DashMap: Modular Dashboard for end to end SLAM deployment with inexpensive robots

Hudson Reynolds, Daigen Burton
Boston University College of Engineering
EC535: Embedded Systems
Profesor Eshed Ohn-Bar
5/6/2025

Abstract

It is generally understood that robots will eventually become commonplace, with broad capabilities and intelligence to justify their cost. Localization and mapping are critical skills for robots to understand our world and attempt complex tasks. As new technologies unlock new projects, creating modular components is a highly efficient strategy. We present ***DashMap***, an easily deployable, modular, customizable dashboard that features components that help users focus on what they want to during SLAM system development. We provide components for SSH/terminal access to Linux-based robots, live camera stream display with FPS counter overlaid and recording features, buttons for controlling the robot, and connection to a powerful SLAM interface to create high-resolution maps with the camera recordings. We paired DashMap with a companion robot, ***Mapper***, to stream frames and drive around. Mapper represents the inexpensive robots that are becoming increasingly popular in households. We hope the easy-to-deploy, easy-to-modify, easy-to-reuse code in DashMap can turn your next robotics project into a slam dunk! https://github.com/HudsonReynolds2/mapper/

Keywords: SLAM, SSH

DashMap: Modular Dashboard for end to end SLAM deployment with inexpensive robots

Simultaneous localization and mapping (SLAM) is a method for autonomous vehicles to find their position within an environment while building an accurate reconstruction of the environment *(What Is SLAM (Simultaneous Localization and Mapping)?*, 2024). Unfortunately, past SLAM algorithms require robots to be equipped with multiple expensive sensors like LiDAR or depth cameras. This poses a significant challenge for low-budget robots and teams. Calibrating, powering, processing, and moving these sensors adds complexity on top of the cost. Existing SLAM models have long-standing compute-side problems like thrashing, over-consumption, unrealistic execution times, low-quality results, and often fall out of support.

Cutting-edge SLAM algorithms are constantly being developed. Sufficient parallel processing capabilities provide computers the ability to run SLAM at resolutions and framerates previously only dreamt of. Now, new models have been developed to remove the need for sensors beyond a single monocular camera onboard the robot. One of the newest of these models, SLAM3R, promises online, real-time, extremely high-resolution SLAM with only a single GPU*(SLAM3R: Real-Time Dense Scene Reconstruction from Monocular RGB Videos*, 2025). With these capabilities, a robot could potentially explore a new environment autonomously while creating a 3D map of the space. SLAM3R works with monocular camera input and doesn't require camera-specific information. This means the model can work with a huge range of inexpensive single-lens cameras.

While these models develop, so do microcontrollers and small board computers. ESP32s and Raspberry Pis are faster, cheaper, lower-power, and have more capabilities than ever before. These capabilities are bringing useful embedded systems projects into the realm of the financially constrained.

However, these boards do not have the necessary computing power to run the new SLAM models. Cloud computing and household networking can solve these problems. Users can satisfy their computing needs in several ways: buy/build a rig, rent a rig in the cloud (bare metal or VM) for some time increment, or use a paid API. Instead, the microcontroller can connect sensors like a camera to this powerful backend and drive motors for exploration.

Deploying a cutting-edge SLAM model on a low-cost robot with a monocular camera and popular microcontrollers serves as a useful starting point to enter the world of advanced robotics. Properly packaging this deployment would allow users to launch their projects with the interface they need to use their robot. For our project, this contribution should be quickly and easily deployable, secure, and customizable. It should be open-source, unconstrained by proprietary software or specific parts, and modular.

We outline the design and implementation of our dashboard, ***DashMap***, and companion robot ***Mapper***, in the Method section below, then evaluate our progress in the Results section.

Method

Description

Our Method section describes each key component of our project, combining them into a fully implemented design. We describe specific components as needed to justify design decisions. The evaluation and more thorough description of our product are fully described in the Results section.

SLAM3R

SLAM3R is a powerful model with many parameters that can be varied to fit a user's needs. The creators provide a browser interface that can be hosted by running their Python script app.py. While the model can be called with Python or otherwise, the browser offers a nice layout and ways to change parameters with sliders, without a user needing to edit any code. We started development by setting up the environment to run SLAM3R (discussed more in Results). Ultimately, using the browser interface proved the most user-friendly way to interact with the model. As the model evolves and updates are made to it, the interface improves. This happened throughout our project. The interface assumes the computer hosting it has the GPU for running SLAM3R itself. This means that a team without access to a sufficient Nvidia GPU rig would need to host the browser interface on a rented rig and serve the interface to themself or access it with some form of remote desktop. With this in mind, the browser interface for SLAM3R will be the primary method we use to interact with the model for the project. Throughout development, we hosted it both remotely and locally. We achieved success with both methods, proving that SLAM3R can be interacted with by beginners on both local and remote setups easily. However, its runtimes vary widely.

The SLAM3R model expects a batch of frames as a video file or a static batch of frames. SLAM3R currently does not provide a way to use the “online” functionality it advertises, even though it is built for this purpose. We reached out to the development team, and they said they will resume work on this part of the project at the end of May. SLAM3R currently provides two demonstration data sets that can be run with their tuned settings using the corresponding bash scripts. The lack of online functionality prevented us from implementing autonomous exploration because the map does not update as new frames are streamed in. As such, our robot (**Mapper**) needs to stream frames or a video file to the SLAM3R model, but it does not have to be in sync. **Mapper** will receive teleoperated commands to move because there is now no ability to localize in real time using SLAM3R. This is discussed in the Results and Future sections. Instead, we will record a video to give to the model at the end of teleoperation.

Mapper

Robots often use Linux as their primary operating system. The baseline method for interacting with these computers is called the terminal, which can be connected to remotely using Secure Shell (SSH). Raspberry Pis (RPi’s) are a series of small board microcontrollers that run Linux and come with WiFi and Bluetooth support. RPis and ESP32s work together very well, and they can be combined to handle multiple tasks in real-time reliably at low power, using a bus/protocol like UART, i2C, SPI, CAN, Bluetooth, or WiFi. They are extremely popular and very inexpensive, with many sensors and accessories available. For running SLAM3R with Mapper, we use a Raspberry Pi 3b as the backbone, and a PiCam Module V3 as the only sensor input. We utilize a dual-core ESP32 to drive motors using a motor driver shield. Combined, these parts add up to less than $100 on Amazon.

To drive Mapper, the ESP32 schedules a FreeRTOS task on each of its two cores. One core communicates inputs to the RPi via Bluetooth, while the other core does input mixing and outputs to the motor driver shield to spin motors. When we turn on a Bluetooth controller in pairing mode, it directly connects to the ESP32 and enables low-latency teleoperated driving. We can also connect the RPi to the ESP32 with Bluetooth. The two ESP32 CPU cores share a data structure to exchange information. This way, each core can reliably handle its tasks without hiccups. The communication core interacts with the Bluetooth modem onboard the ESP32 via the Bluetooth libraries for ESP32 that must be included when compiling its code. This is the even lower-level version of adding a kernel module in Linux. The driving core communicates with the motors via a motor driver shield that has a shift register. The ESP32 also uses two servo motors for a 2-axis gimbal that allows operators to move the camera independently. To push code to the ESP32, we use the Arduino IDE but hope to learn PlatformIO later**.**

Live View

To feed SLAM3R useful data from Mapper, we stream frames to a live view. This live view is useful for any operations with a robot. To save frames, we create start, pause, resume, and stop buttons, where each new recording gets its own folder. The RPi code serves as a helpful starting point for any RPi camera project. We use JavaScript to receive a frame stream from a Python script on the RPi. The RPi establishes a WebSocket connection over WiFi and immediately starts streaming frames with timestamps in their metadata. The server can use this information to overlay a real-time FPS counter. The Results section describes and evaluates this live view.

DashMap

Our dashboard is built to use a browser to support the inherent web-based communication, scalability, and generalizability that this project stack necessitates. React is a modern web framework that allows us to create components that can be treated as modules. These modules, like “live camera view” are powerful because they can be reused, edited, or abstracted quickly while remaining independent from each other. For example, we can create a component for the SLAM3R browser interface to be shown on our dashboard. If a user hosts their SLAM3R on another link or uses an entirely different SLAM model, they can just edit the SLAM component. Similarly, we can create a “Terminal” component that allows for an SSH connection to the robot. DashMap provides a powerful starting point for robot-based SLAM by providing SSH connection/terminal to the robot, camera functionality like livestream and record video, buttons for robot control, and a view of the SLAM3R interface, connecting every level of the stack. We customize components for our specific implementation with Mapper while leaving DashMap easily editable for variations of SLAM projects, while components themselves can be repurposed for entirely different robotics projects. This flexibility satisfies our design goals.

DashMap was developed in a Docker container so users can simply download our code, run a single Docker command, and be good to go. A configuration file stores the user-specific information like IP addresses, while passwords for SSH connection are never stored. This structure allows for quick deployment that fits the situation of the user without sacrificing security or functionality. This aims to reduce reliability on specific operating systems and avoids hurdles users may experience in deploying our code.

Results

We describe specific information about each component, define evaluation criteria for each component, and evaluate these components individually as well as collectively. We create maps of sections of Hudson's apartment using DashMap and describe the successes and failures of this process.

Live View Component

We acquired USB webcams and the common Pi camera modules (V1, V2, and V3), and chose the V3 for its broad user base and programmable autofocus, discussed more in the Future section. We tested Flask, Uvicorn, and TCP-based streaming and kept seeing inconsistent frame pacing and framerates. We used metadata, Python, spreadsheets, and graphs to analyze the performance of the livestream and recordings. We got great results with many methods at low resolutions, but at higher resolutions, the Pi 3 had lag spikes regardless of the method. We verified that the Pi 3 can record great video without lag spikes if it is not transmitting while recording. We tested recording to the Pi's storage while livestreaming to the camera view, then transferring the recorded files at the end. We also verified that this livestream could be received by the remote GPU rig directly if the user wants to go from RPi to cloud without using an intermediate host for a dashboard. We would like to add official support for these methods in a DashMap update. We chose a medium resolution, medium framerate (480p 30fps, 1/60th shutter speed) using simple WebSocket code using compression with TurboJpeg at 90% quality. These settings can be easily modified. The chosen implementation is flexible, secure, reliable, easy to work with, and is resilient to robot failures (no chance of losing an entire clip). We analyse the performance of using websockets and TurboJpeg here.

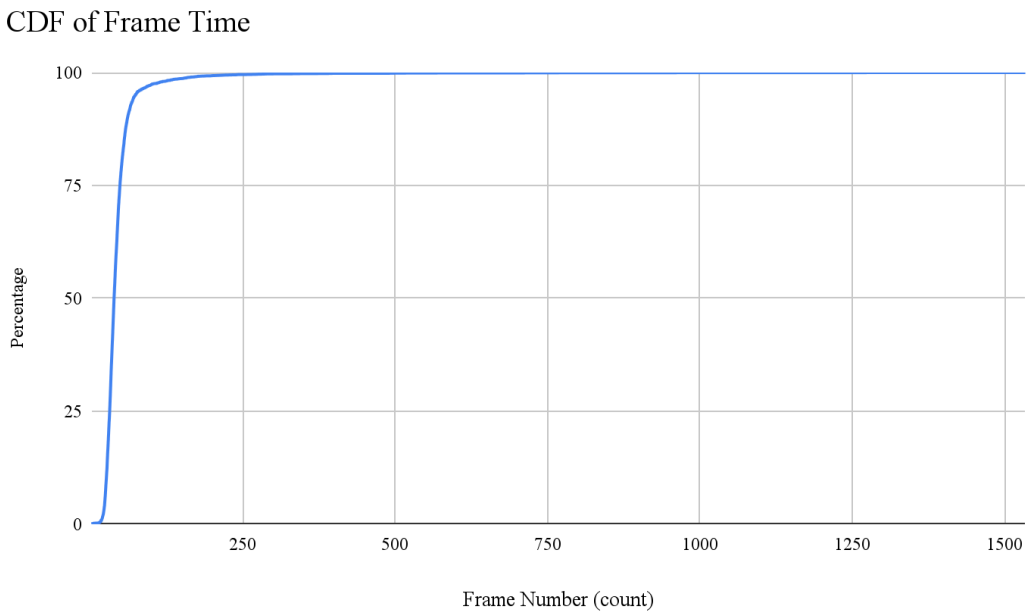

Figure: Frame CDF

The steep increase near the vertical axis shows that a majority of our data takes a short amount of time to transfer frames. More quantitatively, about 94% of frames are transmitted faster than our target of 70ms. About 0.7% of frames stuttered more than 200ms.

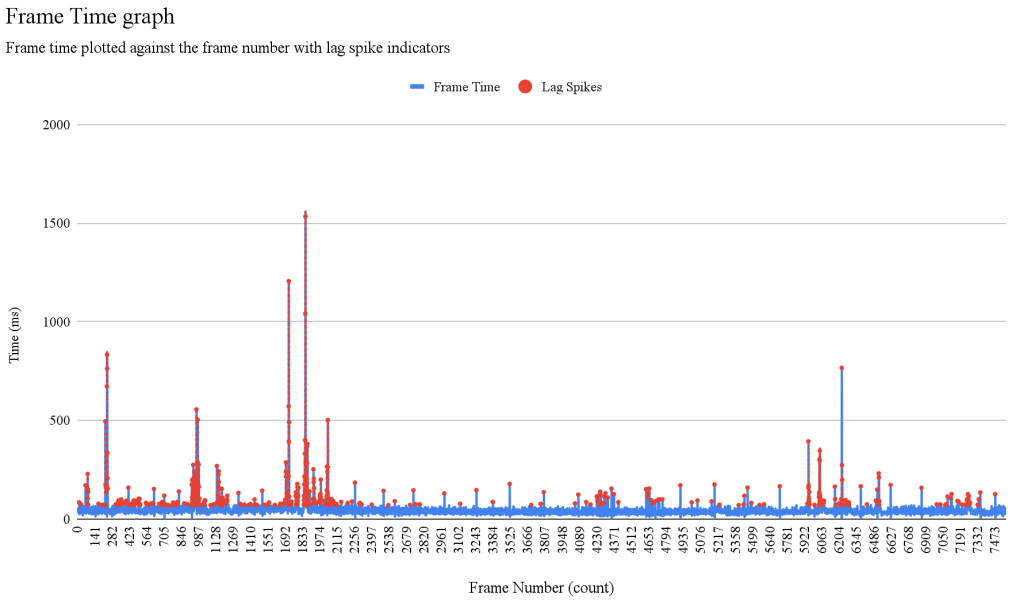

Figure: Frame Timing

This shows frame time (in blue) overlapped with lagged spikes (in red). The lagged spikes are determined by any frame that takes longer than 70ms to transmit, which equates to ~14.3 FPS. There are consistent small spikes every ~200 frames. We believe this is from the RPi's cache filling up, and it needs extra time processing before it can continue sending data. We did not investigate because we intended to just use a Pi 5, which would not have this issue. There are also a couple of large lag spikes, which could be from networking traffic, the websocket needing to reconnect, or processes on the RPi taking priority. This is good enough for our purposes and is discussed in the Future Work section.

The Live View component satisfies our design requirements for being customizable, reusable and modular, and performant while staying flexible. We also tested some footage from it early on with SLAM3R to prove that we could proceed using that camera. Our success with generating the map is discussed later.

SLAM3R Interface Component

The SLAM3R component was tested by connecting both local and remote hosted SLAM3R interfaces to DashMap. We used BU VPN and SSH tunneling to test that we could get the page into our dashboard regardless of its host location. This simple connection proved easy to work with after we figured out how to full-screen the interface within the React component window. DashMap expects the SLAM3R interface to be running when it starts up. If the SLAM3R interface is started afterwards, DashMap will populate it with a simple page refresh, no need to restart the Docker container.

Terminal Component

The Terminal component introduced a host of security concerns but also many new capabilities. It transforms DashMap from being a purely camera to SLAM3R pipeline and adds direct connection to a robot's operating system, with sudo access. This connection is formed via a shared connection to a WiFi network. Adding this feature contributed to a streamlined workflow while working with the robot during our simultaneous deployment and development of DashMap. However, we mentioned security concerns. The terminal acquires its connection to the RPi with an SSH connection. This involves a username and password that our code never stores. We use the industry standard environment variable file and protect a user's connection to their robot. If someone

gets access to their interface after establishing this connection, this could be a problem. However, we tried to protect against some of these vulnerabilities by filtering out certain powerful commands, removing files, adding/deleting users, changing passwords, or fork bombing the system. In our testing, the component functions reliably every time if there is no confounding factor (like accidentally using the BU VPN). It has no noticeable delay and feels like a native SSH terminal. It took significant customization by Daigen to achieve this. It is easily deployable and customizable, reliable, secure with passwords, and we can add more security rules as needed.

Buttons Component

The Buttons Component features a layout of individual button components. These buttons can be customized both functionally and aesthetically on an individual level and reused in other places across DashMap. The Buttons component can be edited to rearrange, add, or remove buttons as needed. With this interface, users can begin using their robot for their specific task. They can launch and kill programs, send teleoperated drive commands, and interact intuitively with their robot. The flexibility of this component satisfies our design. We implement a layout that a user can customize. Our testing proves that commands from the Buttons component reliably get to the RPi, which executes the command, and the Terminal component displays everything. The flexibility and customizability that the Buttons component provides satisfy our design requirements. It is the primary method for adding functionality to DashMap quickly.

Mapper Implementation

For Mapper, our GitHub provides ESP32 code to use a household Bluetooth controller for direct teleoperated driving. The ESP32 can be connected with Bluetooth to the RPi to receive commands from the Buttons component instead of a controller. Due to the broad space of connecting a robot to a control interface, we leave the user to customize this part. The physical space we are mapping is small, and the reliability of a direct connection from controller to the ESP32 is what suited our development needs. For actual deployment, it makes more sense to wire the RPi to the ESP32 using one of the aforementioned buses like I^2C because that connection would be more power efficient, higher bandwidth, resilient to noise, and would not be susceptible to certain vulnerabilities. The space we are using allows the controller to connect directly to the Mapper, while looking at the live view and terminal output, and due to limitations later on, we did not wire the RPi directly to the ESP32. The ESP32 connects to the motor driver shield which is wired to a battery and the motors. We used a set of 4 simple DC motors and mecanum wheels. We created input mixing functions and an intuitive Bluetooth controller interface using the BluePad32 library. The ESP32 code can be customized for the specific robot and drive train, but it is important to recognize the simplicity and low cost of this platform. Mapper satisfies our design requirements for being simple, inexpensive, modular, modifiable, and functional for the purposes of creating a map of our environment. It is representative of many basic robots that might connect to DashMap.

Docker Container

We use Alpine (22-alpine3.21) to run a Node server within a Docker container using the latest and most secure images available. The container installs the following dependencies automatically. Backend: npm, express, ws (websocket), ssh2. Frontend: xterm. The container size is **1.43 GB** which is acceptable. It builds in under a minute and is not computationally intensive to run. If built recently (in memory), it rebuilds in under a second.

Making a map

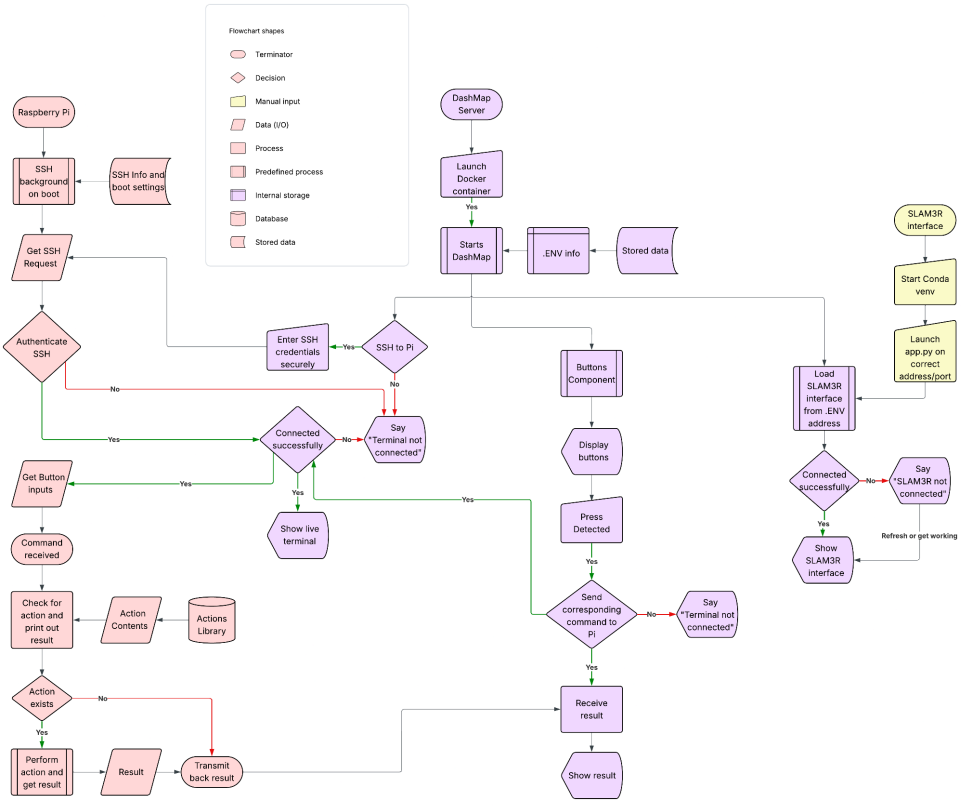

Figure: System Diagram

We will now detail the process used to create the maps of Hudson's apartment, following this system diagram. First, launch the SLAM3R virtual environment, navigate to the correct directory, then launch SLAM3R's browser interface with *python3 app.py*. The setup of this environment is detailed in SLAM3R's repository and is straightforward, although we could never get their optimized kernels to work. Then navigate to Docker and launch the DashMap container which can be simply cloned from the Mapper repository and added to Docker. Go to the link where DashMap is served. Mapper needs to be powered on and have the corresponding RPi code installed from our GitHub. DashMap and Mapper are configured to communicate with each other by default, so as long as Mapper is connected to the network, the SSH button will work. Press that button, enter SSH credentials, and watch as the Terminal component populates. Now, the buttons in the Buttons component should work. If SLAM3R's interface was launched before DashMap, it should auto populate in its component view. If not, simply refresh DashMap. Now that all components are synced, press the launch livestream button and the camera's live view with FPS overlaid will appear. Then, turn on a Bluetooth Controller in pairing mode to auto-pair to Mapper directly. Hold the right bumper to shift into "precision mode" where Mapper drives slowly. Navigate using the camera view and explore the space while recording frames. At the end, press "stop recording" and go to the SLAM3R interface. Click upload directory to start analyzing that recording. When it is done, inspect the map and download it. Now we have created a high-quality map of our space using a simple robot and a powerful interface.

Our results with actually creating maps using SLAM3R are successful. We generated identical demonstration maps to test that our model environment was functional. Then we provided a 27 second clip taken with Mapper at 30FPS 1080p with more than half of the clip being out of focus. The map from this test proved that we could proceed with SLAM3R.

Limitations

The types of limitations we will focus on are accessibility, performance, and difficulty. Accessibility refers to what hardware, systems, and infrastructure we were able to use to accomplish our goal.

SLAM3R's corresponding paper runs all tests with an Nvidia RTX 4090D GPU, where we were provided a rig with 4x 2080 Ti's which each have less than half the VRAM of a 4090. This posed a limitation in comparing our results because we had to use different settings for SLAM3R so it wouldn't crash. This led us to spend significant effort in parallelizing SLAM3R, which is not worth doing while it is still in development. We also used a Windows 10 computer with an RTX 3060 6GB and the Windows Subsystem for Linux (WSL) that needed to be set up for CUDA. Once this worked, it achieved similar quality but suffered from the even smaller VRAM limit, so settings needed to be turned down or else runtimes exploded. We observe a generally linear VRAM consumption rate as the number of frames increases. The model takes exponentially longer when the limit is hit. The proclaimed 25 FPS that SLAm3R's creators mention multiple times is only possible at low resolutions for bursts with preprocessing and a 4090. We show mapped sections of the apartment but were unable to fit an entire map in 6GB of VRAM.

We wanted to achieve both the autonomous exploration and the high-quality maps so we spent more than 50 hours working with other models trying to accomplish this when we learned that SLAM3R wouldn't have the online functionality in time for us. This was mentioned earlier but we did not go into depth about these struggles because we decided to pivot to a more holistic implementation. These models are evolving, so we prefer to look to the future and not get stuck in the past. The lesson with SLAM3R is that cutting-edge models might not support all of their functionality and should not be relied upon for fast deadlines.

The Raspberry Pi's video stream would be much better, at higher resolution and frame rates while being more consistent, if we simply used a Pi 5 instead of a Pi 3. This would make better maps. The lesson from this is to order extra parts early. The Raspberry Pi also needs 5V/3A or 5V/5A which requires a buck converter or a premium power bank that supports the special PD spec. Thankfully, fellow EC535 student Dakota Winslow lent us his large Anker battery for our project. Carefully considering the power requirements of different microcontrollers, especially when motors get mixed in, is important. It can easily create roadblocks. We bypassed this difficulty by keeping the ESP32 on its own battery so the motors never influence the power system of the RPi.

The BU VPN and firewall posed challenges throughout development because the Raspberry Pi does not support the BU VPN. When testing streaming frames directly to the GPU server, we had to set up SSH tunnels suffered from networking complexities with WSL in the middle of that. In our final implementation, we do not use the remote GPU server but could easily connect it if we wanted. This simplified the frame streaming pipeline at the cost of running SLAM3R slower. We had to resort to this because the GPU rig never came back online after the driver update. This is why we don't discuss parallelizing the model; it got removed from the final product. As SLAM3R gets updated, and if we wanted to use it more fast paced, we could connect this remotely as previously discussed.

Future Work

When the online functionality of SLAM3R gets released, it will seamlessly integrate with DashMap with a simple git pull. We would focus on using this online version of the model to explore autonomously, extracting pose information and identifying open doorways as frontiers. We could do this identification on the GPU server and simply stream back directions to Mapper. We would love to parallelize SLAM3R in a way that supports the upcoming online functionality and to create a single map in parallel with multiple GPUs. If possible, we could use two cheap cameras to create our own depth camera and potentially lighten the computation load of SLAM3R with this additional information. We also discussed mixing passes at different focus settings, or using different focal length cameras at the same time to potentially improve SLAM3R's performance in various circumstances like outdoor mapping. Mixing focus distances could be implemented quickly because we chose a camera with autofocus so it would just be a new button with some specific commands mapped to it. It might create sharper maps by keeping the high confidence, sharp focused points from each pass to get the best of both worlds. We would also like to connect our ESP32 to the RPi with a cable bus directly to skip the Bluetooth but this was avoided because the RPi was loaned to us and the ESP32's motor driver shield blocks the pins.

We are excited about these dreams of running SLAM faster, at higher quality, while exploring autonomously. We are investigating using DashMap and Mapper, as well as SLAM3R or another cutting edge model, as part of our Senior Design project in the upcoming fall semester.

Conclusion

Robots are the future. The technological environment that underlies many robotics projects can be distilled into a powerful and flexible tool in the form of a dashboard made of components. We propose DashMap, a modular dashboard with generalized components bridging the gap between a small robot and a big cloud. We create Mapper, a simple robot to stream frames into DashMap to be processed into a high quality 3D map using a cutting edge SLAM model called SLAM3R. DashMap and Mapper are packaged in a GitHub repository and quickly deployable with simple instructions and a thoughtful Docker container. We are excited to use DashMap and Mapper for robotics projects and can't wait to start exploring autonomously with these tools!

References

ChenHoy/DROID-Splat at mcmc. (n.d.). GitHub. Retrieved May 6, 2025, from https://github.com/ChenHoy/DROID-Splat

Homeyer, C., Begiristain, L., & Schnörr, C. (2024). *DROID-Splat: Combining end-to-end SLAM with 3D Gaussian Splatting* (arXiv:2411.17660). arXiv. https://doi.org/10.48550/arXiv.2411.17660

HudsonReynolds2/mapper: End to end implementation of online real time monocular slam for indoor autonomous mapping with no other sensors. Uses offsite GPU rig, raspberry pi, and esp32. Currently working: Real time teleoperated, remote map creation. (n.d.). Retrieved May 6, 2025, from https://github.com/HudsonReynolds2/mapper/tree/main

Princeton-vl/DPVO: Deep Patch Visual Odometry/SLAM. (n.d.). Retrieved May 6, 2025, from https://github.com/princeton-vl/DPVO

Princeton-vl/DROID-SLAM. (n.d.). Retrieved May 6, 2025, from https://github.com/princeton-vl/DROID-SLAM

Rmurai0610/diff-gaussian-rasterization-w-pose at 43e21bff91cd24986ee3dd52fe0bb06952e50ec7. (n.d.). Retrieved May 6, 2025, from https://github.com/rmurai0610/diff-gaussian-rasterization-w-pose/tree/43e21bff91cd24986ee3dd52fe0bb06952e50ec7

SLAM3R: Real-Time Dense Scene Reconstruction from Monocular RGB Videos. (n.d.). Retrieved May 6, 2025, from https://arxiv.org/html/2412.09401v1

What Is SLAM (Simultaneous Localization and Mapping)? (n.d.). Retrieved May 6, 2025, from https://www.mathworks.com/discovery/slam.html