FREEZETAG:
DESIGNING AND BUILDING A SELF-HOSTED IMAGE MANAGEMENT APPLICATION

by

Sathya Tadinada

A Senior Honors Thesis Submitted to the Faculty of
The University of Utah
In Partial Fulfillment of the Requirements for the

Honors Degree in Bachelor of Science

In

Computer Science

Approved:

Dr. Aaron Wood
Faculty Thesis Mentor

Dr. Mary Hall
Director of the Kahlert School of Computing

Dr. Pratik Soni
Departmental Honors Liaison

Dr. Monisha Pasupathi
Dean, Honors College

April 2026

ABSTRACT

FreezeTag is a free and open-source, self-hosted image management application designed for photographers, small businesses, and technically-inclined users who want meaningful control over how their photos are stored and organized. The core premise is straightforward: instead of uploading your photos to a cloud service you don't own, FreezeTag runs on hardware you already have (i.e., a personal computer, a NAS device, or anything capable of running Docker), and your data never leaves your own infrastructure. That eliminates the recurring storage fees that cloud services charge as collections grow, and it means there is no third-party outage that can take your photo library offline. Because FreezeTag's interface is entirely browser-based, any device on the same network can connect to a running instance without any software installed on the client side.

FreezeTag was developed by a four-person team during the University of Utah's Kahlert School of Computing senior capstone sequence, spanning the Fall 2025 and Spring 2026 semesters. We followed an Agile development process organized into three development phases, with regular standups, retrospectives, and code review requirements tracked through GitLab. This thesis covers the design and architecture of FreezeTag, the engineering practices we adopted as a team, and my individual contributions as the frontend interface engineer (such as the core UI implementation, functionality for custom theme creation, and a map widget to view photo locations). It also reflects honestly on what the experience was like, including what worked, what I would do differently, and what building a real, collaboratively-developed application from scratch taught me about software development as a discipline.

TABLE OF CONTENTS

INTRODUCTION AND BACKGROUND

Introduction to FreezeTag

A frustrating aspect about modern image management tools is that they all seem to force essentially the same tradeoff: you can either have something convenient, or something you control, but not both. Cloud services like Google Photos [1] and Apple Photos [2] are very easy to use and widely accessible, but they store data on remote servers owned by third parties, charge recurring fees that grow with the size of a user's collection, and the software is basically a black box that you cannot customize. On the other end of the spectrum, tools like DigiKam [3] provide local storage and more configurability, but require the application to be installed on every device the user wants to access their photos from, and setting up networked access across multiple machines is a cumbersome manual process.

The other thing that bothered us about existing tools is how they all handle organization. Folder hierarchies are the default everywhere, which works fine until your library gets large or complex enough that a single image reasonably belongs in multiple categories at once. Google Photos has this existing folder structure, and they also have a unique approach with Gemini [4] AI-powered search, but it still does not give you a way to define your own categorical labels in any structured way. DigiKam technically has support for tags, but they are clearly secondary to the folder model. Apple Photos has albums, which are basically just folders with a nicer name. None of these tools treat user-defined granular organization as a primary feature, which means (as an example) a photographer who wants to label a batch of images by client, usage rights, subject, and location all at once, and then later query across any combination of those, has no good way to do it.

The target users are people who have already decided they want to own their own data. Freelance photographers get a tagging system built around organizing by client or event, with the guarantee that original file formats are always preserved. Small businesses running on isolated or air-gapped infrastructure can integrate FreezeTag directly through the API into automated workflows without a dependency on a third-party service's uptime or pricing. And technically-inclined hobbyists who are already running a NAS at home get software that fits into that setup naturally, with no accounts, subscriptions, or external services required. In all of those cases, the self-hosting model also means the user is fully in control of their own uptime.

The design decision we kept coming back to was making tags the primary organization unit, as opposed to folders or albums. It sounds simple in theory, but a natural consequence of this is that any image can belong to multiple logical groupings simultaneously, which is something a folder hierarchy just cannot do cleanly. A user would be able to apply tags like "client:SteveJobs", "location:SaltLake", and "format:print" to a batch of images when they upload it, and then later retrieve exactly that subset of their collection by searching across any combination of those tags. This model would also scale much more naturally to large or complex libraries than folder structures do, since a single image is able to exist in multiple categories without needing to be duplicated or moved across directories.

On the deployment side, we wanted the setup to be as painless as possible. The entire system runs through Docker Compose [5] , so getting a FreezeTag instance running takes a single command regardless of what operating system the host machine is running. Since the interface is browser-based, any device on the same network can connect without installing anything on the client side, and photos never leave the host machine. After the capstone sequence wraps up in the summer of 2026, the project is planned for release under the MIT open-source license, so it will be free to use, modify, and distribute. On top of the core gallery and tagging functionality, FreezeTag is also extensible through a Python-based plugin system, which the Tech Stack and Architecture chapter covers in greater detail.

Team Formation

FreezeTag was built by four developers across roughly two semesters of active development time, with each of us spending around ten hours a week on the project. Development began in the Fall 2025 semester and concluded near the end of the Spring 2026 semester. We used GitLab [6] for hosting our code repository, issue board, and contribution guidelines. The team was split into two frontend engineers and two backend engineers, with each member owning a clearly defined area of the codebase from the start of the project. The developers on the project were:

- Ethan Collier, Backend Feature Engineer
- Brayden Jonsson, Frontend Architecture Engineer
- Max Petersen, Backend Architecture Engineer
- Sathya Tadinada (the author of this thesis), Frontend Interface Engineer

On the frontend side, Brayden was responsible for lower-level concerns such as parsing and handling API responses, managing frontend state, and handling authentication on the client side, while I was responsible for the visual design and CSS implementation of all user-facing pages. On the backend side, Ethan owned the API endpoints and the primary user-facing features such as image storage and metadata parsing, while Max was responsible for dependency management, the plugin environment, and deployment infrastructure. The main area where these roles naturally blurred was API contract decisions, since those sit right on the boundary between what the backend exposes and what the frontend needs, and those conversations happened between all four of us pretty regularly. We ran the whole thing as an Agile process organized around four-week sprints, which is covered in more detail in the Software Methodologies and Techniques chapter.

Tech Stack and Architecture

At a high level, FreezeTag consists of two servers that communicate with each other and with a set of Python plugins. The Go [7] backend serves as the core of the application, handling API requests from the frontend, reading and writing image metadata to a SQLite [8] database, invoking ImageMagick [9] for image format conversion and metadata parsing, and managing the lifecycle of Python plugins. The Next.js [10] frontend communicates with the backend over HTTPS and is responsible for all user interaction. The Python plugin layer sits below the backend and receives image and metadata payloads through a pipeline, returning processed results that the backend then stores or acts on. The entire system is deployed and orchestrated using Docker Compose, which allows the frontend server, backend server, and plugin environment to be started together with a single command. Fig. 1 shows how these pieces fit together.

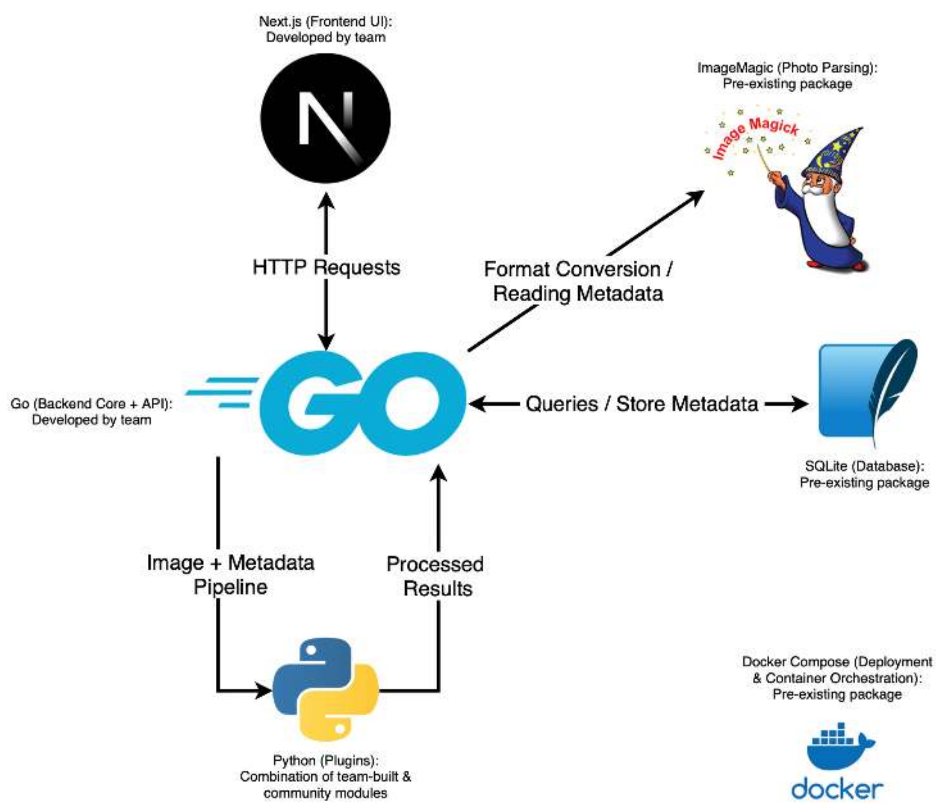


Figure 1: System Architecture Diagram

The backend server is written in Go, which we chose for its performance, strong concurrency support, and straightforward deployment characteristics. Go's goroutine model ended up being a good fit here, since having plugin execution and API request handling running concurrently without the concurrency management overhead we would have had to build ourselves in most other languages. The backend uses the Gin [11] web framework for routing and middleware, which also automatically generates Swagger documentation for all API endpoints through Gin-Swagger. This was useful during development because it gave the frontend team an always-current reference for the available endpoints without needing to read through backend source code directly.

The backend uses SQLite as its database, which stores all image metadata, tag associations, user information, and plugin configuration. SQLite was an easy call here, since the whole point of FreezeTag is that a non-technical user should be able to set it up and run it on hardware they already own, and a database that lives in a single file and requires no separate server process is much more user-friendly in that context than something like PostgreSQL or MySQL. Image format conversion and EXIF metadata extraction are handled through ImageMagick, a widely used open-source image processing library that supports an extensive range of file formats. On upload, ImageMagick extracts embedded metadata fields such as timestamps, GPS coordinates, and camera model, and also generates compressed thumbnail versions of images in the WebP [12] format for efficient display in the gallery interface.

The frontend is built with Next.js, a React-based framework that handles both server-side rendering and static asset serving. We used pnpm [13] as the package manager for the frontend project, and Jest [14] for unit testing. Image uploads are handled through react-dropzone [15] , which provides the drag-and-drop upload interface, and all communication with the backend API is done with a standard fetch API via undici [16] .

The plugin environment is managed using uv [17] , a fast Python package manager, and each plugin runs in its own isolated virtual environment to prevent dependency conflicts between plugins. Communication between the Go backend and the Python plugins happens over a loopback port using a custom REST protocol, with the backend acting as the orchestrator that decides when to invoke a given plugin and what data to pass to it. Plugins can also read from a shared SQLite table that the backend exposes to them in read-only mode, which allows plugins to make decisions based on existing tags and metadata without needing a separate API call.

The plugin system is one of the most distinctive aspects of FreezeTag's architecture. Rather than building every possible image processing feature directly into the backend, we designed a plugin interface that allows Python scripts to hook into the image upload pipeline and contribute new tags, captions, or other metadata based on the content of the uploaded image. Plugins are listed and managed through the plugin management page in the frontend, where users can enable or disable individual plugins and view their version and available hooks. The first-party plugins that shipped with FreezeTag by the end of development were:

- Face Recognition: detects and identifies faces in uploaded images using a local machine learning model, and automatically applies name-based tags to images containing recognized individuals.
- Google Gemini Tagger: sends uploaded images to Google's Gemini API [18] and uses the model's vision capabilities to generate descriptive tags automatically, providing a cloud-assisted tagging option for users who are comfortable with that tradeoff.
- ML Tagger: runs a local machine learning model to classify image content and suggest tags without sending any data to an external service, making it suitable for users who want automated tagging with full data locality.
- RAM Tagger: a lightweight local tagger that operates with a smaller memory footprint than the full ML Tagger, intended for users running FreezeTag on hardware with limited RAM.

Each team member was also responsible for implementing at least one substantial Rank 3 feature independently. My two Rank 3 features were adding UI for displaying photo locations on an interactive map and functionality for importing custom themes. These are covered in detail in the Individual Contributions chapter.

INDIVIDUAL CONTRIBUTIONS

Role in the Project

As the frontend interface engineer on FreezeTag, my primary responsibility was the visual design and CSS implementation of every user-facing page in the application. The frontend team was divided between two distinct roles. Brayden, as the frontend architecture engineer, was responsible for the lower-level concerns of the frontend, such as managing API communication, handling authentication state, and structuring how data flowed through the application. My role sat on top of that foundation, taking the data and API integrations Brayden built and turning them into the actual interface that users see and interact with. In practice, the two roles required close and ongoing coordination throughout both semesters, since good UI components need real data behind them and data-handling code needs a UI surface to be meaningful. The pages I did primary development on were the overall application UX, the gallery page, the image detail view, the tags management page, the settings page, and the image detail sidebar. My two Rank 3 features were adding UI for displaying photo locations on an interactive map within the image detail view and custom theme importing functionality accessible through the settings page.

Frontend Interface Implementation

The gallery page is the primary view of the application and the page that users land on after logging in. It displays all uploaded images in a responsive masonry-style grid, with each image rendered as a WebP thumbnail generated by the backend at upload time. The top of the page contains a search bar that accepts a custom query syntax, with example queries shown as hints below the bar to help new users understand how to construct searches. Alongside the search bar, there are three controls: a Tags dropdown, a Sort dropdown, and a Select button.

The Tags dropdown was one of the more interesting design problems. The naive implementation would just show every tag in the library, but that gets useless fast with a large collection. Instead, I built it around an intersection model, where selecting a tag filters the gallery, and the dropdown would immediately update to show only the tags that actually appear on those filtered images. You would then be able to progressively narrow your search, and at every step, the dropdown continues only showing you options that are actually relevant. This behavior directly reflects FreezeTag's tag-first philosophy, where combinations of tags are the primary way users navigate their library. Fig. 2 shows how the dropdown looks when open, with the available tags updating in real time as the user selects from them.

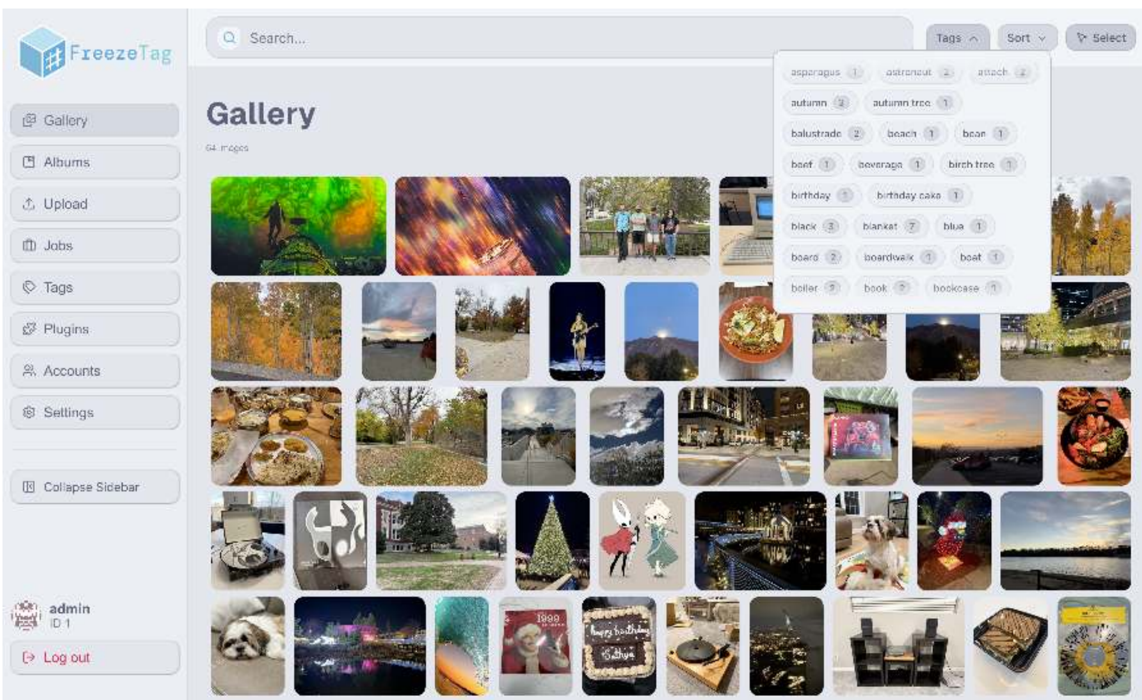


Figure 2: Gallery Page showing the Tags dropdown open

As shown in Fig. 3, the Sort dropdown allows users to sort the gallery by either date created (the date the photo was taken, extracted from EXIF metadata) or date added (the date the image was uploaded to FreezeTag), and to order the results either newest first or oldest first. These two sort dimensions are distinct and have their own use cases, where a user who just uploaded a batch of old scanned photos would want to sort by date added to find what they just uploaded, while a user browsing their collection chronologically would want to sort by date created.

The current design went through some iteration before landing here, since the first method that we had for sorting photos was to add a filter query directly to the search bar. For example, users would type “sortBy=DateAdded; sortOrder=Newest” to sort by date added in descending order. This worked fine, but it was not very discoverable or user-friendly, especially for non-technical users who might not be familiar with the query syntax. Moving the sorting controls into a dedicated dropdown made the feature much more accessible and intuitive, and it also allowed us to show the currently applied sort method and order directly in the UI without needing to parse it out of the search query string.

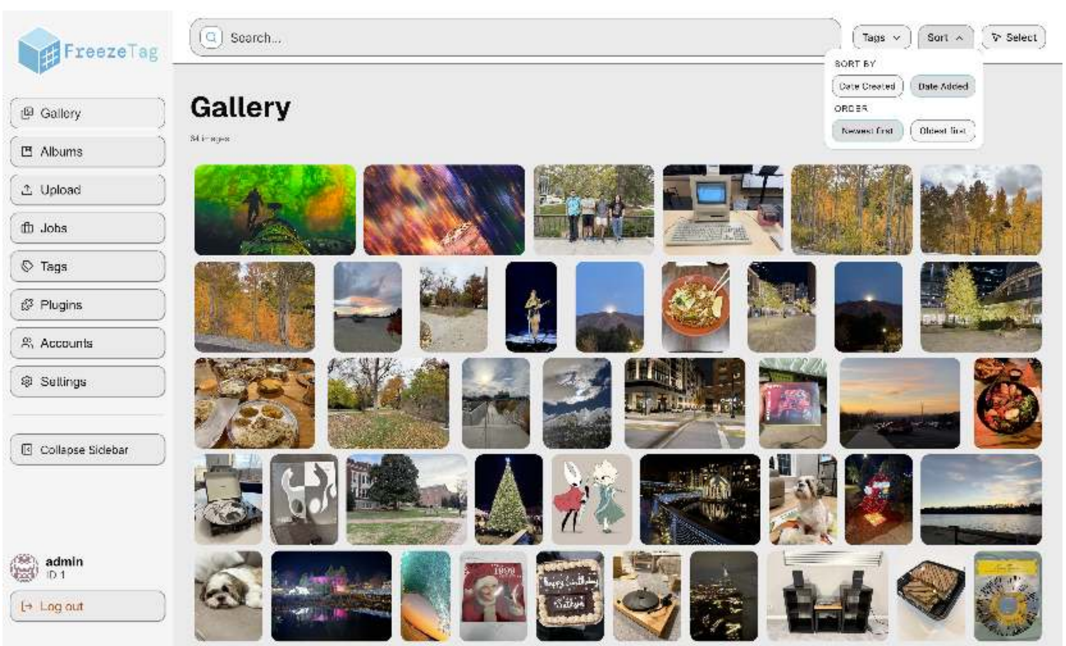


Figure 3: Gallery Page showing the Sort dropdown open

The Select mode was motivated by a pretty specific workflow, which is that a user who just bulk-uploaded a large amount of photos from a shoot would need to be able to tag all of them at once without opening each image individually. Fig. 4 shows how clicking Select adds a tag panel onto the right side of the screen and lets you check off images to build a selection, then apply or remove tags across all of them in one action. This workflow also enables users to run various plugins across a selection of images, which can be extremely helpful for similar reasons.

This feature was added primarily because as we were using the app, we realized how slow it was taking for us to tag photos one-by-one after uploading tens or hundreds of images in one go. Conveniently for us, this component could be reused from our Upload page (mentioned later in the chapter), saving us a lot of development time and giving the user a consistent interface for bulk-tagging. (Unfortunately, we spent around an hour trying to build the functionality from scratch before we came to that realization.)

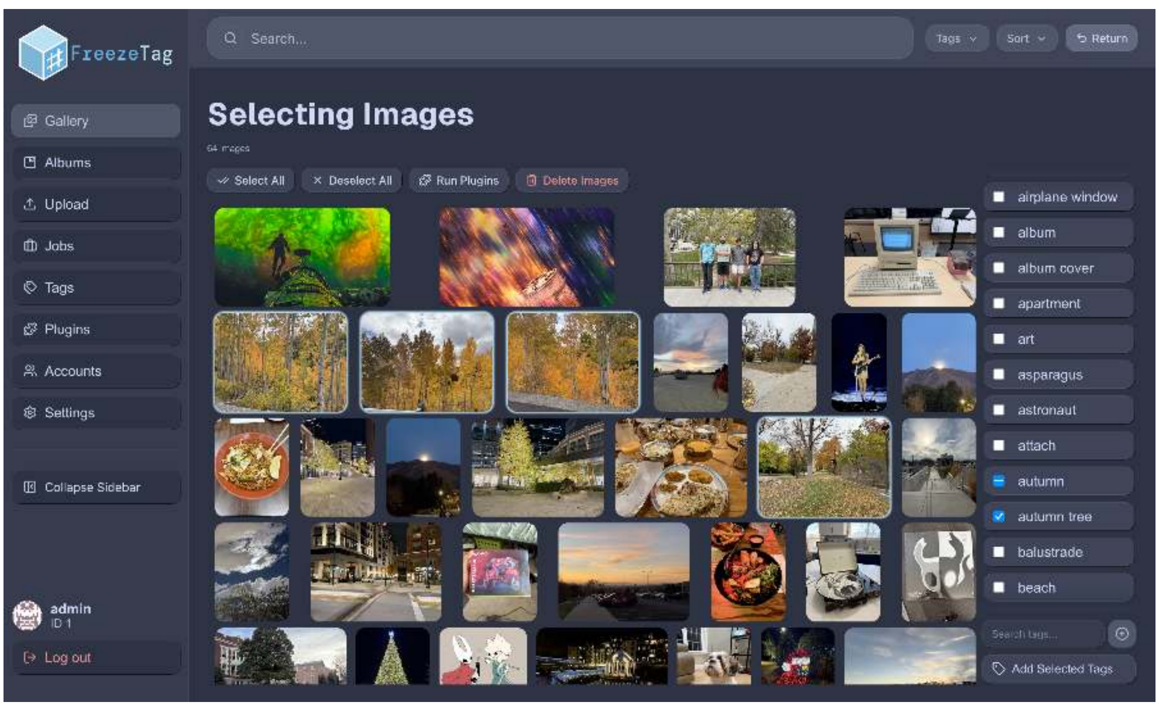


Figure 4: Gallery Page showing the Select mode with the tag panel visible

Clicking any image in the gallery opens an image preview view, demonstrated in Fig. 5, which is a full-screen overlay showing the image at full resolution alongside a collapsible metadata sidebar. The image viewer supports zoom controls at 1x and 2x magnification, and left and right arrow buttons allow the user to navigate through the current gallery view without closing the overlay. The metadata sidebar displays all EXIF data extracted from the image at upload time, including resolution, date taken, date uploaded, GPS coordinates, and camera model. Below that is a scrollable tags section where users can view, remove, and add tags on a per-image basis. The sidebar can be hidden entirely if the user wants to focus on the image itself.

This component ended up being one of the most technically challenging parts of the frontend to implement, since there were a lot of moving parts that were not immediately apparent when I first tackled this feature. As an example, the zoom functionality was surprisingly complex to zoom in/out relative to the location of the cursor. On top of this, the navigation buttons also had some tricky edge cases to work through, since they needed to be aware of the current gallery filters and sort order to know which images were next in line.

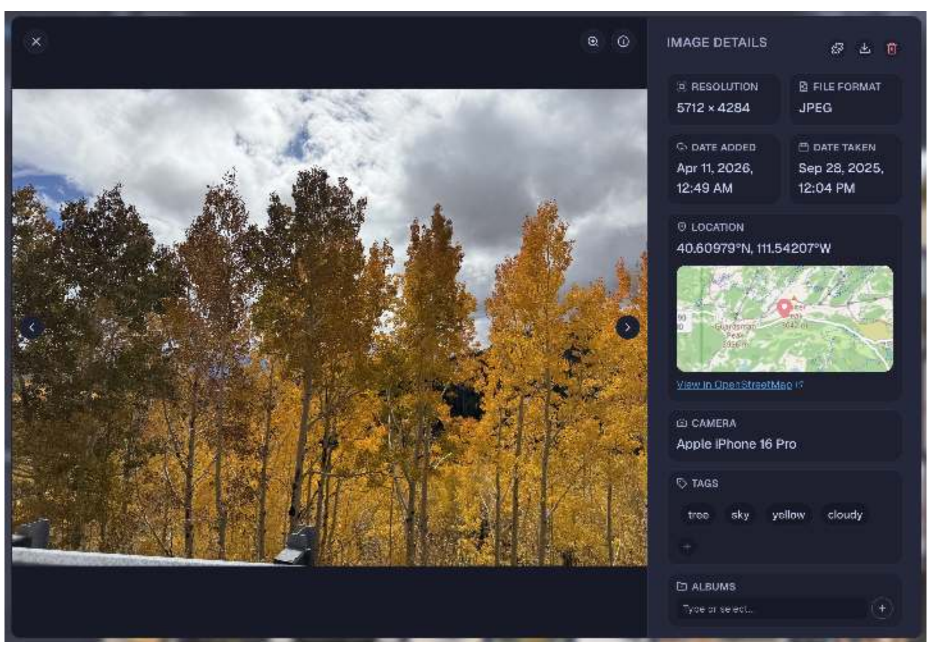


Figure 5: Preview Window Panel

The upload page allows users to add new images to their FreezeTag library, as shown in Fig. 6. Users can either click an “Upload Images” button to open a file picker or drag and drop images directly onto the button. Once images are staged for upload, they appear as thumbnails in the main area of the page, and the tag panel from the gallery’s Select mode appears on the right side of the screen. This allows users to select from existing tags or create new ones and apply them to the entire batch before finalizing the upload. The Select All and Deselect All buttons at the top make it easy to apply a set of tags to every image in the batch at once.

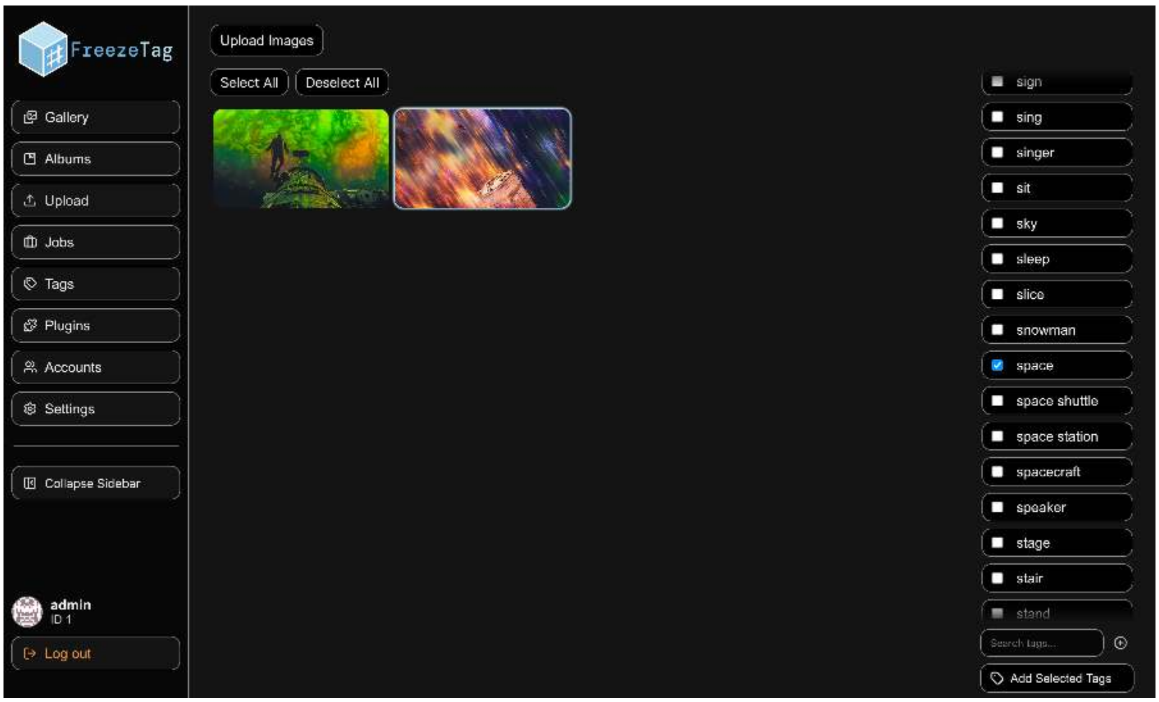


Figure 6: Upload Page

The tags management page provides a dedicated interface for managing all tags stored across the entire library. Fig. 7 demonstrates how it displays a paginated list of every tag in the system alongside a count of how many images currently carry that tag. Users can search across all tags using a fuzzy search bar at the top of the page, which makes it easy to find a specific tag in a large library. Each tag row has a navigation arrow that filters the gallery to show only images with that tag, and a delete button that removes the tag from all images. Users can also select multiple tags using the checkboxes and mass-delete them in a single action, which is useful when a plugin has generated a large number of unwanted or duplicate tags.

This was another feature that ended up being something we realized after-the-fact that we needed. Initially, my design for tag management was just the tag panel in the image metadata sidebar. That worked fine for managing tags on a per-image basis, but we quickly realized that we also needed a way to manage tags across the whole library, especially for bulk-deleting tags that were generated by plugins or applied to large batches of images.

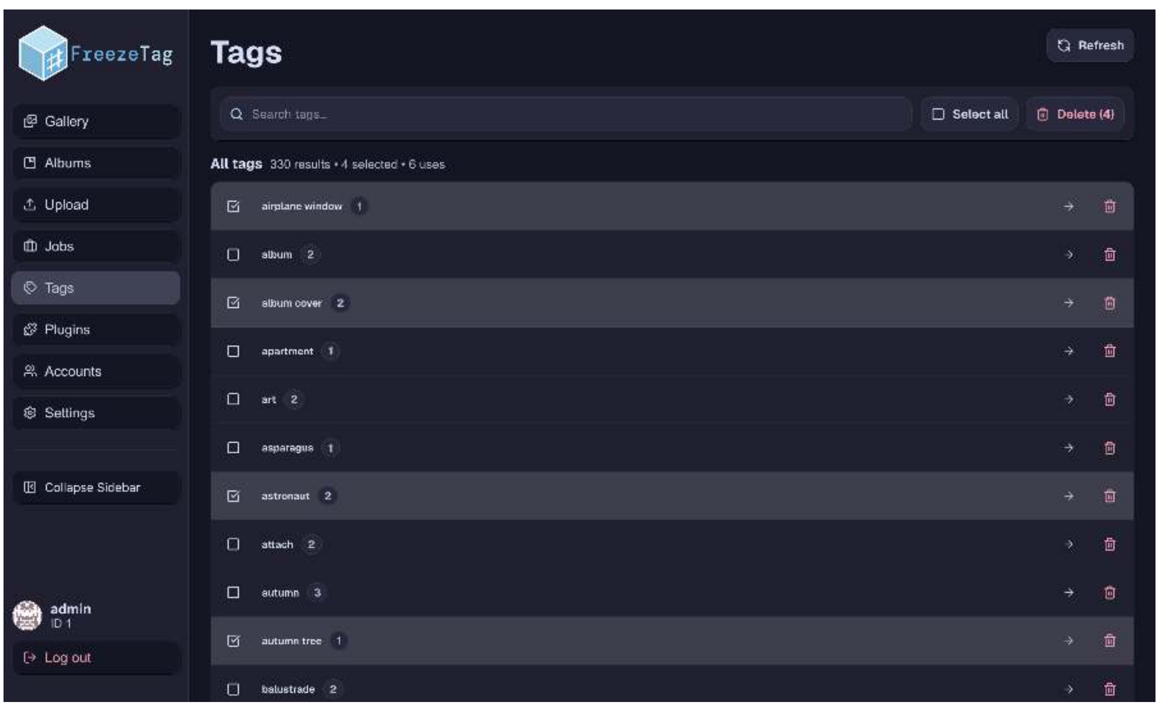


Figure 7: Tags Management Page

Fig. 8 shows how the Settings page is organized. The Profile section allows users to upload and change their profile picture, which appears in the bottom left corner of the sidebar throughout the application. The Preferences section contains three settings: a theme selector, a unit toggle, and an option to disable the map widget. The theme selector currently allows users to choose from a set of built-in Catppuccin [19] themes, which is a popular open-source color scheme. (The observant readers will notice the various themes utilized throughout the screenshots in this chapter.) The unit toggle switches between metric and imperial default units. The Security section contains a password change form.

Similar to the Preview Window feature, this page initially seemed straightforward to implement, but it ended up having a lot of hidden complexity once I got into the details. An example of this is the custom theme creation feature (mentioned in detail in the Rank 3 section), which required significant development effort to implement properly while maintaining good responsive design throughout the page. Most of these details stemmed from us wanting to give users as much flexibility as possible with customizing various aspects of the application, since we did not want to force them into a one-size-fits-all design.

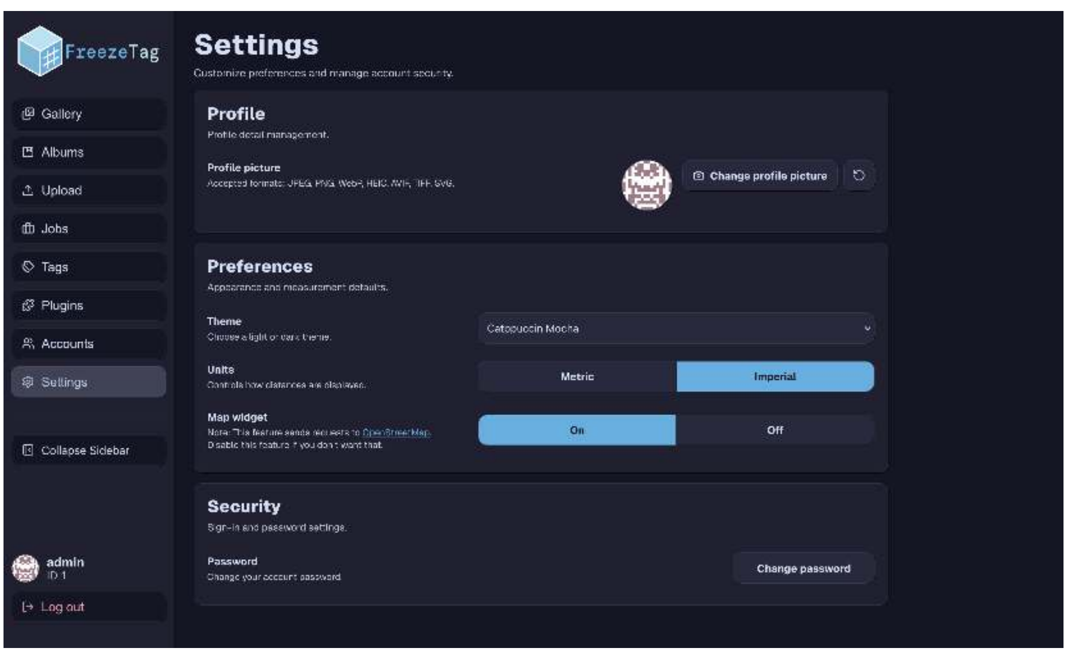


Figure 8: Settings Page

Rank 3 Features

My two Rank 3 features were the map widget within the metadata sidebar and custom theme creation through the settings page.

The map feature is integrated directly into the metadata sidebar when an image has GPS coordinate data available, located above the camera information section. The design intent is similar to how Google Photos surfaces a small embedded map within its photo detail view, showing the approximate location where the photo was taken as a pin on a mini map. Fig. 9 shows the map rendered using Leaflet [20] with OpenStreetMap [21] tile data, both of which are open-source and do not require an API key (consistent with FreezeTag's goal of being fully self-hostable without requiring accounts or paid services). Because the GPS coordinates are already extracted from EXIF data at upload time and stored in the database, the map component only needs to read those coordinates and pass them to Leaflet to render the pin.

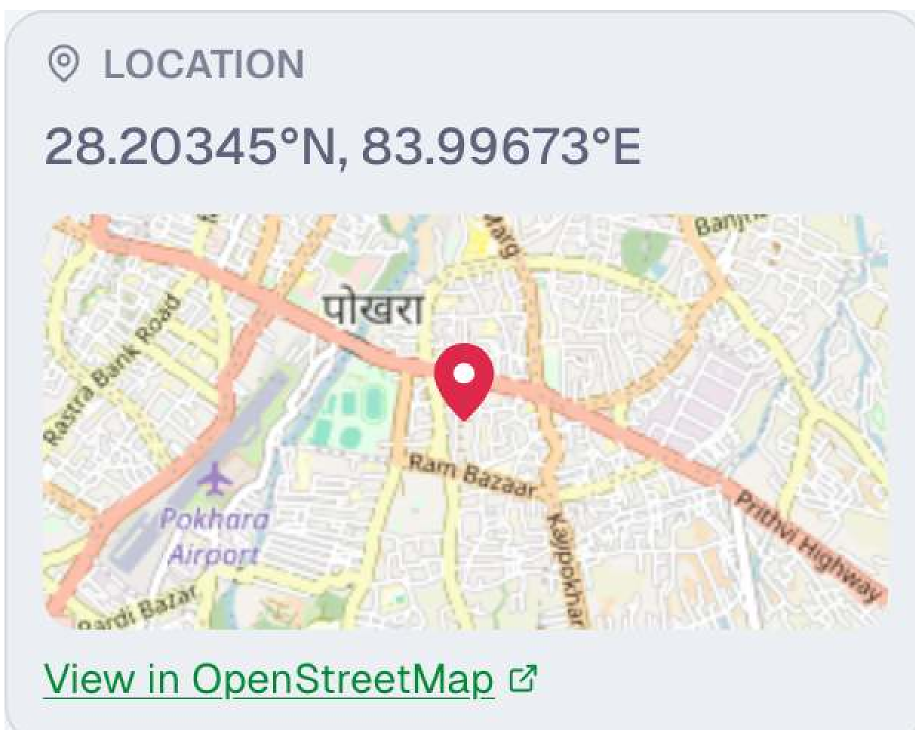


Figure 9: OpenStreetMap Feature

The custom theme creation feature, shown in Fig. 10, is accessible through the Preferences section of the settings page. The goal is to allow users to create a custom color palette for their FreezeTag instance by specifying a set of Catppuccin CSS variable names and the color values they want to assign to each one. This approach builds on the Catppuccin theming system already in place in the application, so users who are familiar with Catppuccin's variable naming conventions can customize their instance without needing to write any CSS directly.

Additionally, users can import and export their custom themes as JSON files, which allows them to share their themes with other users or back them up for safekeeping. When a user imports a custom theme, the application validates the JSON structure to ensure it contains valid Catppuccin variable names and color values before applying it. The themes are stored in the browser's local storage, so they persist across sessions but are specific to each user on a given machine.

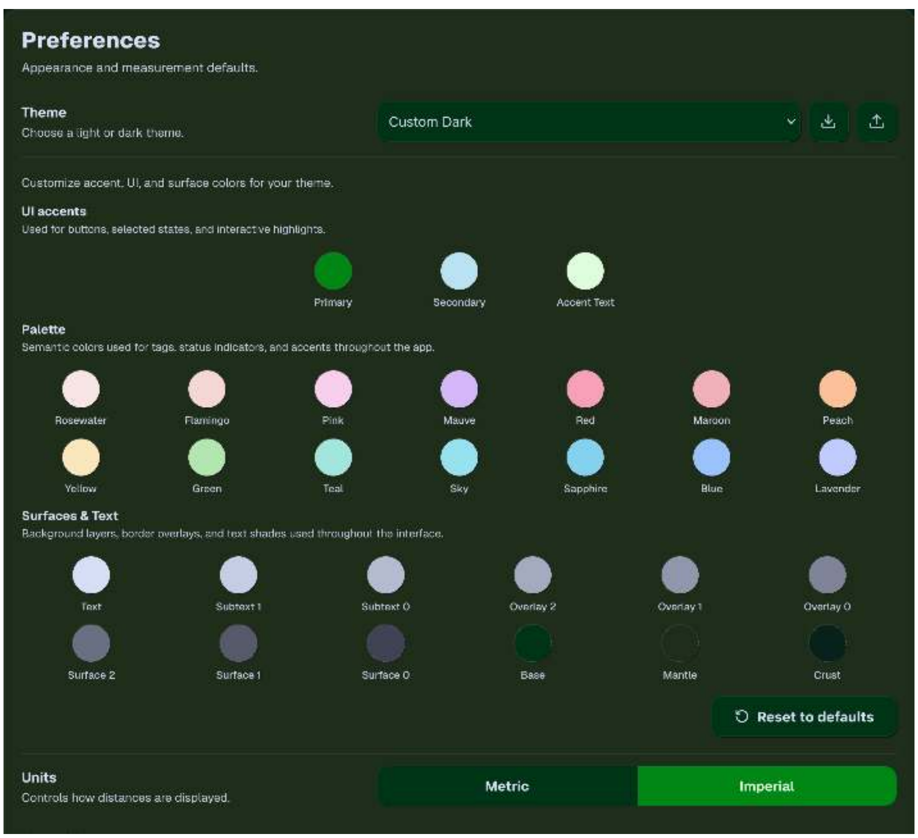


Figure 10: Custom Theme Creation Feature

SOFTWARE METHODOLOGIES AND TECHNIQUES

Agile and Development Process

From the beginning of the project, our team adopted an Agile [22] development process. Agile was a natural fit for FreezeTag because the application had a large number of features that were interdependent on each other, meaning that the specific implementation details of one part of the system frequently had implications for how another part needed to be built. Rather than trying to plan every detail upfront and stick to a rigid schedule, Agile gave us the flexibility to respond to those kinds of discoveries as they came up during development without derailing the project.

Development was divided into three phases (Alpha, Beta, and Release), each roughly four weeks long. In theory, these were checkpoints for deliberate reprioritization. In practice, however, especially by the Beta phase, they mostly served as a forcing function to confront whatever had slipped and decide whether to carry it forward as a priority or cut it. Despite the challenges, the checkpoints gave us a structured moment to recalibrate without needing to hold separate retrospective meetings throughout the phase.

Within each phase, work was tracked using GitLab's [6] issue board, as highlighted in Fig. 11. We created issues for every feature, bug, and task across both the frontend and backend, and used a set of labels to organize them. Phase labels (Alpha, Beta, Release) indicated which phase an issue was targeted for. Domain labels (Frontend, Backend) indicated which side of the codebase the issue belonged to. Each issue also carried a state label (Backlog, To Do, In Progress, Done) that was updated as work progressed, giving the whole team a clear and current picture of where development stood at any given time. Fig. 12 shows an example of a specific issue, with the phase, domain, and state labels visible alongside the issue description and acceptance criteria.

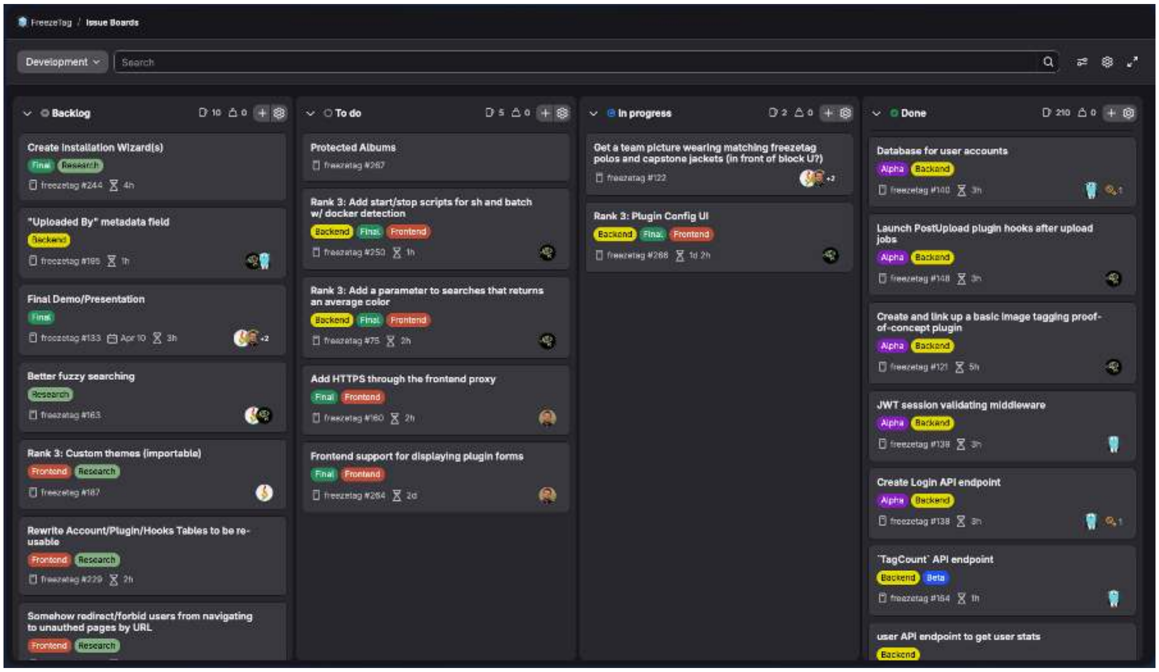


Figure 11: GitLab Issue Board showing label columns and issue states

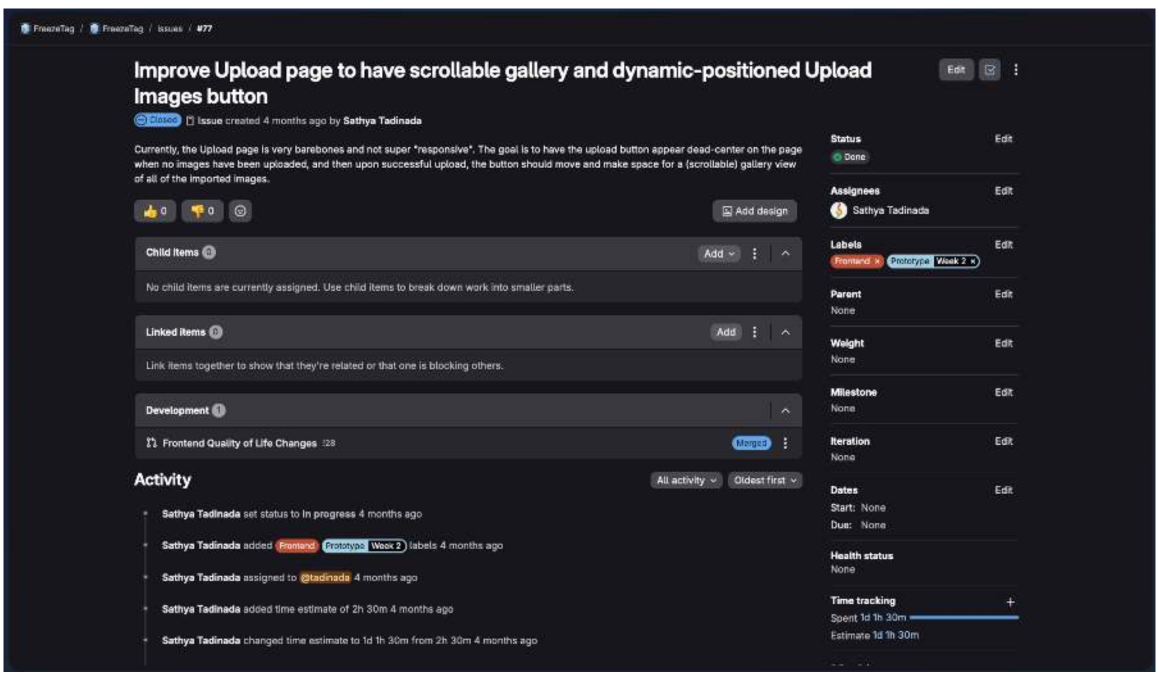


Figure 12: GitLab Specific Issue showing labels, assignee, and description

Meetings and Communication

Our team met three times per week throughout most of both semesters. The Friday staff meeting with the course instructor was held in person. Outside of that, we met remotely on Monday evenings to align on goals for the week ahead, and again on Wednesday or Thursday evenings to review progress, work through any blockers, and prepare for the Friday meeting. Most communication between meetings happened asynchronously over Discord, which gave us the flexibility to work on our own schedules while still staying coordinated. Scheduling was not always straightforward, given that four students have very different weekly commitments, so we kept the number of required meetings small and relied on the asynchronous channel to fill in the gaps.

Tools and Infrastructure

Before any application code was written, the team prioritized setting up a CI pipeline that would automatically enforce quality standards on every merge request targeting the main branch. Max was the primary person responsible for building and maintaining this system. Every merge request was required to pass three automated checks before it was eligible for human review: a format check, a lint check, and a unit test run.

The format check required all code to be formatted with the standard tool for its language. We used Prettier [23] for the frontend and gofmt [24] for the backend. Consistent formatting across the codebase reduces unnecessary noise in diffs and makes the repository history easier to follow when multiple people are working in overlapping areas.

The lint check required all code to pass the appropriate linter with no warnings or errors. We used ESLint [25] for the frontend and golangci-lint [26] for the backend. Both tools were configured with their default rulesets as the baseline, with a small number of additional rules added on top to suit our specific needs.

The unit test check required all existing tests to pass and enforced a minimum code coverage threshold of 80% across both the frontend and backend. Frontend visual components were largely exempt from the coverage requirement since they are primarily layout and styling code that is better verified by hand than by automated tests. By the end of development, the frontend had reached approximately 95% code coverage and the backend approximately 85%, both comfortably above the minimum. Fig. 13 shows an example of a merge request with the CI pipeline status visible, including the results of the format, lint, and test checks, as well as the reviewer approval section below.

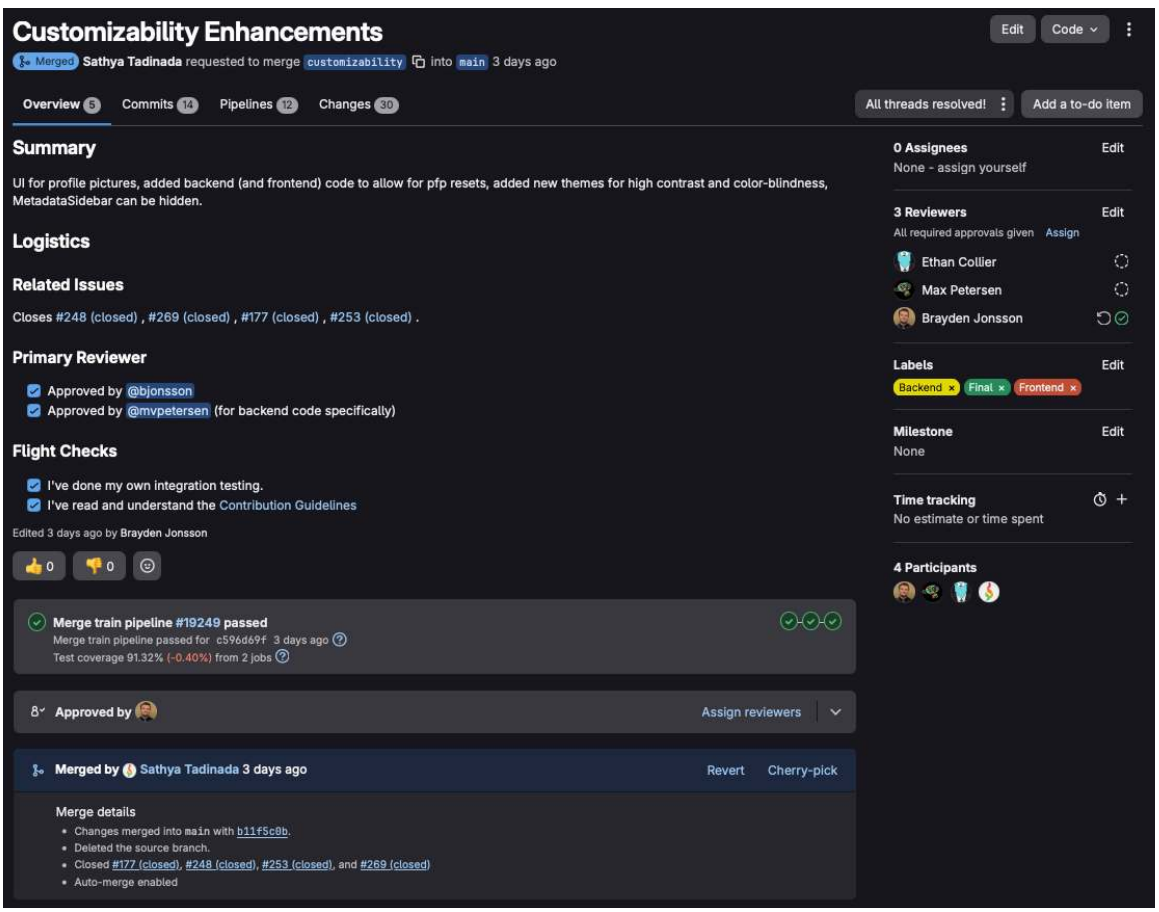


Figure 13: GitLab Merge Request showing CI pipeline status and reviewer approval section

Code Reviews

In addition to the automated pipeline checks, every merge request required a manual review and approval from at least one designated primary reviewer before it could be merged. The author of each merge request was responsible for assigning a primary reviewer with relevant knowledge of the area of the codebase being changed. Frontend merge requests were reviewed by whichever frontend engineer had not authored the request: if I submitted a merge request, Brayden would review it, and if Brayden submitted one, I would review it. The same pattern applied on the backend between Ethan and Max. Any team member could also leave a blocking review if they found a significant issue, regardless of domain, though this was rare in practice. The combination of automated checks and mandatory peer review meant that code reaching the main branch had passed both machine-enforced quality gates and a human assessment before being integrated.

Documentation

Our team's general approach to documentation favored writing clear, readable code over heavy inline commenting. The main structured exception to this was the backend API layer, where all endpoints were annotated with comments specifically to enable automatic Swagger documentation generation through Gin-Swagger. This gave the frontend team a reliable, always-current reference for every available endpoint without needing to read through backend source code directly. At the project level, we maintained documentation in READMEs throughout the repository. This was done with an eye toward FreezeTag's planned release as an open-source project under the MIT license following the conclusion of the capstone sequence in the summer of 2026, at which point external contributors will need enough documentation to understand and work with the system without guidance from the original team.

User Studies and Feedback

Our team participated in the formal user studies required by the capstone course. Beyond those, we also gathered informal feedback throughout development by showing in-progress features to peers and collecting reactions in the moment. This kind of lightweight, continuous feedback helped us catch usability issues early, before they became deeply embedded in the design. Because we used FreezeTag ourselves as part of the development process, we also gave each other ongoing feedback on UI decisions as new interface elements landed, which helped the team converge on design choices efficiently without needing dedicated meetings for every decision.

REFLECTION AND ANALYSIS

Successes and Challenges

Overall, I consider FreezeTag to be a successful project, both as a piece of software and as a team experience. The thing I am most proud of is how seriously all four of us took it from start to finish. FreezeTag was not just a class assignment that we put in the minimum effort to complete; it was something all four of us were invested in building, and I think the final product reflects that. We started with a clear project vision in Fall 2025 and executed on that same vision through Spring 2026 without significant pivots or course corrections. The core idea, a self-hosted, tag-first image management application with an extensible plugin model, was the idea we shipped. That kind of consistency from initial concept to finished product is not something every capstone team achieves, and it speaks to how well-defined our goals were from the beginning.

From a purely technical standpoint, what I am proudest of is how close FreezeTag came to feature parity with established commercial solutions like Google Photos and Apple Photos. It is not a perfect comparison in every dimension, but the features we do have work well, and the combination of tag-based search with a custom plugin model is something that no major image management product currently offers. That makes FreezeTag unique in its category, which is not something I expected to be able to say about a student capstone project.

On a personal level, this project pushed me well outside of my comfort zone. Coming into capstone, I had very limited experience with React and Next.js, and almost no formal background in UI/UX design. Taking on the frontend interface role meant I was responsible for the visual design and implementation of every page in the application, which was a steep learning curve early on. By the end of the project, I felt genuinely competent in those skills in a way I did not at the start, and I can see myself applying them directly in a professional context.

The challenges we ran into were mostly technical rather than interpersonal. The most recurring difficulty was infrastructure compatibility, with versioning conflicts between libraries, packages that lacked support for the specific functionality we needed, and interactions between different parts of the stack that did not behave as expected. These kinds of issues tend to be time-consuming in a way that is hard to plan for because they are often invisible until you are already deep into an implementation. The two that hurt the most were user authentication and the job tracking system. We had estimated authentication at about four weeks of dev time, but we ended up getting it done faster than that because bcrypt [27] (a widely used password hashing library) handled most of the hard parts. The jobs system, however, went the other direction entirely, where what looked like a straightforward progress-tracking feature turned out to require a whole new subsystem to even make it tractable.

Some features were also scoped down or deferred over the course of development. A plugin marketplace, which would have allowed users to discover and install community plugins directly from within the application, was pushed out of scope in favor of keeping the core experience polished. The location map feature, which was originally envisioned as a fully interactive embedded map in the image detail sidebar, ended up being implemented as a toggleable native UI component due to the complexity of integrating an interactive Leaflet map cleanly into the sidebar layout at that stage of development. These were deliberate tradeoffs rather than failures; the plugin system's architecture is flexible enough that a marketplace could be added post-capstone without requiring changes to the core application, and the map component achieves the core goal of surfacing location data from image metadata in a way that is useful to the user.

There were occasional disagreements on frontend design decisions, as is natural when multiple people have opinions on subjective visual choices. In most cases, we reached a consensus quickly through discussion, and when we could not, we asked external users for their preference, which gave us an objective enough tiebreaker to move forward without much friction.

For the custom theme importing feature, the main challenge was designing an import format that was flexible enough to be useful while remaining simple enough that non-technical users could work with it. Building on Catppuccin's existing CSS variable naming conventions gave us a reasonable foundation, since a meaningful portion of the self-hosted software community is already familiar with that system, but determining the right level of abstraction for the import file format required several iterations.

Future Directions

The most immediate future direction for FreezeTag is performance at a larger scale. The Go backend served us well during development and handles concurrency cleanly, but now that we have a solid understanding of what the application's data access patterns actually look like in practice, there is a reasonable case for eventually rebuilding the backend in Rust for a more production-ready implementation. Rust's memory model and performance characteristics would make it better suited to handling very large photo libraries or multi-user deployments without the overhead that a garbage-collected language introduces.

On the frontend, the priority would be continued feature development and polish to close the remaining gap with commercial solutions. There are parts of the interface that work correctly but would benefit from more refinement, and there are features that were scoped out during the capstone that would meaningfully improve the user experience if added.

The plugin ecosystem is probably the area with the most long-term potential. Right now, discovering and installing plugins requires some manual effort on the user's part. A plugin marketplace built into the application itself, where users could browse, install, and update plugins without leaving FreezeTag, would make the extensibility of the platform much more accessible to non-technical users. Given the flexibility of the existing plugin architecture, building a marketplace on top of it should be relatively straightforward compared to the work that went into building the plugin system itself.

FreezeTag is also planned to be released under the MIT open-source license following the conclusion of the capstone sequence in the summer of 2026. That transition will bring its own set of priorities around documentation, contributor onboarding, and maintaining a stable public API for the plugin system, all of which the team has been building toward throughout development.

Lessons Learned

If I were starting this project over, the most significant technical change I would make on the frontend would be to move away from Next.js toward a more conventional Node.js server setup. Next.js introduced a recurring set of friction points throughout development, particularly around the distinction between client and server components and various React hydration issues that were difficult to diagnose and fix. Some of these may have been React issues rather than Next.js specifically, but the framework added enough complexity to the development experience that a simpler server setup would likely have let us move faster and with less confusion. This is not a critique of Next.js as a technology in general, it is well suited to many use cases, but for an application like FreezeTag where the interface is relatively stateful and the server-side rendering model does not provide much benefit, the tradeoffs were not in our favor.

I would also have started active development earlier in the Fall 2025 semester. A significant portion of that semester was spent on planning, designing, and evaluating options, which was valuable but could have been compressed given how clear the project vision was from the start. Starting earlier would not necessarily have changed what we built, but it likely would have given us more time to add features and polish in the final stretch.

On the design side specifically, I would have sought out formal UI/UX guidance and design system references much earlier in the process. A large part of the visual design work ended up requiring iteration and backtracking, largely because I was figuring out design principles and patterns as I went rather than starting from a well-defined design language. As previously mentioned in the Frontend Interface Implementation section, the Sort dropdown is a good example of this, where the first approach of embedding sort syntax directly into the search bar worked, but it took seeing non-technical users interact with it to make clear how undiscoverable it was. The Select mode and the tags management page both followed a similar pattern, where the feature was not in the original design at all and only became necessary once we were actually using the app with real photo collections. In retrospect, consulting with more experienced designers or studying established design systems earlier would have saved time and produced a more cohesive result from the start. However, the experience of working through those problems without a safety net also taught me more about interface design than I would have learned in a more guided context, and I came out of it with a much stronger instinct for UI decisions than I had going in.

More broadly, this project reinforced a lot of the ideas introduced in earlier software development coursework at the University of Utah around effective team collaboration, version control practices, and the value of building in small, reviewable increments. The difference is that in a capstone context, those ideas are no longer abstract; you feel the consequences of following or ignoring them directly in the quality and pace of your own work. FreezeTag is the most complete and technically involved piece of software I have built as a student, and the lessons from building it are ones I expect to carry into my career.

CONCLUSION

The development of FreezeTag was, above all else, a demonstration of what a motivated team can build when they commit fully to a shared vision. Over two semesters, we designed, developed, and deployed a full-stack, self-hosted application with a responsive web interface, a novel tag-based search system, an extensible Python plugin architecture, and a suite of first-party plugins powered by local machine learning models. None of us had built something of this technical caliber before as a team, and the fact that we were able to see it through from an initial concept in Fall 2025 to a polished, feature-rich application by Spring 2026 is something I am genuinely proud of. Along the way, we all grew significantly as engineers. For my part, I came into this project with limited experience in React, Next.js, and UI/UX design, and I am leaving it with a level of frontend competence that I feel confident bringing into a professional setting. That kind of growth, across all four members of the team, is probably the most lasting outcome of this project.

From a technical standpoint, the achievements that FreezeTag made are worth taking a moment to appreciate. The application reaches near-feature parity with established commercial image management solutions like Google Photos and Apple Photos, and in certain respects goes beyond them. The tag-based organizational model combined with a flexible, user-extensible plugin system is something that no major image management product currently offers, and it positions FreezeTag as a novel piece of software in its category rather than just another student imitation of an existing product. The core functionality works reliably, the interface is clean and usable, and the architecture was designed with extensibility and long-term maintainability in mind from the very beginning.

What makes all of that feel most meaningful is what comes next. FreezeTag is planned to be released under the MIT open-source license in the summer of 2026, following the conclusion of the capstone sequence. That means the work we did over these two semesters will, instead of sitting in a university repository and gradually becoming irrelevant, be out in the world as a real, production-ready piece of software that anyone can download, run on their own hardware, contribute to, and build on. For a field like self-hosted software, where the community depends on passionate developers choosing to share their work openly, releasing FreezeTag feels like a meaningful contribution. The idea that someone we have never met could run FreezeTag on their own server, use it to manage their photo library, or write a plugin that extends it in a direction we never thought of is precisely the kind of outcome that made this project worth building in the first place.

REFERENCES

[1] Google, “Google Photos.” Accessed: Mar. 29, 2026. [Online]. Available: https://photos.google.com/

[2] Apple, “Apple Photos.” Accessed: Mar. 29, 2026. [Online]. Available: https://www.apple.com/ios/photos/

[3] digiKam Team, “digiKam.” Accessed: Mar. 29, 2026. [Online]. Available: https://www.digikam.org/

[4] Google, “Google Gemini.” Accessed: Mar. 29, 2026. [Online]. Available: https://gemini.google/about/

[5] Docker, “Docker Compose.” Accessed: Mar. 29, 2026. [Online]. Available: https://docs.docker.com/compose/

[6] GitLab Inc., “GitLab.” Accessed: Mar. 29, 2026. [Online]. Available: https://gitlab.com/

[7] Google LLC, “The Go Programming Language.” Accessed: Mar. 29, 2026. [Online]. Available: https://go.dev/

[8] SQLite Consortium, “SQLite.” Accessed: Mar. 29, 2026. [Online]. Available: https://www.sqlite.org/

[9] ImageMagick Studio LLC, “ImageMagick.” Accessed: Mar. 29, 2026. [Online]. Available: https://imagemagick.org/

[10] Vercel Inc., “Next.js.” Accessed: Mar. 29, 2026. [Online]. Available: https://nextjs.org/

[11] Gin Contributors, “Gin Web Framework.” Accessed: Mar. 29, 2026. [Online]. Available: https://gin-gonic.com/

[12] Google Inc., “WebP.” Accessed: Mar. 29, 2026. [Online]. Available: https://developers.google.com/speed/webp

[13] pnpm Contributors, “pnpm: Fast, Disk Space Efficient Package Manager.” Accessed: Mar. 29, 2026. [Online]. Available: https://pnpm.io/

[14] Meta Open Source, “Jest: Delightful JavaScript Testing.” Accessed: Mar. 29, 2026. [Online]. Available: https://jestjs.io/

[15] react-dropzone Contributors, “react-dropzone.” Accessed: Mar. 29, 2026. [Online]. Available: https://react-dropzone.js.org/

[16] Node.js Contributors, “Undici.” Accessed: Mar. 29, 2026. [Online]. Available: https://undici.nodejs.org/

[17] Astral, “uv: An Extremely Fast Python Package Manager.” Accessed: Mar. 29, 2026. [Online]. Available: https://github.com/astral-sh/uv

[18] Google, “Google Gemini API.” Accessed: Mar. 29, 2026. [Online]. Available: https://ai.google.dev/

[19] Catppuccin Contributors, “Catppuccin.” Accessed: Mar. 29, 2026. [Online]. Available: https://catppuccin.com/

[20] V. Agafonkin, “Leaflet: an Open-Source JavaScript Library for Mobile-Friendly Interactive Maps.” Accessed: Mar. 29, 2026. [Online]. Available: https://leafletjs.com/

[21] OpenStreetMap Contributors, “OpenStreetMap.” Accessed: Mar. 29, 2026. [Online]. Available: https://www.openstreetmap.org/

[22] Agile Alliance, “What is Agile?.” Accessed: Mar. 29, 2026. [Online]. Available: https://www.agilealliance.org/agile101/

[23] Prettier Contributors, “Prettier: Opinionated Code Formatter.” Accessed: Mar. 29, 2026. [Online]. Available: https://prettier.io/

[24] Google LLC, “gofmt.” Accessed: Mar. 29, 2026. [Online]. Available: https://pkg.go.dev/cmd/gofmt

[25] ESLint Contributors, “ESLint: Find and Fix Problems in Your JavaScript Code.” Accessed: Mar. 29, 2026. [Online]. Available: https://eslint.org/

[26] golangci-lint Contributors, “golangci-lint.” Accessed: Mar. 29, 2026. [Online]. Available: https://golangci-lint.run/

[27] W. Contributors, “bcrypt: A Password Hashing Function.” Accessed: Mar. 29, 2026. [Online]. Available: https://en.wikipedia.org/wiki/Bcrypt

APPENDIX

Appended is the FreezeTag Design Document, authored during the 2025-2026 school year.

FreezeTag

Design Document

Ethan Collier, Brayden Jonsson, Max Petersen, Sathya Tadinada

December 8, 2025

Table of Contents

1 Introduction

FreezeTag is a new *free* and open-source image *tag*ging software designed for photographers, businesses, or anyone else who appreciates easy-to-use and adaptable image storage. FreezeTag works by running locally on hardware you provide, such as a personal NAS or a computer, and then collects and stores metadata about your photos. In addition, the user is able to provide tags to indicate what groups an image may be part of. For example, a tag could indicate a particular subject, a specific event, or the client the photo was taken for. This data is then stored in a local database, and FreezeTag's easy-to-use interface allows you to search these photos based on the metadata and tags.

The FreezeTag server is also able to load, run, and manage Python-based plugins that customize the functionality of FreezeTag to fit every possible need. FreezeTag ships with some first-party plugins and third-party custom plugins are also supported. Because these plugins are based in Python, this unlocks the powerful artificial intelligence and machine learning capabilities available through modules such as PyTorch and TensorFlow. AI/ML can be used through plugins directly in the app to support facial recognition, automatic captioning/tagging, natural language searching, and more. Plugins can also contribute to a workflow; one first-party plugin supports selecting a group of images, converting them to a specified format, and emailing them to a client. FreezeTag supports concurrency in these plugins so that you don't have to entirely compromise speed for features.

FreezeTag will be designed as a web server, which users will host on their own managed hardware. This generally avoids expensive cloud costs, as most users will have compatible hardware already. The web server will be designed in Go, a language which is fast, garbage-collected, and has strong support for concurrency. The plugins will be written in Python, which is easy-to-use and has strong AI/ML capabilities. Both the servers and plugins will be managed with Docker Compose, which provides some security benefits for protecting against malicious plugins and also allows us to guarantee a consistent environment. The web server will also use a SQLite database for storing metadata and tagging information for photos, which will also be partially accessible to plugins. This entire backend infrastructure will connect to a Next.js frontend, so that the GUI will be designed and usable as if it was a webpage. This has the added benefit of allowing multiple users on a network to easily access the same FreezeTag instance.

2 Background

2.1 Project Market

FreezeTag provides a solution for individuals or businesses that want to store and categorize images without using a cloud-based service. Flexibility through plugins means that FreezeTag can be adapted to even niche use cases with a little configuration and work, and the tagging system allows granular storage and retrieval of very specific images.

Using Python plugins, developers can leverage machine learning libraries like Pytorch and Tensorflow to run their own image classifiers locally, adding another layer to the customizability of the app.

FreezeTag is ideal for freelance photographers, digital artists, web archivists, and especially anyone who already runs their own NAS (a small but very enthusiastic market)

2.1.1 Similar Apps

- DigiKam
  - DigiKam is a local app written in QT that supports similar tagging and plugins.
  - The plugins in DigiKam are the open-source style of adding a DLL to the project (hard to develop).
  - Networked storage is difficult (DigiKam has to be locally installed on each device, and networked drives have to be set up manually)
- Google/Apple Photos
  - Similar image classification tools to ours, but no configuration or customization
  - Easy to use web interface
  - Only on the cloud: no control over your data, sometimes expensive!
  - Folder structure over tag structure: harder to use, less granular

2.2 Required Technology

At a high level, FreezeTag is a pair of servers (backend written in Go, frontend written in Next.js) that communicate back and forth over an API to store and serve images.

2.2.1 List of Leveraged Technologies

- Docker + Docker Compose
- Go Backend

- SQLite
- ImageMagick (for dealing with a range of formats)
- uv Python package manager
- JsonRPC or custom REST over a loopback port for communication between Python and Go
- Gin and Gin-Swagger

- Next.js Frontend
  - Node and pnpm
  - react-dropzone
  - Javascript/undici fetch
  - Jest

2.2.2 List of Technologies to Implement

- Easy Deployment
  - Will be accomplished with a Docker Compose configuration file.
- API Handling on both the frontend and backend
- Image Gallery
- Image Tagging
- Plugin Environment Management
- Plugin Interaction
- Authentication and Admin Management

2.3 Software/Hardware Requirements

Our application is designed from the ground up to have flexible hardware requirements so that it can be used and hosted by a broad range of people and organizations.

2.3.1 Running a FreezeTag Instance

2.3.1.1 Architecture

To run your own FreezeTag instance, all you need is a normal consumer-grade computer running Windows or Linux (or an Intel Mac). Whatever you choose, it has to be capable of running Docker (this includes basically every computer and OS produced recently). During initial development, we'll only support x86 architectures, but in the future we might expand to ARM64 (specifically to run on Apple Silicon).

2.3.1.2 Storage/Bandwidth

If you're going to use FreezeTag as your primary place for storing photos, you might want to use a drive that can store a couple terabytes. There's no requirement on drive speed, but an SSD or a faster hard drive will definitely lead to faster load times.

In the browser interface, we'll display lower resolution high compression thumbnail formats like WEBP, so drive speeds and bandwidth will be as minimal a constraint as possible while you scroll and search your collection.

2.3.1.3 Optional Plugins

The Python plugins supported by FreezeTag can come with their own extra hardware requirements, which is why they're optional! Certain machine learning applications might benefit from a fast CPU or GPU acceleration on top of our basic hardware requirements, and we want to keep the basic hardware requirements as minimal as possible.

2.3.2 Storing Photos on a FreezeTag Instance

To connect to your own (or someone else's) FreezeTag instance, all you need is an internet connection and a modern web browser. The GUI is completely web-based, so there's no particular architecture requirements (it works across all mobile and PC platforms).

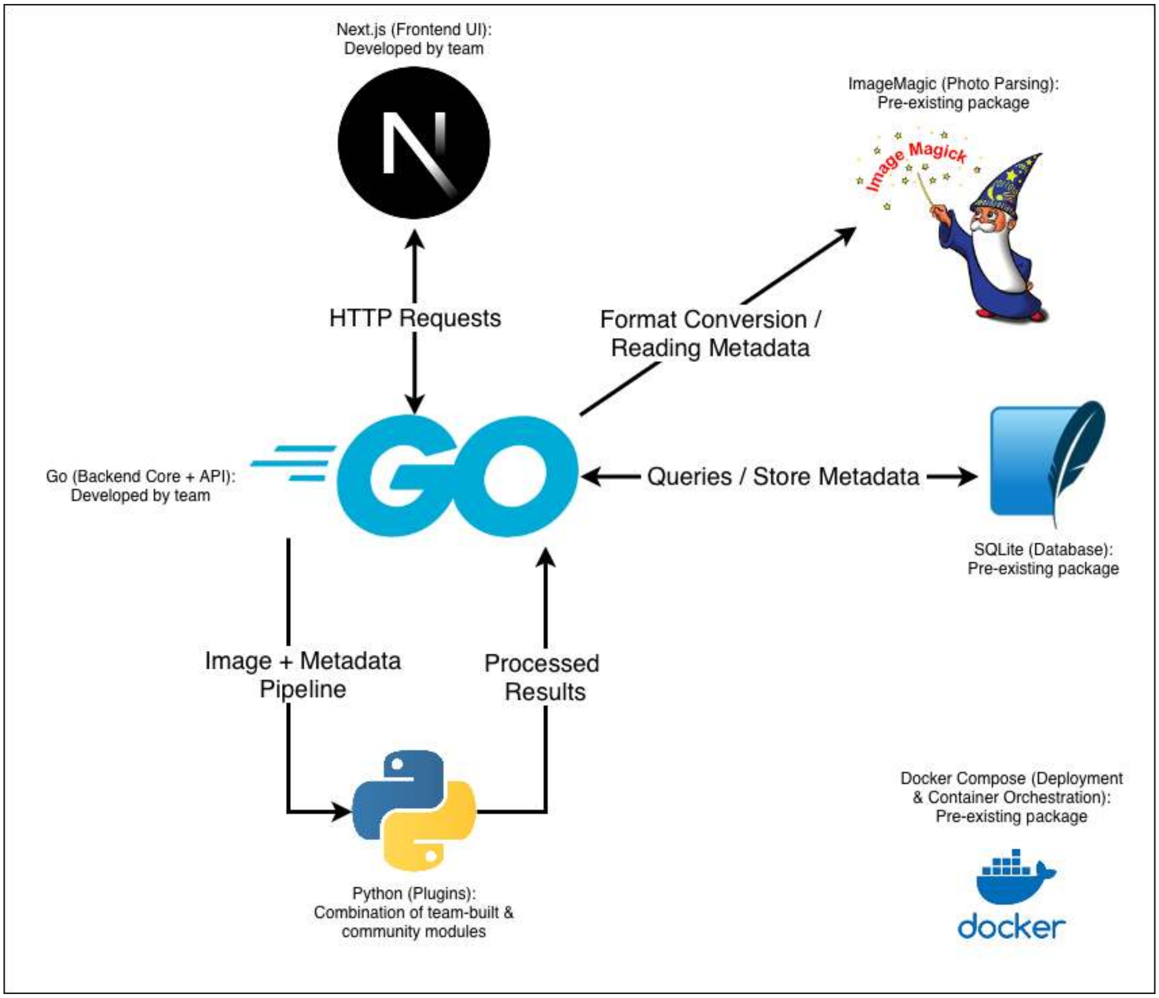
3 Requirements

3.1 System Architecture

The system will be a Go backend server and a Next.js frontend server working together. The Go server will manage plugins, as well as the photo storage and database. The frontend will manage displaying images to the user, accepting uploads from the user, and searching for photos, as well as some authentication (when we add it). The backend will also implement its own authentication in the form of API keys that can be managed from the frontend by an admin user. This way, the frontend can be used by users and the backend can be used by other programs the users might want through the API.

The frontend and backend are both stored in the same monorepo, but in different folders with independent structure. The frontend project structure will utilize Next.js's page routing for the different possible interfaces, but will store common components (buttons, images, etc.) in separate folders. The intent here is that actual pages and their source code will be relatively minimal and focused, with the common components abstracted away.

The backend project structure will be mostly written in Go, utilizing a lot of different Go/C binding libraries to connect to utilities like ImageMagick and SQLite to manage databases and image/metadata parsing. The folder structure will be a pretty standard Go project, but when built it will have a secondary folder for storing Python plugins and their related data, virtual environments, etc.



3.2 Personnel

Brayden will be the frontend "architecture" engineer and Sathya will be the frontend "interface" engineer. The intent here is that Brayden can be in charge of lower-level tasks, such as parsing and handling API responses from the backend, and Sathya can be in charge of actual user interface design and implementation. In practice, it is extremely likely that the frontend will have a lot of mixed development because APIs need UI to be consumed and UI needs APIs to work, especially at the beginning of the project, but we think it's fair to assign ownership this way. Brayden has direct experience working in a frontend architecture at Lucid, and Sathya has comparatively more experience working on user interfaces. This split also encourages clear separation between user-facing and API-facing parts of the application, which improves separation of concerns in the codebase and makes updates easier.

Likewise, Max will be the backend architecture engineer and will be focusing on tasks such as interacting with plugins and handling containers. Ethan will be the backend "feature" engineer which includes the API and most of the tasks that relate to the actual user-facing features. Max is taking the role as backend architecture engineer due to his experience in the exact type of application management we will need for managing plugins. Ethan likewise has some relevant experience in defining APIs on backend applications.

In addition, both backend engineers will have the primary task of implementing our plugins, but due to the flexibility of plugins the frontend engineers will participate as well when we reach a Rank 3 polish stage. All engineers will be completing at least one "major" plugin when we reach Rank 3.

3.2.1 Primary Assignments

- Sathya, Frontend Interface Engineer
  - Interface Design
  - CSS Implementation
- Brayden, Frontend Architecture Engineer
  - Frontend Data Management
  - Backend Communication
- Ethan, Backend Feature Engineer
  - API Endpoints
  - Primary User Features
    - Image Storage
    - Metadata Parsing
    - Etc.
- Max, Backend Architecture Engineer
  - Dependency Management
  - Plugin Management

3.3 System Features

3.3.1 Rank 1

- Photo Gallery UI
  - A responsive grid-based gallery displaying all uploaded photos.
  - Includes sorting and basic filtering (by upload date, filename, or tag).
  - Each image opens a details modal showing metadata, tags, and captions.
- Uploading UI

    - Drag-and-drop or file picker interface to upload images to the server.
    - Displays upload progress and confirmation upon completion.
    - Uses multipart API requests to transmit file and metadata to the backend.
- Server Connectivity
    - Full client-server integration: frontend served via Next.js, backend serving API requests.
    - Initial deployment supports local testing and remote access (localhost and Docker).
- Tagging and Captioning Database
    - SQLite schema that associates each image with user-defined tags and captions.
    - Supports exact-match search and partial text queries.
- EXIF Metadata Parsing
    - Automatically extracts embedded metadata from uploaded images.
    - Parses fields such as timestamp, camera model, and GPS coordinates.
    - Stores extracted metadata in the database as searchable fields.

3.3.2 Rank 2

- First-Party Plugins
    - Facial Recognition Auto-Tagging
        - Uses a local machine learning model to detect faces and auto-tag known individuals.
    - Local ML Image Captioning
        - Integrates a lightweight image-captioning model for automatic caption generation.
        - Runs offline and stores captions in the metadata database.
    - Location Tagging via OpenStreetMap
        - Uses EXIF GPS data to generate approximate location tags (e.g., nearest city or country).
    - Bulk Exporting Tools
        - Allow users to export selected images and metadata in common formats (ZIP, TAR, or email attachments).
- Community Plugin API
    - Exposes a public API endpoint for developers to submit and manage community plugins.
- Docker Deployment Support
    - Provides a Docker Compose file that automatically sets up the backend, frontend, and database, enables simple-command deployment.

- Responsive and Polished UI
  - Finalized CSS with device-reactive breakpoints and consistent color scheme and branding for the FreezeTag identity.
- User Authentication
  - Integrates with backend API keys for user-scoped data access.

3.3.3 Rank 3

- Additional First-Party Plugins
  - UI Extension Plugins
    - Plugin that displays photo locations on the map
      - Owned by Sathya
    - More robust "bulk tag" plugin for large file uploads
      - Owned by Ethan
  - More UI API exposure to support non-pipeline plugins
    - A plugin can add a button/component/page/etc and run custom plugin code based on user input
      - Owned by Brayden
- Architecture-Aware Deployment Support
  - Adds compatibility for ARM-based systems (e.g., M-series Macs).
  - Attempts to utilize available accelerators (GPUs, Metal, etc.)
  - Includes architecture-specific Docker images and build optimizations.
  - Owned by Max

4 Software Engineering Techniques

4.1 Development Process

We believe that Agile will be the best process for our project. We are developing an app that has a lot of features which are cross-dependent on each other, so while we will be planning most features ahead of time, we will need to plan around the specific implementation as we build the software. Relying on an Agile process means that we can be flexible in responding to particulars of any implementation.

4.2 Tools

We are developing in Go and Next.js (which is React-based), both of which have strong support in VSCode. All of us plan on using VSCode because of that support and our familiarity with it.

We plan on generously using libraries that give us desired functionality in a well-implemented way, which is well-supported by using Go Modules for Go code and NPM for Next.js code. Some libraries/tools that we will heavily rely on include Gin, ImageMagick, SQLite, and Docker Compose. Through Gin, we will also get Swagger docs for our backend automatically.

Both the backend and frontend will have strong unit testing. More is specified in Section 4.3, but we intend on enforcing a minimum testing coverage and requiring that unit testing (alongside formatting and linting checks) passes before a branch is merged. Go has unit testing built in, and we are using Jest for testing Next.js code.

Bug tracking, issue tracking, and versioning all will be handled in Gitlab using the provided tooling. This also means we will be using Git specifically for versioning.

4.3 Code Review

We have crafted a set of Contribution Guidelines available on Gitlab. We have summarized/rephrased that text here:

Authors of an MR are expected to perform manual QA/integration testing on their new code. This mostly applies to UI items, but API items can also be tested by hand and it is a good idea to do that as well. Authors of a PR are also expected to write unit tests for any new features they implement and uphold an 80% code coverage minimum.

An MR will be required to be reviewed and approved by a “primary reviewer”, who should be the “next best expert” in that area of the codebase. For example, a frontend MR should be reviewed by at least a frontend developer, if not the frontend developer who most recently worked in that code area. The primary is required to review and approve, all other team members are optional but have 12 hours to leave a review (the 12 hour period can be, and is frequently, waived by a discord “voice vote”). All reviewers can leave a blocking review, but this is disruptive and should be a last-resort. All reviews given should have genuine concerns, and all reviews should be taken seriously, even if the comments are just nitpicks. It is better to change code at the MR stage before it becomes an issue later.

We use an automated pipeline to run formatting checks, linting checks, and unit tests. All stages of the pipeline must pass for a merge to happen, except in emergencies.

4.4 Documentation

Generally, we believe that code should be well-written and self-documenting. We hope that this minimizes the number of unnecessary/redundant inline comments that are liable to becoming stale. Our code review process is designed to help enforce this. We will likely have some portions of our project that will need large, overall documentation, and we will accomplish this primarily by READMEs in the repository. Our intent is for this project to become open source after capstone, and so we don’t want documentation to rely on the Capstone instance wikis which may not be accessible to the public in the future.

4.5 Team Communication and Meetings

We primarily communicate using Discord. This allows us to be flexible, by using text or voice as appropriate, and working synchronously or asynchronously as appropriate or necessary. We all agree that, while it is unrealistic for responses to be instant, that we should be prompt with our responses to help keep up the velocity of the project.

We plan on meeting at least three times a week. We will meet on Monday after Capstone lecture for weekly planning, Wednesday during the regular lecture time for a weekly in-person standup, and Friday morning for our staff meeting with Professor Wood. Other meetings can be planned as needed, and can be in-person or remote as appropriate. Because of our commitment to be communicative over Discord, more regular meetings is most likely not necessary.

5 Timeline

	Ethan Collier	**Brayden Jonsson**	**Max Petersen**	**Sathya Tadinada**
Alpha				
Week 1	Upload progress endpoint	Clean up messy frontend gallery code	Add sorting to the query endpoint	Metadata shown in the image context panel
Week 2	User authentication API endpoints	Upload progress on the frontend	Begin plugin environment development	Fix the search location parser
Week 3	User authentication API endpoints	Docker Compose deployment	Finish plugin environment (to at least a usable state)	User Authentication UI
Week 4	Lock down API endpoints for authentication as configured	Frontend User Authentication Handling	Image Captioning First-Party Plugin	Plugin manager UI
Week 5	Lock down API endpoints for authentication as configured	Frontend User Authentication Handling	Finalize Plugin Manifest/ Plugin Handler Polish	Location Tagging with Map First-Party Plugin
Beta				
Week 6	Plugin Import From Git Repo	Responsive Frontend Design (Mobile Support)	Expose read-only database to plugins	Responsive Frontend Design (Mobile Support)
Week 7	Plugin Import From Git Repo	Third-Party .zip Email Plugin	Give each plugin its own SQLite table	Display errors to users with a common flow
Week 8	Bulk Export First-Party Plugin	Build the marketing website	Facial Recognition	Build the marketing website

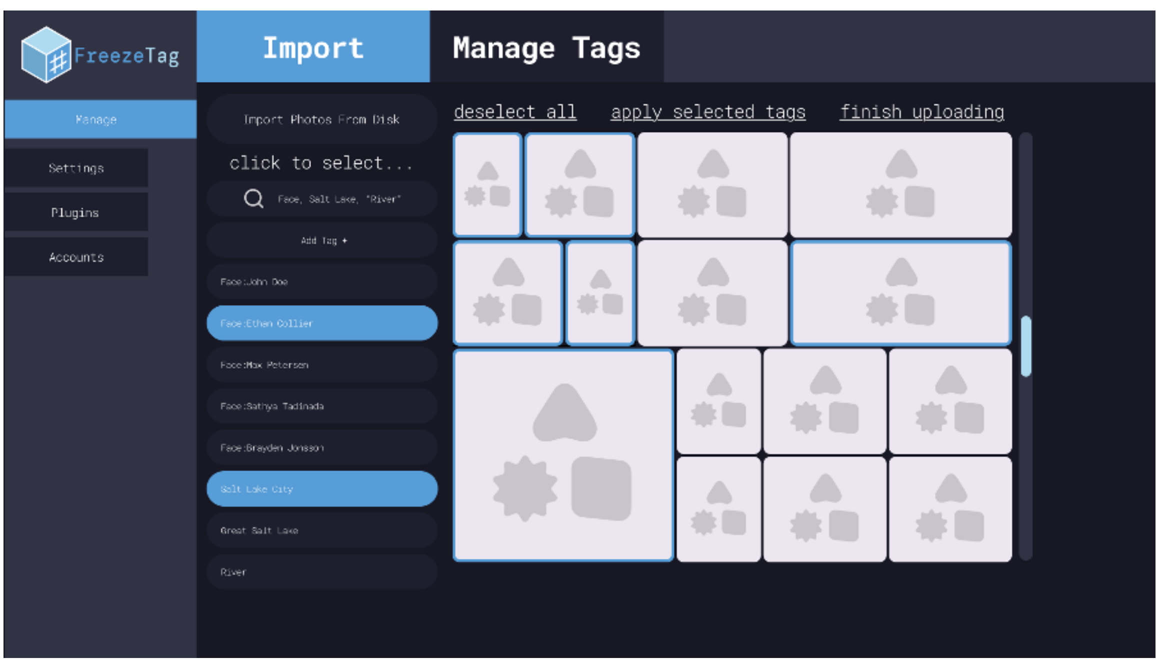
			First-Party Plugin	
Week 9	Bulk Tag First-Party Plugin	Settings UI	Facial Recognition First-Party Plugin	Polish marketing website, add basic documentation
Week 10	Bulk Tag First-Party Plugin	Custom UI Plugin Support	Plugin Flows Backend Support	Plugin Flows Frontend Support
Release				
Week 11	Enabling/ Disabling Plugins	Custom UI Plugin Support	Plugin Flows Backend Support	Plugin Flows Frontend Support
Week 12	Release-Ready Implementation of First-Party Plugins	Custom UI Plugin Support	System-Aware Optimized Deployment	Photo Locations on Map First-Party Plugin
Week 13	Testing and Bugfixing Backend	Testing and Bugfixing Frontend	Testing and Bugfixing Plugins	Photo Locations on Map First-Party Plugin
Week 14	Test, Bugfix, Polish	Test, Bugfix, Polish	Test, Bugfix, Polish	Test, Bugfix, Polish
Week 15	Test, Bugfix, Polish	Test, Bugfix, Polish	Test, Bugfix, Polish	Test, Bugfix, Polish

6 Appendix A: UI Sketches

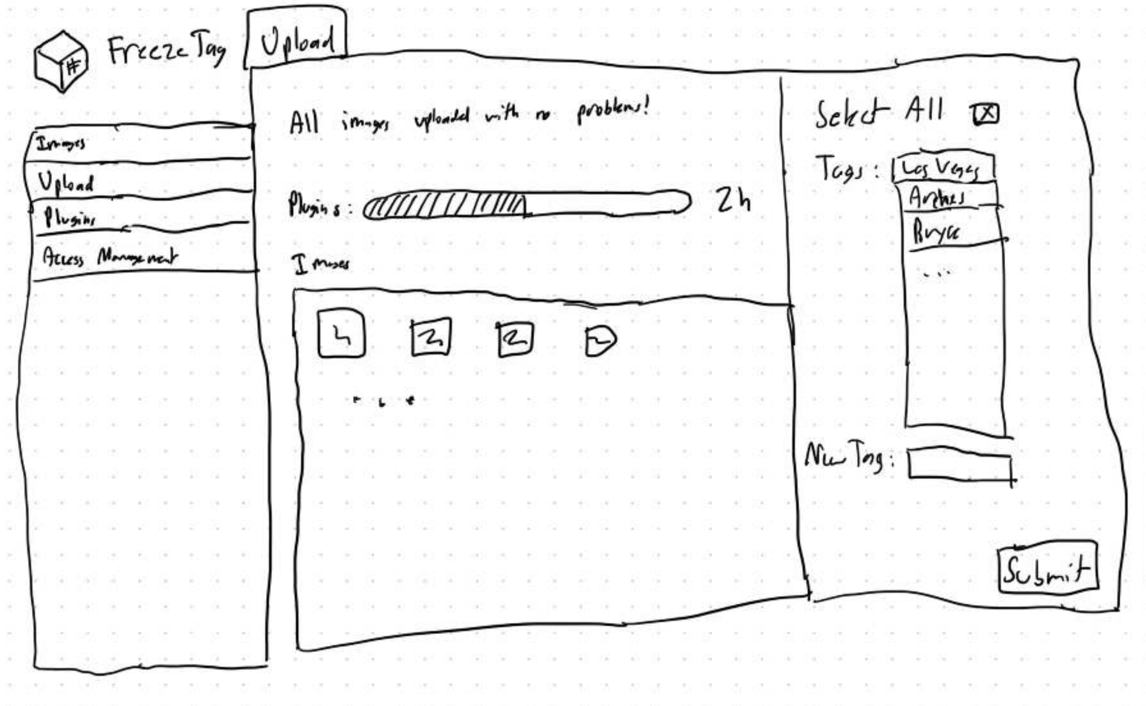
6.1 Gallery Page

6.2 Import/Upload Page

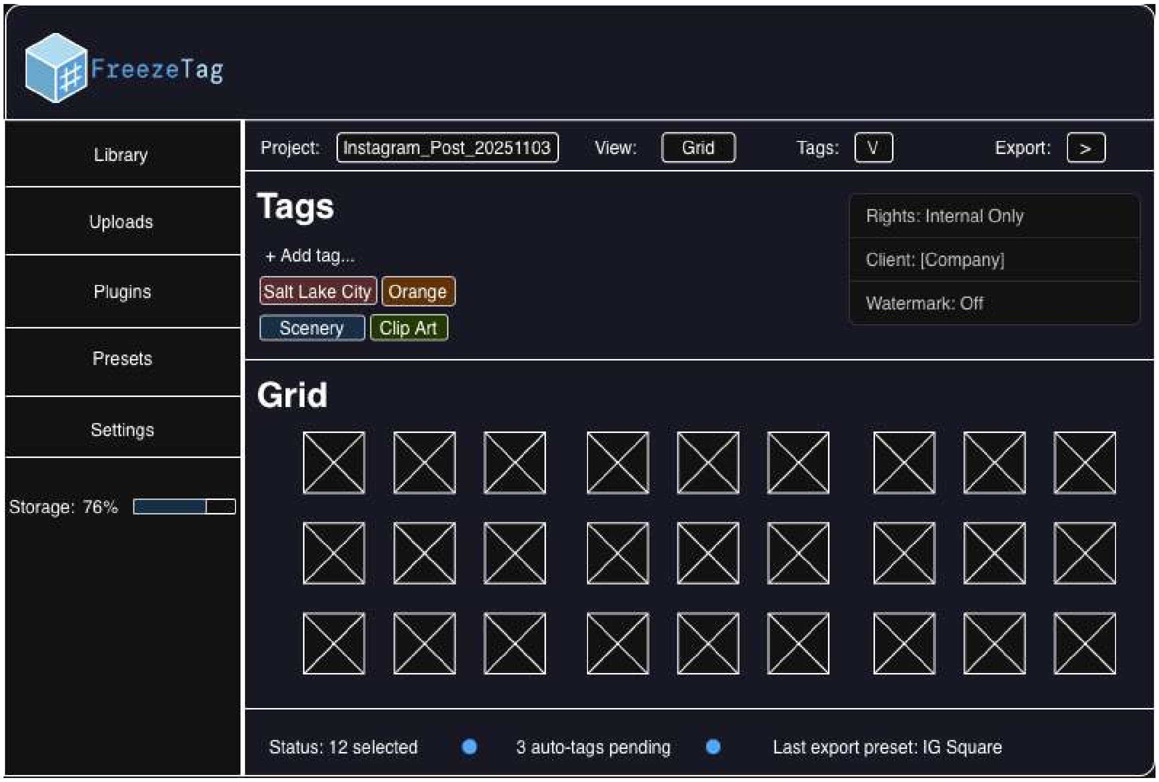
6.2.1



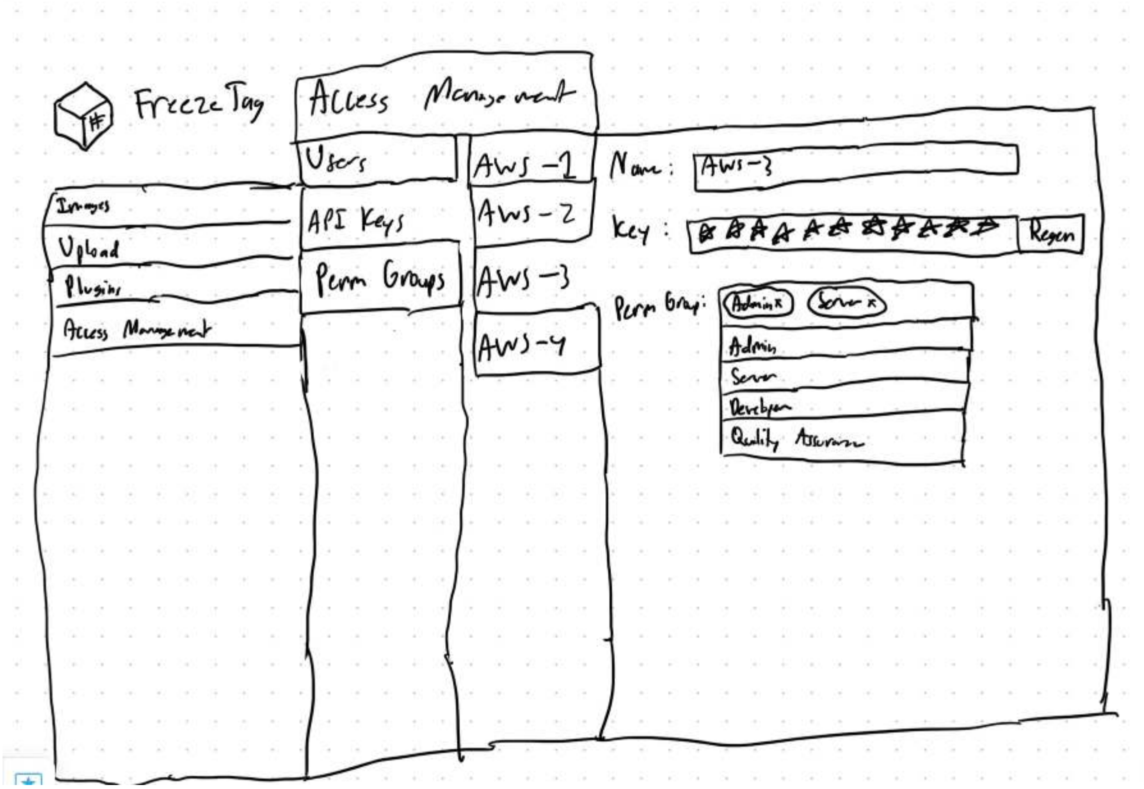
6.2.2


FreezeTag
Upload
Images
Upload
Plugins
Access Management
All images uploaded with no problems!
Plugins :
2h
Images
Select All
Tags :
Los Vegas
Arches
Bryce
New Tag :
Submit

6.3 Export Page



6.4 Admin Page



7 Appendix B: Use Cases

7.1 Use Case 1

Number	**Use Case 1**
Title	**Family Pictures**
Preparer	Max Petersen
Actor	Ordinary (Maybe tech-savvy) person
User Story	Whenever I have family get-togethers, we're always taking a lot of photos on our phones. It would be nice if I could easily upload pictures from my phone and look through them in a couple different ways later.
Course of Events	1. Connect to a running FreezeTag instance from my phone's browser 2. Tap the upload button and select photos from my gallery 3. Give the entire upload a global tag (e.g. "Thanksgiving") 4. Finalize the upload so that my pictures are stored. 5. Later, come back and scroll through pictures in chronological order • If I want to view a specific event, I could search for a tag (e.g. "Thanksgiving") in the search bar. • I could even view a specific time frame, e.g. "Thanksgiving, 2025".
Exceptions/Alternatives	• Manual Tags - Aside from the global tag provided during upload, groups of 1 or more photos can be given more tags.
Related UI	See Section 6.1 and Section 6.2

7.2 Use Case 2

Number	**Use Case 2**
Title	**Business**
Preparer	Brayden Jonsson
Actor	SRE at a Tech Company
User Story	As an SRE at a document-based SaaS company, I need to find an image storage solution that we can self-host (for security and data integrity reasons) which is easy to interface with our current back-end, like a database.
Course of Events	1. Research the FreezeTag backend API, with our documentation 2. Implement connections to our API in the current company codebase 3. Using the infrastructure that the company already owns or has access to, deploy FreezeTag and access image tagging and plugin features directly through our API 4. Manage API keys, access, and other admin tasks through the user-friendly FreezeTag frontend, which is still accessible. 5. Extend FreezeTag with plugins to automatically delete and remove old, unused images based on tagging data. Use the frontend as an admin panel to control the rules of this plugin.
Exceptions/Alternatives	N/A
Related UI	See Section 6.4 and Section 6.2.2 (plugin progress bar)

7.3 Use Case 3

Number	**Use Case 3**
Title	**Brand Curator**
Preparer	Sathya Tadinada
Actor	Graphic Designer
User Story	As a freelance graphic designer, I want to import project assets, apply consistent tags (that keep track/organize into various categories like client, project, usage rights, format, etc.) and quickly find/export the exact variants I need so that hand-offs to my clients are fast and streamlined.
Course of Events	1. Create or open a project/collection for a client or project 2. Bulk-import images/exports (PNG/JPG/WEBP/PSD Previews) 3. Run selected plugins (color extraction, auto object/scene tagging, etc.) to tag the batch 4. Add batch tags (client, project, rights, etc.) and various per-image tags ("Instagram square", "SVG logo", etc.) 5. Review and edit tags in the tagging section/panel. 6. Search by tags or fuzzy/natural language to retrieve needed assets with ease 7. Select results and run an export workflow (watermark, sizes/formats/preset naming, etc.) 8. Store the exported package in a handoff folder for clients.
Exceptions/Alternatives	• If GPU/ML plugins are unavailable or cannot be run on-device, proceed with manual/batch tagging instead. • If certain files cannot render a preview, show a default/fallback thumbnail
Related UI	See Section 6.3 and Section 6.2

7.4 Use Case 4

Number	**Use Case 4**
Title	**Client Pictures**
Preparer	Ethan Collier
Actor	Professional Photographer
User Story	Recently, I had 2 separate large photoshoots including one of my clients. This client needed the original images of himself sent to them. These photoshoots are large, and as such it is hard to find specific images of him, and it is impractical to send over such a large amount of data.
Course of Events	1. User connects to the FreezeTag instance web UI 2. User has enabled the ML Face ID plugin via a Plugin Manager Menu 3. User enters the name of the person into the Tag Based Search Bar 4. search results generate an Gallery filled with every picture of the client taken in every session 5. Gallery gets further refined with additional tags to ensure that it is only pictures of the client, and no group pictures 6. the Gallery is then Exported to be delivered to the client
Exceptions/Alternatives	• The user may have a plugin enabled that automatically sends the Gallery to the client via email, skipping the middleman device
Related UI	See Section 6.2, Section 6.1, and Section 6.3

7.5 Use Case 5

Number	**Use Case 5**
Title	**Family Guy**
Preparer	Max Petersen
Actor	Mother
User Story	I have lots of pictures of my family members, but they are unsorted and uncategorized. I would like to be able to search the pictures by who is in them, without needing to manually tag every picture with their name.
Course of Events	1. Connect to FreezeTag and go to the "Plugins" menu 2. Download/enable the face recognition plugin from the official FreezeTag first party plugins 3. Tag good pictures of each family member's face with `face:[name]` 4. Newly uploaded pictures will be scanned for each person with a `face:` tag, matching people will get their name added as a tag
Exceptions/Alternatives	N/A
Related UI	See Section 6.1 and Section 6.2

8 Appendix C: Revisions

8.1 Introduction

No changes.

8.2 Background

- Changed `List of Technologies` to `List of Leveraged Technologies` and added `List of Technologies to Implement` to clearly delineate what we will leverage through libraries and what we will write.

8.3 Requirements

- Added more information about each of the engineer's roles.
- Defined clear ownership over the current suite of Tier 3 features.
- Added a list of "Primary Assignments" for each engineer.
- More clearly defined Max's primary Tier 3 feature

8.4 Software Engineering Techniques

- Turned the copy of our contribution guidelines into a summary, to fit within the page limit established by the rubric.

8.5 Timeline

No changes.

8.6 Appendix A: UI Sketches

- Moved UI Sketches in front of Use Cases, changed title to include "Appendix".

8.7 Appendix B: Use Cases

- Moved Use Cases behind UI Sketches, changed title to include "Appendix".
- Reformatted use cases to use tables.
- Added Use Case 5 "Family Guy"

Name of Candidate: Sathya Tadinada

Date of Submission: April 11, 2026