 1-6. doi: 10.1109/WCNCW.2019.8902852

[7] R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar and J. Ott, "Consolidate IoT edge computing with lightweight virtualization," *IEEE Network*, vol. 32, no. 1, pp. 102-111, Jan. 2018. doi: 10.1109/MNET.2018.1700175

[8] S. Lee, D. Yoon, S. Yeo, and S. Oh, "Mitigating cold start problem in serverless computing with function fusion," *Sensors*, vol. 21, no. 24, p. 8416, 2021. doi: 10.3390/s21248416

[9] P. Silva, D. Fireman, and T.E. Pereira, "Prebaking functions to warm the serverless cold start," in *Proceedings of the 21st International Middleware Conference (Middleware '20)*, Delft, Netherlands, December 7-11, 2020, pp. 1-13. doi: 10.1145/3423211.3425682

[10] S. R. Poojara, C. K. Dehury, P. Jakovits, and S. N. Srirama, "Serverless data pipeline approaches for IoT data in fog and cloud computing," *Future Generation Computer Systems,* vol. 130, pp. 91-105, May. 2022. doi: 10.1016/j.future.2021.12.012

[11] B. Cheng, J. Fuerst, G. Solmaz, and T. Sanada, "Fog function: Serverless fog computing for data intensive IoT services," in *2019 IEEE International Conference on Services Computing (SCC),* Milan, Italy, July 8-13, 2019, pp. 28-35. doi: 10.1109/SCC.2019.00018

[12] R. Mahmud, S. Pallewatta, M. Goudarzi, and R. Buyya, "Ifogsim2: An extended ifogsim simulator for mobility, clustering, and microservice management in edge and fog computing environments," *Journal of Systems and Software*, vol. 190, no. 111351, pp. 111351, Aug. 2022

[13] Amazon Web Services, "AWS Serverless Application Model – Developer Guide," AWS, 2021. [Online]. Available: https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-application-model.pdf

[14] The Cloud Computing and Distributed Systems (CLOUDS) Laboratory (2021), iFogSim [Source code]. https://github.com/Cloudslab/iFogSim

[15] Raspberry Pi, "Raspberry Pi 3 Model B+," 2018. [Online]. Available: https://datasheets.raspberrypi.com/rpi3/raspberry-pi-3-b-plus-product-brief.pdf

[16] K. S. Awaisi, A. Abbasm S. U. Khan, R. Mahmud, and R. Buyya, "Simulating Fog Computing Applications Using iFogSim Toolkit," in *Mobile Edge Computing*, Cham: Springer International Publishing, 2021, pp. 565-590.