

1 Description of Overall Design & Architecture Diagram

Major Components:

- **Key-Value Storage (KVS)**
 - Serves as the primary storage system for all data, including user accounts, files, and email information
- **Backend Coordinator**
 - Monitors the status of KVS nodes through heartbeat signals and informs KVS of server status
 - Maintains the health and status of backend servers
 - Elects primary servers for each cluster
 - Ensures that each cluster always has a primary
 - Maintains mappings of server cluster values and
 - Load Balances incoming requests from frontend and email servers across backend nodes to balance the load efficiently.
- **Frontend Server**
 - Handles HTTP requests from browsers, generates appropriate HTTP responses, and renders web pages.
 - Supports functionalities including User Accounts, Storage, and Email.
- **Admin Console**
 - Displays the current status of all servers and offers an interface and shutdown and restart KVS nodes.
 - Provides access to view raw data for troubleshooting and analysis.
- **WebMail Server**
 - Manages viewing both sent and received emails along with sender/recipient, subject and timestamp.
 - Supports sending, forwarding, deleting and replying to emails both within PennCloud and to external users (e.g., to seas.upenn.edu accounts).
 - Integrates with Thunderbird and supports the SMTP protocol for external communication.
 - **Extra credit:** Sent tab to interact with emails in a user's outbox.
- **Frontend Load Balancer**
 - Distributes incoming HTTP requests across multiple frontend servers to ensure optimal performance and availability and monitors the status of frontend servers, keeping track of which ones are active.

How these Components Interact:

Our system is a resilient network of interconnected servers designed for seamless, efficient, and reliable user experiences.

Fault tolerance ensures continuous operation, even during failures.

At the core is the **Key-Value Storage (KVS)**, a distributed database storing user data, metadata, and system states. Each primary server has two replicas for fault tolerance—replicas take over seamlessly during failures, minimizing downtime.

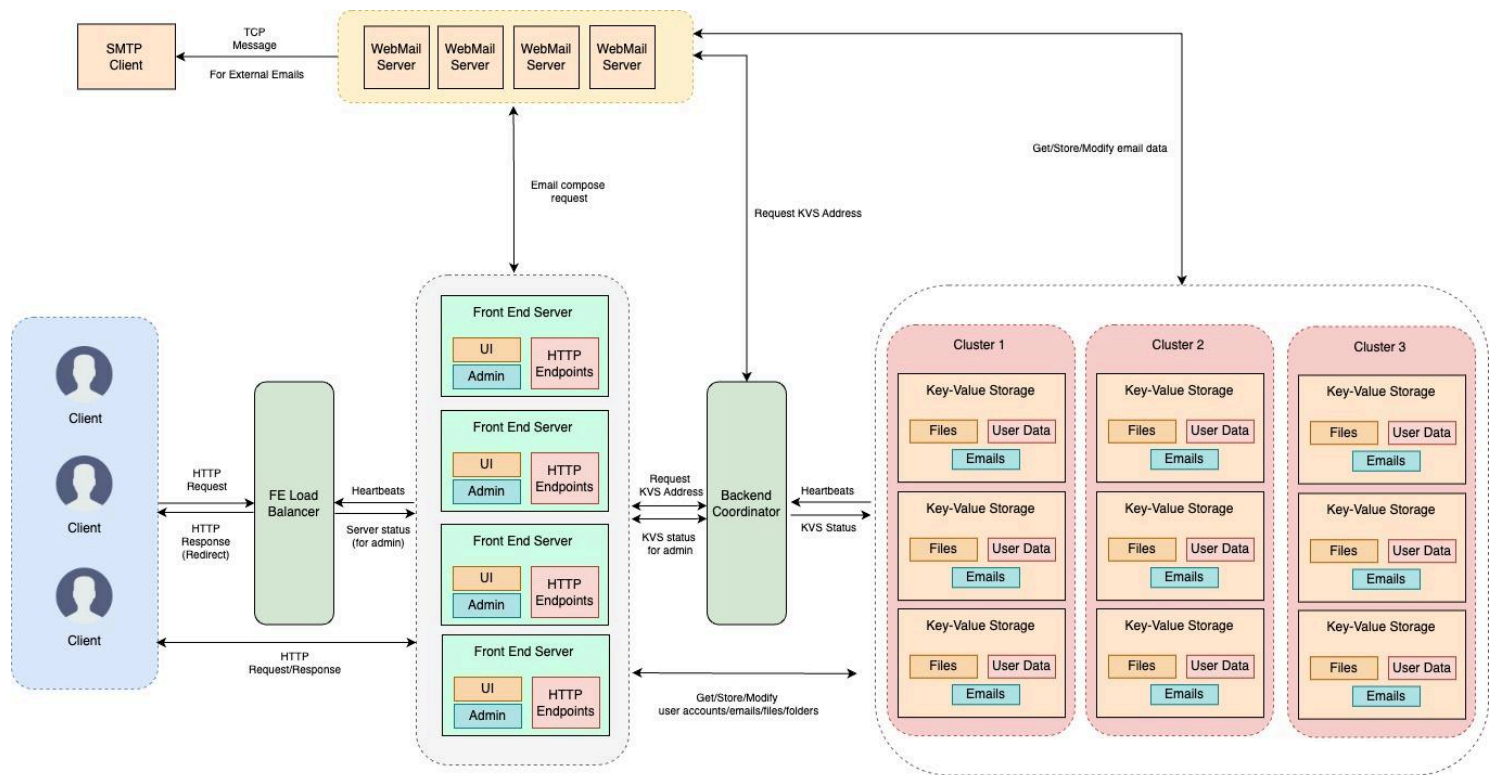
The **Backend Coordinator** monitors KVS servers' health via heartbeats, maintaining system stability by identifying and addressing issues in real time.

The **WebMail Server** manages email operations (compose, delete, retrieve) using SMTP for delivery, interacting with KVS for efficient email storage and retrieval. An SMTP client handles outgoing requests to external domains.

Frontend Servers handle user requests, rendering dynamic web pages (HTML, CSS, JavaScript). They connect with the Backend Coordinator, KVS, and WebMail Server to deliver an intuitive user experience. A **Load Balancer** distributes traffic efficiently among frontend servers, ensuring performance and preventing overload.

The **Admin Console** provides system management tools, allowing administrators to monitor, activate, or deactivate servers and access raw KVS data for transparency and control.

Architecture Diagram:



2 Overview of the Implemented Features & Design Decisions (Screenshots of interface in Appendix)

Key-value Storage

- **Data Storage**
 - The Key Value Store acts as a database where all the data is stored. We distribute the data across 3 groups of 3 KV replica nodes each where each group is assigned a range of keys which they will cater to. In our case, our key ranges were a-i,j-r and s-z. All the requests will go to one of the corresponding nodes in their respective group which is decided by the Backend Coordinator.
 - Every node stores the data in tablets. The data is stored in key-value pairs where the keys(rows) are strings where each string is the username#<function> e.g. username#account, username#files, and username#email. This is done to ensure a single row doesn't get too large. The values corresponding to each key is a map of columns and the inner values which can be of any type.
 - Whenever any data needs to be accessed for read or write, the data is loaded into memory and then the changes are applied. This tablet is kept in memory until some other data not present in this tablet is required. Then we evict the current tablet and load the required one.
- **Fault Tolerance**
 - Whenever a write request comes in, the operation is first logged in a log file which is present for each tablet and this is done across all the replicas in that cluster. Thus if a node dies, the system can still continue to accept any kind of operation.
 - The system is designed to handle multiple failures where several of the nodes can crash. This is done by implementing replication and recovery mechanisms. We also have checkpointing and logging in place to ensure that the data in the nodes doesn't get lost even when the majority of the nodes crash.
- **Consistency**
 - Each row also has a mutex associated with it. This is to ensure sequential consistency and to avoid race conditions when multiple clients attempt to perform operations on the same row simultaneously.
 - Each file (log and checkpoint file) also has a mutex associated with it. This is to ensure thread safety and to avoid race conditions when multiple threads attempt to access or modify the same file concurrently.
- **Replication**
 - We implemented a primary-based replication protocol. When any replica receives a write request, it is first sent the primary replica of that cluster. The primary replica then forwards this request to all the replicas in the cluster and each replica performs the required operation. When the primary gets the success response from the others, it sends the response back to the sender.
- **Checkpointing and logging**

- Checkpointing is achieved by first recording all the write operations to a log file. This log file is essential in the case of recovery if any node wants to recover transactions which were applied to the memory but not written to disk. When a node recovers, it can simply read the log file after getting the checkpoint.
- The in-memory tablets are checkpointed to disk on a regular basis. We kept a track of the number of write transactions taking place since the last checkpoint and then checkpoint after we reach a threshold.
- We implemented a centralized checkpointing mechanism which is initiated by the primary node when the threshold is reached. The primary node will make calls to all the replicas and initiate checkpointing. The checkpoint version gets updated and gets stored as metadata to disk along with the contents.
- Recovery
 - When a node dies, the remaining nodes start getting more requests and the primary server is notified about its status. So we need to restart the nodes and ensure that it is in sync with the other nodes and has the latest data. So during recovery, the restarted node first tries to get the latest checkpoint version from the primary server. If the checkpoint number of the primary server is the same as that of the recovering server, then it fetches the difference in logs from the primary server as the checkpoints are already in sync, just the log files need to be updated.
 - If the checkpoint versions are different, it fetches the contents of the checkpoint file from the primary in chunks and also does the same for log file entries. Once it has recovered, the coordinator marks it as recovered and informs the primary server.
 - In case, the last remaining node also crashes and later comes back up, it gets to know that there are no servers in its cluster by the coordinator. Then the server does local recovery after which the coordinator makes it as the primary server.
- Design Decisions and Challenges
 - We had initially implemented a basic version of 2PC commit protocol for replication where the primary waits till it gets acknowledgments from all replicas and only then informs them to perform the operation. But it was slightly slowing down the system when we were uploading large files. Also due to time constraints we couldn't test the various failure cases. So we decided to drop that idea
 - Currently we only have one tablet in memory and this tablet gets evicted when we receive a write operation for a separate tablet. This can be improved by having some kind of a mechanism like LRU where we just evict the least recently used tablet instead.
 - Initially all data was stored in strings in our log file and we were logging each operation done in the log file line by line. We then shifted to storing data in bytes instead as that was much faster but we started facing issues when logging. During recovery of nodes, when the log files were applied to the in memory tablet, we were getting slight data corruption as some bytes were getting lost. In order to fix this, we started appending the length of the operation with each operation while adding to the log file.
 - We also came across a rather difficult scenario. If a node crashes, the other nodes continue accepting operations. Now if they too crash without checkpointing and the node that crashed in the beginning came back up, then this node becomes the primary. Now if the other nodes come back up, even though they have the same version as primary, they will have a conflict in log files. For this some kind of a conflict resolution mechanism will need to be in place.
 - All communication between the replicas, coordinator, frontend and webmail servers was done using gRPC. This saved a lot of time as it took care of serialization, thread management etc.

Backend Coordinator

- Periodic heartbeats

Heartbeat checks are critical for real-time monitoring of server health. The Coordinator periodically sends heartbeat requests to servers in a loop, with a frequency of 500ms. If a server does not respond, it is marked as "down," and the Coordinator continues attempting reconnections at the same interval. If the server remains down, the primary server of the corresponding cluster is notified of the failure. Similarly, the primary server is notified when the server comes back online.

To avoid excessive calls, overwhelming the gRPC stub channels, or blocking communication with the primary server, status notifications are sent only once when a change in the server's status is detected. If the delivery of the notification fails, redelivery attempts are made until the message is successfully delivered and marked as such. This mechanism ensures efficient communication without compromising reliability.
- Server-Cluster Mappings

The Coordinator organizes 9 servers into 3 clusters, with 3 servers in each cluster, to streamline operations and distribute workloads. Cluster numbers are associated with character ranges in the alphabet to evenly distribute users across the clusters. Each server's address (IP:port) is linked to a specific cluster, enabling the Coordinator to identify which cluster a server belongs to. This mapping simplifies server management and monitoring within the system, allowing for seamless coordination across clusters.
- Primary Servers and Role Assignment

Primary servers are central to cluster operations, handling critical tasks such as checkpointing, logging, recovery, replication, and coordinating with other servers for these operations. The Coordinator maintains a data structure that maps the primary servers to their respective clusters. This mapping is updated in real time to ensure robust and uninterrupted service.

- **Assigning and Reassigning Primary Servers**
The Coordinator dynamically assigns primary servers to clusters based on live server statuses. In the event of a primary server failure, a live secondary server within the cluster is promoted to the primary role. This redundancy ensures operational continuity and minimizes disruptions caused by server downtime.
- **Live Cluster Details**
The Coordinator maintains a comprehensive list of the live status of all 9 servers. This live mapping is used by the frontend servers to populate the admin console page.
- **Round Robin Load Balancing**
The Coordinator ensures efficient load balancing across backend servers to evenly distribute workloads and prevent any single server from becoming a bottleneck. For requests redirected by frontend and mail servers, the Coordinator uses a round-robin strategy, distributing requests sequentially among backend servers within a cluster. This is achieved by tracking the current server index for each cluster in use, ensuring workloads remain evenly balanced. The Coordinator identifies the appropriate cluster for a user based on their username. The frontend and mail servers then continue to use the assigned backend server for the user throughout their session. This approach ensures optimal performance, high availability, and fault tolerance across the system.
- **Design Decisions and Challenges**
 - Initially the co-ordinator sent a message to the primary of the cluster after every heartbeat indicating the status of the servers in its cluster. On testing this turned out to be a huge overhead in terms of communication between the coordinator and the primaries in each cluster. Thus the coordinator was modified to deliver notifications only on status change and more than one message were only sent when redelivery attempts were being made.
 - Another design decision that was made was to use round robin for load balancing over random assignments within a cluster. Round-robin is a better load balancing strategy because it ensures even distribution of requests across servers, preventing overloads and maximizing resource utilization. Unlike random assignments, it provides predictable and fair workload distribution, avoiding the risk of uneven server usage.
 - All communications to and from the coordinator are done using gRPC stubs and calls for reasons stated in the previous sections.

WebMail Server

- **Storing emails in the KVS**
 - The storeEmailData function handles the storage and management of email data within a distributed email system. It starts by retrieving the sender, recipient(s), and email content, parsing the recipient string to handle multiple addresses. For each recipient, the function determines their username and domain, identifies their Key-Value Store (KVS) address using gRPC, and differentiates between PennCloud users and external recipients. For PennCloud users, it updates the KVS with the sender's "sentEmails" and each recipient's "rcvdEmails" columns, ensuring proper serialization of email metadata and chunks for efficient storage. If the recipient is external, the function sends the email using gRPC to an SMTP service. Metadata, including the timestamp and chunk information, is constructed and stored in the KVS. Throughout, the function ensures error handling for KVS updates, metadata storage, and external email dispatch, providing a comprehensive mechanism for managing email transmission and persistence in a distributed system.
- **SMTP Communication**
 - The Mail Server implements SMTP protocols for sending emails to external domains. This includes handling the entire command flow (HELO, MAIL FROM, RCPT TO, DATA, and QUIT), ensuring compliance with SMTP standards.
- **Inbox Management**
 - The Mail Server maintains an organized inbox for users, displaying email headers such as subject, sender, and time. Users can view, reply to, or forward emails and delete messages from the inbox and outbox.
- **Email Composition**
 - The server supports composing new emails for both local and remote recipients. For remote emails, it queries the recipient's domain, resolves the SMTP server's IP address, and connects directly for email transmission.
- **Integration with Thunderbird**
 - The server enables local email retrieval through integration with Thunderbird. It reads configuration files and initializes connections for seamless interaction.
- **Outgoing emails**
 - For outgoing emails, the server acts as an SMTP client, establishing connections and sending data to remote servers.

- Design Decision: Separation of local and remote email handling ensures modularity. The gRPC framework facilitates structured communication for querying domain-specific SMTP servers, enabling scalability.

Frontend Load Balancer

- Frontend Server Status
 - The Frontend Load Balancer periodically receives gRPC heartbeat calls from Frontend Servers and maintains a status map for each server.
- Load Balancing
 - Upon receiving an HTTP request, the Frontend Load Balancer uses its status map to identify a live Frontend Server and sends a 302 redirection response to the browser.

Frontend Server

- Design
 - The Frontend Server is a multithreading server that has a socket listening to a specific port by reading the server configuration file. It processes http requests from the browser and sends HTTP responses to render HTML pages with CSS and JavaScript or redirect the browser where needed.
 - The PennCloud system is supported by 4 Frontend Servers for fault tolerance and load balancing. The server is kept stateless to ensure smooth recovery when faults happen.
- User Accounts
 - The application supports user login, registration, password changes, and logout, all backed by the Key-Value Storage. During login and registration, user data is verified, ensuring credentials are correct or usernames are available. Password changes validate the current password before updating to a new one.
- Session Management
 - When a user successfully logs in or registers, a session object is created containing the session ID, username, and the assigned Key-Value Storage based on the username's first character. The session ID is included in the HTTP response to set browser cookies.
 - Subsequent requests include the session Id in the cookies, which the server verifies against the session object to maintain persistent connections until the session expires or the user logs out. If authentication fails, the user is logged out. Additionally, users with a valid session in the cookies are redirected to the menu page when attempting to access the login or registration pages.
- Menu Page
 - The menu page allows users to navigate the application via links to the storage service, email service, change password, admin console, and logout options.
- Storage Service
 - The storage page provides an interface for a user to upload a file, create a folder, delete, rename, and move items by integrating with the Key-value Store.
 - Files are stored in the Key-Value Storage using a structured format. Each file is assigned a unique identifier, and its metadata, including location and size, is stored separately. Large files are divided into chunks to ensure efficient storage and retrieval. Folders are stored similarly to files. Instead of chunks, folders store their child folders separately to maintain the relative file directory structure. This structure optimizes simple operations such as rename and move, since the first one involves only the change of metadata, while the second one simply needs to change the child of the original folder, and update the content of the destination folder.
- Email Service
 - Inbox Display
 - Displays received and sent emails in a tabular format with sortable headers and includes action buttons (e.g., View, Reply, Forward, Delete) for each email.
 - Design Decision: Tabbed design separates received and sent emails, improving organization.
 - View Email Page shows detailed information for a selected email, including sender, recipient, timestamp, subject, and content and provides the option to return to the inbox.
 - Compose New Email
 - Allows users to initiate email composition directly from the inbox. The modal collects email details (to, subject, body) and submits via a POST request.
 - Design Decision: The modal approach avoids unnecessary page reloads, enhancing user experience. Mandatory fields and placeholders also guide users in completing the form.
 - Dedicated reply and forward mail pages that auto-fill the “From”, “To” (for replies) and “Subject” fields with appropriate values from the selected email along with the content of the original email. Users can also edit the body before sending the email.
 - Design Decisions
 - All frontend pages use Semantic UI for styling, ensuring a visually appealing and consistent user interface.
 - Responsive design adapts the layout for various devices, enhancing accessibility.

Admin Console

- Server status display
 - The admin console retrieves the live status of the Key-Value Storage and Frontend Servers by making gRPC calls to the Frontend Load Balancer and Backend Coordinator. This happens whenever a user navigates to the admin page. Additionally, a refresh button allows users to manually update the server status.
- Shutdown and Revive Key-value Storage
 - Users can temporarily halt operations of a Key-Value Storage node by clicking the "Shutdown" button, which sends a gRPC call to the targeted node. While inactive, the node cannot process requests until a "Revive" command is issued via the "Start" button in the admin console.
- View Raw Data
 - Users can view raw data from any Key-Value Storage node by selecting the "Raw Data" button. The admin console retrieves the keys (rows and columns) and, if the data isn't file or email content, displays it directly. For file or email content, users can click "View Content" to fetch and display the data on a separate page.
- Design Decisions and Challenges
 - We decided against implementing pagination as the Key-Value Storage lacks indexing to support it, and iterating through the entire dataset would slow performance. Instead, we opted not to display certain data types directly, allowing faster rendering while retaining access through a "View Content" link for specific cases.

Inter-server Communication

- To handle the large volume of messages exchanged between nodes efficiently, we opted to use **gRPC** for inter-server communication. This choice eliminated the need to manually serialize objects for transmission and handle TCP connections, saving significant development time. Additionally, gRPC's built-in multithreading simplified the management of concurrent message processing, reducing the effort needed to implement multithreaded logic. However, while gRPC simplifies much of the communication process, our primary challenge was the initial setup and ensuring that all components adhered to aligned server protocols for seamless integration.

3 Extra Credit

The sent emails tab allows users to view emails in their outbox. It also supports all actions - viewing, composing, replying, forwarding, and deleting - that are supported for received emails.

4 Contributions

Austin: Frontend (User Accounts + Storage Service + Admin Console + Frontend Load Balancing) + FE/Email gRPC calls

Mahika: Webmail server + Frontend

Mikhael: KVS

Shriya: Frontend (Inbox) + Coordinator + Load Balancing (Backend) + Webmail (Storing emails in the KVS) +

Thunderbird + Integration testing + bug fixes

5 Appendix

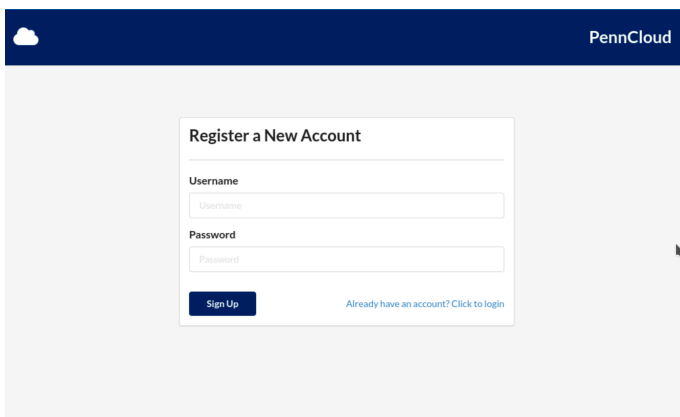


Image 1: Sign up page

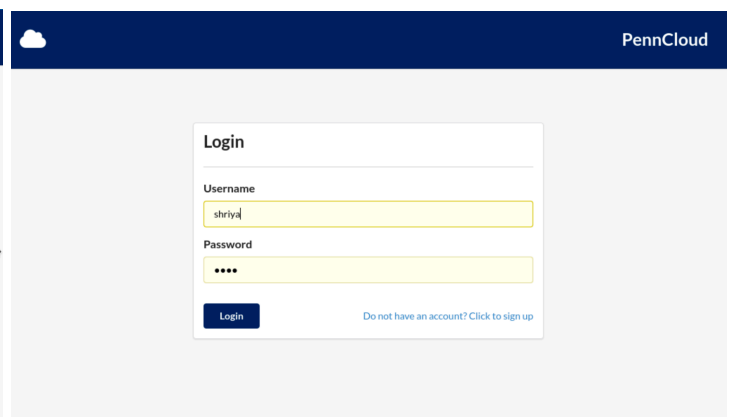


Image 2: Login page

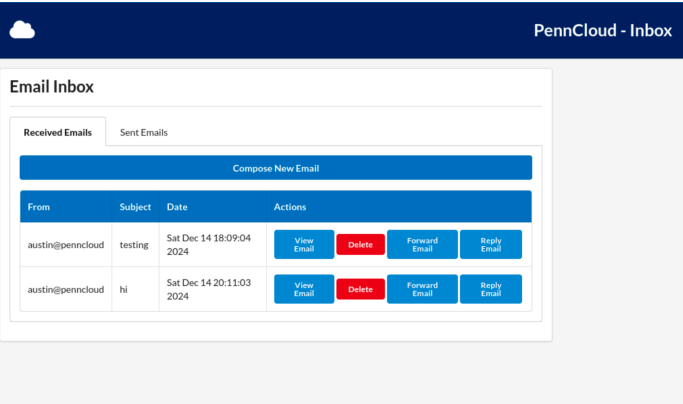


Image 3: Received emails tab

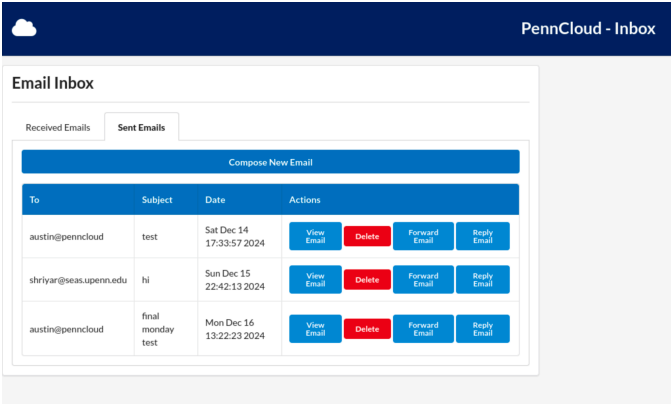


Image 4: Sent emails tab (extra credit)

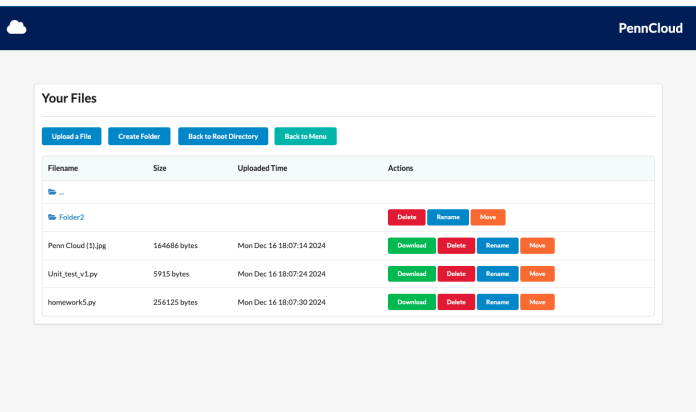


Image 5: Storage page

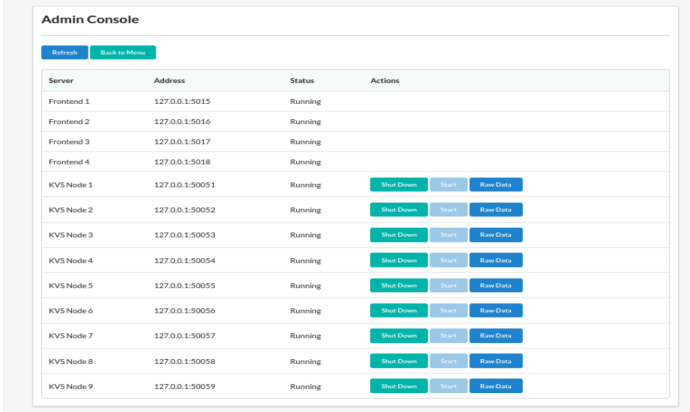


Image 6: Admin Console

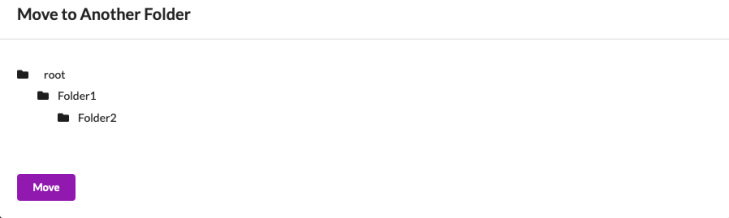


Image 7: Folder Structure

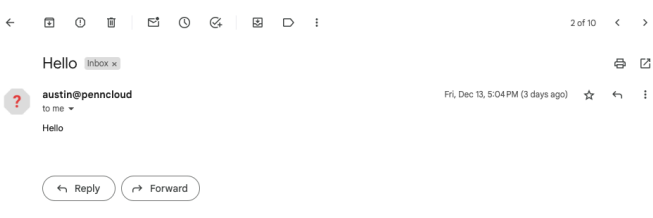


Image 8: Email sent to external domain

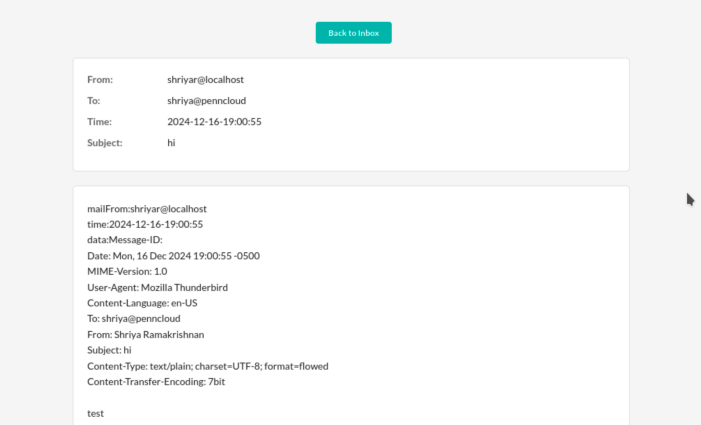


Image 9: Email from Thunderbird