Cloud Application Development Continual Assessment 2024

How can Cloud Services be used to Develop a Serverless Microcontroller Temperature Monitoring Application

Deirdre Lee T00126583

**Abstract**

This project investigates the use of coud services for developing a serverless microcontroller temperature monitoring application. The purpose of this work is to explore the feasibility and effectiveness of using cloud infrastructure, in this case AWS, to enhance the scalability, reliability and efficiency in a proof of concept temperature monitoring system.

The research methodology involves review of research papers relating to this topic, online documentation regarding both AWS and micro:bit, as well as user manuals pertaining to microcontroller integration in a number of areas. This study examines the integration of cloud services such as AWS Lambda, DynamoDB, API Gateway, and Amplify.

The main conclusion reached highlights the ease of use of AWS services for developing serverless microcontroller applications for temperature monitoring, highlighting the potential of cloud-based solutions in IoT applications.

*Keywords: Microcontroller, Monitoring, AWS, Serverless, API, Alarm, Notification, Lambda, DynamoDB, Amplify*

# **Introduction**

This report explores the development of a serverless microcontroller temperature monitoring application using cloud services, particularly AWS. It addresses the question: How can Cloud Services be used to Develop a Serverless Microcontroller Temperature Monitoring Application?

The background involves the growing interest in cloud-based IoT solutions due to scalability and efficiency. The investigation is relevant due to increasing demand for IoT solutions and cloud-based approaches' potential benefits. The method includes literature review and practical experimentationusing a number of AWS services and aa micro:bit microcontroller.

The research also aims to provide insights into cloud-based microcontroller applications' feasibility and effectiveness. It is important to note that device security and application authentication aspects will not be explored for the proof-of-concept system due to time and research limitations.

# **Literature Review**

The search for relevant publications involved employing various methodologies, including keyword searches in academic databases, review of conference proceedings, and examination of relevant journals in the field of IoT and cloud computing. Notable trends observed during the search included an increasing emphasis on cloud-based solutions for IoT applications, particularly in the context of scalability, efficiency, and cost-effectiveness.

During the research, it was noted that the devices used for monitoring were Wi-Fi, or LoRaWAN, enabled to some extent.(Leal Sobral et al., 2023) Additionally, multiple devices were employed for the monitoring process, ensuring a more accurate reading. The system architecture described in "Cloud-based Sensor Network for Environmental Monitoring" depicts a robust system with three layers, with the middle 'bridge layer' containing the cloud component. Despite not having access to the same range of devices, this project was modelled on the proposed architecture.(Corbellini et al., 2017)

Throughout the search, it was also observed that the devices used for monitoring required a standalone power source. In instances where they were not connected to a device via USB, they could use a battery pack for power, which is relevant in the case of using a micro:bit. For the Proof of Concept application, it was not deemed necessary to source a battery pack due to there being only one monitoring device, as well as the fact that without an intermediary communication device, the connection via USB was not only vital for power but also for communication with a receiver program.

A very interesting new concept that emerged through research was LoRa technology. This technology is

a wireless communication technology designed for long-range communication with low power consumption, making it suitable for applications such as Internet of Things (IoT) devices, environmental monitoring, and asset tracking, as outlined in Rapid IoT Prototyping: A Visual Programming Tool and

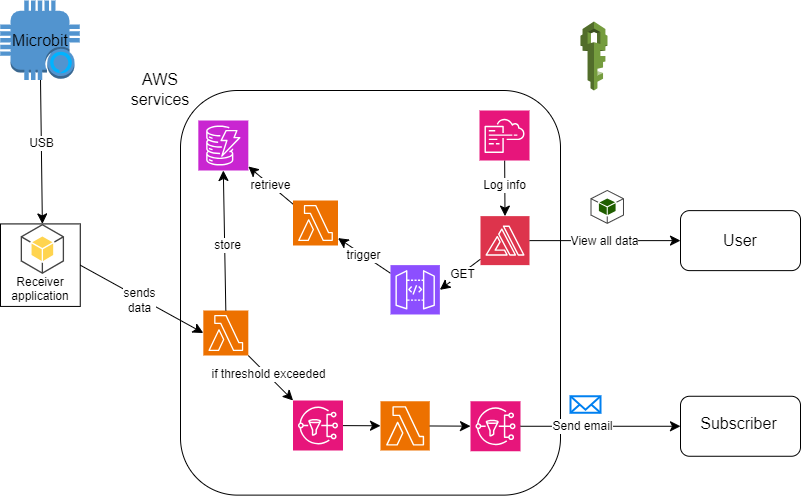
Hardware Solutions for LoRa-Based Devices (López & Lamo, 2023). Micro:bit LoRa devices are available for purchase, so this will be an area of interest for further exploration and development of this project.

In conclusion, the literature review provided valuable insights that informed the research in several ways. It highlighted the growing trend towards cloud-based solutions for IoT applications, emphasising their scalability, efficiency, and cost-effectiveness.

The review also identified the use of LoRa devices for monitoring, along with the importance of using multiple devices to ensure accurate readings. Most influential was the examination of the system architecture in "Cloud-based Sensor Network for Environmental Monitoring", which offered a framework to model the current project, despite device access limitations.



# **Methodology**



## **Develop Sensor Data Collection**

For this project, Mu Editor was installed to work on the program locally, as suggested in the book ‘*Beginning BBC micro:bit: A Practical Introduction to micro:bit Development’* (Seneviratne, 2018). A basic script was created to check if the temperature received from the micro:bit exceeded predefined thresholds, printing a warning across the microcontroller screen as well as the current temperature.

### **Transmit Readings to Laptop Device**

Utilising Bluetooth as a communication option was considered for this project. However, upon investigation, it was determined to not be viable for the current scope of the project. This decision was influenced by the need for an intermediary device to facilitate communication over Bluetooth Low Energy (BLE) via a ‘bridge layer’. (Corbellini et al., 2017) The micro:bit v1.3B, being the primary sensor device, cannot transmit data via BLE. While it possesses built-in radio capabilities, it operates on a custom protocol based on, but incompatible with, BLE. To enable BLE communication, an intermediary device is essential. One feasible option involves using a Raspberry Pi, which supports BLE. Acting as the bridge, the Raspberry Pi would receive data from the micro:bit via its radio module, process it, and transmit it over BLE to the receiving device. Given these considerations, the decision was made to rely on an alternative approach that involves using a USB cable to transmit data directly to a laptop. As part of this setup, the micro:bit code was updated to log readings at specified intervals and trigger events when threshold exceedances occurred (microbit, 2022), and the temperature logs were stored in memory. To ensure continuous logging, I developed a Python script that imported the ‘pyserial’ library to establish communication with the micro:bit via the serial port, testing communication initially using PuTTY. (Liechti, 2020) This script reads the data from the micro:bit and stores it in a Python dictionary. This printed data was then received through the program.

## **Set up AWS Identity & Access Management (IAM)**

As services were created, any necessary permissions were assigned by creating policies and assigning them to the required roles, following the principle of least privilege. Throughout the description of steps taken, there are included indications of which services this was implemented for.

## **Create AWS Lambda Function**

After completing the previous steps of creating the initial microcontroller program and setting up communication with the receiving laptop, it was necessary to begin the cloud application process. The initial plan for the cloud implementation included utilising AWS IoT Core to manage communication with the IoT device. However, since the device is not WiFi-enabled and needs to communicate through the Python receiver program, as a result of the research and exploration in ['3.1.1 Transmit Readings to Laptop Device’](#_Transmit_Readings_to) this service is not suitable for the proof of concept version. The receiver program created will need to communicate with the cloud, and to do this, it was necessary to create a Lambda function to receive the data through a HTTP endpoint. This also necessitated the creation of an API Gateway service to facilitate the storage to the database, which will be discussed further in section [‘5. Set up API Gateway’](#_Set_up_API). The Lambda function, HttpRequestHandlerLambda, was then created. This function manages incoming sensor data, storing it in DynamoDB, and triggering alerts via SNS when temperature thresholds are exceeded.

The ThresholdExceededAlarm Lambda function processes incoming SNS messages, extracting temperature-related information such as the current temperature, location, and predefined threshold. It constructs a warning message if all required information is present, alerting about temperature breaches. The function then publishes this message to a specified SNS topic. Error handling ensures that any issues encountered during execution are logged and properly raised for troubleshooting. The HttpGetAllData Lambda function uses the DynamodDB scan operation to retrieve all items from the MicroSensor table. It then iterates through each item to check for instances of type Decimal and converts them to type float Finally, the items are returned along with a 200 status code.

## **Set up API Gateway**

For the ‘HttpRequestHandlerLambda’ lambda function created in ['5 Create AWS Lambda Function'](#_Create_AWS_Lambda) it was necessary to create a REST API gateway with a POST endpoint and add it as a trigger to the Lambda function. To facilitate the retrieval of the table items using the ‘HttpGetAllData’ lambda function, adding a GET endpoint found at ‘/alltempdata’ was crucial. Calling this endpoint triggers the aforementioned serverless function.

## **Set up Amazon Simple Notification Service (SNS)**

Before creating the ‘ThresholdExceededAlarm’ Lambda function, an SNS topic and subscription were created. A topic called ‘TempThresholdAlarm’, which by default can only be published and received by the topic owner, was also created. The default specifying the topic owner as the only subscriber can be updated but it was deemed sufficient for this POC. Subsequently, a subscription was created, using an email protocol, and defining the author's college email to be the only subscriber. (Amazon AWS, 2024) A subscription confirmation email was then sent to the email address, to which the user was then able to successfully subscribe. After creating the ‘ThresholdExceededAlarm’ function a new IAM policy was added to allow the Lambda function permission to publish to SNS. (AWS Knowledge Center, 2024) The ‘HttpRequestHandlerLambda’ function was then updated to invoke the ‘ThresholdExceededAlarm’ function, passing the current temperature, location, and threshold variables. These variables are then included in the updated message sent to the subscriber. This was achieved by using Amazon SNS to trigger the second Lambda function. (Amazon AWS, 2024) A new topic, called ‘NotifyAlarmLambda’, was created and a subscription with a Lambda protocol set up, which triggers the ‘ThresholdExceededAlarm’ function from the ‘HttpRequestHandlerLambda’ endpoint. The receiving function, ‘ThresholdExceededAlarm’, was then updated to parse the incoming message and perform the SNS actions. Additionally, both Lambda functions were ensured to have the necessary IAM roles to perform the required actions. The ‘HttpRequestHandlerLambda’ function, when invoked, publishes to the SNS topic ‘NotifyAlarmLambda’, providing the necessary variables ‘curr\_temp’, ‘location’, and ‘threshold’. In short, the ‘ThresholdExceededAlarm’, subscribed to the ‘NotifyAlarmLambda’ topic, parses the incoming message details and incorporates them into a message published to the ‘TempThresholdAlarm’ topic. This message is distributed to all subscribers. Overall, the SNS topics and subscriptions were easy to manage and updateable to include multiple subscribers as the application grows.

## **Configure Amazon DynamoDB**

### **Database Plan for Storing Sensor Readings**

DynamoDB is used for storing the temperature readings and other associated data. DynamoDB was chosen due to its ability to handle large quantities of data and scale efficiently. It is also suitable for real-time processing, which is crucial for this application. Users will need to be able to access historical data, and there will be continuous write operations as the temperature data is stored.

Currently, the data being collected includes the temperature reading in degrees Celsius to the nearest whole number. It also records when the threshold is exceeded, as well as holding the value of the upper and lower temperature thresholds. Additionally, the schema needs to include the following information: Sensor ID, a unique identifier for each sensor device; Location, details about the location of each sensor device (such as room name); Time of Reading, a timestamp indicating when each temperature reading was recorded; and Exceeded Threshold, a flag indicating whether the temperature reading exceeded the predefined upper or lower thresholds. There is only one sensor device being used for this proof of concept application, but as part of the non-functional requirement of scalability, the schema needs to reflect the fact that multiple sensors may be present. Considering all this, the following schema was proposed:

|  |  |
| --- | --- |
| TableName | MicroSensor |
| SensorID | String (PK - Partition Key) |
| Location | String |
| TimeDate | String (PK - Sort Key) |
| Temperature | Number |
| Upper Threshold | Number |
| Lower Threshold | Number |
| Exceeded Threshold | Boolean |

The decision was made to use a composite primary key for the table, designating ‘sensorID’ for the partition key. This decision was made due to the need for scalability, as the system is expected to interact with multiple sensors beyond the proof of concept phase. By distributing data across multiple partitions based on ‘sensorID’, DynamoDB can efficiently handle increased data or workload. (Amazon AWS, 2024) ‘TimeDate’ was allocated as the sort key. This choice aligns with DynamoDB's design principles and helps with efficient querying, especially for accessing historical data. (Amazon AWS, 2024) By organising data based on the event time, the table supports optimised data retrieval and storage. The microcontroller code was updated to pass only the required variables to the receiver program. Despite spending a lot of time researching and testing, time data was unable to be passed from the microcontroller. Instead, a variable is initialised on the receiver side, which is then stored with the passed variables.

### **Set up Storage Solution with DynamoDB**

The next step in the project was to set up the DynamoDB table to store the data received by the Lambda function via the API Gateway. A table called ‘MicroSensor’ was created, setting ‘SensorID’ as the partition key and ‘DateTime’ as the sort key as previously discussed. An IAM policy was then created, called ‘DynamoDBReadWrite’, to allow permissions to the Lambda functions to read and write to the table. After creating the policy, it was attached to the ‘HttpRequestHandlerLambda’ role.

## **Set up AWS Amplify**

Continuing with the project, a simple Vue.js application was deployed using AWS Amplify. Amplify's streamlined deployment process was utilised, with the project first being initialised with Amplify CLI, with necessary settings configured. The application is set up to deploy on a push to the GitHub repository, which is listed in AWS as the source provider. Deployment is also achievable by using a webhook. Finally, the deployment of the Vue app to the cloud was achieved with a simple command, amplify publish, instantly making the application accessible to users.

# 

# **Results/Data/Findings**

The lack of capabilities of the micro:bit presented a challenge, necessitating an alternative approach as the initial plan to connect directly using AWS IoT Core was not feasible. Instead, alternative methods were explored to establish communication with the microcontroller, leading to the adoption of a workaround solution.

AWS services proved to be user-friendly, with well-documented resources and simple-to-follow examples. This facilitated the integration of various AWS services into the project. However, the vast array of available AWS services also posed a challenge, as it was easy to become overwhelmed by the options. At certain points in the project, it was necessary to take a step back to prioritise and refrain from adding additional services simply because they were available.

The absence of a budget constraint limited the exploration of certain paid AWS services. While some services were not viable options due to budget constraints, the project highlights the potential benefits of using paid services for enhanced functionality and performance.

While the project encountered some minor challenges such as device limitations and budget constraints, it more importantly demonstrated the ease of use and effectiveness of AWS services.

Include about issue with POST necessitating redesign

# **Discussion**

The project successfully implemented a temperature monitoring system using the micro:bit, AWS services, and the Vue.js application. However, it encountered initial limitations due to unfamiliarity with IoT devices and the complexities of AWS services. Despite these challenges, the project progressed, utilising direct USB communication as an efficient alternative to Bluetooth.

IAM policies were created to ensure secure access control within the AWS environment. The development of various Lambda functions enabled seamless integration with DynamoDB and SNS for data storage and alerting. The integration of API Gateway facilitated smooth communication between the microcontroller receiver application and cloud services. With DynamoDB configured to store temperature readings and associated data, the development of the project illustrated the ease of use and effectiveness of AWS services for data management.

Finally, the deployment of the Vue.js application using AWS Amplify ensured rapid deployment and instant accessibility to users, again highlighting the user-friendly nature of the AWS services. CloudFormation was an invaluable tool in debugging the Amplify application. Overall, the project’s success emphasises the effectiveness of the AWS documentation, which offered thorough guidance throughout the development process.

Make more personal, experiences with working with AWS and the device. Any issues or difficulties?

# **Conclusion**

Moving forward, if the project were to be redone or built upon, priority would be given to enhancing security measures to ensure device and data protection and confidentiality. Additionally, with more time and research, it is believed that additional AWS services could be integrated to further enhance system functionality, such as CloudWatch for Monitoring. Furthermore, existing services could be implemented more efficiently with a deeper understanding and refinement of their configurations, such as an expansion on Amplify for the application backend. Considering that there was no budget allocated, service costs had to be carefully considered in the decision-making process.

Overall, while the project achieved its objectives, there is room for improvement and optimisation. With continued exploration and refinement, the system could be evolved to meet more specific requirements, as well as offer enhanced performance and security for both the application and users.

Link to GitHub Repository containing all code files: <https://github.com/EmoSense/therm_alarm>

Link to AWS Hosted Application: <https://main.d39t6hls28dk0h.amplifyapp.com/>

# **References**

Amazon AWS, 2024. *Best practices for designing and using partition keys effectively.* [Online]   
Available at: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-partition-key-design.html  
[Accessed 5 April 2024].

Amazon AWS, 2024. *Best practices for using sort keys to organize data.* [Online]   
Available at: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-sort-keys.html  
[Accessed 5 April 2024].

Amazon AWS, 2024. *Fanout to Lambda functions.* [Online]   
Available at: https://docs.aws.amazon.com/sns/latest/dg/sns-lambda-as-subscriber.html  
[Accessed 9 April 2024].

Amazon AWS, 2024. *Getting started with Amazon SNS.* [Online]   
Available at: https://docs.aws.amazon.com/sns/latest/dg/sns-getting-started.html  
[Accessed 9 April 2024].

Amazon AWS, 2024. *Tutorial: Build a CRUD API with Lambda and DynamoDB.* [Online]   
Available at: https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-dynamo-db.html  
[Accessed 8 April 2024].

Anon., n.d. [Online].

AWS Knowledge Center, 2024. *How can I publish a message to an Amazon SNS topic using a Lambda function?.* [Online]   
Available at: https://repost.aws/knowledge-center/sns-topic-lambda  
[Accessed 9 April 2024].

Golovnia, S., 2022. *15 NON-FUNCTIONAL REQUIREMENTS IN SOFTWARE ENGINEERING [2024].* [Online]   
Available at: https://www.sfappworks.com/blogs/non-functional-requirements-in-software-engineering  
[Accessed 1 April 2024].

Liechti, C., 2020. *pyserial.* [Online]   
Available at: https://pyserial.readthedocs.io/en/latest/shortintro.html#opening-serial-ports  
[Accessed 3 April 2024].

microbit help & support, 2021. *Outputing serial data from the micro:bit to a computer.* [Online]   
Available at: https://support.microbit.org/support/solutions/articles/19000022103-outputing-serial-data-from-the-micro-bit-to-a-computer#:~:text=You%20can%20send%20and%20receive,and%20Makecode%20for%20micro%3Abit.&text=In%20Python%2C%20the%20  
[Accessed 4 April 2024].

microbit, 2022. *Serial.* [Online]   
Available at: https://makecode.microbit.org/device/serial  
[Accessed 4 April 2024].

Nuclino, 2024. *A Guide to Functional Requirements.* [Online]   
Available at: https://www.nuclino.com/articles/functional-requirements  
[Accessed 1 April 2024].

Seneviratne, P., 2018. *Beginning BBC micro:bit: A Practical Introduction to micro:bit Development.* 1st ed. Sri Lanka: Apress.

Corbellini, S., Di Francia, E., Grassini, S., Iannucci, L., Lombardo, L., & Parvis, M. (2017). *Cloud based sensor network for environmental monitoring*. https://doi.org/10.1016/j.measurement.2017.09.049

Leal Sobral, V. A., Nelson, J., Asmare, L., Mahmood, A., Mitchell, G., Tenkorang, K., Todd, C., Campbell, B., & Goodall, J. L. (2023). A Cloud-Based Data Storage and Visualization Tool for Smart City IoT: Flood Warning as an Example Application. *Smart Cities*, *6*(3), 1416–1434. https://doi.org/10.3390/SMARTCITIES6030068

López, J. J., & Lamo, P. (2023). Rapid IoT Prototyping: A Visual Programming Tool and Hardware Solutions for LoRa-Based Devices. *Sensors*, *23*(17). https://doi.org/10.3390/S23177511