AN EXPLORATION OF TURING PI-BASED EDGE CLOUD

TEAM SDMAY24-03

Nicholas Bergan, Cooper Caruso, Owen Henning,

Kale Kester, Owen Perrin, Andrew Phelps

1. INTRODUCTION

1.1 Problem Statement

Cloud abstraction, containerization, and orchestration: three pivotal concepts that have shaped how software is deployed and managed at scale. Our project is primarily an exercise in exploring these concepts as they pertain to private and hobbyist clouds. While consumer cloud offerings like Amazon Web Services, Azure, or Google Cloud offer services that abstract resource consumption but provide a medium to deploy applications, some use cases necessitate a locally-administered private cloud. Security concerns, legacy systems, existing capital, and need for more fine-grained control of infrastructure may mean a self-hosted cloud is the best option. In other scenarios, system administrators or developers may want to explore and learn the fundamentals of container-based clouds using real hardware, just as our team has done.

Given a Turing Pi 2 cluster board with three compute nodes, our team has designed and implemented a proof-of-concept, self-hosted cloud which explores new, fundamental paradigms and principles. The cloud will support three main components: relational data storage, hierarchical file storage, and process management. Succinctly: databases, files, processes. The team will administer the cloud using continuous integration and deployment (CICD) via Gitlab pipelines and Gitlab runners. To facilitate distributed file storage while demonstrating replicability of the database and containerization of software, our team has deployed a custom-built software application stack which allows members to register and use the cloud for hierarchical file storage. This means the team must make additional efforts to provide an intuitive interface for users. Designing, provisioning, maintaining, and deploying on a private cloud has given the team an excellent opportunity to learn about private clouds and the principles used to create them.

1.2 Intended Users and Uses

The primary benefactors of our project are the project team members. This project provides an opportunity to research, design, and implement a cloud computing solution that may not otherwise be possible. Future groups may also benefit from the design and research we have done. Furthermore, individuals or hobbyists that wish to experiment or implement their own computing cluster on similar hardware may implement our solution or use it as a starting point for their own. Specifically, this technology could be useful to engineering teams that need to take advantage of a CI/CD workflow and Docker containers to dynamically provide server computation for handling client requests. Additionally, the usage of a low-power and scalable hardware platform provides a use case for a company to own in-house compute servers. A data-sensitive user would also take advantage of this in-house, locally hosted server and workflow platform as it replicates the advantages of cloud-based scalability with the addition of having absolute control over the implementation in terms of security and accessibility. Small business owners could make use of a similar platform and design to either run a handful of low-resource-intensive programs or test out the concept of a private cloud based on scalable containers.

2. Revised Design

2.1 Requirements

Hardware:

- The private cloud will be deployed atop a Turing Pi 2 mainboard
- The system will be able to support between 1-4 compute modules at any time

Cloud and Containerization:

- The system will support containerized applications via Docker
- The system will support container orchestration across all nodes via Kubernetes
- The system will be able to scale containerized applications across all clusters according to their resource needs
- The system will have a web API to deploy scalable containerized applications to the private cloud
- The system will expose API endpoints which support blob storage
- At least 90% of the API endpoints will be supported by a served website which allows users to perform all major actions (e.g. upload, download, deploy containers, monitor resource usage)
- The website will be visually simple and aesthetically pleasing, using modern web components and UI principles
- The file storage will be distributed across all compute nodes
- The system will have robust monitoring via its interface which reports functional status (e.g. nominal, process failures) and resource utilization
- The system will support a containerized video streaming application
- The system will show performance improvement for scalable containerized applications as more compute clusters are added

2.2 Engineering Standards

• NIST SP 800-145	The NIST Definition of Cloud Computing
• ISO/IEC 19941:2017	Cloud Computing Interoperability
• ISO/IEC 19944-1:2020	Cloud Computing and Distributed Platforms
• ISO/IEC/IEEE 90003:2018	Software Engineering

2.3 Security Concerns and Countermeasures

The main security concerns for our project are appropriate authentication and authorization—we must ensure both the cloud and users' data are protected. In order to do this, we utilize a reverse proxy that directs all requests through our authentication backend. Once the user is authenticated and authorized, the request is forwarded to the necessary service or handled by the backend itself, if appropriate. before handling them and serving content.

In order to protect the cloud itself, we make use of Kuberenetes's built-in web API. This utilizes a token- and role-based authentication and authorization scheme, similar to our backend. By only provisioning tokens to developers which need access to deploy resources, we can limit direct access to the cloud. Indirect access to deploy is granted via our Gitlab runners on the cloud, which can apply manifests on behalf of users during pipeline runs.

2.4 Design Evolution

2.4.1 Initial Design

Our system model and diagram consist of a layered design in which resources are translated to user applications through various steps, designs, and abstractions. At the bottom, a Turing Pi 2 mainboard, which provides us with networking and interconnect for compute nodes, three Raspberry Pi Compute Module 4s (CM4s). The Turing Pi 2 was designed as a minimal block for a consumer-grade edge infrastructure that is cost-effective. The Turing Pi 2's onboard management controller and network switch allows for the CM4s to work in tandem, pooling their resources so that they can be managed via a singular abstraction wherein Kubernetes orchestrates containerized workloads across devices. To allow software to be deployed, an interface layer making use of Django is implemented. Applications already containerized with docker can use this API as a medium to be transferred downwards and orchestrated with Kubernetes. For example, while the core functionality of uploading a file to a system can be implemented in a variety of ways, an initial POST request to the API which submits the manifest or file to be uploaded is the fundamental point of interaction for uploading. The uploaded container is then handed off to Kubernetes amongst the three Pis in an optimized and load-balanced manner, or transferred to storage via a Docker volume.

2.4.2 Design Iteration 1

The largest change in-between designs 0 and 1 was the inclusion of the distributed file system (DFS). Originally, we had planned to use built-in Linux programs in conjunction with Docker volumes for persistent storage. However, in order to balance reads/writes as well as stored objects across all three compute nodes and their respective storage, we opted to instead use a distributed file system. This provides many benefits, foremost being replication and load balancing. The DFS can manage the concurrent reads and writes to the files system in order to ensure the integrity of files across all compute nodes and additional storage. This iteration also included the usage of S3 (simple storage service) APIs for interacting with the DFS. In Iteration 1, we planned to use SeaweedFS, an open-source, community-based distributed file store.

2.4.3 Design Iteration 2

The largest design iteration came early in the implementation phase. There were three major additions during this time: database replication, reverse proxying, and a private container registry.

Relational data is vital for almost all business-related applications. So, it makes sense that one of the main tenets of a cloud is the reliability of its relational data. Database replication ensures that when one node fails, users' data is not just safe, but accessible. Using *Kubegres,* a Kubernetes operator for Postgres, we're able to maintain three separate instances of the database at each point in time: one primary instance and two replicated instances. Redundancy is guaranteed via the replicas, and accessibility is guaranteed via a mechanism that promotes a replica to become the primary database when primary failure occurs.

The next major change came to the ingress architecture. While it's possible to serve the frontend web page via the API, we instead opted to use software purpose-built to handle and route large amounts of traffic. NGINX, a popular reverse proxy with great containerization support, was chosen to serve as our reverse proxy that handles forwarding requests to the backend, serving users' files (after authentication with the backend), and serving the static files for the web page. We have 3 replicas of NGINX distributed across the cluster; requests are routed to a particular instance of NGINX via a Kubernetes ingress controller and load balancer. Reverse proxying adds much greater flexibility in our design, letting us change proxy routes and configurations with ease.

As the number of custom-built containers grew, we needed a reliable, private storage solution for the images. While public container registries like DockerHub do exist and are widely used, we decided it would be better to integrate with GitLab's private container registry service. With this, we couple our deployment pipeline, cluster, and the images used in the former together with privacy in mind.

3. Implementation Details

3.1 Detailed Design

Our design is constituted by three major parts:

- Infrastructure: The major hardware and software components of the system and how they interact
- Interface: How users, developers, and administrators interact with the system
- Pipeline: How the system is managed over time via Gitlab integrations, continuous integration, continuous deployment

3.1.1 Infrastructure Design

Our design diagram is included below for reference. In our infrastructure, we work logically upwards in increasingly higher levels of abstraction. When considering the design, it's important to remember the three users of the system: (a) application users, who need to interact with services on the cluster and store data, (b) developers who need to deploy applications and infrastructure to the cluster, and (c) system administrators, who need to ensure system uptime and availability for application users and developers.

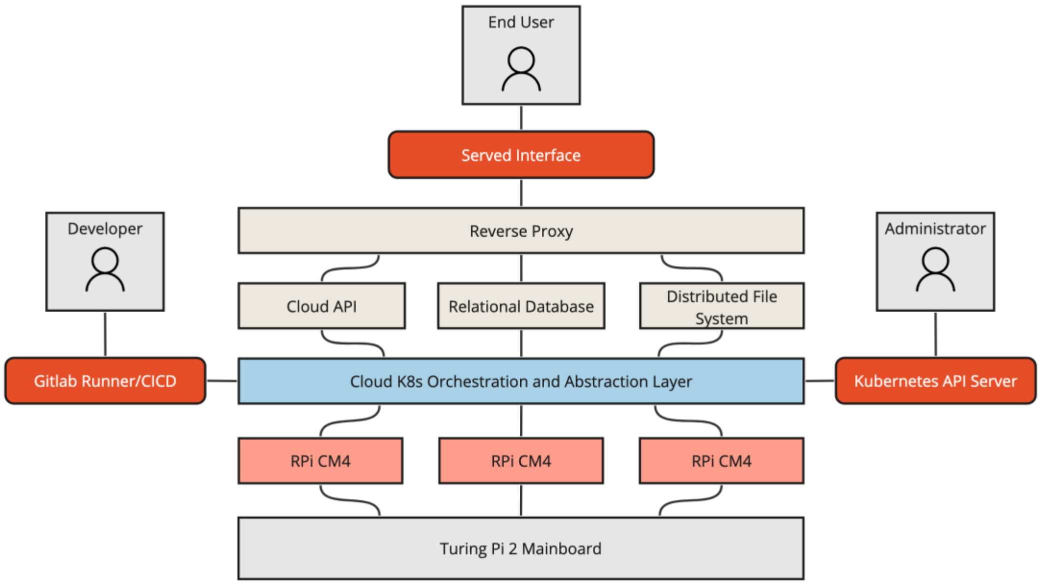


Fig. 1 High-level design diagram of the cloud system. Higher levels of abstraction devolve into specific hardware as the diagram moves from top to bottom. Each main user of the cloud is also shown.

Turing Pi 2 Mainboard:

At the very bottom is the Turing Pi 2 Mainboard, which contains an *Allwinner T113-S3* system-on-chip (SOC). This chip runs an embedded Linux operating system that is used to manage the onboard switch and allow low-level configuration of the interconnects of the Raspberry Pi Compute Module 4s. We're able to remotely login to this board management controller (BMC) when necessary to configure the network bridge and switch.

Raspberry Pi Compute Module 4s:

The main workhorses of our private cloud are three Raspberry Pi Compute Module 4s. Each contains 4 GB of memory and quad-core ARM processors. We've allocated 64GB SD cards to each for operating system use, and 500GB solid-state drives (SSDs) to each for the cloud's distributed file storage. The devices are fitted to carrier boards which allow them to slot into the Turing Pi 2 mainboard. The Turing Pi board interfaces the compute modules to the SATA SSDs through SATA headers for one module, and mini-PCIe cards for the other two compute modules that we're using.

Cloud K8s Orchestration and Abstraction Layer:

At the heart of our private cloud is the software abstraction layer that turns discrete computers into a singular compute cluster. We specifically use *K3s*, a Kubernetes (K8s) implementation built for lightweight deployments. Because of networking constraints, we make use of an air-gapped installation in which necessary binaries are directly placed on the cluster (as opposed to located in repositories online).

As mentioned before, two parties must be able to interact at this level in the system: cloud administrators and cloud developers. Administrators can provision supporting, vital infrastructure via the Kubernetes API server. Tokens are manually provisioned to each administrator as need be, allowing them to access the API server with role-based permissions so they can interact with least-privilege permissions. Cloud developers, on the other hand, interact with cloud via Gitlab pipelines. A Gitlab runner agent is deployed on the cluster, accepting jobs from developers' repositories and enabling them to deploy their infrastructure configuration and apps.

***Cloud API*:**

Allowing users to access the cloud via the API server is superfluous and certainly doesn't adhere to best security practices. Instead, we offer a more user-friendly backend that lives on top of the cloud abstraction layer. Using Django and the Django Rest Framework, we've developed an application that can interact with other components on the cloud, including the distributed file store, relational database, the reverse proxy, and any other deployed apps.

The main interfaces for the API involve the two fundamental building blocks of cloud: users and their data. The API functions as the sole method of authentication and authorization on the cloud—users wanting to access any part of the cloud will interact first with the API to authenticate and authorize. Afterwards, the request is re-routed to the intended destination. For example, if a file is requested, the API must first authenticate and authorize the user before directing the reverse proxy (NGINX) to serve the file. All other components of the cloud (the distributed file store, database, etc.) are only available internally, ensuring this order of operations.

We utilize Django's user management tools for password hashing, permission checking, and session/token management. The backing database and file store, *Kubegres* and *Rook-Ceph*, are discussed next.

***Relational Database*:**

For relational data on the cloud, we've provisioned *Kubegres*, a Postgres operator for Kubernetes. The relational data on the cloud is sensitive and vital to operation, so maintaining security and redundancy is key.

For security, we utilize Kubernetes's secrets, which facilitate what the name implies: secure, protected secrets for infrastructure passwords, tokens, and keys. Only those with direct administration access to the cluster are feasibly privy to these.

Redundancy is the reason we chose *Kubegres* over other options. While we considered a host-based Postgres database in the beginning, we knew we wanted replication across the cluster to ensure that data was accessible and intact even in the case of node failure. *Kubegres* guarantees this. We make use of a configuration which has one primary and two replica databases. The replicas can be used for reads, while the primary can be used for reads and writes. If the primary were to fail, *Kubegres* promotes a replica to become the primary. Thus, we can almost always guarantee uptime of the database, which means uptime of our main authentication and authorization schemes.

Distributed File System:

Persistent volumes and persistent volume claims are the basic way of sharing data amongst processes on Kubernetes. However, manually provisioning these is error-prone and leaves room for downtime, less scalability, less redundancy, and overall less reliability. Seeing as file storage was at the heart of our application and cloud, we made use of *Ceph*, a distributed file system which ensures all the properties of data we care about. If data is lost, Ceph can use advanced self-healing techniques to recover and repopulate the data onto healthy nodes. If more storage is added, Ceph can scale appropriately to consume it and balance data across it. And perhaps most importantly, it's performant. While it's certainly the "heaviest" application on our cluster, it still performs all these tasks with minimal memory and CPU utilization.

Actually, managing Ceph, on the other hand, can still be difficult and cumbersome. Ceph has seemingly billions of knobs and limited Kubernetes integration. To make it a more seamless part of our infrastructure, we use *Rook*, an orchestrator for Ceph (referred to together as *Rook-Ceph*). Rook reduces the complexity of deploying Ceph properly to a Kubernetes cluster.

The interaction with the DFS from the other services' point of view is simple: we're able to provision parts of the DFS via persistent volumes, then containers (such as our backend or other deployed apps) can mount these persistent volumes using persistent volume claims. This allows developers to access files in the way they know best in the format they prefer (as determined by their container).

Reverse Proxy:

Having processes and data deployed on a device is without value unless they can be accessed by users. Combined with a Kubernetes ingress controller and load balancer, our reverse proxy routes connections to the relevant processes.

The first point of contact for our cloud is the Kubernetes ingress controller. Our configuration uses this as a load balancer, routing and balancing requests across the cluster. Each of these requests gets sent to one of three instances of our reverse proxy. We utilize NGINX, a popular choice for cloud-based reverse proxying. Any of the three instances may handle the request. The request is forwarded to the appropriate service. In almost all cases, this is the Cloud API for authentication and authorization purposes. However, some resources may be accessed unprotected—the static website files, for example, may be requested and used by anyone.

It's important to note that NGINX doesn't just route users' incoming requests. After authentication and authorization, NGINX may need to reroute requests based on the `X-Accel-Redirect` header set by the Cloud API server. This is how, for example, large files are served—the Cloud API server validates access to the resource and directs NGINX to serve it (since NGINX is much more suitable and performant for these tasks than Django).

Served Interface:

Most users likely don't want to interact with the cloud solely through API requests. For this reason, we've provided a web page which facilitates interaction with the API through a user interface. To build this interface, we used the React framework. We aimed to expose our file-handling endpoints through an interface that would feel familiar to users and seem akin to software like Windows Explorer or MacOS Finder as well as sites like Google Drive, Box, and Microsoft OneDrive.

3.1.2 Interface Design

The purpose of our interface is the abstraction of the underlying layers of software and hardware. It should be easily accessible and intuitive to use, so the user interface takes the form of a React based webapp with lightweight design elements pulled from common file system interfaces.

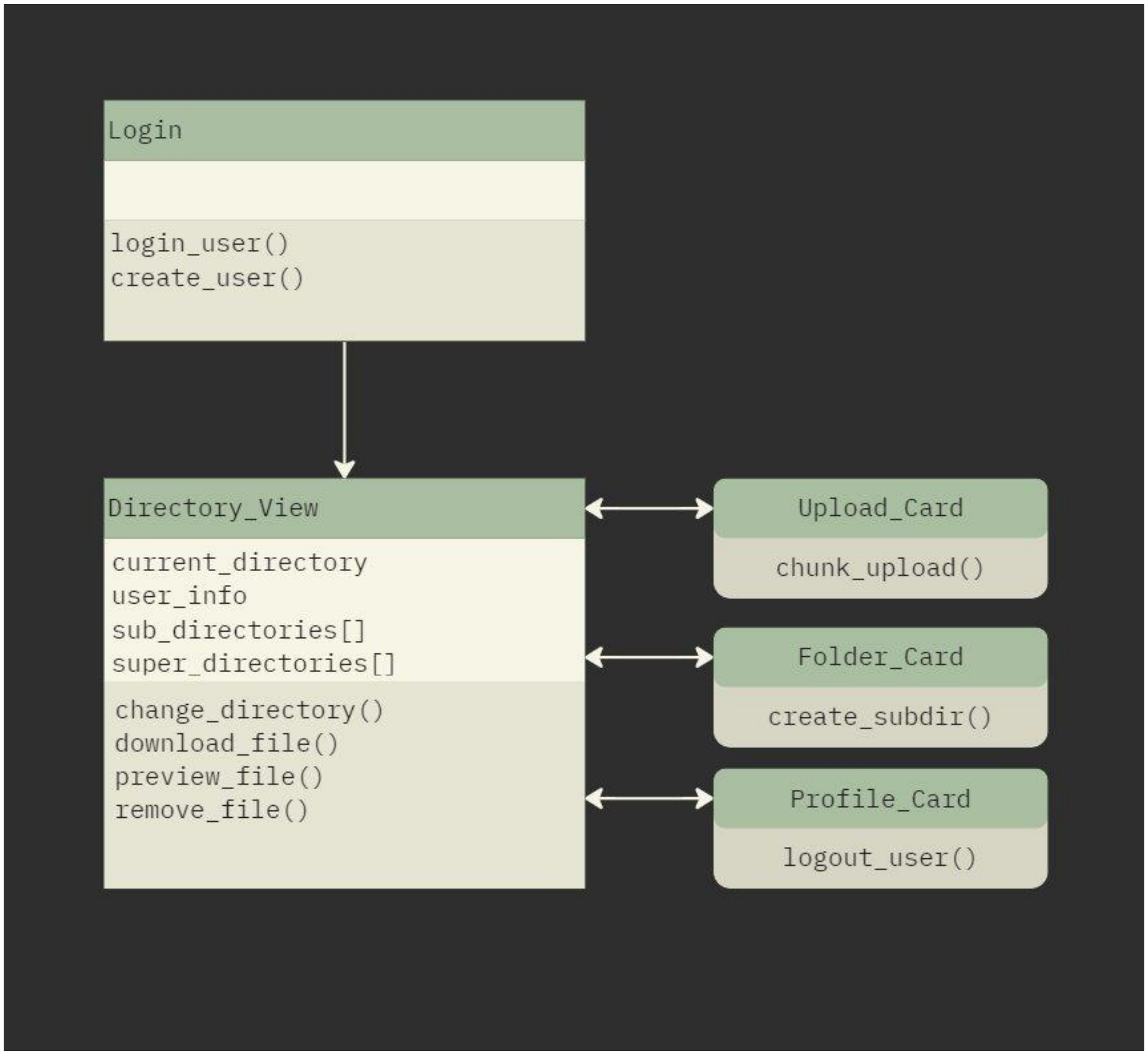


Fig. 2 Final Screen Navigation Layout

Good principles should be followed for the design, including:

- A single page web-app-like design

- Responsiveness

- Excellent feedback for user actions

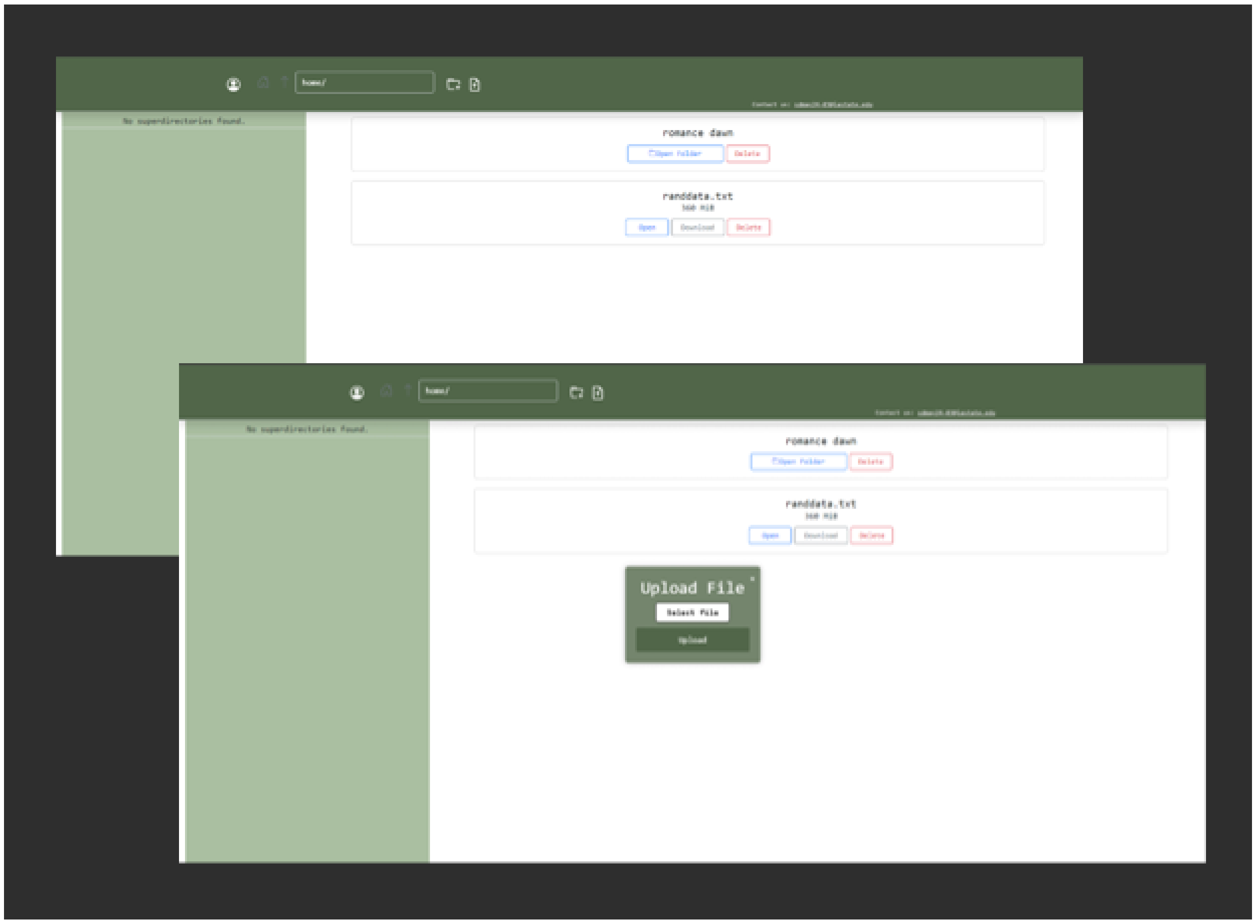


Fig. 3 Final Screen Design

To meet these criteria, we adhered to the precedent set by other file managers including a list-like visual of files in the current directory. Buttons are prominent for the actions users are most interested in: uploading, downloading, and viewing files. We attempted to eliminate any visual fluff which may otherwise bog down the reactiveness of the page or take focus away from the essential parts of the design.

3.1.3 Pipeline Design

Maintaining consistency and coordination between developers' workflows, automated testing, and deployment is essential for a private cloud. Analogous to how a Kubernetes cluster has a "control plane" node which manages configuration for worker nodes, we make use of a "controlling" Gitlab project and pipeline that helps keep the cluster provisioned with the latest changes at all times.

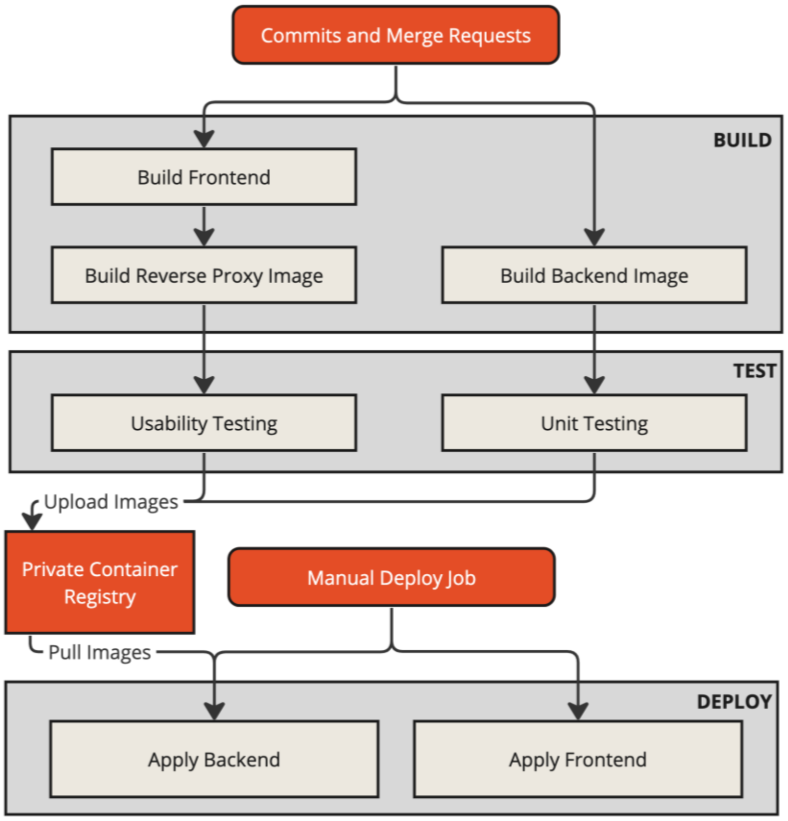


Fig. 4 High-level flow of the continuous integration and delivery pipeline.

The pipeline first comes into play when merge requests are created on Gitlab. Before changes are merged into the main branch, they must be vetted. This includes a build and test process. Building, in-and-of itself, is a quick way to ensure the software is valid—there is no point in attempting to test or deploy software that will not build. At this point, we compile the frontend from React into static files. These files are then used in the next part of the build: constructing the NGINX Docker image. Our NGINX docker image contains both configuration of the reverse proxy as well as static files which must be served. These static files for the frontend are included in this build. In parallel, the backend is also built into a Docker image. If either build fails, the pipeline is halted and a failure is indicated.

Following the build stage, testing is conducted. We've written a suite of tests for each major component, and this is further described in *Section 4: Testing*. It is as this point we ensure regression hasn't occurred with the new changes.

Upon success of earlier pipeline stages, administrators have the option of deploying the most recent build to the hardware. At this point, the Gitlab runner will apply or reapply all infrastructure manifests to the cluster. This ensures infrastructure matches the configuration specified within the Git repository. The runner will also issue rolling restarts. These restarts prompt the backend and NGINX deployments to pull the latest Docker images from where they are safely stored within the Gitlab container registry. This way, all containers running on the cloud platform are fresh and never stale.

3.2 Description of Functionality

When deployed, users will be able to use the cloud for hierarchical and relational data store. What may be a simple button press for the user translates into an intricate interweaving of requests to different parts of the infrastructure.

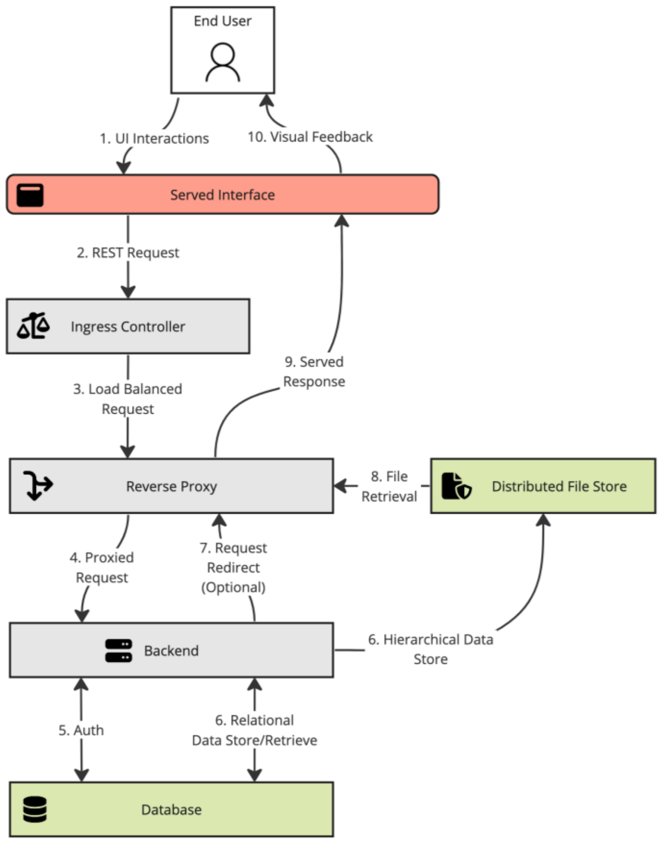


Fig. 5 Functional diagram which illustrates how an end user can upload and receive relational and/or hierarchical data.

The request's journey begins at the served web page. When a user interacts with an element of the webpage such as a button (1), a RESTful request is sent to the cluster at its designated URL (2). Here, the Kubernetes ingress controller begins handling the request, employing Kubernetes' algorithms for selecting a node to handle the request. From there, the request is passed on to an instance of the reverse proxy (3). Note that at any point in this process, it's possible that requests may jump from compute node to compute node based on a variety of factors.

The reverse proxy inspects the request. If the request is asking for public, static files, the proxy is capable of serving them directly. This is how the user would acquire the files to render the website in the first place. If the request is destined for the API, the reverse proxy forwards it to the backend (4). At this point, the backend may need to authenticate and authorize the user, depending on the nature of the request (5). A request to login, for example, requires authentication. A request for private files also needs authorization.

After the request has been deemed valid, the backend must construct a response. Often, the response is just a status code (e.g. *200 OK*). Other times, the request must include relational data (such as what files a user has access to) which must be queried from the database. If this is the extent of the response, the backend may respond directly to the client. In other cases, we must revisit the reverse proxy. Serving files, for example, is better handled by a reverse proxy than an API. For these responses, the backend makes use of a redirection back to the reverse proxy, indicating that the reverse proxy should intercept and serve the requested resource (7, 8).

Whichever route is taken, the user's web page will receive a served response, which it can then render a feedback and visualization for. (9, 10).

3.3 Notes on Implementation

While we've worked around any hurdles that have arisen, we'd like to note some difficulties which may be referenced by future teams. These may also add context to certain design decisions.

- The network settings assigned to our devices prohibited reaching some sites. This prevented downloading K3s directly, pulling Helm charts for Gitlab Kubernetes agents, and pulling some images/software from Github or Gitlab repositories. To circumvent this, we generally adapted an "air-gapped" approach in which necessary installation materials were copied over and installed manually.
- Additionally, our network settings made building Docker images from *within* Docker containers on the machine difficult. We opted to use shell runners for our pipeline instead, allowing us to manage permissions and network settings more easily on the host machine.
- Gitlab's AgentK, its operator which allows a more complete Kubernetes integration, is supposed to be multiplatform. However, the latest images which are compatible with the Electrical and Computer Engineering Department's Gitlab instance do not support the ARM64 architecture (the architecture of the Raspberry Pi Compute Module 4s). This prevented us from using Gitlab's repository integration with Kubernetes. To work around this, we again used shell runners located *on* the cluster which could interact with the Kubernetes deployment.

4. TESTING

4.1 Process

We have two major designs which must be under test: the private cloud stack, and the developed software application.

Infrastructure Testing

We perform unit tests to ensure baseline requirements are met, functional tests to measure throughput, monitoring to measure workloads and utilization, and intermittent health checks to test availability. Initially, we also performed tests of the physical hardware. After installing the storage adapters and configuring the network, we made sure that both storage and internet were healthy.

Unit testing of the cloud API occurs within the CICD pipeline. We make use of Django' Rest Framework's built-in testing framework which allows for simulating requests in addition to regular testing of backing user and file models. We maintain 85% code coverage with these unit tests. If each unit test passes, deployment can continue. This helps prevent regression bugs, too; changes which break the API will cause previously-written tests to fail.

Deployment via the pipeline helps catch integration errors, too. If, for example, the frontend build cannot be integrated with the NGINX proxy image, the build will fail. The other way in which we check both integration, availability, and resource utilization is intermittent health checks. Our infrastructure has plenty of interconnected and moving parts, we use several dashboards and tools which allow us to validate that the platform is not under duress, healthy, and integrated. Foremost are Grafana and Prometheus, respectively dashboard and monitoring frameworks. Prometheus allows us to scrape data from each major infrastructural component using hooks they've exposed. Grafana helps us *visualize* that data with concise dashboards. With both combined, we're able to see CPU and memory utilization broken down by each process over time. Additionally, we can receive alerts for system failures. With the above, we can maintain good knowledge of the integration and uptime of our cloud.

Frontend Testing

The frontend usability was tested by putting a variety of people with no experience with the design in control of the webapp, then asking them to perform a series of basic tasks. All test subjects were able to complete all given tasks.

4.2 Results

To observe best-case performance testing of the overall design, we first tested transfer speeds of the DFS. Since the DFS is designed to be hardware-abstracted, somewhat like RAID, we only needed to consider the existence of the file somewhere in the DFS, opposed to being in a specific physical drive. Additionally, the theoretical best-case transfer speed of the DFS was to either the SD-cards, or to the ethernet interface. The results of the SD-card testing to the DFS produced about a 4.5 minute duration to transfer one gigabyte of raw random data. This translated to about 3.5 megabytes per second, which is slower than the physical transfer speed of either the SD-card or the SATA SSDs. The slowdown is inevitable with the overhead produced by the DFS, though the degree was surprising. Additionally, testing a copy speed of a file within the DFS to the DFS, produced a transfer speed of 41.5 megabytes per second, a significant speedup than interfacing to the SD-cards. This indicates that a DFS interface, offering a professional storage design, provides a speedup over direct SD-card management.

Further measurements of the above tests revealed that the Raspberry Pi Compute Modules were performing below ratings due to them being insufficiently cooled. Kubernetes opts to pin most of the load to the first core on the control plane, making it a possible bottleneck to job distribution, and is more critical to be performant than the other nodes. When CPU-intensive tasks are ran on any node, the compute modules often peak at 85 degrees Celsius and thermal throttle, reducing computational performance. The obvious performance metric is of parallel computing and file serving per Watt. The cluster averages 20 Watts under moderate computational load, with a peak around 25 Watts. This indicates that any computation on the cluster only uses a fraction of the power that a typical PC or PC-based server, producing highly-efficient distributed computation.

5.1 Public health, safety, and welfare:

While our design isn't intended to be consumed or affected by the general public, many private clouds *are* backing industries which are vital to public health, safety, and welfare. In fact, most private clouds are created because of the sensitivity to these concerns. It'd be important in those scenarios to consider things like distributed file store encryption, certificates for HTTPS, and potentially even air-gapping entirely and isolating from the network.

5.2 Global, cultural, and social:

As our global society shifts to a more "cloud-first" mindset, it's important to understand what clouds are, i.e. they *are not* magic. While our project doesn't reach lengths that deem it impactful globally, culturally, or socially, the concepts it works certainly do. Making sure developers know the societal impacts of amassing data and the importance of redundancy is a key part of making clouds work for the common good. With the increasing needs for affordable and secure computing resources, this project has shown that the Turing Pi platform makes server-style computing much more accessible than what would otherwise be exclusive to large enterprises, satisfying the social needs of our more interconnected world.

5.3 Environmental:

As noted in our testing, our private cloud doesn't consume much energy—our environmental impact is limited. This is an important consideration for clouds. More headlines appear by the day with concerns around the amount of energy going toward (and therefore often environmental destruction as a result of) large-scale distributed computing. When possible, it's worth considering the option to move away from large, power-hungry servers and toward smaller, ad-hoc devices like the Turing Pi to conserve.

5.4 Economic:

This is a relatively affordable product for users to obtain. The only things they need to purchase are the hardware components. Typically, public cloud providers lure customers with pricing that scales. With

Turing Pi, you know *exactly* the cost of your cloud, forever. The only variable cost over time is the cost of energy. Additionally, the Turing Pi offers a ready-to-use platform, which a user could opt to design themself, for additional cost savings. To be more detailed, the Turing Pi platform supports up to four compute modules of various kinds, each with different cost versus performance. The overall platform pales in comparison to a typical PC-style server, making it a much more economical option for small system projects, with the intent of scalability easily attainable.

6. CONCLUSIONS

6.1 Review Progress

Overall, we made great progress this semester and accomplished nearly all of what we set out to do. We created an efficient private cloud with robust redundancy in case of failure, replication, and resilience. The infrastructure allows for the key tenets of a cloud: relational data storage, hierarchical object storage, process containerization, and hardware abstraction. By the end of the semester, we've created a very tight development cycle that makes it simple for developers to simulate the cloud environment before deploying with a hands-free pipeline.

We did make several iterations along the way which included revisions of the requirements—these mostly came about because of our team learning more about best practices using the technologies and principles the project is based on. For example, while we had initially set out to allow users to upload containers via the web API, we opted to instead route these actions through the Kubernetes API server instead; this, as we learned, is a security best practice. While we had hoped to address authorization and encryption further, we instead opted to focus on the fundamental tenets noted above (circumventing issues with obtaining SSL certificates, requesting network settings changes for our devices, etc.).

6.2 Design Value

One of the key goals for our project was for our team to learn about and explore cloud computing and its underlying infrastructure. Our design brings value to this goal by displaying our exploration and tying in several fundamental aspects of cloud computing, such as running a program in a containerized environment. Our project also brings value to our intended user by providing an easy step-by-step guide on how to create a cloud computing environment. Our design allows people to create their own file storage system, that is fast, reliable, and secure. A user could also develop their own cloud application using the infrastructure that we installed onto the Turing Pi 2.

6.3 Potential Next Steps

There are near-limitless directions in which a further group could take the project. Our hardware platform was the Turing Pi 2, but future groups could take the infrastructure further to the edge using a cyber-physical system. Integrating sensors and actuators using either Raspberry Pis or other devices could enable the cloud to reach new use cases and users.

If future teams would like to continue in a more web-oriented direction, it'd be wise to obtain certificates and set DNS records appropriately for the cloud. Ensuring requests are load balanced at the

DNS resolution level could help reduce uneven strain on the devices. This then gives the project more options at the backend level. Browsers are more keen to cooperate with the backend's request to store session and authorization cookies when a TLS connection is used.

From a hardware point of view, there's always room for more compute power. Clusters are often used for things like training machine learning models. Adding more compute nodes, cooling, storage, and memory could allow the cloud to perform more data-intensive tasks. Deploying the same infrastructure on rack-mounted servers obtained from surplus could be interesting. If a non-ARM64 architecture is used, there is also a good chance for greater support for Gitlab's official Kubernetes integration. Lastly, Dockerizing the pipeline and having runners run within containers could simplify deployment.

Appendix 1: Operations Manual

Our team maintains the approach that the usage of the software should accompany the software itself. We maintain READMEs which instruct the user on building and applying the infrastructure throughout the relevant parts of the repository. We direct those wishing to use the cloud infrastructure to our software repository (see Appendix 4).

For those wishing to utilize the deployed web page, we direct them to our demonstration video, which walks through the common usage of the application.

Appendix 2: Design Iterations

Accompanying Section 2.4, we have provided the diagrams for the design revision prevented to the faculty panel before beginning the implementation page of our design.

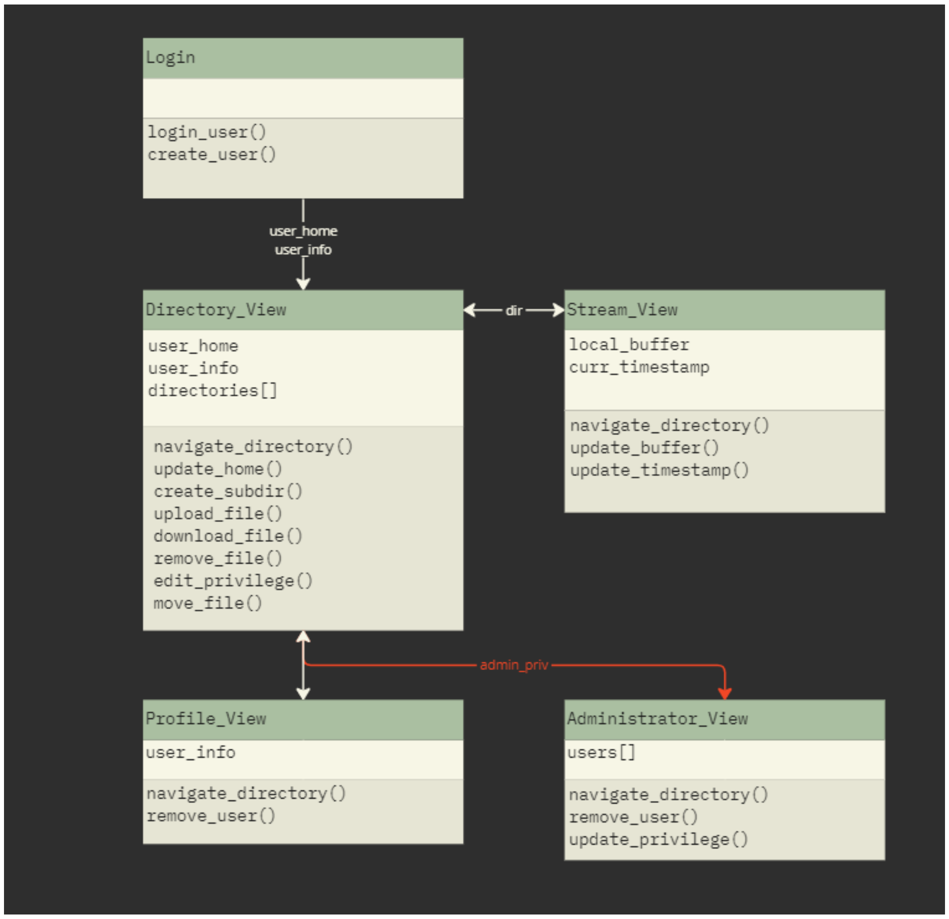
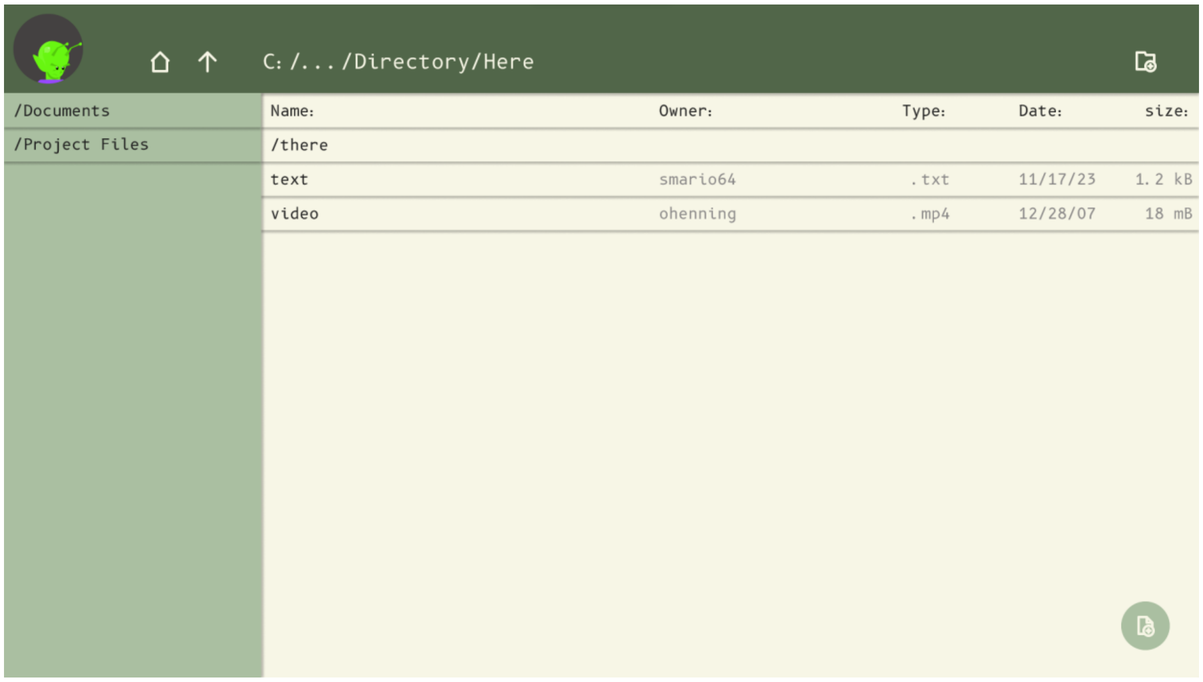


Fig. 6 An early concept for webapp views with expected stored values and intended functionality.



Name:	Owner:	Type:	Date:	size:
/there				
text	smario64	.txt	11/17/23	1.2 kB
video	ohenning	.mp4	12/28/07	18 mB

Fig. 7 Plan for directory view layout.

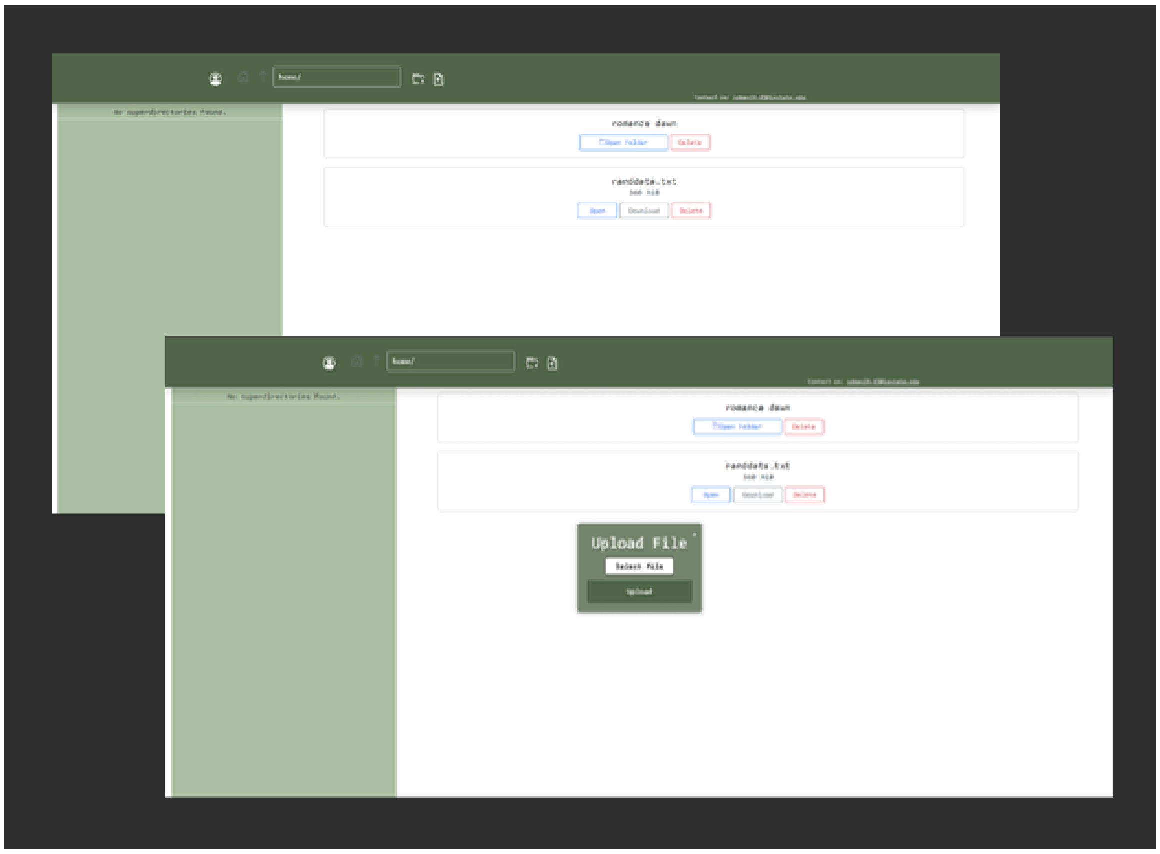


Fig. 8 Final front-end directory view.

APPENDIX 3: OTHER CONSIDERATIONS

We would like to thank Dr. Akhilesh Tyagi for his advice and support throughout the past two semesters. In addition, we'd like to extend a thank you to Iowa State's Electronics and Technology Group (ETG), who was always more than willing to help us acquire equipment and.

APPENDIX 4: SOFTWARE

We use Gitlab to host our software repository. This is available from Iowa State's network at:

http://git.ece.iastate.edu/sd/sdmay24-03

The authors do not maintain a publicly-available mirror of this repository. Please reach out to the team at sdmay24-03@iastate.edu to request a copy of the software.