HCLTech | Supercharging Progress™

AI solution for constrained edge gateways: An architectural approach

Table of contents

Abbreviations 3

1. Introduction 4

2. Business challenges 4

3. Problem statement 5

4. AI solution 5

5. Model development phase 6

5.1 Data collection 7

5.2 Model design 8

5.3 Model training 8

5.4 Model testing 9

5.5 Gesture recognition- details 10

5.6 Edge impulse code generation - Problems 10

6. Model deployment phase 11

6.1 Model interpretation 11

6.2 Embed model 12

6.3 Interference programming 13

6.4 Deploy interfence 14

6.4.1 Reference architecture 14

6.4.2 Gesture recognition- Architecture 16

6.5 Test results 18

6.5.1 Size 18

6.5.2 Memory consumption 19

7. Benefits 20

8. Conclusion 20

9. References 21

10. Authors info 22

Abbreviations

Acronym	Full form
AI	Artificial Intelligence
API	Application Programming Interface
GUI	Graphical User Interface
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
ML	Machine Learning
OS	Operating System
RTOS	Real-Time Operating System
TinyML	Tiny Machine Learning
UART	Universal Asynchronous Receiver-Transmitter
uTask	micro-Task

1. Introduction

Technological advancements have had a significant impact on various fields, including medicine, consumer electronics, aerospace and industry. Among these innovations, AI is significantly transforming human lives. Integrating these AI into resource-limited devices like microcontrollers has long been a challenge, but the advent of Tiny Machine Learning (TinyML) has made this possible.

TinyML enables the deployment of ML models on resource-constrained devices such as microcontrollers with low processing capability and limited memory. TinyML revolutionizes edge computing by enabling resource-constrained gateways to receive data from sensors and perform real-time analytics without sending data to the cloud. By reducing cloud dependency, latency-sensitive applications can run at the system's edge, improving real-time responsiveness and enhancing privacy. TinyML is used across various domains, including Smart Homes, Smart Cities, Healthcare and Agriculture.

Despite its broad applications, most TinyML solutions are monolithic in nature, designed with a single large task of sensor data acquisition and edge analytics. Consequently, these solutions often lack support for non-functional requirements such as readability, modifiability, reusability, testability and configurability. These challenges emphasize the need for an architectural solution that facilitates the seamless deployment of TinyML in constrained edge gateways. This paper proposes a solution to address these challenges.

2. Business challenges

The development of any reference architecture presents several business challenges, including efforts to identify the engineers with the right skills to meet non-functional requirements such as modifiability, readability, testability, configurability and reusability. In the absence of a reference architecture, different teams within an organization may develop their designs, leading to inconsistent solutions, multiple maintenance streams and a lack of uniformity. An additional challenge is finding engineers with the proper skill set for developing AI modules on resource-constrained microcontrollers, a rare combination.

This reference architecture needs to be both domain and OS-agnostic. It shall act as an accelerator for the software development of constrained edge gateways and shall be applied across multiple domains, such as medical, aerospace, telecommunications and industrial sectors. Additionally, it shall support the integration of new functionalities.

3. Problem statement

Currently, no open-source reference architectures are available for constrained edge gateways (e.g., microcontrollers) for deploying TinyML applications. Existing solutions are proprietary. Existing solutions are proprietary. Some frameworks, such as Edge Impulse, generate single monolithic applications that lack key non-functional requirements such as readability, testability, modifiability, reusability and configurability as shown in Figure 1. As a result, architecture teams must invest significant efforts to design and develop an architecture during the architectural phase to deploy TinyML applications. Developing a reference architecture with guidelines for TinyML deployment can significantly reduce this effort, allowing architecture teams to focus more on integrating functional requirements.

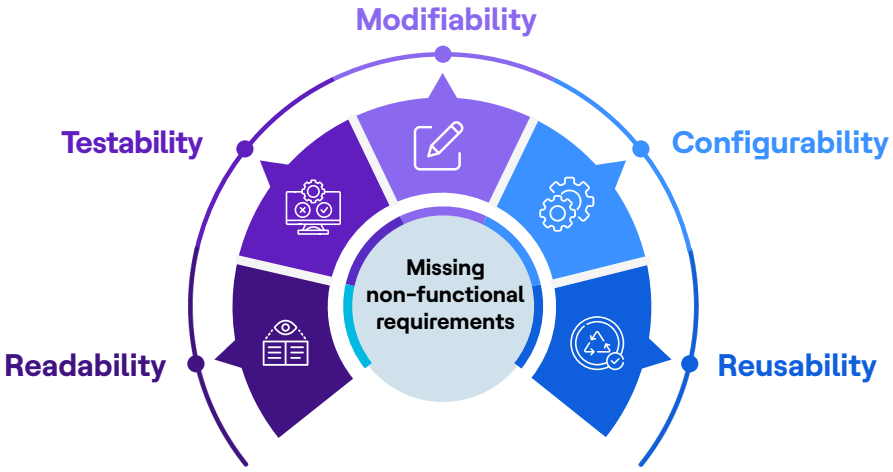

Figure 1 - Problem statement

4. AI solution

Any AI solution involves two major phases which are model development and model deployment. The proposed AI solution uses Edge Impulse for model development and a reference architecture for model deployment. Figure 2 shows an overview of the tasks in each phase.

Let us examine these phases one by one using the case study of a gesture recognition application that identifies various motions such as left-right, front-back and up-down by performing analytics on the collected accelerometer data. The details of this application are shown in the later section.

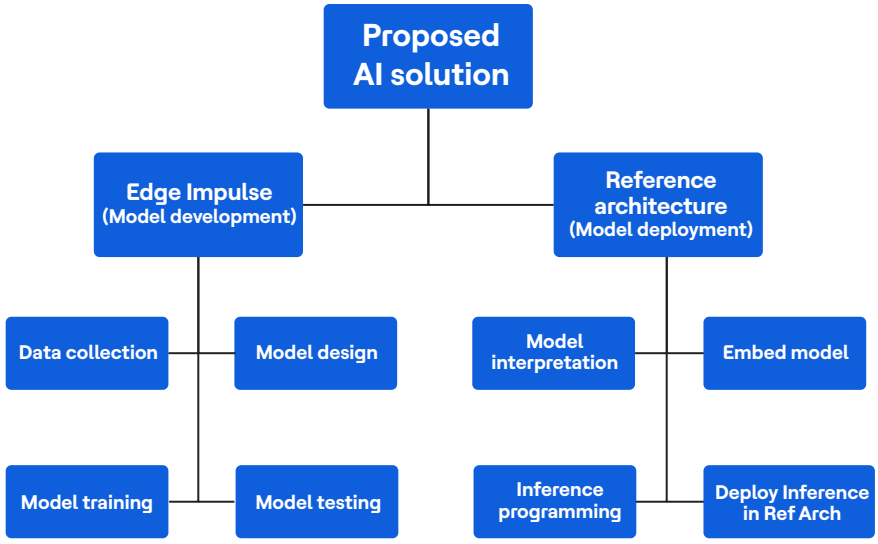

Figure 2 - TinyML Solution - Phases and Tasks

5. Model development phase

The model development phase involves tasks such as data collection, design, training and testing. These are cyclic tasks, as shown in Figure 3 and need to be repeated until the model gives accurate predictions.

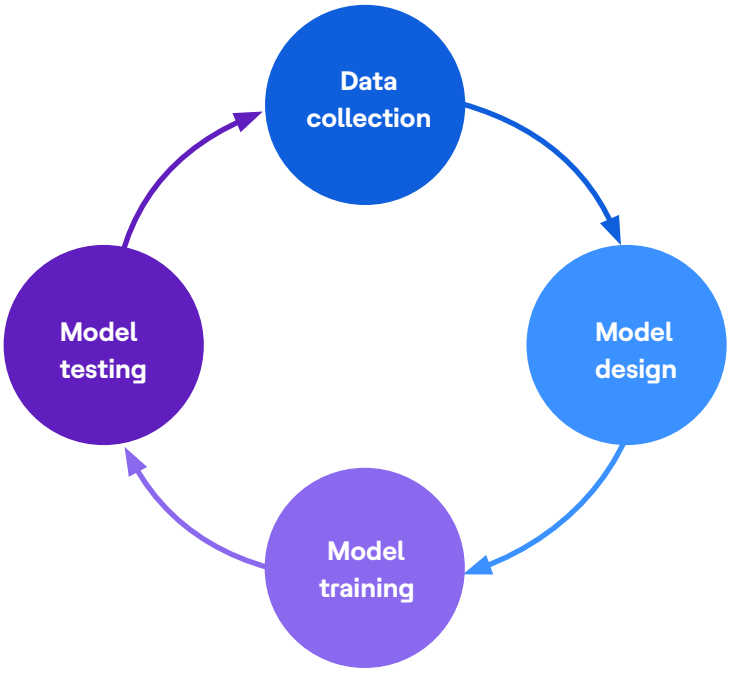

Figure 3 - Tasks in model development phase

Instead of writing programs that require significant effort, open-source tools such as Edge Impulse can speed up the work. Edge Impulse is an open-source tool that can be used for tasks such as data collection, model design, model training and model testing. It is used in this gesture recognition application, which addresses the problem of the need for significant effort. It is highlighted in Table 1. Refer to [1] for the documentation of Edge Impulse. It provides detailed instructions and extensive guidance.

Table 1 - Edge Impulse for model generation

Problem identified	Proposed solution
Developing programs for data collection, model design, model training, model testing and model generation requires significant effort.	Edge Impulse, an open-source graphical user interface tool, significantly reduces those efforts.

5.1 Data collection

This step involves gathering accelerometer data for different board movements. To facilitate this process, an application for data collection must be developed and executed on the target board, enabling it to read the accelerometer data and transmit it to Edge Impulse via Universal Asynchronous Receiver-Transmitter (UART).

1. After creating an Edge Impulse account and logging in, create a project in Edge Impulse.
2. Connect the target board to the host machine using a USB to TTL converter.
3. Run the "edge-impulse-data-forwarder" command on the host machine and select the port.
4. Connect the device and begin data collection by following the on-screen instructions.
5. Label the data as "left-right" and shake the board in that direction, ensuring you collect data at different speeds and motions.
6. Repeat this for "front-back" and "up-down" directions as well.

Refer to these links [2] and [3] to learn more about this procedure and the data forwarder in Edge Impulse.

Figure 4 shows the data acquisition in Edge Impulse with accelerometer data collected and labeled for different motions.

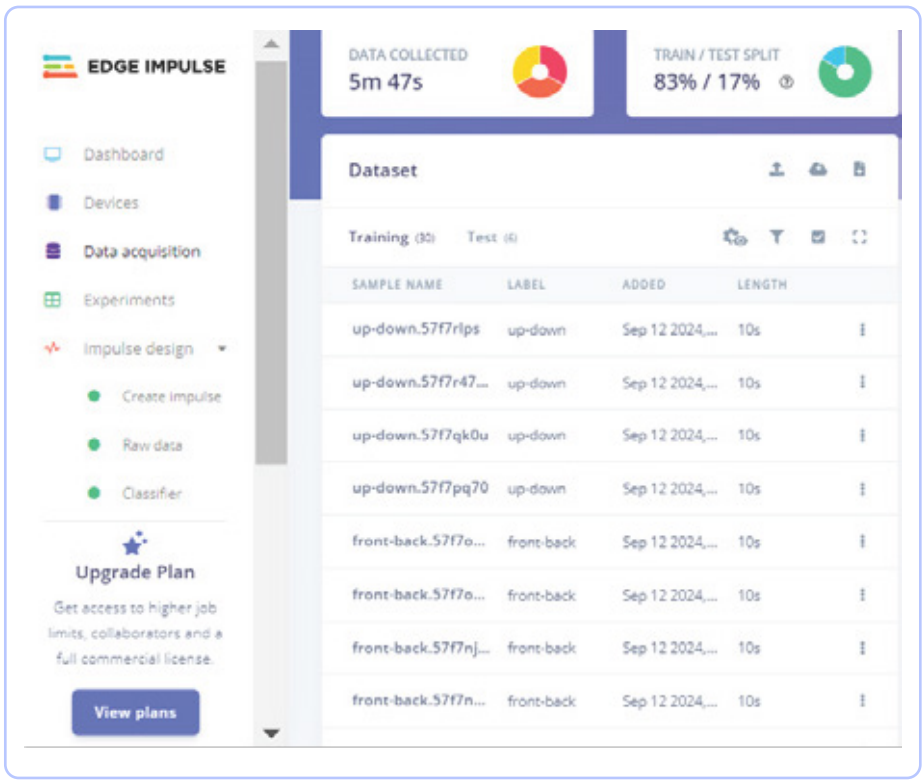

Figure 4 - Edge Impulse data acquisition

5.2 Model design

Model design is crucial as it involves selecting the right algorithms and configurations to interpret the accelerometer data effectively and design the model. In Edge Impulse, this can be facilitated using prebuilt processing blocks and learning blocks. The processing block extracts meaningful features from the input accelerometer data and the learning block trains the model. For gesture recognition applications, raw data was chosen as the processing block and classification as the learning block. For more information, refer to [4].

5.3 Model training

Model training in Edge Impulse is a streamlined process where the collected data is used to train the model using the platform's integrated training pipeline. Figure 5 shows the model training parameters after the features are extracted.

Figure 5 - Model training parameters

After the features are extracted, the model needs to be trained by setting the number of training cycles. The end of this training cycle generates a confusion matrix. The confusion matrix evaluates a model by comparing the actual values against the predicted ones. More data should be collected at various speeds and motions to improve the accuracy of the confusion matrix. Model optimizations may increase on-device performance but decrease accuracy.

Figure 6 and Figure 7 show the confusion matrix and the model performance for both optimized and unoptimized models, respectively.

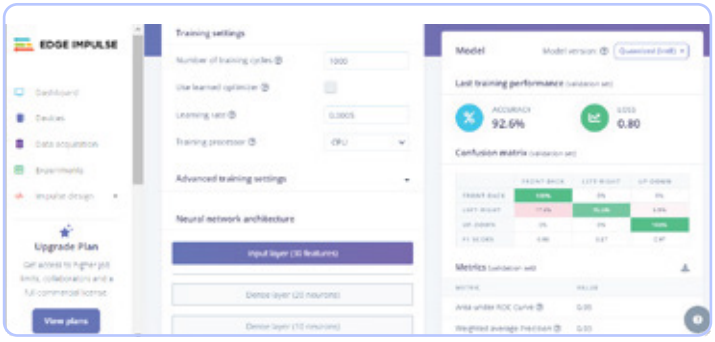

Figure 6 - Confusion matrix and model performance for optimized model

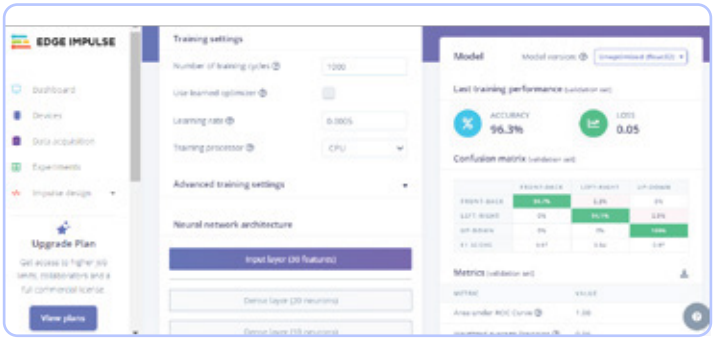

Figure 7 - Confusion matrix and model performance for the unoptimized model

5.4 Model testing

Model testing ensures that the trained model performs well on unseen data. Edge Impulse provides tools for validating the model using a separate test dataset and visualizing its performance through metrics like accuracy, precision and recall. Both optimized and unoptimized models can be used for model testing. This phase is critical to confirm that the model predicts well and is ready for deployment.

In the model testing section of Edge Impulse, clicking "Classify all" classifies all the test data and the live data and updates the results.

We have now developed and tested the model. The auto-generated code from Edge Impulse has some problems, which can be understood using the detailed case study of gesture recognition.

5.5 Gesture recognition – details

Below are the requirements for the gesture recognition application:

- Collect accelerometer data using the built-in accelerometer.
- Detect motions such as left-right, front-back and up-down movements.
- Perform real-time actions by activating different LEDs upon identifying the motions.
- Transmit the identified motion to the external system via UART.

To improve modularity, each requirement requires a separate task.

The development platform chosen for this application is the STM32F407Disc1 board, which features 1 MB of flash memory and 192 KB of RAM. Development is carried out using STM32CubeIDE, while the host machine operates on Ubuntu 22.04.

Figure 8 shows the Target board – STM32F407G-Disc1, connected to the host machine via a USB to TTL cable.

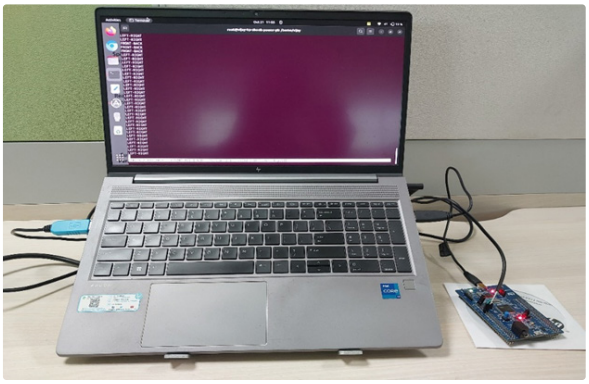

Figure 8 - STM32F407G-Disc1 Setup with Host

5.6 Edge Impulse code generation – Problems

Next, let's generate the code in Edge Impulse. The deployment section offers two options: generating a complete application or a C++ library for integration into applications. Both options generate a complex, monolithic code which is difficult to understand and integrate. For instance, the README.txt of the generated code directs users to references [6], [7] and [8] for guidance, which shows the complexity of integration. Table 2 lists all the identified issues with this auto-generated code.

Table 2 - Problems identified with auto-generated code

Problem identified	Proposed solutions
The integration process is complex.	Utilize the model from Edge Impulse and develop your own inference application.
Difficult to read and understand	Develop your own code with clear comments, utilizing the Inference programming flow indicated in Table 3 and the inputs from the Netron app referenced in Figure 10. This approach enhances readability.

Monolithic code combining both data collection and inference functionalities clubbed together	Create a modular reference architecture that separates tasks into individual modules.
The code has all the parameters hard-coded.	Employing a JSON configuration file allows the task parameters to be configurable.
Implementing new features like activating the LED and communication tasks is challenging.	With a modular architecture, a new module can be easily added.
Testing individual functionalities is challenging due to the code's monolithic structure.	With a modular architecture, each module can be independently tested by changing the inputs and verifying the expected outcomes. This approach also removes the need for other modules to be available during testing.
Not reusable for similar software or product lines	With a modular architecture, the required modules can be reused for similar software or product lines

Next, we will see how implementing a reference architecture addresses each of the above problems.

6. Model deployment phase

The objective of the model deployment phase is to complete the necessary steps and deploy the model into the reference architecture as an inference module. Figure 9 illustrates a series of tasks that we will see step by step.

Figure 9 - Tasks in model deployment phase

6.1 Model interpretation

This task is essential for comprehending the model and the different activation functions that are present in it. This information serves as input for inference programming. Netron is a graphical user interface tool that visually displays the model, helping developers understand its activation functions. You can install it or use it directly from online at https://netron.app/

Figure 10 shows the generated model of the gesture recognition application opened in the Netron app. The activation functions present in it are FullyConnected, Relu and Softmax.

6.2 Embed model

Due to the memory limitations of resource-constrained devices in TinyML, we can't use a file system to store the trained model file. Instead, a Linux command line tool called "xxd" can be used in the host machine to convert the model into an array, which can then be used in the inference program to access the model.

To illustrate, embedding a tflite-micro model file named ei-gr-freq10--float32-model. lite into memory requires the following command. This command generates an output file containing an array definition and its length, as shown below. This array needs to be used in the inference program.

```
xxd -i ei-gr-freq10--float32-model.lite > model.cc
```

The above command creates a file called model.cc containing the following content. Note that some parts have been omitted for clarity.

```
unsigned char ei_gr_freq10_float32_model_lite[] = {  0x1c, 0x00, 0x00,
0x00, 0x54, 0x46, 0x4c, 0x33, 0x14, 0x00, 0x20, 0x00,  0x1c, 0x00, 0x18,
0x00, 0x14, 0x00, 0x10, 0x00, 0x0c, 0x00, 0x00, 0x00,  0x08, 0x00, 0x04,
0x00, 0x14, 0x00, 0x00, 0x00, 0x1c, 0x00, 0x00, 0x00,  0x8c, 0x00, 0x00,
0x00, 0xe4, 0x00, 0x00, 0x00, 0x7c, 0x0f, 0x00, 0x00,  0x8c, 0x0f, 0x00,
0x00, 0xd0, 0x14, 0x00, 0x00, 0x03, 0x00, 0x00, 0x00,  0x01, 0x00, 0x00,
0x00, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0a, 0x00,  0x10, 0x00, 0x0c,
0x00, 0x08, 0x00, 0x04, 0x00, 0x0a, 0x00, 0x00, 0x00,
..... // some data omitted for brevity  0x00,
0x00, 0x00, 0x19, 0x0c, 0x00, 0x0c, 0x00, 0x0b, 0x00, 0x00, 0x00,  0x00,
0x00, 0x04, 0x00, 0x0c, 0x00, 0x00, 0x00, 0x09, 0x00, 0x00, 0x00,


  0x00, 0x00, 0x00, 0x09
};
unsigned int ei_gr_freq10_float32_model_lite_len = 5428;
```

6.3 Inference programming

Tensorflow Lite Micro is chosen due to its seamless integration with Edge Impulse, rich Tensorflow ecosystem and effective performance.

However, the documentation for programming with tflite-micro is limited. The sequence for any inference program using tflite-micro is shown in Table 3 below.

Table 3 - Inference programming sequence using tflite-micro

Step	Description
Read model	Load the TensorFlow Lite model file into memory using const tflite::Model* model = tflite::GetModel(ei_gr_freq10_float32_model_lite); ei_gr_freq10_float32_model_lite is the array generated in the previous step of the embedding model
Set up a resolver	Initialize the operator resolver to map the model's operations shown in the Netron app to implementations For example, the Netron output in Figure 10 uses three operations – FullyConnected, Relu, Softmax. The equivalent code is below tflite::MicroMutableOpResolver<10> resolver; resolver.AddFullyConnected(); resolver.AddRelu(); resolver.AddSoftmax();
Set up an interpreter	Create an interpreter instance to interpret and execute the model. It also does resource allocation for tensors. interpreter = new tflite::MicroInterpreter(model, resolver, tensor_arena, kTensorArenaSize); TfLiteStatus allocate_status = interpreter->AllocateTensors();
Set up input and output tensors	Define input and output tensors based on the model's requirements. input = interpreter->input(0); output = interpreter->output(0);

Fill input tensors	Populate input tensors with the input sensor data required for inference. Ensure data format and dimensions match model's expectations.
	input->data.f[i] = xi[i]; // x-value
	input->data.f[i] = yi[i]; // y-value
	input->data.f[i] = zi[i]; // z-value

Invoke interpreter	Execute inference by invoking the interpreter. interpreter->Invoke();
Extract the probability prediction	etrieve the predictions from the output tensor after inference. Analyze the output tensor to extract the probability distribution. float left_right_prob = (output->data.f[1]); float front_back_prob = (output->data.f[0]); float up_down_prob = (output->data.f[2]);

Infer result	Interpret probability predictions to derive the result. Map probabilities to meaningful labels or outcomes. Check which probability has the highest score to determine the motion prediction.

6.4 Deploy inference

To deploy the inference, a reference architecture is proposed to tackle the problem of missing non-functional requirements like readability, modifiability, configurability, reusability and testability.

6.4.1 Reference architecture

The proposed reference architecture is modular and incorporates the idea of small, autonomous services from microservices architecture. It also utilizes the messaging pattern from the publish-subscribe architecture, where each task acts as a publisher or subscriber. Hence, it can be called a micro-task-based architecture.

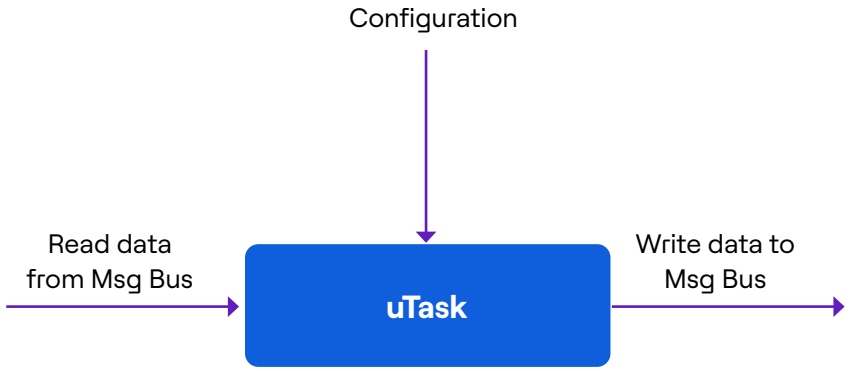

Figure 11 - Structure of a uTask

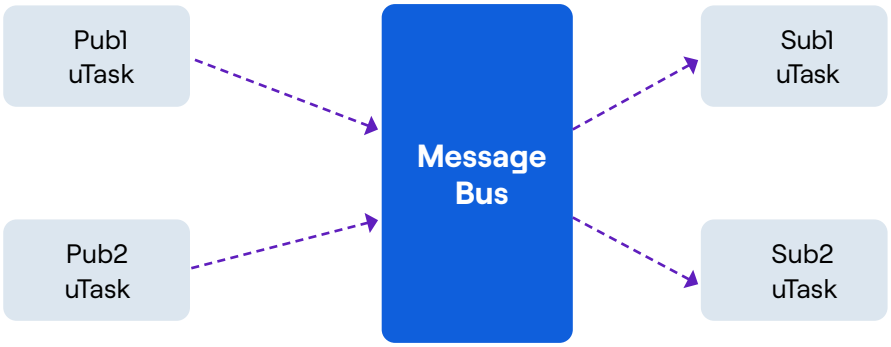

Figure 12 - messaging pattern

The components of this architecture involve a message bus and a group of micro-Task or uTasks that act either as publishers or subscribers, as shown in Figures 11 and 12. This design ensures seamless communication and data exchange between uTasks, enabling a flexible and modular system that can be easily maintained and scaled. The message bus is the backbone, facilitating the decoupling of uTasks and enabling asynchronous communication, which is essential for the smooth operation of resource-constrained devices.

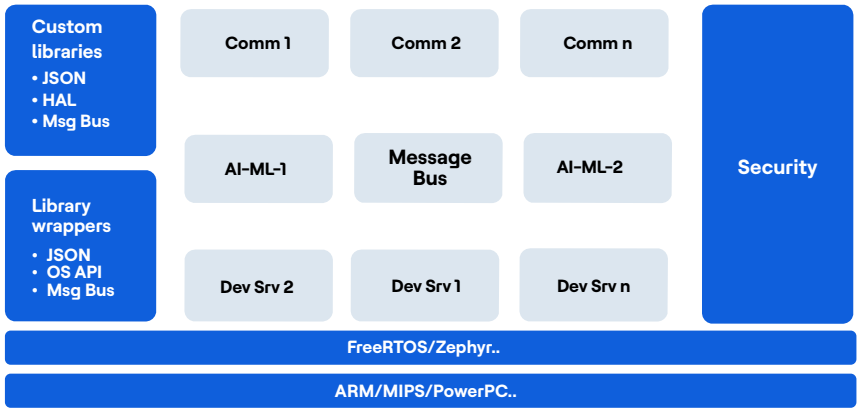

Figure 13 - Overview of the reference architecture

Figure 13 shows an overview of the reference architecture. At startup, each uTask loads a JavaScript Object Notation (JSON) configuration and sets itself up according to the specified parameters. These parameters consist of topics for publishing and subscribing, which address the problem of configurability. Subsequently, the uTask subscribes to the message bus for the designated read topics. After processing, it writes or publishes data back to the message bus.

6.4.1.1 Device service

These south-bound uTasks interact with end devices, read data and publish it to the message bus. It uses HAL to communicate with the end devices.

6.4.1.2 Communication service

These north-bound uTasks subscribe to the message bus for data, read it and then relay the information to external systems.

6.4.1.3 Message bus

A critical component of this architecture is the lightweight message bus, which serves as a channel for communication between various tasks. The message bus employs a publish-subscribe mechanism, allowing uTasks to decouple their operations while maintaining a robust flow of information.

The message bus facilitates asynchronous communication, allowing uTasks to operate independently, enhancing the system's modifiability and scalability. This modular approach not only aids in the system's maintainability but also allows for easy integration of new tasks without disrupting the existing workflow. As the uTasks operate independently, they can be tested by simulating the inputs and verifying the expected results without depending on other uTasks.

6.4.1.4 AI-ML

These analytics uTasks read data from the message bus, perform inference using a deployed model and then publish the results back to the message bus.

6.4.1.5 Custom libraries

This architecture is flexible, allowing us to use our own libraries to implement functionalities like the message bus, Real-Time Operating System (RTOS) and JSON configuration. These customizable libraries can be replaced as needed.

6.4.1.6 Library wrappers

This layer encapsulates all the library APIs, ensuring every task interacts with this wrapper. This design allows us to switch libraries in the future without modifying the uTasks themselves.

6.4.1.7 RTOS

A lightweight RTOS such as FreeRTOS can leverage its scheduling capabilities, ensuring tasks are executed promptly and efficiently. If RTOS is not used, we might need to develop our own scheduling algorithms, which again demands significant effort.

6.4.1.8 Security

This security block indicates that we can implement security according to the requirements. However, it is outside the scope of this document.

6.4.1.9 Processor

It denotes any microcontroller with limited resources.

6.4.2 Gesture Recognition – Architecture

Figure 14 illustrates the proposed reference architecture employed for the gesture recognition application mentioned in the section Gesture recognition - Details

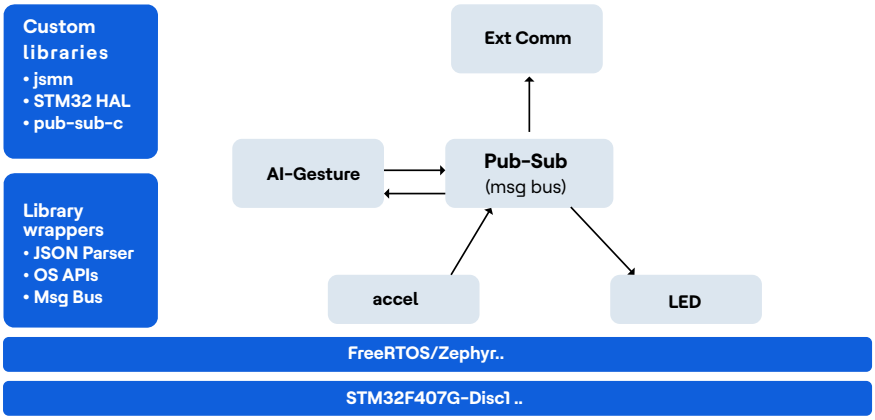

Figure 14 - Gesture recognition architecture

The application has been divided into four uTasks and the proposed reference architecture has been used. These uTasks are reusable across product lines as they are modular and independent of other uTasks. This also enables the addition of new uTasks, such as LED and Ext Comm, apart from data collection and analytics from the auto-generated code. Let us see all four uTasks one by one.

6.4.2.1 accel

This south-bound uTask uses STM32 HAL, reads the accelerometer data and publishes it to the message bus.

6.4.2.2 AI-Gesture

This AI uTask reads the accelerometer data published by accel, runs inference on it and publishes the inference result back to the message bus.

Tflite-micro is used for inference operations. The source code is available in https://github.com/tensorflow/tflite-micro . It is written mostly in C++.

6.4.2.3 LED

This south-bound uTask reads the inference result published by AI-Gesture uTask and glows different LEDs as shown in Table 4.

Table 4 - LED Color for various motions

Motion	LED Color
Left-Right	Red
Front-back	Green
Up-Down	Blue

6.4.2.4 Ext Comm

This north-bound uTask reads the inference result published by the AI-Gesture uTask and forwards the data to an external system through UART.

All these uTasks can be reused for similar software or product lines as they function as independent modules.

6.4.2.5 Pub-Sub library

This is the message bus library used for this project. The Github path is https://github.com/jaracil/pubsub-c. It is written in C and has an MIT license.

The various topics used in this message bus along with the relevant publishers and subscribers are shown in the below Table 5.

Table 5 - Topics with Publishers and Subscribers

Publisher	Topic	Subscribers
accel	x, y, z values	AI-Gesture
AI-Gesture	Inference	LED, Ext Comm

6.4.2.6 JSMN Library

The GitHub for the JSON library is https://github.com/zserge/jsmn. It is written in C and has an MIT license. All the uTasks read a JSON configuration that was embedded into the code using the xxd tool. JSMN library was used to read and parse these JSON configurations.

STM32Cube IDE

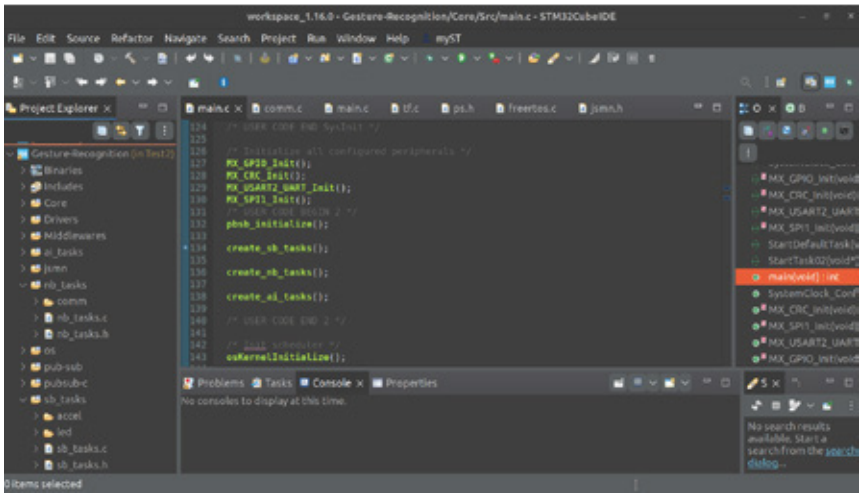

Figure 15 – STM32IDE with Gesture Recognition

STM32CubeIDE needs to be setup which provides the HAL for STM32F407Disc1 board. All the downloaded libraries need to be integrated into this environment. Figure 15 shows the STM32 IDE screenshot.

6.5 Test results

After building and testing, it was found that all the uTasks functioned correctly as expected and the inference uTask accurately predicted the output. Figure 16 is the output of minicom, which receives data from comm uTask through a USB to TTL converter. It shows that comm uTask received the final prediction for all motions, such as LEFT-RIGHT, FRONT-BACK and UP-DOWN.

6.5.1 Size

As shown in Figure 17, the application's binary size was 191 KB.

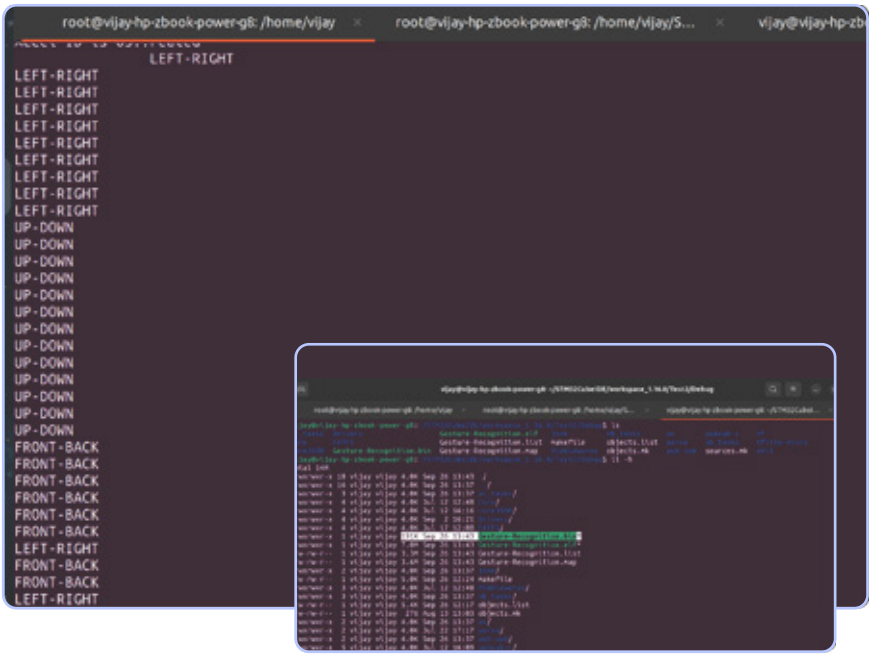

Figure 16 – Minicom Output of comm uTask

Figure 17 – Gesture recognition binary size

6.5.2 Memory consumption

To find out the RAM usage of this application, the command "arm-none-eabi-size Gesture-Recognition.elf" can be used, which shows the output in Figure 18. The memory region breakdown and consumption are shown in Tables 6, 7 and 8.

Figure 18 - Output of arm-none-eabi-size

Table 6 - Memory regions

Memory region	Description
text	The size of the code (Flash memory usage)
data	The initialized global and static variables (stored in RAM and occupies Flash)
bss	The uninitialized global and static variables (stored in RAM)
dec	The total size of text + data + bss in decimal
hex	The total size in hexadecimal

Flash usage = text + data

Section	Size (bytes)	Size (KB)
Text section	187,624	183.23
Data section	7,500	7.3
Total flash usage	195,124	**191 KB** approx.

RAM usage = data + bss

Section	Size (bytes)	Size (KB)
data section	7,500	7.3
bss section	60,024	58.6
Total RAM usage	67,524	**66 KB** approx.

7. Benefits

There are numerous advantages to adopting this architectural solution for TinyML. This approach is not restricted to a specific domain and can be applied across various fields. By employing Edge Impulse for data collection and model training, there is no need to develop standalone software for these tasks, which saves huge amounts of effort. Additionally, Edge Impulse offers ease of use with comprehensive documentation and strong community support. A ready-made reference architecture reduces development time, facilitating quicker product launches and enabling businesses to respond swiftly to market demands and secure a competitive advantage. Implementing non-functional requirements such as modifiability, readability, configurability, reusability and testability also ensures the software's stability and quality. The maintenance team ensures the reference architecture software is reliable and can fix the bugs, reducing the load on the project development team. Figure 19 consolidates these benefits.

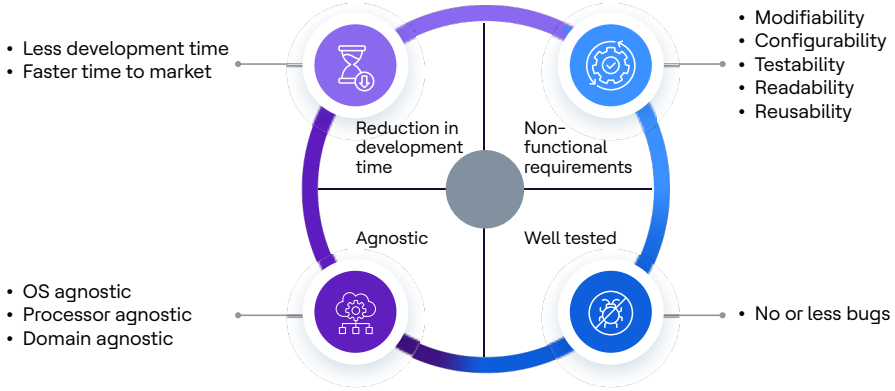

Figure 19 - Benefits of the architectural approach for TinyML

8. Conclusion

In conclusion, addressing the need for more support for non-functional requirements and the absence of standardized approaches in TinyML development is crucial. The problems identified with using the existing solutions have been listed in Tables 1 and 2. By proposing a solution with a reference architecture and clear guidelines, this paper aims to overcome these challenges, facilitating the efficient development and deployment of TinyML applications in constrained edge gateways. The results and the memory consumption show that this architectural approach is highly efficient and reliable. This approach enhances readability, modifiability, configurability, reusability and testability and ensures a more streamlined and effective process for data collection, model training, model generation and inference development.

9. References

1. https://docs.edgeimpulse.com/docs (Edge Impulse Documentation)
2. https://docs.edgeimpulse.com/docs/tools/edge-impulse-cli/cli-data-forwarder
3. https://docs.edgeimpulse.com/docs/edge-impulse-studio/data-acquisition
4. https://docs.edgeimpulse.com/docs/edge-impulse-studio/impulse-design-and-experiments
5. https://community.silabs.com/s/article/32-bit-flash-ram-usage?language=en_US (Memory Consumption details)
6. https://docs.edgeimpulse.com/docs/deploy-your-model-as-a-c-library
7. https://docs.edgeimpulse.com/docs/running-your-impulse-locally-1
8. https://docs.edgeimpulse.com/reference/inferencing-sdk

10. Author information

Vijay Annamalaisamy

Vijay is an Embedded Software Architect at HCLTech with extensive experience in architecting, designing and developing embedded software across various domains. He has strong expertise in products related to edge computing, IoT protocols and cybersecurity. Vijay is part of the Consulting and Solutioning team within the Embedded and Testing Capability Unit.

Sathya Durai

Sathya is a Solution Director at HCLTech. He has extensive experience in architecting, designing and developing embedded software for products across various domains. He is skilled in solving challenges related to products from multiple industries. Sathya is part of the embedded and application team within the Embedded and Testing Capability Unit.

HCLTech | Supercharging Progress™

HCLTech is a global technology company, home to more than 220,000 people across 60 countries, delivering industry-leading capabilities centered around digital, engineering, cloud and AI, powered by a broad portfolio of technology services and products. We work with clients across all major verticals, providing industry solutions for Financial Services, Manufacturing, Life Sciences and Healthcare, Technology and Services, Telecom and Media, Retail and CPG, and Public Services. Consolidated revenues as of 12 months ending December 2024 totaled $13.8 billion. To learn how we can supercharge progress for you, visit hcltech.com.

hcltech.com