BoilerNet: A Compute-Enabled Mini-NAS

Akshath Raghav Ravikiran, Aneesh Reddy Poddutur,
Gautum Kottayil Nambiar, Gokulkrishnan Harikrishnan

02.02.2025

GTA: Jack Garrett Blowers
Professor: Ryan Beasley

Design Document

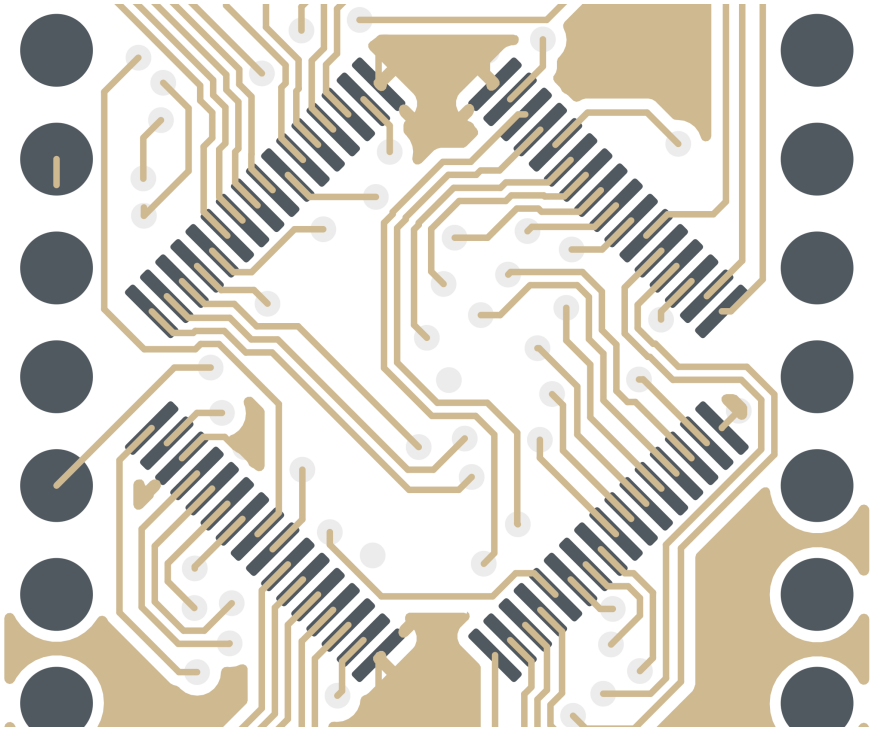

Figure 0.0.1: BoilerNet Logo

Contents

List of Figures

List of Tables

Revision Log

Date	Revision	Changes
2/2/2025	v1.0	Design Document 1 Completion
3/15/2025	v2.0	Design Document 2 Completion
4/20/2025	v3.0	Design Document 3 Completion

Table 1: Revision Log

Glossary

- **NAS** - Network Attached Storage. A system, of varying complexity, that is made primarily for data storage purposes.
- **A.P.I. (API)** - Application Programming Interface.
- **M.C.U. (MCU)** - Micro Controller Unit.
- **I/O File** - Smallest unit of data that is utilized within the cluster. For the current use-case, this is a PNG File.
- **Job** - Collection of I/O Files given by or to the user. Buffered within the API Backend.
- **Task** - Each I/O File present in a job. Only one task exists within the system at any given moment.
- **Node** - Individual device or computer that is part of the network.
- **Network Node** - Nodes required for receiving incoming/outgoing Tasks.
- **Internal Switch Node** - Nodes focusing on buffering and storing patches.
- **Compute Node** - Nodes used specifically for computational workloads.
- **Compute Cluster** - Group of Compute Nodes that can parallelize computation tasks.
- **Distributed System** - System where multiple independent nodes work together to achieve a common goal.
- **Operation** - User-selected operation to be performed on I/O files in the job.

1 Introduction

1.1 Executive Description

This project aims to develop a Network Attached Storage (NAS) system utilizing ESP32 MCUs. The aptly-named BoilerNet aims to enable In-Network-Compute alongside swappable disks and compute workloads, neatly brought together by a cloud-hosted Dashboard interface. There are a few core ideas that form the foundation for this "cluster" – Low Power Usage, Plug-and-Play Workloads, High Scalability.

1.2 User Stories

Dr. Jenny Matthews

Dr. Jenny Mathews is a marine biologist conducting research on coral reef ecosystems in remote locations. During her expeditions, she collects vast amounts of data from drones and sensors, including environmental metrics like water temperature, salinity, and pH, as well as high-resolution imagery of reefs.

To process this data, Dr. Mathews would like a compute solution designed to handle tasks in the field, such as filtering noisy sensor data, aggregating data streams, and manipulating collected image data. She would also like to swap out the workloads she runs for fixed periods of time. Sometimes, she prefers to retrain her models for better accuracy.

The compute solution would ideally utilize less power than her computer since she often finds herself in energy constrained situations, such as being aboard a research vessel that relies on solar power. Her workloads are not bound by low-latency requirements, but do require fast and remote access to data. Moreover, Dr. Matthews has very limited internet connectivity, so sending data to a cloud computing server is not feasible. Along with that, her research is a long-term expedition, so the costs of cloud computing would add up over time.

Jeffrey Smith

Jeffery Smith, a graduate student studying wildlife, frequently collects image data of animals in the field. The images need to be stored in a collective drive used by his collegues. This data cannot be stored on Google Drive or Microsoft OneDrive due to data privacy concerns. Due to his limited budget, he also seeks a low-cost, energy-efficient solution for storing a redundant copy of these photos.

He has one set stored on his laptop, but wants to have a secondary storage device that integrates seamlessly into his workflow and is accessible by others in his lab. The solution should allow him to transfer images efficiently over a wireless connection, and into a disk that he can quickly eject as need be.

2 Design Requirements

2.1 Requirements

1. The Frontend User-Interface will allow the User to choose the Operation and define a Job with a (limited) number of Tasks for the cluster to work on.

2. The Backend will be responsible for buffering Jobs and sending in Tasks serially into the Network Node. It will contain a message queue, working in a Last-In-First-Out manner.

3. The Network Node will send the job type, along with the task to the Switch that is not-in-use or is least recently accessed. This decision is taken dynamically.

4. The Internal Switch Nodes will be responsible for managing an internal file directory and lookup table for storing files.

5. The Internal Switch Nodes will also be responsible for coordinating the Task transfer into the Compute Nodes, as well as data aggregation for results. It will also be responsible for tagging the results with the actual Task on-disk.

6. The Compute Nodes must run a full Deep Neural Network, meant for object classification. They must also need to efficiently manage on-chip data accesses to prevent failures.

7. The Compute Nodes are to be developed onto Cartidge-like PCBs, allowing for slotting and unslotting with ease.

8. The Compute Cluster, as a whole, must be contained within it's 3-D printed enclosure, made in Black and Gold colors.

2.2 Factors influencing requirements

2.2.1 Public Health, Safety, and Welfare

1. The system must be safe to use, with protection against electrical hazards, overheating, and physical damage.
2. It should work reliably in different environments without posing risks to users.

2.2.2 Global Factors

1. The design should work in different regions, supporting various power sources and conditions.
2. It should follow open standards so it can integrate easily with other technologies.

2.2.3 Cultural Factors

1. The system should be easy to use, regardless of the user's technical background.
2. It should allow customization to fit different needs and preferences.

2.2.4 Social Factors

1. The system should be simple to set up and maintain, even for non-experts.
2. It should enable collaboration by means of good documentation.

2.2.5 Environmental Factors

1. It should utilize power at a rate slower than the average laptop, defined by current CPU standards, ensuring working in low-energy conditions.
2. Sustainable materials and proper disposal methods should be considered for recycling of parts.

2.2.6 Economical Factors

1. The system should be affordable to build, use, and maintain.
2. It should be modular so users can upgrade parts without replacing everything.

3 System Overview

3.1 System Block Diagram

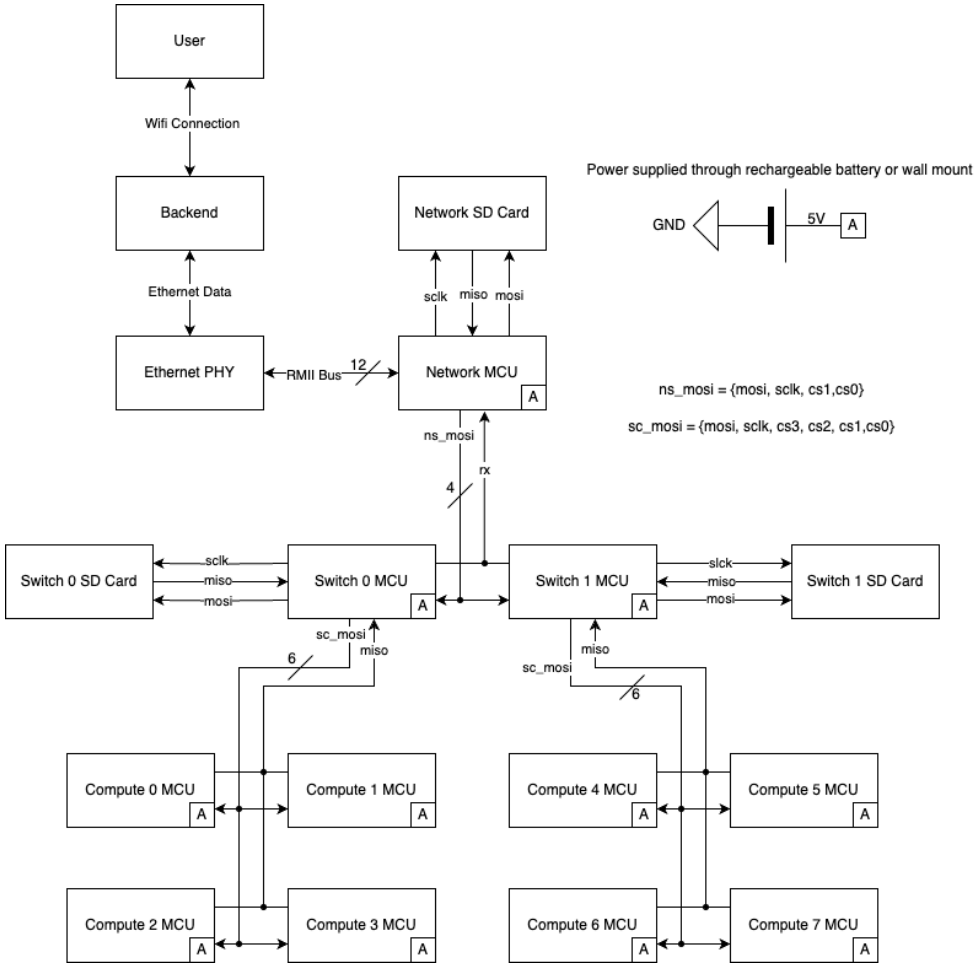

Figure 3.1.1: System Block Diagram

3.2 System Activity Diagram

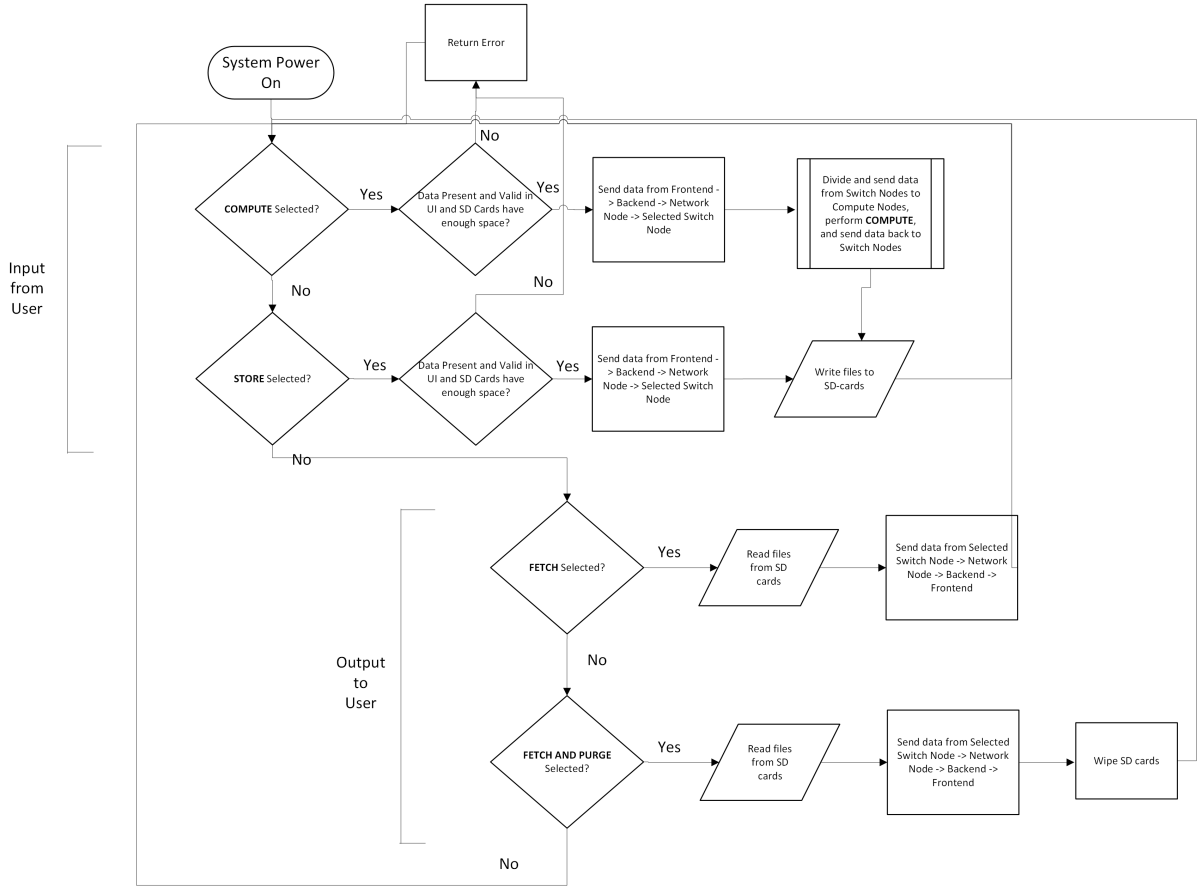

Figure 3.2.1: System Activity Diagram

3.3 System Mechanical Design (Extra Credit)

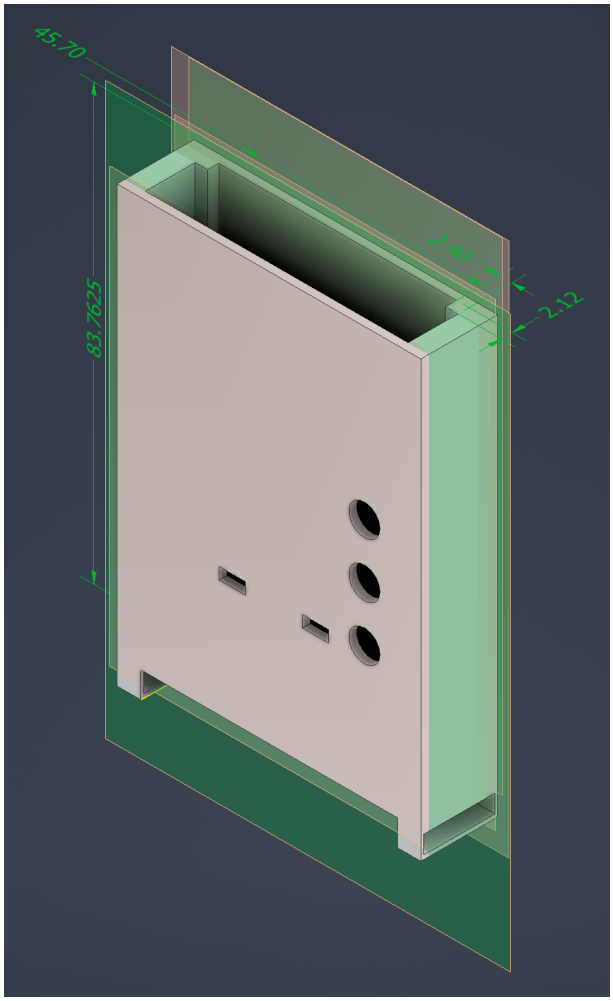

Figure 3.3.1: Compute Node Enclosure

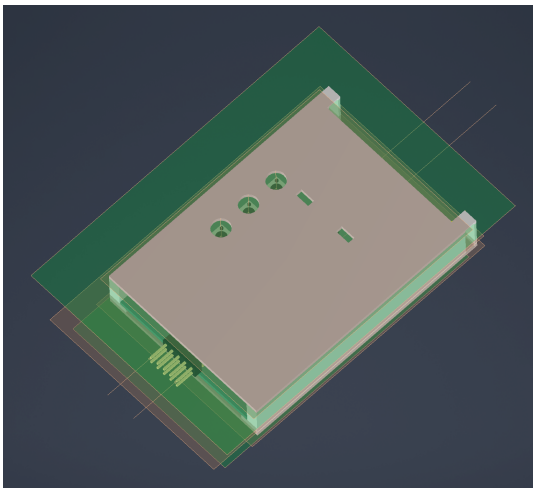

Figure 3.3.2: Compute Node Enclosure w/ Compute Node

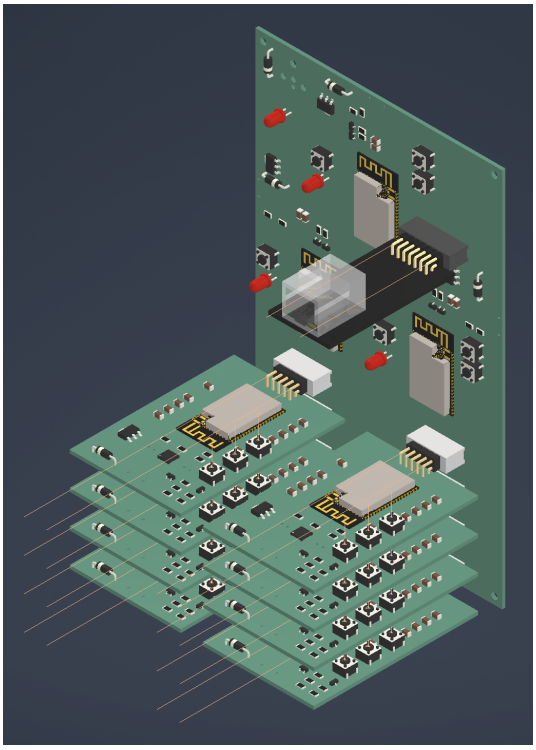

Figure 3.3.3: Final Node Assembly without Enclosures

3.4 Integration Approach

Frontend to Backnd

- The Frontend is built on the Streamlit framework
- The Backend is built on the FastAPI framework, and has endpoints defined in a REST manner.
- The Frontend is hosted on Streamlit's Cloud Service, and calls the Backend's endpoint URLs for the respective communications.

Backend to Network Layer Integration:

- The Backend communicates with the Network Node through a Router. The Router is necessary since the Network Node acts like a DHCP client, and requires a DHCP Server to register with an IP Address.
- The Network Node has a LAN 8720 PHY Board attached, to allow TCP over Ethernet using an RMII Interface.
- The System has a two-hop routing process, where the Backend integrates a load-balancer as a middle-layer to choose which BoilerNet system to route to. To simplify, the system does not allow for a direct Frontend to Network Node connection for scalability reasons.

Network Layer to Internal Switch Integration:

- The Network Node communicates with the Switch Nodes via an SPI communication.
- The Network Node and Switch Nodes are placed on the Main PCB, such that their SPI connection can be contained entirely within the PCB traces. Length matching, impedance matching, and ground stitching are utilized to improve signal integrity.
- Both the Network Node and Switch Nodes act as independent bodies, made to dynamically interact as a mesh. This is different from traditional approaches of SDN (Software Defined Networking), where a Control Plane and Data Plane is employed. The current method was chosen to minimize control logic across layers, and localize decision making.

Internal Switch to Distributed Compute Integration:

- The Switch Nodes communicate with the Compute Nodes via SPI communication.
- The Compute Nodes are assembled on separate boards in order to achieve modularity, where they can be easily removed and added as required.

- Each Compute Node runs a Deep Learning Model, which inferences over the Task sent over by the Switch Node.
- The connection points are assembled in the form of a standard Compute Cluster Rack, in 4 rows and 2 columns.
- Each Switch Node utilizes a unique chip select and handshake line to isolate compute nodes during transactions.
- Each Switch Node is responsible for transactions with 4 of the Compute Nodes. Each of the Switch Nodes isolates its Compute Nodes in 1 column of the Compute Cluster Rack.
- Power and Ground reference are provided to the Compute Nodes through the connection point, such that the entire cluster can be powered with the barrel jack on the Main PCB.

3.5 System Photographs

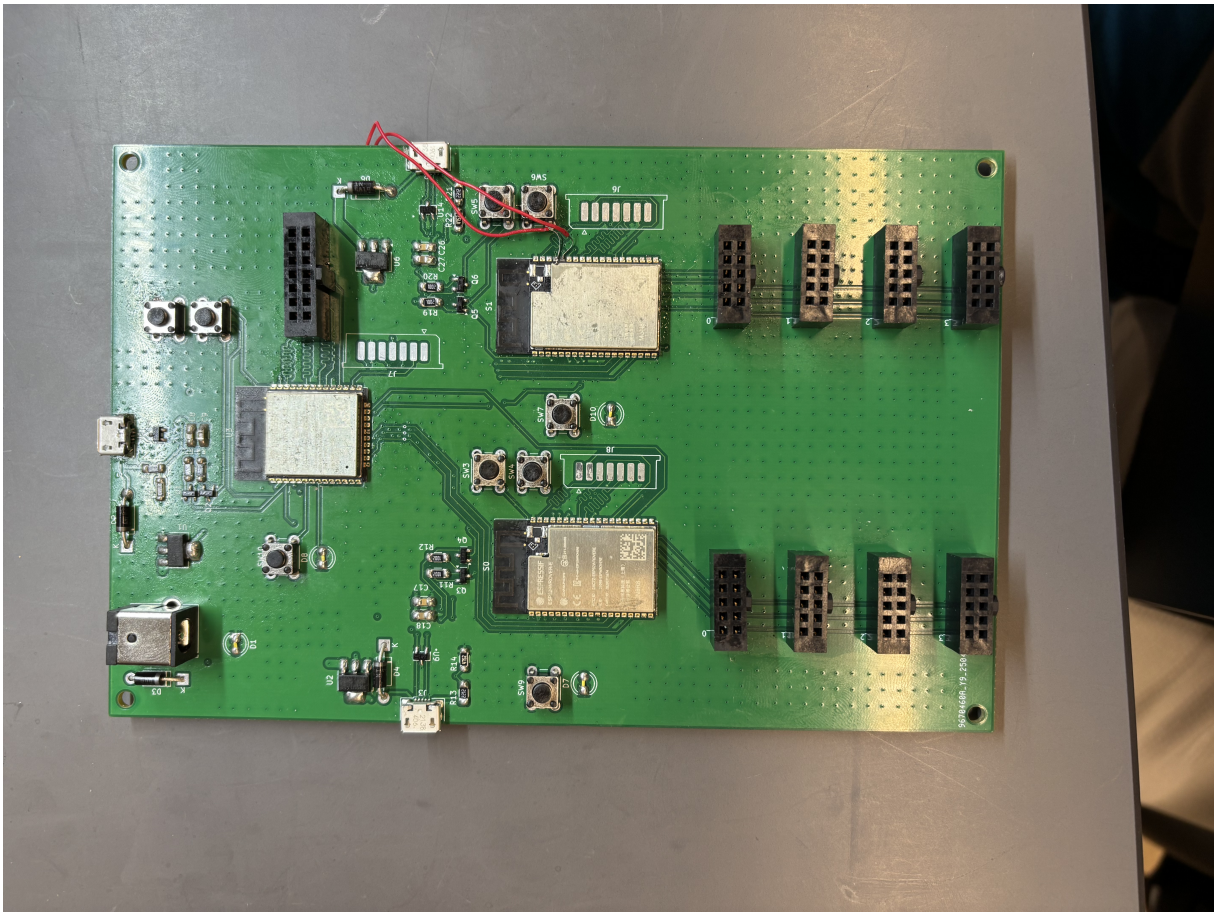

Figure 3.5.1: Network and Switch Board

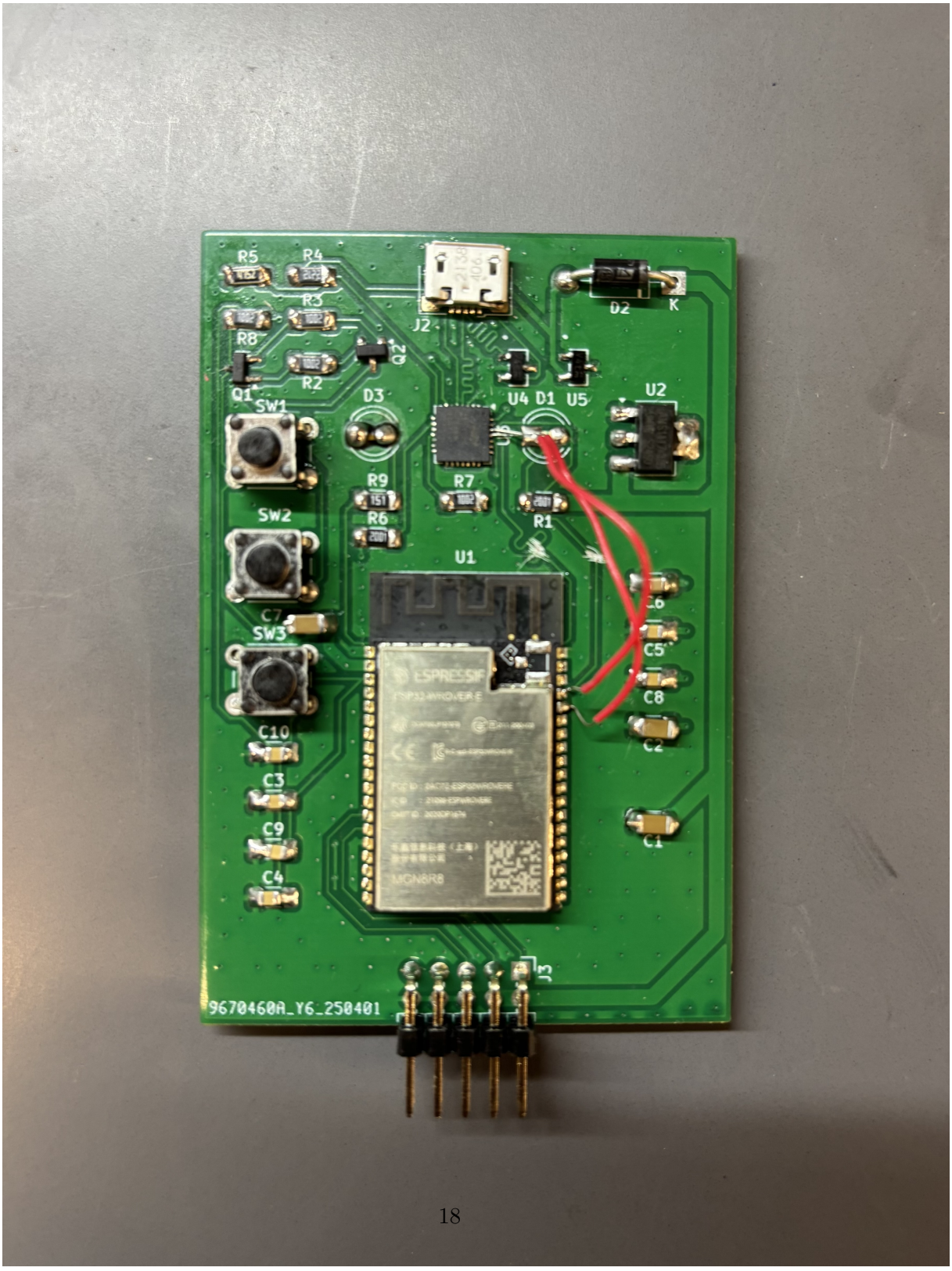

Figure 3.5.2: Single Compute Board

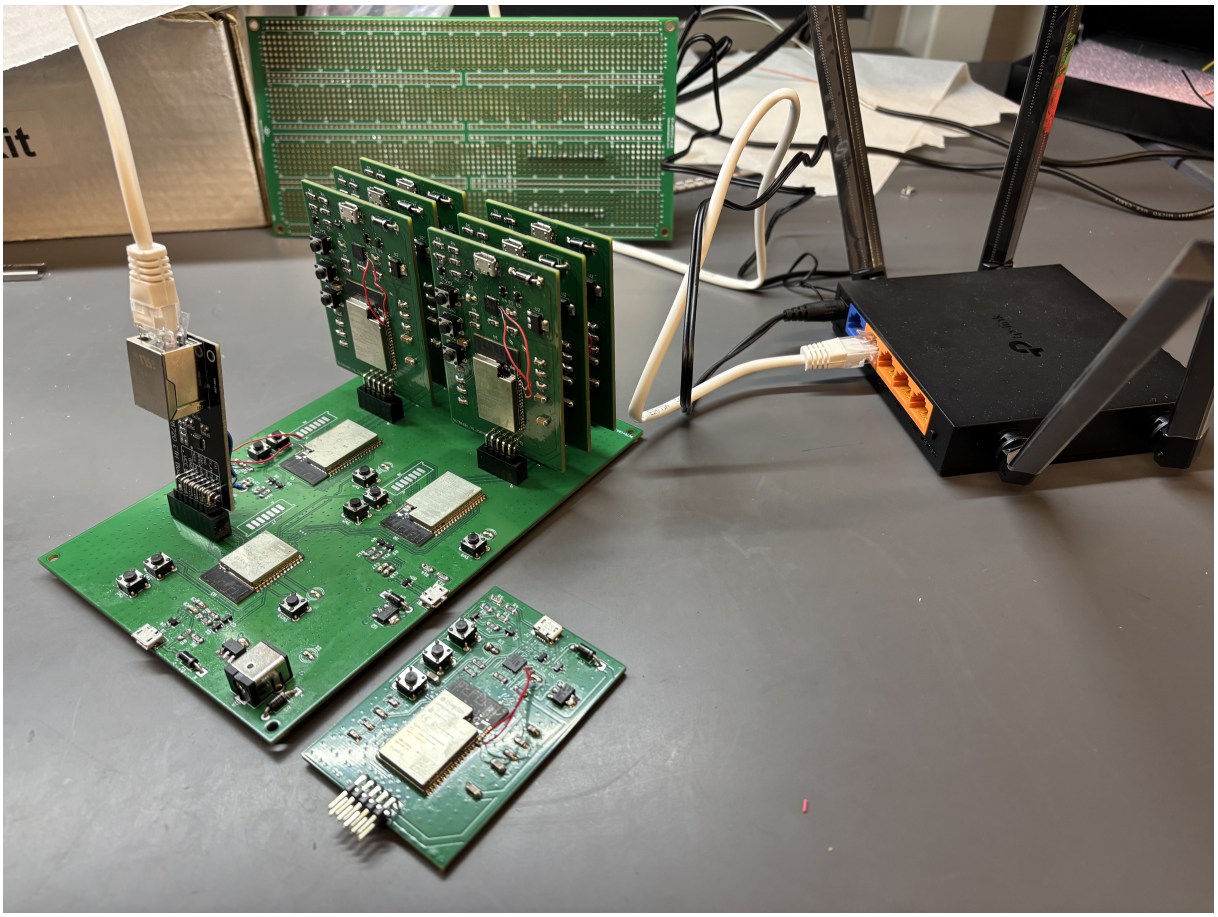

Figure 3.5.3: Assembled Compute Cluster

4 Subsystems

4.1 Subsystem 1: User-Interface / Back-End Layer

4.1.1 Subsystem Diagrams

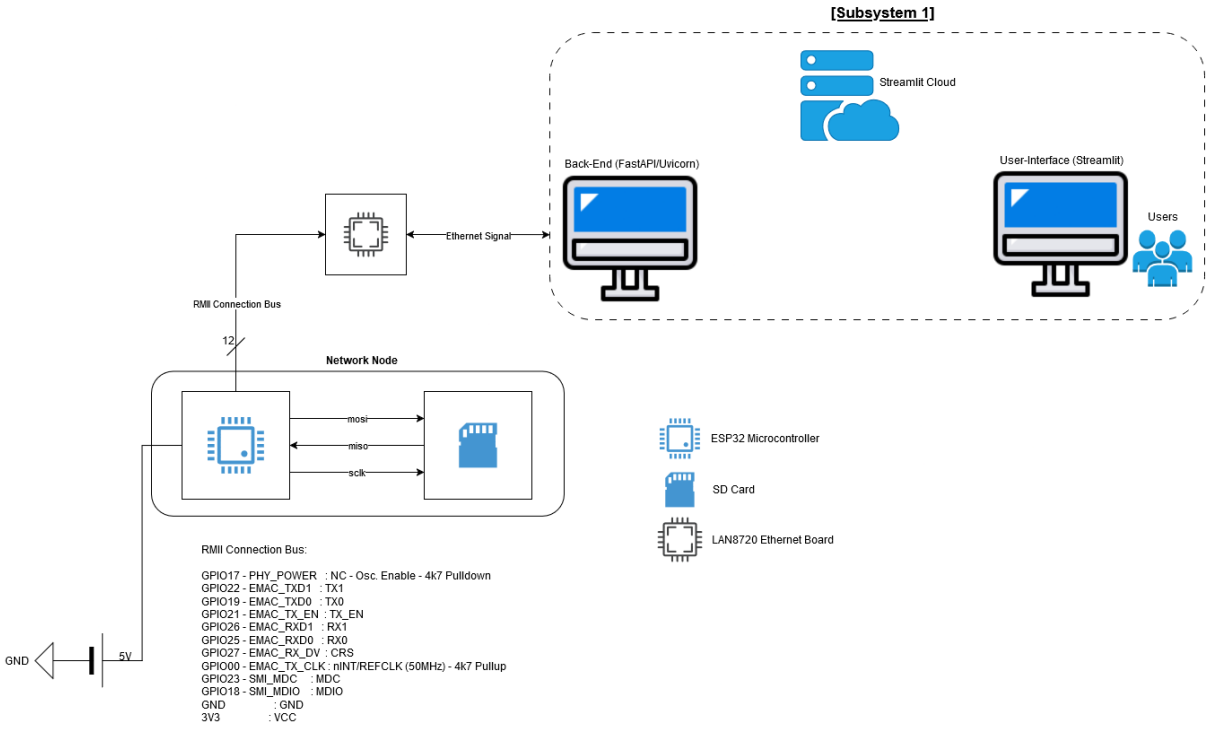

Figure 4.1.1: User-Interface Block Diagram

4.1.2 Specifications

1. The Front-End must be capable of accepting requests on any number of Tasks per Job.
2. Backend CPU Utilization should not exceed 30%.
3. The Backend must be able to complete a 2KB transfer into the Network Node in under 100ms.
4. Codebase passes Black (Python Code Formatter) with no errors.

4.1.3 Subsystem Interactions

This subsystem is broken into two parts.

1. The User-Interface is responsible for helping the Users navigate the system, help them understand what they can do with it, and accept their Tasks.
2. The User-Interface works with the Back-End to send data from Streamlit's Cloud Provisioning tools to the local-laptop-hosted Uvicorn server (interface with FastAPI framework). It needs to be reliable and send multiple tasks as bundled packets, ensuring compression as needed.
3. The Back-End will finally handle the chunking of the queue-d Tasks within the acquired Job for transmitting over the Ethernet protocol.

4.1.4 Core ECE Design Tasks

- **ECE 20875**: Practical understanding of applying Python to build error-free software, as well as Code Formatting standards.
- **ECE 36800**: Basics of Software-defined data structures, and utilizing them for asynchronously transmission queues.
- **ECE 36200**: Knowledge of SPI Protocol, and MCU programming/interfacing. Understanding of GPIO Pin Alternate Function modes.

4.1.5 Psuedo-Code

- **Front End:**
  - **Session:** Maintain a unique session ID, a job list, and a job counter.
  - **New Job:**
    - Upload up to 5 files (each $<$ 3MB) (if input task).
    - Generate a unique job identifier and create tasks from files (if input task).
    - Send tasks to the backend.
  - **Job Queue:**
    - List jobs (by number) and display task file names.
    - Allow multi-selection of tasks and choose an operation (Compute, Store, Fetch, Fetch and Purge).
    - Execute operations on selected tasks and update the UI.
    - Provide a button to delete the entire job.

- **Back End:**
  - **Models:**
    - **Task:** Contains task ID, job ID, file data, and filename.
    - **Event:** Contains event ID, task ID, job ID, command, status, and result.
  - **Global Structures:** Job Registry (maps job IDs to tasks and events), Event Queue, and Completed Event Store.
  - **Worker:** Continuously process events by:
    - Looking up the associated task,
    - Executing the command (store, compute, fetch, delete),
    - Updating the event status and result.
  - **Handlers:**
    - For each task operation: Check for an existing event; if missing, create and enqueue one; return status/result.
    - For job deletion: Immediately remove all non-processing tasks and events from the job.

4.1.6 Parts

- 1x Ethernet Cable - RJ45 Connector
- Laptop/Desktop with 8GB Ram and 512GB Disk

4.1.7 Algorithm

1. **Static IP Detection & Handshake**: Uses ARP (Address Resolution Protocol) to identify ESP32's static IP. Performs a TCP handshake (3-way handshake: SYN, SYN-ACK, ACK) to establish a reliable connection.
2. **HTTP Request Handling (REST API)**: Uses FastAPI's AsyncIO model to process multiple requests efficiently without blocking execution.

4.1.8 Theory of Operation

- **Job and Task Management:**
  - Users create a Job by uploading up to 5 files (each under 3MB); each file becomes an individual Task.

  - Tasks are grouped under a unique Job identifier and stored in a registry, on disk.

- **Task Processing and Event Handling:**
  - Each Task operation (Compute, Store, Fetch, or Fetch and Purge) is initiated by the UI, which generates an associated Event.
  - Events are enqueued for processing; a background worker processes these events, updates status, and stores results.

- **Job and Task Deletion:**
  - Users may delete individual Tasks or the entire Job.
  - Job deletion immediately removes all non-active Jobs from the registry.

- **System Resilience:** The system ensures real-time status monitoring and automatic clearance of pending operations.

4.1.9 Specifications Measurement

1. The Front-End must be capable of accepting requests on any number of Tasks per Job.

   The following images outline the ways in which the Front-End UI provisions for a user wanting to perform specific operations on N tasks per job. To elaborate, this means that the UI is able to allow selection of any number of tasks within a job, and gives the user to perform any of the operations on them collectively.

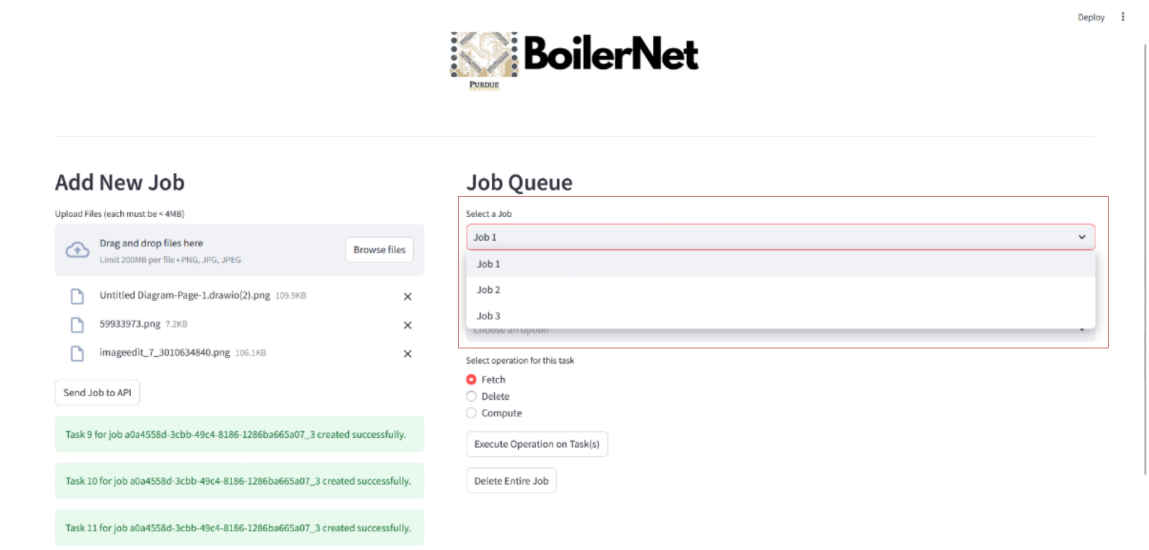

Figure 4.1.2: Front-End View - Multiple Task Selection

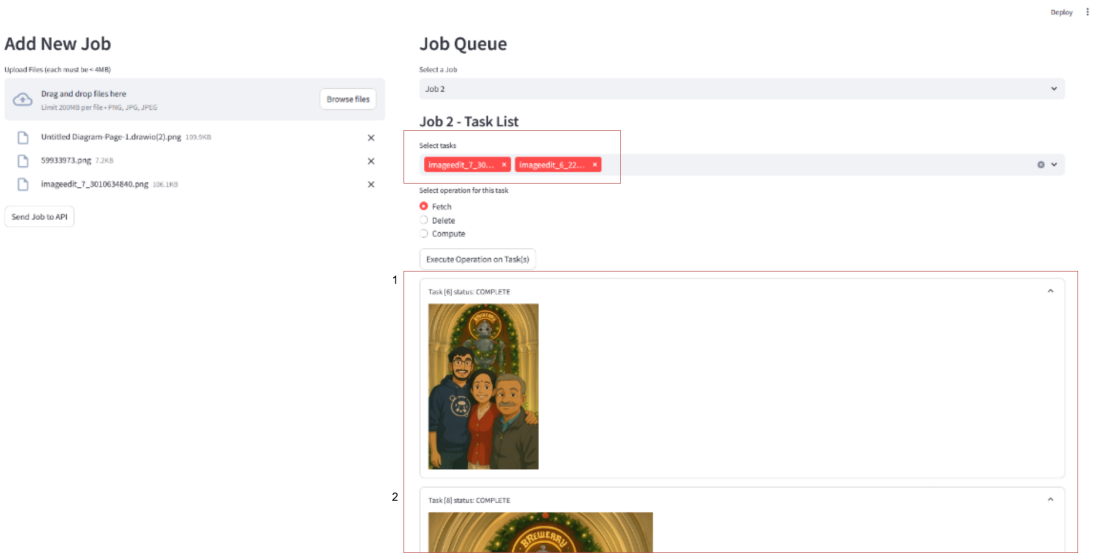

Figure 4.1.3: Front-End View - Multiple Task Fetch

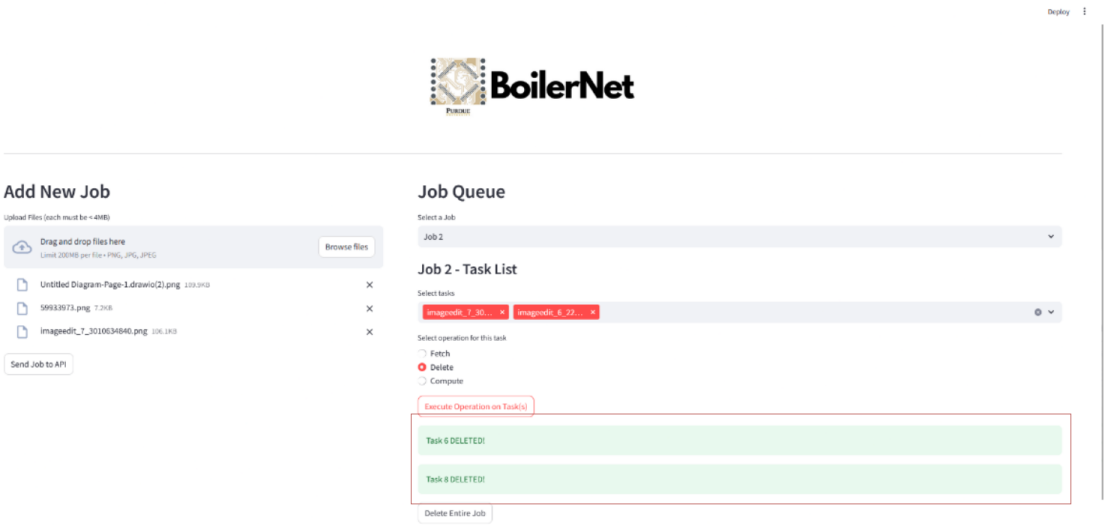

Figure 4.1.4: Front-End View - Multiple Task Delete

2. Backend CPU Utilization should not exceed 30%.

   We provision for this specification by rate-limiting the user to no-more than 5 Tasks per Job, as well as ensuring that each Task is at-most 4MB. Beyond this, we encounter a scalability problem, which is taken care of by our load balancer in the cloud. This service is provided to us, but we do not ancitipate our cluster to be used in a high-traffic environment.

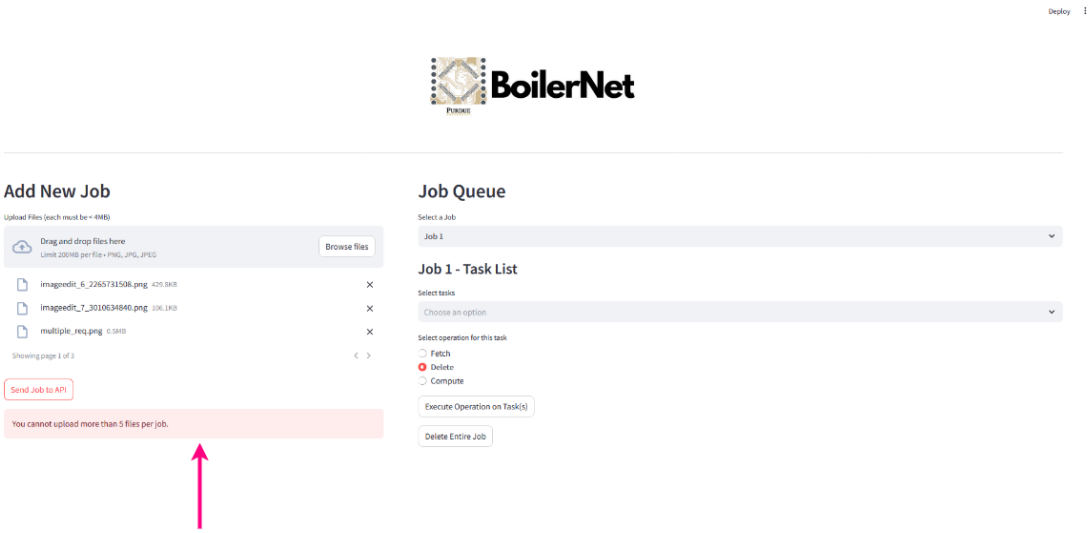

Figure 4.1.5: Front-End View - Rate Limiting Users

3. The Backend must be able to complete a 2KB transfer into the Network Node in under 100ms.

4. Codebase passes Black (Python Code Formatter) with no errors.

   The following image confirms that our codebase adheres to all of Python Black's formatting requirements.

```
(venv) (base) root@araviki:/home/araviki/workbench/boilernet/streamlit_app# black --check app.py
All done! ✨ 🍰 ✨
1 file would be left unchanged.
(venv) (base) root@araviki:/home/araviki/workbench/boilernet/streamlit_app# cd /home/araviki/workbench/boilernet/fastapi_backend
(venv) (base) root@araviki:/home/araviki/workbench/boilernet/fastapi_backend# black --check main.py
All done! ✨ 🍰 ✨
1 file would be left unchanged.
```

Figure 4.1.6: Black Formatter Confirmation

4.1.10 Standards

- **IEEE 802.3 (Ethernet Standard)**: Ensures reliable wired communication between the laptop, router, and network node ESP32.
- **RFC 2616 (HTTP 1.1) / RFC 9110 (HTTP 2.0)**: Governs data transmission between the FastAPI backend and Streamlit frontend using RESTful communication.

4.2 Subsystem 2: Network Coordination Layer

4.2.1 Subsystem Diagrams

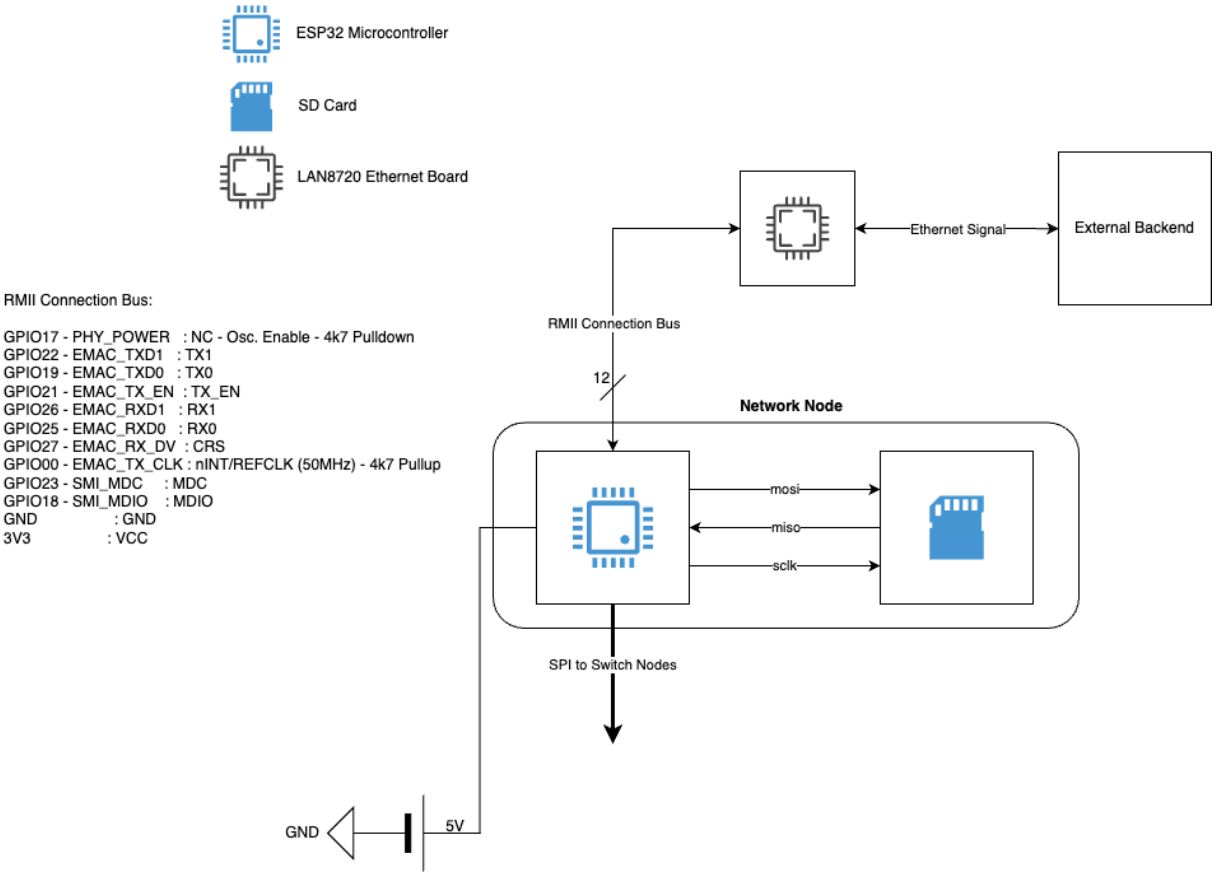

Figure 4.2.1: Network Coordination Block Diagram

4.2.2 Specifications

1. Average Response Time between a Backend-Compute transaction should be less than 1s.
2. The Network Node must be able to guarantee 100% packet transmission into the Switches.
3. The Network Node must clear the packets in-flight when there is an error in the pipeline.

4.2.3 Subsystem Interactions

The data will be converted from compressed file formats to a raw file format in the Backend before being sent out through the Ethernet port to the Network Node with a corresponding image id. The network node will break the image into 2 patches and send the patches via SPI to the switches.

4.2.4 Core ECE Design Tasks

- **ECE 362**: Utilization of reference manuals, technicalities of microcontroller development in C, and register manipulation.
- **ECE 368**: Generalized raw image data structures in C and algorithmic techniques.
- **ECE 337**: Understanding protocol implementation and bit banging.
- **ECE 40862**: Understanding IoT and inter-module implementation, as well as familiarization with the ESP32.

4.2.5 Schematics

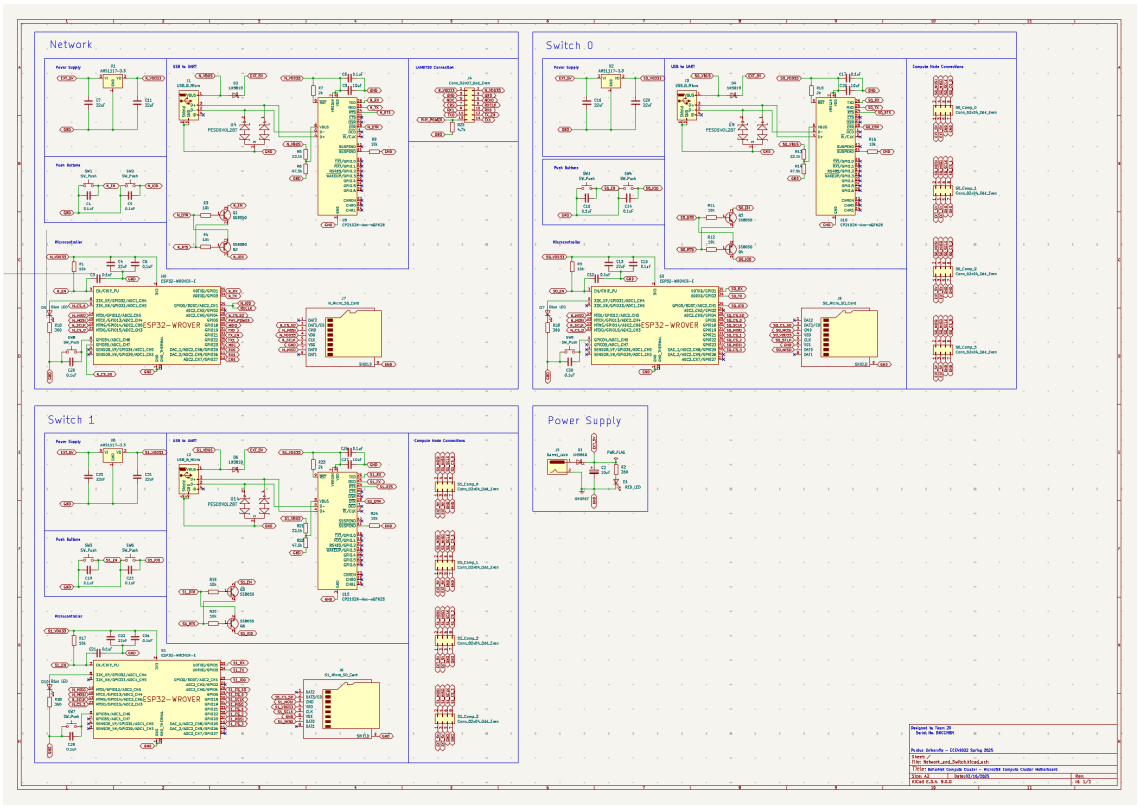

Figure 4.2.2: Network and Switch Node Schematic

4.2.6 Parts

- 1x ESP32-C6-WROOM-1
- 1x LAN8720A Ethernet Board
- 1x Onn 32G SD Card

4.2.7 Algorithm

The Backend will use the CV2 library to convert images into raw binary data and then send an image process request to the Network Node. The Network Node will send either a ready or a stall based on compute node availability. If a ready is received by the Backend, the image will be broken up into patches that can be sent and buffered on the Network

Node PSRAM before the network node places them on the SD card. Once the full image is received, an "image received" acknowledgment is sent to the Backend. The full image is then pulled from the SD card in patches and sent via SPI in equal parts to the switch nodes.

4.2.8 Theory of Operation

- **Backend Operation:**
  - The Backend will begin any operation by sending a process request to the Network Node, and will not proceed with any process until receiving a valid ready from the network node
  - The Backend will utilize a watchdog timer to ensure the network node does not reach an unresponsive state, and will forward the necessary errors to the Front end if an unresponsive state is reached.
  - The Backend will wait for a valid acknowledgment from the network node after the transmission of each packet to avoid overloading the system.
- **Network Node Operation:**
  - The Network Node will only send a valid acknowledgment for each received packet from the Backend after all processing for the packet has completed, in order to avoid overloading the system
  - The network node will store a global table of all image and patch IDs within the system, such that reconstruction can occur seamlessly at the backend, and patch misalignment does not occur.
- **Transmission Operation:**
  - **"Compute" and "Store" operations**
    1. The Backend sends the Network Node a process request for "Compute" or "Store" via Ethernet. If the system is busy, a "stall" is sent as a reply. Otherwise, the process continues after sending a "ready" reply. The Network Node receives an image in patches from the Backend and stores the image patches in the SD card. Each patch size is 1/8 the size of the full image. Each transmission is has a valid acknowledgment sent as a reply
    2. After sending the patches to the Switch and receiving relevant acknowledgments, the Network Node sends a task complete transmission to the Backend.
  - **"Fetch" and "Fetch and Delete" operations**

1. The Backend sends the Network Node a "Fetch" or "Fetch and Delete" request. If the system is busy, a "stall" is sent as a reply. Otherwise, the process continues after sending a "ready" reply.
2. After forwarding the "Fetch" or "Fetch and Delete" request to the Switch Nodes and receiving the image patches from the Switch Nodes, the patches are buffered in the SD card. The Network Node then sends the image in patches to the Backend. Each patch is 1/8 the size of the full image. Each transmission will have a valid acknowledgment sent as a reply from the Backend.

- **Transfer Rate Specification:** The Backend sends patches of the image to the Network node over the Ethernet and RMII interface, which function at a raw theoretical 100 Mbps. However, with the TCP overhead and due to the use of an external PHY chip, we can expect to see 80 Mbps in good conditions. In poor conditions, this may drop to 10 at the lowest, which still satisfies the 0.1MBps transfer rate specification. Kindly note the MBps vs. Mbps rates.

- **SD card Utilization Specification:** SD card utilization is based on the file-system that is being used. The use of a 32GB SD card allows us to use FAT32, which guarantees around 92% utilization as a worst-case scenario. This satisfies the 90% SD card utilization specification.

4.2.9 Specifications Measurement

1. Average Response Time between a Backend-Compute transaction should be less than 1s.

   This specification is solely about the total Round-Trip-Time for any packet in the Cluster. To elaborate, even if the Switch Nodes or Compute Nodes get into modes where they will be busy for a while, we need to ensure that we respond to the Backend quick!

   In the next few images, the important parts of the Timing Summary are highlighted to prove that the specification has been met.

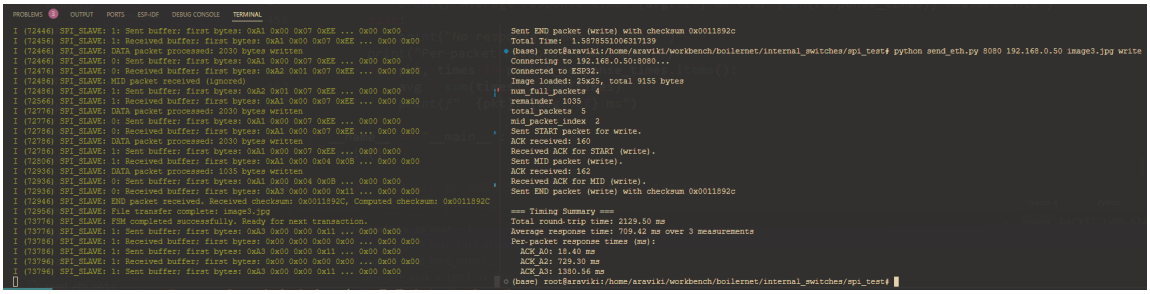

Figure 4.2.3: Backend + Network Node View: Timing Summary for Writes

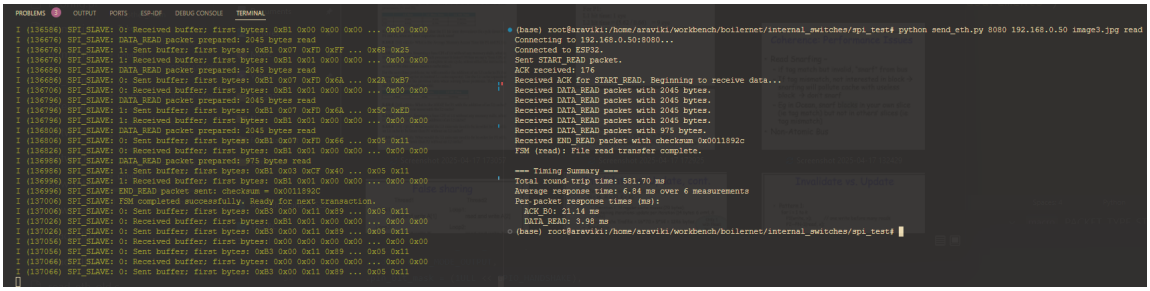

Figure 4.2.4: Backend + Network Node View: Timing Summary for Reads

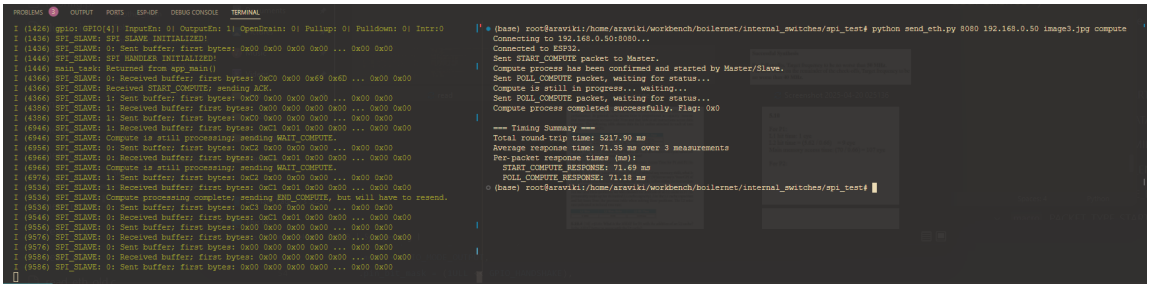

Figure 4.2.5: Backend + Network Node View: Timing Summary for Compute

2. The Network Node must be able to guarantee 100% packet transmission into the Switches.
   A retry mechanism is utilized to ensure that packet corruption or loss is accounted for. Thus, our system has a 100% packet transmission rate. We wish to note that after

moving to the PCB, signal integrity has moved to near 100% packet transmission as-is. The retry mechanism has not been triggered during testing post-prototype.

3. The Network Node must clear the packets in-flight when there is an error in the pipeline.

   In the following image, we combine the Routing Logs across the Network, Switch and Compute Nodes. It highlights how a simulated error in the Switch – disconnecting prematurely – causes a flush across all stages. The second image highlights how a simulated error in the Compute – disconnecting completely – causes a flush throughout the pipeline. Note that this will be re-discussed in Subsystem 3's Specification Measurement.

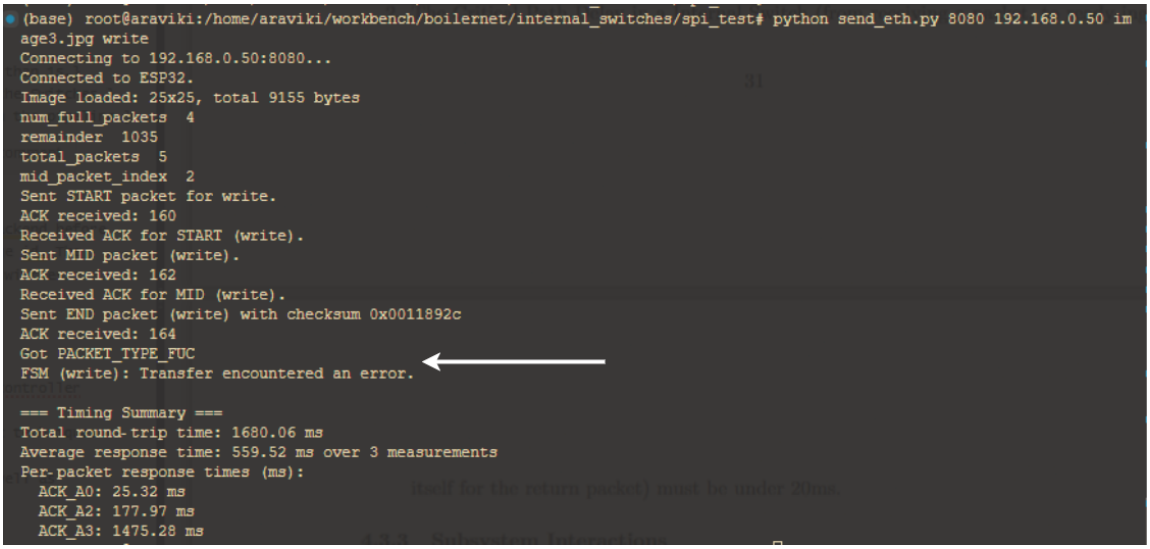

Figure 4.2.6: Error in Packet

4.2.10 Standards

- [**IEEE 802.3 (Ethernet Standard)**]: The Ethernet standard implemented is 100BASE-TX, which is an extension of 10BASE-TX. A 50MHz oscilator is used to implement a 2 bit per oscillation communication, at 100Mbps. The RMII interface is used to connect to the Physical MAC on the ESP32. This was chosen due to easy interface with a backend laptop while also having access to high throughput.
- [**SPI**]: Contains four lines (SCLK, MOSI, MISO, CS); was chosen due to high bandwidth and support for multiple slave devices

4.3 Subsystem 3: Internal Switch Routing

4.3.1 Subsystem Diagrams

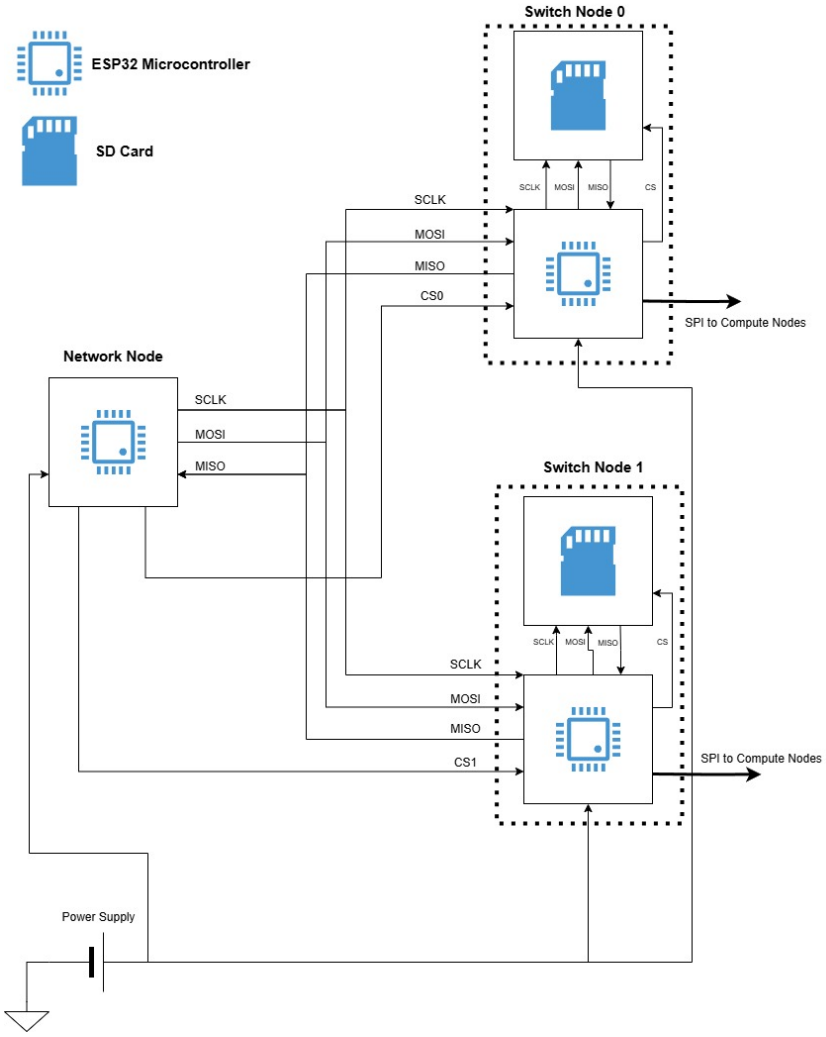

Figure 4.3.1: Internal Switch Routing Block Diagram

4.3.2 Specifications

1. Individual packets can be received and sent at greater than 50KBps over SPI.
2. The Critical Path Delay in a Internal Switch – from receiving a packet, to preparing

for the next one – must be under 200ms.

3. The Switches must be able to communicate errors to both the Network Node and Compute Nodes.

4.3.3 Subsystem Interactions

1. The job and task data from the Network Coordination Layer is parsed and split in the Network Node. The Internal Switch Routing Layer operates on this data from the Network Node.

2. The completed task data from the Distributed Compute Layer is assembled and sent to the Network Coordination Layer.

4.3.4 Core ECE Design Tasks

- **ECE 36200**: Fundamental for programming MCUs and learning about different communication protocols.
- **ECE 56800**: Introduction for working with ESP32s.
- **ECE 36800**: Creating data structures and enhancing C programming skills directly influences ability to write effective MCU code.
- **ECE 43700**: Can be applied to MCU architecture, covers parallelization and compute speed metric calculations.

4.3.5 Schematics

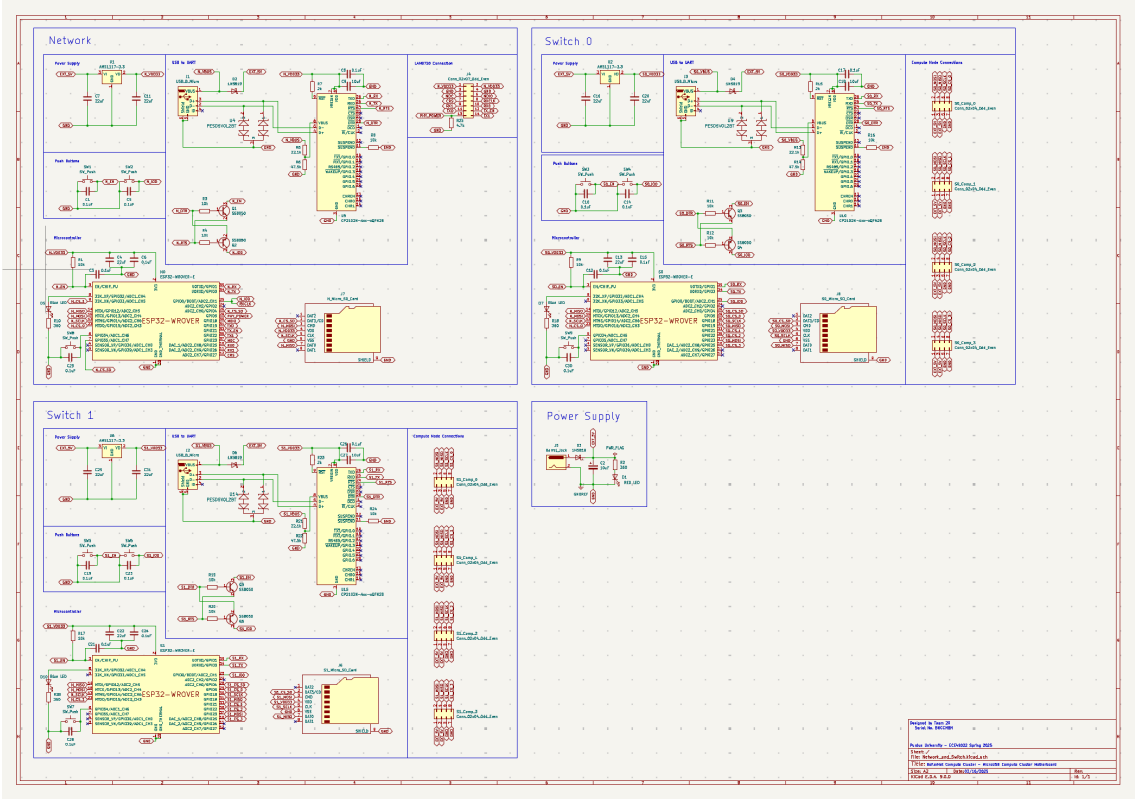

Figure 4.3.2: Network and Switch Node Schematic

4.3.6 Parts

- 2x ESP32-C6-WROOM-1
- 2x Onn 32G SD Card

4.3.7 Algorithm

1. The Network Node sends the split task data and patch ID to the Switch Nodes.
2. The Switch Nodes determine whether the task needs to go to the Compute Nodes or the SD Card.
3. Upon a **FETCH** or **FETCH AND PURGE** operation, data is collected from the SD Cards and transmitted back to the Network Node.

4.3.8 Theory of Operation

A task request (and possibly task data) arrives at the network node. The Network Node sets a wait signal high and then identifies the task type in order to take one of the following actions:

- For a **Compute** operation, the Network Node assigns metadata and divides the task into N parts, where N denotes the number of Switch Nodes (in this system, N = 2). Data is sent to Switch Node 0, and then Switch Node 1. Each Switch Node divides the incoming data into J patches, where J denotes the number of Compute Nodes per Switch Node (in this system, J = 4). Each Switch Node sends and receives data from the Compute Nodes. Upon receiving data from J Compute Nodes, the Switch Nodes reassemble the incoming information and send the data and associated metadata in binary file format to their respective SD cards. The Switch Nodes then send a signal to the Network Node that they are finished with the operation.

- For a **Store** operation, the Network Node assigns metadata and divides the task into N parts, where N denotes the number of Switch Nodes (in this system, N = 2). The data is sent to each Switch Node. Data is sent to Switch Node 0, and then Switch Node 1. Each Switch Node sends the data and its associated metadata in binary file format to its SD card. Each Switch Node then sends a signal to the Network Node that it is finished with the operation.

- For a **Fetch** operation, the Network Node receives metadata for the file to be retrieved. This metadata is sent to each Switch Node. Each Switch Node examines the metadata, retrieves the corresponding binary file from its SD Card, and sends it back to the Network Node. The Network Node then reassembles the packets received from each Switch Node and sends the final result back over Ethernet.

- For a **Fetch and Purge** operation, a **Fetch** operation is performed, but a deletion command is sent by each Switch Node for each binary file that is retrieved from the SD Card after the file has been retrieved.

All data transfer will occur utilizing the SPI (Serial Peripheral Interface) protocol at a data rate of 80 MHz, meeting the data rate transmission specifications.

4.3.9 Specifications Measurement

1. Individual packets can be received and sent at greater than 50KBps over SPI.
   This specification is currently not met. We are hoping to reach the target with optimization from the code as well as switching to the PCB.

2. The Critical Path Delay in a Internal Switch – from receiving a packet, to preparing for the next one – must be under 200ms.

This is required to ensure that the Network Node is not stalled for too long, which will lead to TCP overflows, or WDT-related (WatchDog Timer) Panics. Even if the Switch will be busy for longer than normal, specifically when Reading or distributing data to Compute Nodes, the Network Node needs to be informed of it immediately.

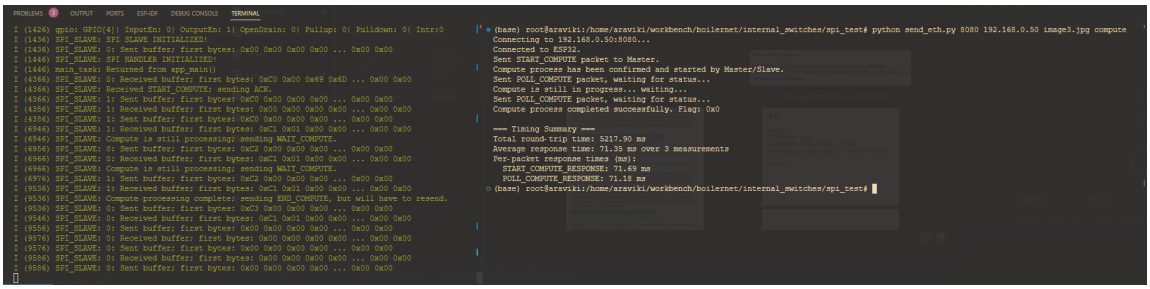

Figure 4.3.3: Critical Path Delay

3. The Switches must be able to communicate errors to both the Network Node and Compute Nodes.

   To clarify, errors can be generated in any of the layers – even Backend – and at any time. Here, the specification is meant to measure how **robustly** the Switch can handle this situation.

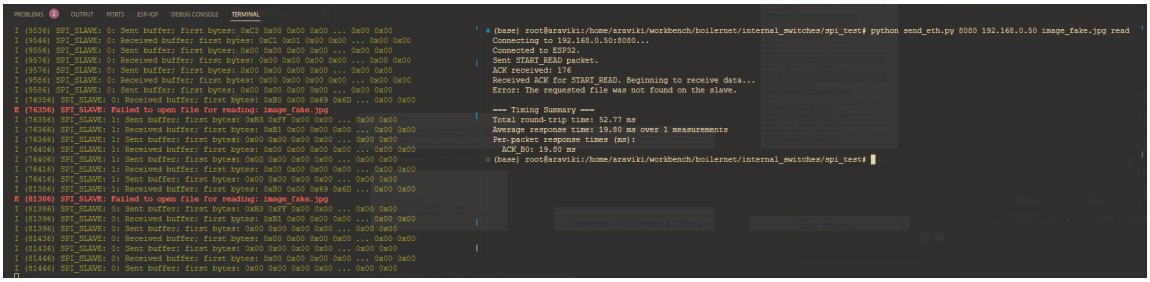

Figure 4.3.4: Network Coordination (File Not Present)

4.3.10 Standards

- **SPI**: Contains four lines (SCLK, MOSI, MISO, CS); was chosen due to high bandwidth and support for multiple slave devices.

4.4 Subsystem 4: Distributed Compute Layer

4.4.1 Subsystem Diagrams

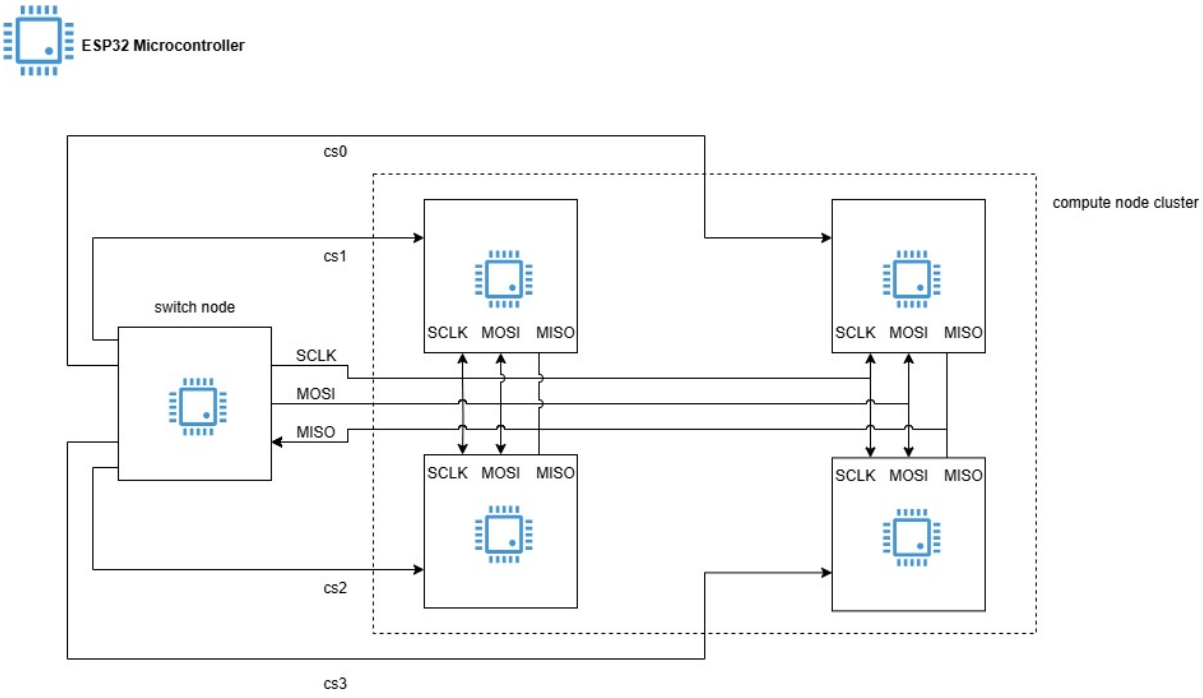

Figure 4.4.1: Distributed Compute Layer Block Diagram

4.4.2 Specifications

1. The Deep Learning Model running on a Compute Node must not take more than 6 seconds for inference.
2. Compute Nodes must decode PNG buffers into Byte Arrays in under 2s.
3. Compute Nodes must be able to fit the entire PNG Buffer, Quantized Activations and intermediate Buffers in RAM.

4.4.3 Subsystem Interactions

Communicates to the internal switches through the SPI protocol to receive the input image patch and return the output processed patch.

4.4.4 Core ECE Design Tasks

- **ECE 36200**: Using C to program MCU pins and learning about different communication protocols
- **ECE 56800**: Introduction to working with ESP32s
- **ECE 36800**: Data Structures and Algorithms to increase efficiency of compute tasks

4.4.5 Schematics

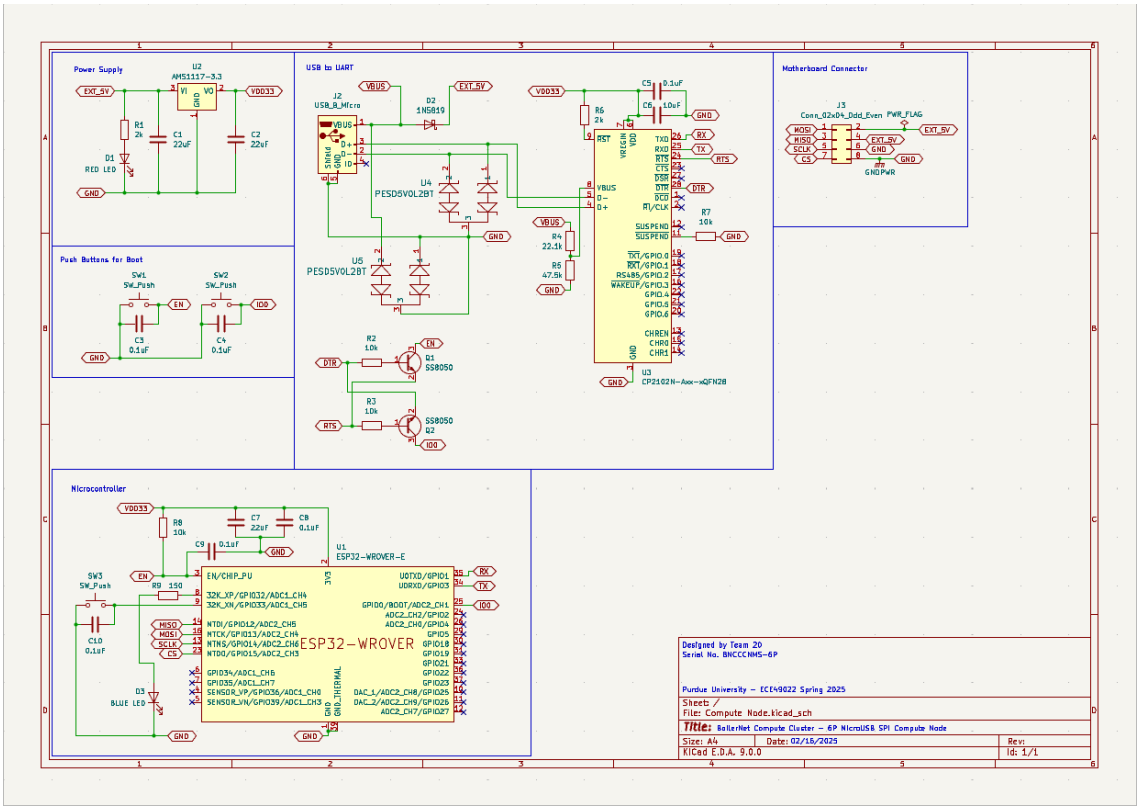

Figure 4.4.2: Compute Node Schematic

4.4.6 Parts

- 8x ESP32-C6-WROOM-1

4.4.7 Algorithm

The DNN models running on the Compute Nodes are fine-tuned variants of the MobileNetV2 Backbone, trained on grouped subsets of the 2012 Imagenet Object Classification dataset. Below, we specify our reasoning behind choosing the MobileNetV2 architecture for our use-case.

- Uses *depthwise separable convolutions* to factorize a standard convolution into a lightweight depthwise convolution (per-channel spatial filtering), drastically reducing both multiply–accumulate operations and parameter count..
- Alpha (width multiplier) controls model size—using $\alpha = 0.35$ yields a sub-million-parameter model that still extracts rich feature maps.
- Enables low-precision quantization (e.g. INT8) with minimal accuracy loss, making it ideal for microcontroller deployment.

4.4.8 Theory of Operation

Each compute node embeds the TFLite-Micro interpreter, which uses a small *tensor arena* in RAM and a fixed C++ array of quantized weights graph in Flash (generated by converting the trained TF Lite model to a `.tflite` flatbuffer and then to a C header). One of the models trained to identify different dog breeds has a C header size of 3.8MB and 700KB TFLITE binary. At runtime:

1. The node waits for a framed PNG image over SPI. This will be streamed directly into a SPIRAM buffer.
2. Upon complete reception, it decodes the PNG into a raw `uint8_t[224×224×3]` buffer, in uint32_t format.
3. The TFLite-Micro interpreter is initialized once; `AllocateTensors()` lays out input, intermediate, and output tensors within the pre-allocated arena.
4. The image is quantized into INT8, fed into the input tensor, and `Invoke()` runs the MobileNetV2 network entirely in Flash/RODATA, producing a 5-class softmax output.
5. A small C++ helper builds a fixed-size (2 KB) response packet containing the predicted class index (or name) and the inference time, suitable for DMA over SPI back to the switch node.

4.4.9 Specifications Measurement

1. The Deep Learning Model running on a Compute Node must not take more than 5 seconds for inference.
   This specification is currently not met. We managed to optimize and quantize our activations to help during inference, however were met with an average of 7s. We outline our tests below, with varying PNG sizes.

2. Compute Nodes must be able to fit the entire PNG Buffer, Quantized Activations and intermediate Buffers in RAM.
   We utilize the SPIRAM-enabled WROVER IE boards to ensure internal RAM is not overflowing. With 224x224 Tiles taking over 300KB and TF-Lite's Tensor Arena requiring over 500KB of working memory, internal RAM would have failed us. We ensure robustness by cancelling transactions overflowing a preset utilization limit of 4MB. Below, we outline three scenarios of failure, and corresponding handler packets.

3. Compute Nodes must decode PNG buffers into Byte Arrays in under 2s.
   With the help of LodePNG's open-source decoder, and the WROVER's SPIRAM, we were able to override the general 'malloc()' calls to help in larger working buffers, and sub-2s decoding time!

4.4.10 Standards

- **SPI**: Wired communication following master-slave architecture. Consists of 4 wires for MOSI (master out, slave in), MISO (master in, slave out), SCLK (clock), and CS (chip select).

5 PCB Design

5.1 PCB Schematics

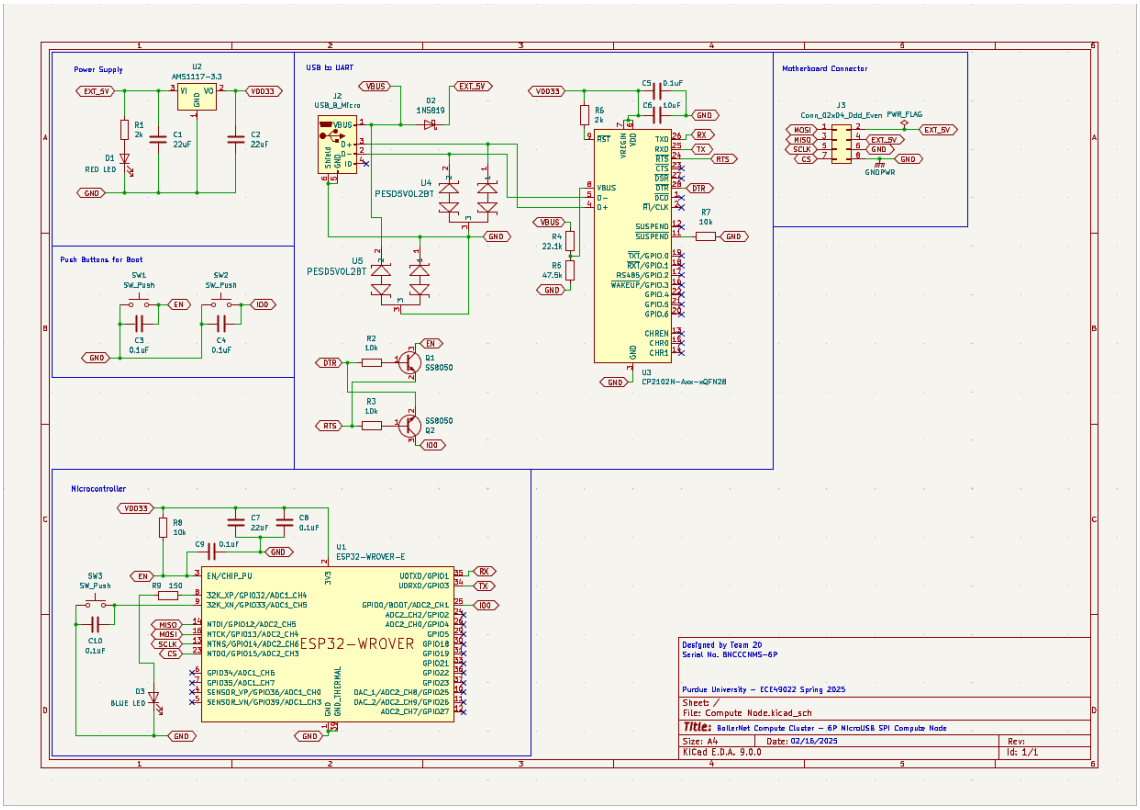

Figure 5.1.1: Compute Node Schematic

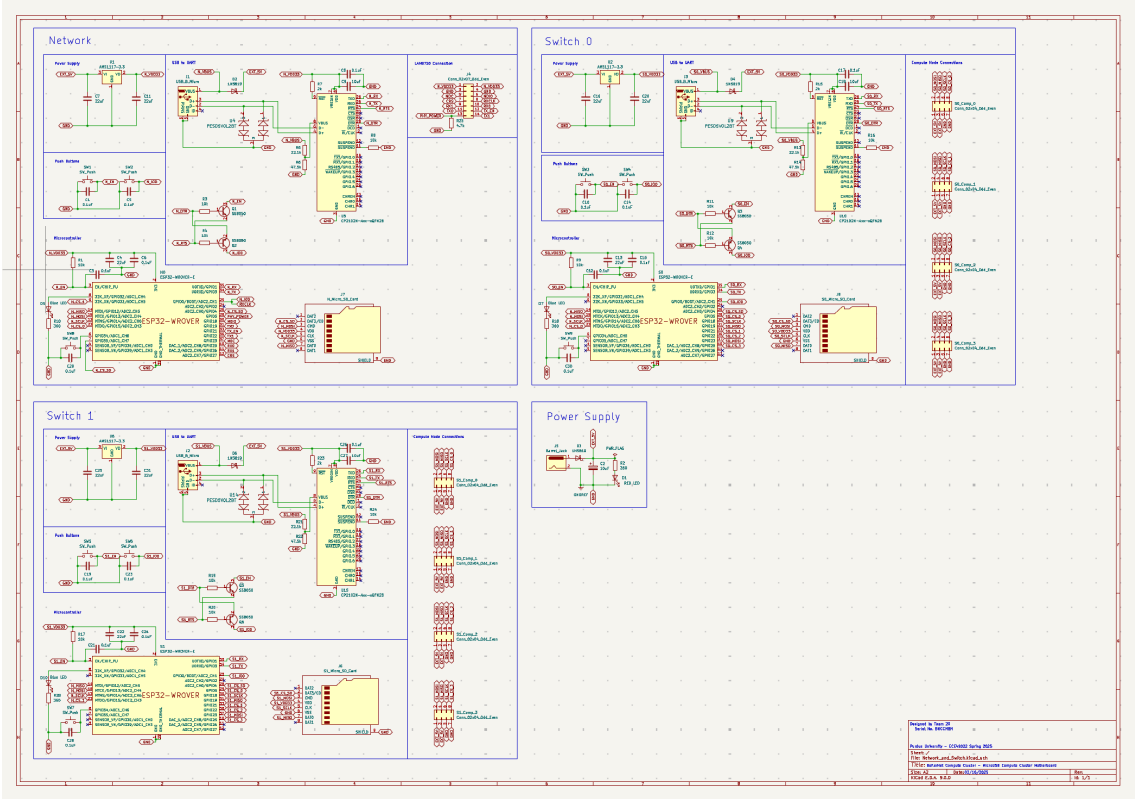

Figure 5.1.2: Network and Switch Nodes Schematic

5.2 PCB Layout

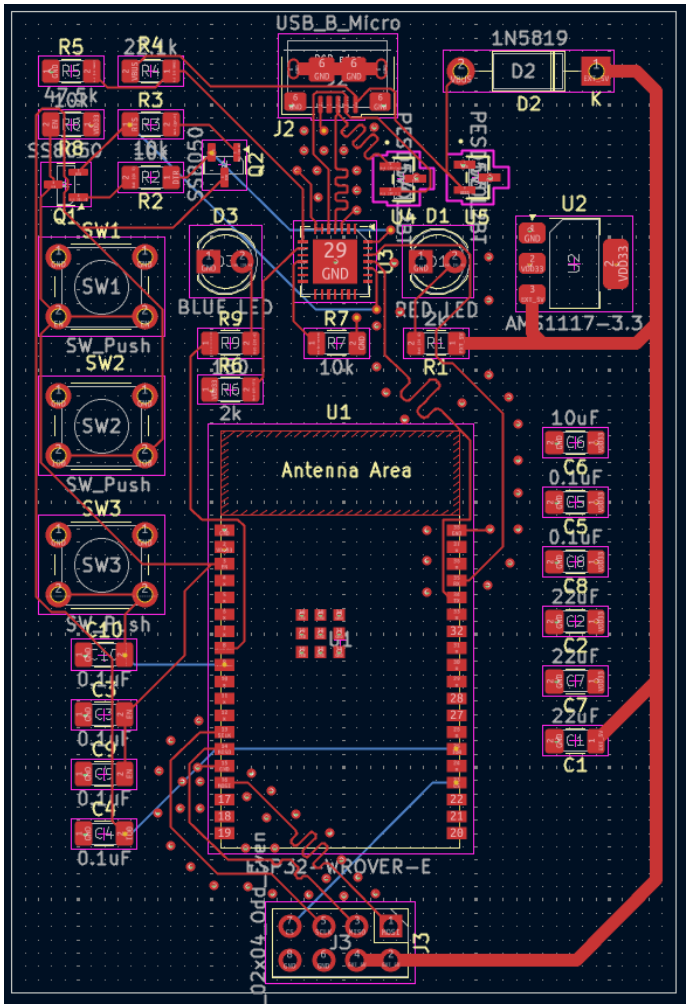

Figure 5.2.1: Compute Node PCB Layout

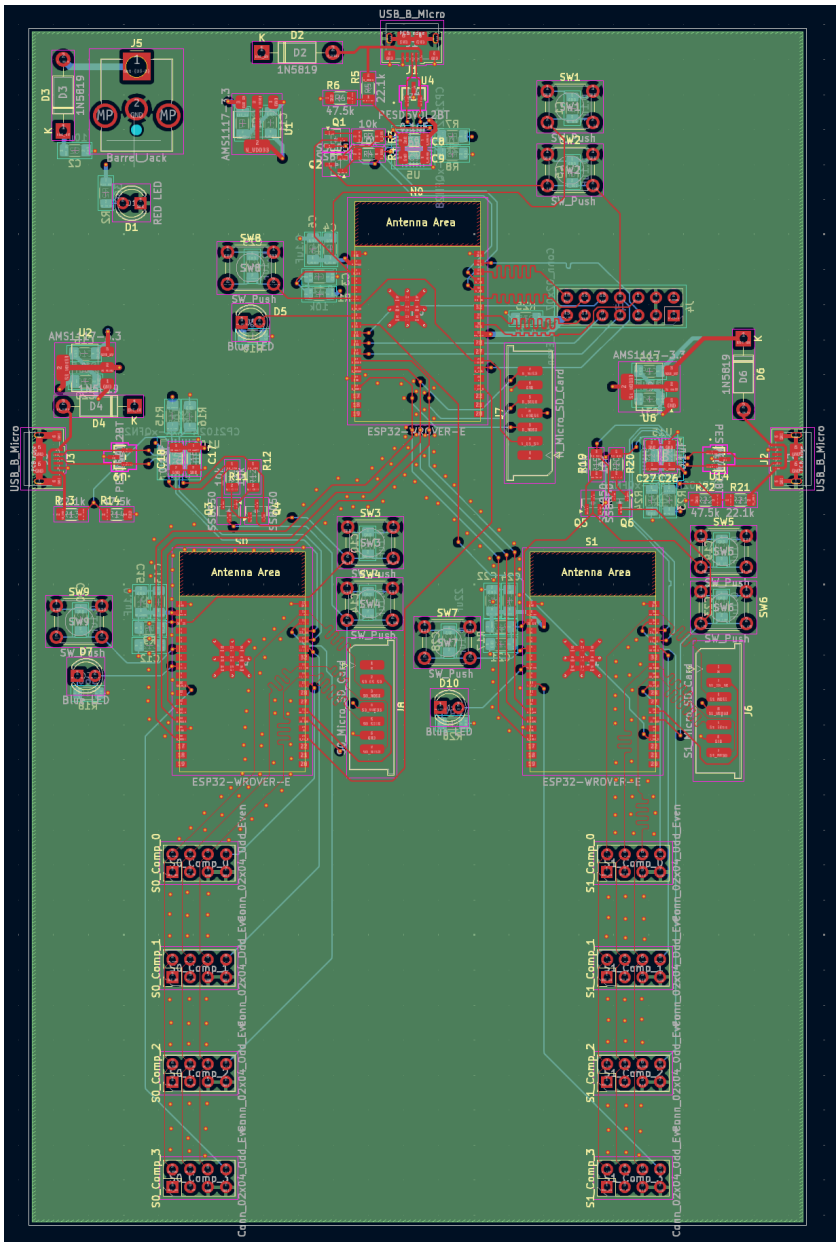

Figure 5.2.2: Network and Switch Nodes PCB Layout

6 Final Status of Requirements

1. Requirement 1: The Frontend User-Interface will allow the User to choose the Operation and define a Job with a (limited) number of Tasks for the cluster to work on.
   Met: Users are currently faced with a webpage that shows options to upload files and assign tasks that will be sent to the compute cluster.

2. Requirement 2: The Backend will be responsible for buffering Jobs and sending in Tasks serially into the Network Node. It will contain a message queue, working in a Last-In-First-Out manner.
   Partially Met: The Backend currently buffers jobs, but currently does not use a Last in First Out message cue.

3. Requirement 3: The Network Node will send the job type, along with half of each task to each switch node.
   Partially Met: The network node currently splits the jobs between the 2 switches, but does not clarify job type as only one job has been implemented. This will be resolved with future job implementations.

4. Requirement 4: The Internal Switch Nodes are required to split incoming Tasks and "patch" them into Patches and store them in SD-Card.
   Met: The internal Switches can successfully patch and store images into the SD card

5. Requirement 5: The Internal Switch Nodes will also be responsible for splitting up the Patches among the compute nodes in serial, reading-and-writing asynchronously.
   Not Met: This is not currently implemented within the design.

6. Requirement 6: The Compute Nodes will process the Patches, as received, synchronously, depending on the Operation type chosen by the User.
   Partially Met: The compute nodes require the patch to move through the entire system before reaching them and do not receive the patch to process with the current design. They are capable of working in isolation but not within the current system.

7. Requirement 7: The Compute Nodes are to be developed onto Cartidge-like PCBs, allowing for slotting and unslotting with ease.
   Met: An enclosure for the compute node has been designed and printed. The compute node's 10-pin male receptacle is exposed in the enclosure, and is able to be slotted into the female receptacle on the motherboard.

8. Requirement 8: The Compute Cluster, as a whole, must be contained within it's 3-D printed enclosure, made in Black and Gold colors.
   Partially Met: The enclosures for the slottable compute nodes have been designed and printed in black. The motherboard enclosure still needs to be designed and printed.

7 Team Structure

7.1 Team Member 1

Akshath Raghav Ravikiran
Major: Computer Engineering
Contact: araviki@purdue.edu
Team Role: Software/Microcontroller (Technical) and Communication (Professional)
Bio: Akshath is an aspiring computer engineer, excited about Machine Learning Hardware. He is a researcher at Purdue's SoCET group, focusing on computer architecture enabling AI-specific compute. He plays tennis in his free time, and loves Calvin & Hobbes comics.

7.2 Team Member 2

Aneesh Reddy Poddutur
Major: Electrical Engineering
Contact: apoddutu@purdue.edu
Team Role: Microcontroller (Technical) and Team Lead (Professional)
Bio: Aneesh is an electrical engineering student at Purdue University. His interests lie in FPGA and ASIC development for high-speed compute, along with embedded system design. When not working, Aneesh likes to play basketball, spend time in nature, and play video games.

7.3 Team Member 3

Gautam Kottayil Nambiar
Major: Computer Engineering
Contact: gnambia@purdue.edu
Team Role: Microcontroller (Technical) and Purchasing (Professional)
Bio: Gautam is an aspiring Computer Engineer at Purdue University with a focus on low-level systems design. He enjoys working with C, SystemVerilog, and embedded systems, specializing in RTL design, digital logic, and microprocessor interfacing. In his free time, he enjoys dancing on Purdue's premier bhangra team, Boiler Bhangra.

7.4 Team Member 4

Gokulkrishnan Harikrishan
Major: Computer Engineering
Contact: gharikri@purdue.edu
Team Role: Microcontroller (Technical) and Facilitator (Professional)
Bio: Gokul is a Computer Engineering student at Purdue University. He is interested in computer systems, ASIC, and digital design. In his free time, he enjoys reading as his main hobby.

8 Bibliography

References

[1] Ravikiran, Poddutur, Nambiar, Harikrishnan. "AkshathRaghav/BoilerNet: Code release for a distributed mini-compute-cluster prototype, demo'd at SPARK S'25 as a Purdue Senior Design Capstone.", Github, https://github.com/AkshathRaghav/boilernet. Accessed 2 Feb. 2025.

[2] Zelmoghazy. "Zelmoghazy/ESP32-Ethernet-LAN8720: Everything You Need to Know on How to Connect the LAN8720 PHY to the ESP32." GitHub, github.com/Zelmoghazy/esp32-ethernet-lan8720/blob/main/README.md. Accessed 2 Feb. 2025.

[3] ESP32 SPI Communication: Pins, Multiple SPI, Peripherals (Arduino) — Random Nerd Tutorials," Aug. 18, 2022. https://randomnerdtutorials.com/esp32-spi-communication-arduino/