Hybrid Backup Service with Decentralized Sovereignty

Neil Roy, Kevin Lee, Edwin Medrano Villela, Awin Zhang, Suhrit Padakanti

Team Mysten

December 8rd, 2025 ~ PRD v2

Introduction

1.1 Problem Statement

Modern cloud backup services force users into a difficult and persistent trade-off between convenience, security, and long-term data ownership. While mainstream cloud storage platforms offer clean interfaces, fast uploads, and reliable uptime, they also maintain near-total control over user data, even when encryption is part of the offering. Users ultimately rely on a single centralized provider whose operational changes, outages, or corporate decisions directly impact their ability to access their own files. This creates risks:

- **Vendor lock-in:** Providers may shut down, increase prices, or alter policies (e.g., reduce encryption guarantees), leaving users with limited options.
- **Regulatory Pressure:** Centralized companies can be compelled to weaken encryption guarantees, provide backdoor access, or restrict data retrieval during disputes or compliance checks.
- **Data Loss:** If the central service fails, users may permanently lose access to backups.

Alternatively, decentralized storage backed by cryptographic protocols can eliminate many of these risks by distributing data across independent nodes. However, decentralized systems present their own barriers:

- Users must understand complex concepts such as wallet management, gas fees, blob expiration, and cryptographic identifiers.
- Decentralized networks often lack the low-latency performance that mainstream users expect.
- The learning curve for interacting with blockchain-based systems is prohibitively steep for non-technical audiences.

Thus, existing solutions force consumers into a false choice:
Choose convenience with no sovereignty, or sovereignty with severe usability costs.

Our project addresses this gap by merging centralized speed with decentralized ownership in a single system that behaves like familiar cloud storage while guaranteeing long-term recoverability independent of the service provider.

1.2 Motivation and Innovation

Most existing backup solutions force users to choose between convenience and control. Traditional cloud services prioritize speed and usability, but often trap users in closed ecosystems. In contrast, decentralized storage platforms emphasize sovereignty at the expense of accessibility and ease of use. Our project eliminates this trade-off through a **hybrid architecture**, ensuring that users retain **full control** of their **encrypted data** while maintaining the simplicity and speed expected from modern backup services.

What makes our approach impactful:

- **Hybrid sovereignty model:** By layering Walrus as a decentralized backend, our system ensures that users retain full ownership and retrieval rights even if the centralized provider fails or changes their policies/pricing.

- **Decoupled control and infrastructure:** Encryption keys and user data remain separate. Users keep control of their encrypted backups, while infrastructure remains interchangeable. This separation enforces true data sovereignty rather than simply offering *encrypted storage*.
- **Operational simplicity for mainstream users:** Our caching and lifecycle-management layer hides blockchain complexity, handling WAL/SUI payments, renewal events, and bandwidth optimization. Allowing decentralized storage to become viable for everyday consumers, not just technically skilled users.
- **Centralized Wallet / Payment Abstraction:** A major advancement in our current implementation is the introduction of a centralized wallet system that completely abstracts away the complexities of blockchain payments. Instead of requiring users to manage their own crypto wallets, seed phrases, or gas fees, the system provides a familiar funding workflow through Stripe (operating in test mode to avoid real charges). The centralized backend maintains a single on-chain Walrus/Sui wallet and automatically handles all payment-related responsibilities, including estimating upload costs, accounting for storage amplification, calculating renewal fees, and managing blob lifetimes. This design transforms decentralized storage into a seamless, subscription-like experience while preserving all underlying Web3 guarantees.
- **Sustainable, user-first ecosystem:** By standardizing how centralized services interact with decentralized backends, we enable future interoperability among backup providers, promoting a more open and competitive data-storage ecosystem.

In short, our **Hybrid Backup Service with Decentralized Sovereignty** delivers the best of both worlds: the seamless performance of centralized cloud backups and the autonomy of decentralized storage. It redefines what 'ownership' means in personal data management, offering **usability without compromise, resilience without complexity, and sovereignty without friction.**

1.3 Background and Assumptions

Background

Our project builds upon several established technologies:

- **Cloud infrastructure** (AWS S3-style object storage) provides low-latency caching.
- **Walrus decentralized blob storage** offers persistent, verifiable, and content-addressable data storage across independent nodes.
- **The Sui blockchain** provides fast, parallel transaction execution and security guarantees suitable for high-throughput storage interactions.
- **AES-GCM encryption** is a cryptographic industry standard widely applied in TLS 1.3, secure messaging, and modern zero-trust systems.

Assumptions

To ensure technical feasibility, we assume:

1. Users have basic familiarity with cloud storage interactions (uploading/downloading).
2. Users should not need to understand blockchain mechanics or key management.
3. The client device is trusted to generate and store AES-GCM keys.
4. Browser support for the WebCrypto API is widely available.

5. Network latency between the centralized server and Walrus nodes is acceptable for background synchronization.
6. A centralized service can reliably manage the funding wallet on behalf of all users.
7. Users are comfortable interacting with fiat-based payments (Stripe) instead of cryptocurrency wallets.

1.4 Team Goals, Objectives, and Project Specifics

The team's objective is to build a full-stack backup system that combines decentralized permanence with centralized performance.

Team Goals

- Develop a functional, production-grade web application.
- Integrate independent systems (frontend, backend, Prisma/Postgres, Walrus, Stripe).
- Ensure users retain control of encryption keys at all times.
- Provide a seamless user experience that hides all decentralized complexity.
- Build robust caching and deduplication layers for scalable performance.
- Enable recovery pathways even when centralized layers fail.
- Establish a clean architectural foundation suitable for future extensions.

Project Specifics

- **Frontend:** login, upload, download, file history, cost previews, payment page.
- **Client-side cryptography:** AES-GCM encryption, base64 encoding, and authenticated metadata.
- **Centralized backend:** request routing, caching, deduplication, payments, lifecycle management.
- **Persistent storage:** blob upload, renewal, and retrieval via the Walrus network.
- **Database:** Postgres + Prisma for metadata, history, and user accounting.
- **Payment system:** centralized wallet + Stripe for funding and cost abstraction.

This architecture enables the hybrid sovereignty model to function in a real-world environment.

1.5 Core Technical Advance

As stated before, the system implements a **hybrid storage architecture** that unites centralized performance with decentralized resilience. Its design ensures that users enjoy the speed and simplicity of a conventional cloud service while maintaining full control and recoverability of their encrypted data.

Centralized Layer

- **Fast-access caching:** Files are encrypted locally, then uploaded to the central server for deduping, stored temporarily in an S3-like bucket, and then pushed to Walrus. Cached copies allow instant downloads, addressing one major performance gap in decentralized storage systems.
- **Efficient synchronization:** The centralized server manages all bandwidth-heavy communication with the Walrus network, insulating users from the complexity of blockchain transactions and high data amplification (up to 4–5x).

- **Payment and lifecycle automation:** The server abstracts Web3 interactions by automating WAL/SUI payments, renewal tracking, and blob expiration management. This allows users to interact with decentralized storage using familiar subscription or credit-card models.
- **Transparent encryption handling:** All files received by the server are already **AES-GCM-encrypted** on the client side. The centralized layer only handles ciphertext and metadata, ensuring **zero-knowledge operation**; the provider never has access to plaintext data or keys.

Decentralized Layer (Walrus)

- **Persistent, sovereign storage:** The Walrus network stores encrypted user data as **content-addressed blobs**, each verifiable and retrievable through a unique cryptographic identifier.
- **Independence and resilience:** Even if the centralized cache or payment service becomes unavailable, users can directly recover their encrypted data from Walrus using their local AES-GCM key.
- **Portability and trust minimization:** Because data persistence depends on the decentralized network rather than a single provider, users can migrate to other services or implement their own retrieval clients without re-uploading data.

Client-Side Encryption with AES-GCM

Every file is encrypted before transmission using **Advanced Encryption Standard (AES)** in **Galois/Counter Mode (GCM)**. This algorithm provides both **confidentiality** and **integrity** through authenticated encryption:

- **Confidentiality:** Only users who possess the encryption key can decrypt the file; the server and Walrus nodes see only ciphertext.
- **Integrity:** Each encrypted object includes an authentication tag that detects any modification to the ciphertext or metadata.
- **Zero-knowledge, end-to-end encryption:** All encryption and key management occur on the client side; the server and storage nodes never possess decryption capability.

This cryptographic approach ensures that user sovereignty is preserved at a mathematical level—data remains secure even in untrusted environments.

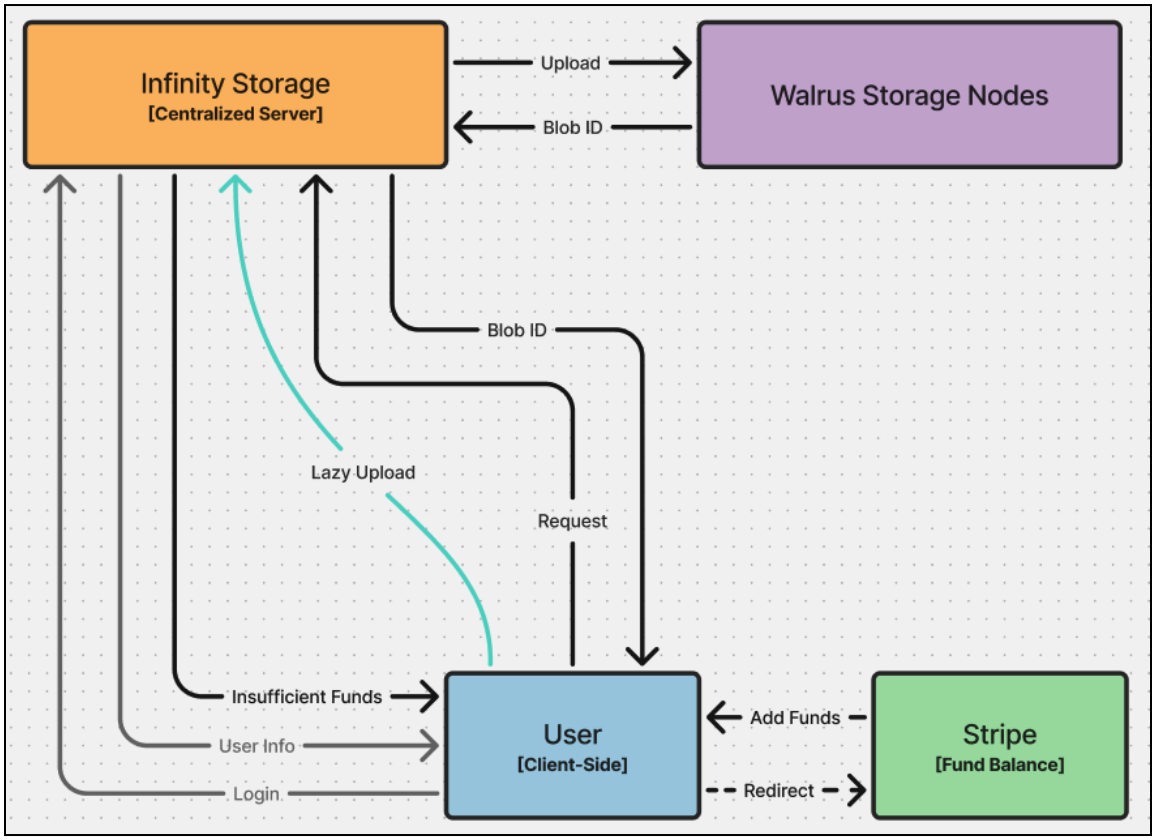
System Advantages

- **Usability without compromise:** Users interact with a familiar web interface and instant caching layer while benefiting from end-to-end encryption and decentralized redundancy.
- **Resilience and direct recovery:** If the centralized cache fails, users can retrieve the same AES-GCM–encrypted blob directly from Walrus.
- **Security and compliance:** AES-GCM meets modern encryption standards (NIST SP 800-38D) and is widely used in secure communication protocols such as TLS 1.3, ensuring both strong protection and regulatory alignment.

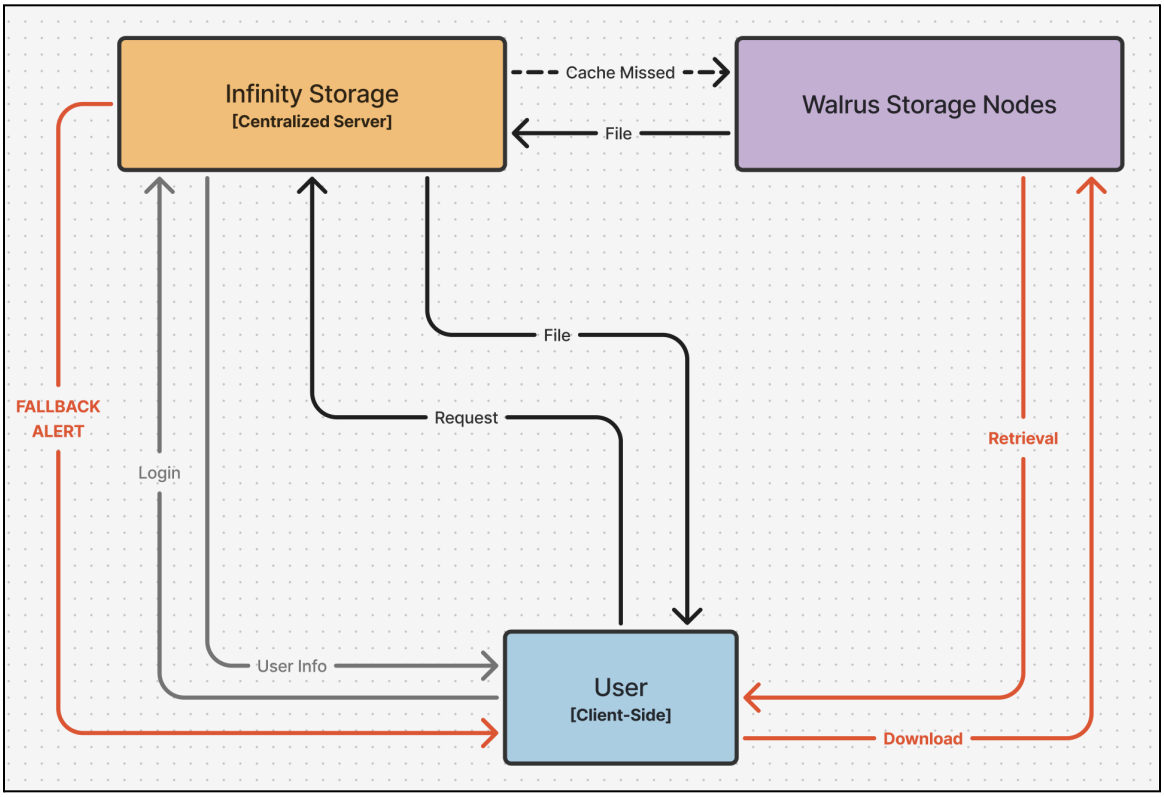
System Architecture

2.1 High-Level System Diagram

2.1.1 Uploads



2.1.2 Downloads



2.2 User Interaction and Design

This section describes how users are to interact with Infinity Storage, our hybrid backup system. Core workflows include: uploading encrypted files, verifying file metadata, restoring data, managing storage, and managing balance. The design prioritizes operational simplicity for the user by concealing the complexity of the blockchain without compromising data sovereignty. Each interaction is shaped around the hybrid architecture detailed in Section 2.1, where the centralized server provides usability while Walrus ensures long-term recoverability.

2.2.1 User Login and Session Initiation

Interaction Overview

Users authenticate with a standard username/password login. We ensure the safety of the user's account by providing minimum password requirements like special characters, numbers, and uppercase characters. Upon successful authentication, the client loads the user's AES-GCM encryption key and initializes a session token with the centralized server.

Design Objectives

- Maintain a familiar login experience while preserving security using zero-knowledge properties
- Ensures users can begin interacting with the system immediately after authentication
- Secure handling of the user's AES-GCM encryption keys

Flow

1. User navigates to the web interface at our URL
2. User enters credentials; server validates identity
3. Client loads or generates an AES-GCM encryption key for the user
    a. This key is made visible to the user in the "Profile" section
4. UI transitions to the dashboard, showing balance and prompting the user to upload files

2.2.2 File Upload Workflow (Lazy Upload + Caching)

Interaction Overview

The user selects one or more files either by dragging and dropping into our UI or the file picker. The user is then prompted to either upload now or upload later. Once a choice is made, a confirmation request appears with the expected cost, storage duration, and user balance. After confirmation is made, the files are then encrypted using AES-GCM, and the process of verifying the files begins. After the files are verified, uploads begin (in a queue) to the Walrus storage nodes for long-term storage. *Note that files are cached when downloads occur, not uploads.

User Experience Goals

- Uploads "feel" instantaneous, similar to modern mainstream cloud storage
- Users should not need to understand WAL/SUI payments, blobs, or replication factors
- The system transparently manages multi-file uploads and background synchronization

Flow

1. User selects files (single or multiple)
2. Client verifies the file's metadata
3. UI prompts a confirmation forthe user (including cost and duration)

4. Files are encrypted, and the upload process begins
5. Encrypted blobs are uploaded and replicated across Walrus storage nodes
6. Once persistence is confirmed, UI transitions the file status to “completed.”
7. Uploaded files then appear under the "history" section

2.2.3 File Retrieval and Download Workflow

Interaction Overview

Users can download any previously uploaded file through the dashboard’s history section. Retrieval follows a cascading fallback model:

- Primary path: download from centralized cache (fastest)
- Fallback path: download directly from Walrus using the blob ID
- Local decryption: client uses the user’s AES-GCM encryption key to restore file contents

This design fulfills the resilience and sovereignty guarantees we promise our clients. Users remain able to access their data even if the centralized provider fails. Accessing data if our centralized server fails is possible through direct interactions with Walrus through CLI or some other interface (another centralized server, perhaps). Upon failure, we expect the user to have their encryption key saved as well as their SUI address backed up, allowing the user to recover their files with no cost.

Flow

1. User clicks “download” on a file in history (or downloads by blobID)
2. System attempts to fetch the encrypted blob from the cache
3. If cache retrieval fails, UI indicates fallback; client then fetches directly from Walrus storage nodes (slower)
4. If cache hits, the user’s file is downloaded directly from there (fast)
5. Update the cache (if this file is popular, this is when the cache is updated)
6. Client decrypts the file contents locally
7. The file is then downloaded to the user's local files at the destination requested by the user
    a. If downloaded from history, the file name remains intact
    b. If downloaded by blobID, the user can optionally provide a file name(default = blobID)

2.2.4 User Management Dashboard

Interaction Overview

The dashboard provides a unified view of all the uploaded files, metadata, and storage state (i.e., how many days until removal). Included in the dashboard is a balance center where users are able to view balance, add balance, and view the live SUI-USD conversion rate. Further, there is a section for user's “profile” where attributes about the user’s account can be viewed and changed (i.e., can change password here).

Key UI Components

- **Recent Uploads View**: History section displays the upload status of each upload (this includes the date uploaded, time remaining among walrus storage nodes, and the file name and size)
- **File Actions**: From the history section, the user is able to directly download, delete (from cache), decrypt, and view the blobID of each file

- **Balance Display:** Shows user's pre-loaded balance (in USD), live SUI-USD exchange rate, and buttons to quickly add cash to balance
    - **Adding Balance:** There are quick-add buttons available for the user to add cash amounts, including: $5, $10, $25, $50, and $100
    - **Stipe Checkout**: Users are able to use familiar checkout methods in order to pre-pay for a balance on our centralized server. By clicking any of these quick-add buttons, the user is then redirected to a Stripe checkout page where they can pay via Apple Pay, Google Pay, Credit, Debit, Link, etc.
- **Profile:** User is able to change password (by entering current password first). The new password must still satisfy the same password criteria when creating an account. The new password must be entered twice and match to ensure the user remembers their password
    - **Encryption Key:** User's encryption key is stored here, visible to the user if they choose. Hidden by default for security reasons.

2.3 System Models & Interactions

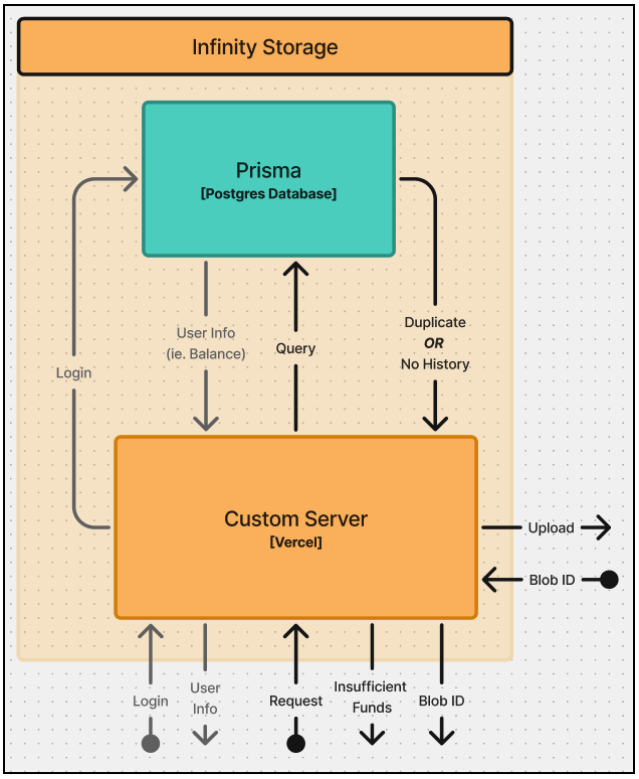
Below are diagrams that break down the two main interactions (upload/store and download) on our application, and summaries on how they are properly handled within our centralized server.

2.3.1 Handling Uploads/Store

We start with the assumption that the user has provided their credentials, allowing the centralized server to have access to any relevant account information (such as funds). With that in mind, a request to upload a file was to come in (having passed all checks, such as file size, etc.), we would then see if the user has sufficient funds to carry out the transaction.

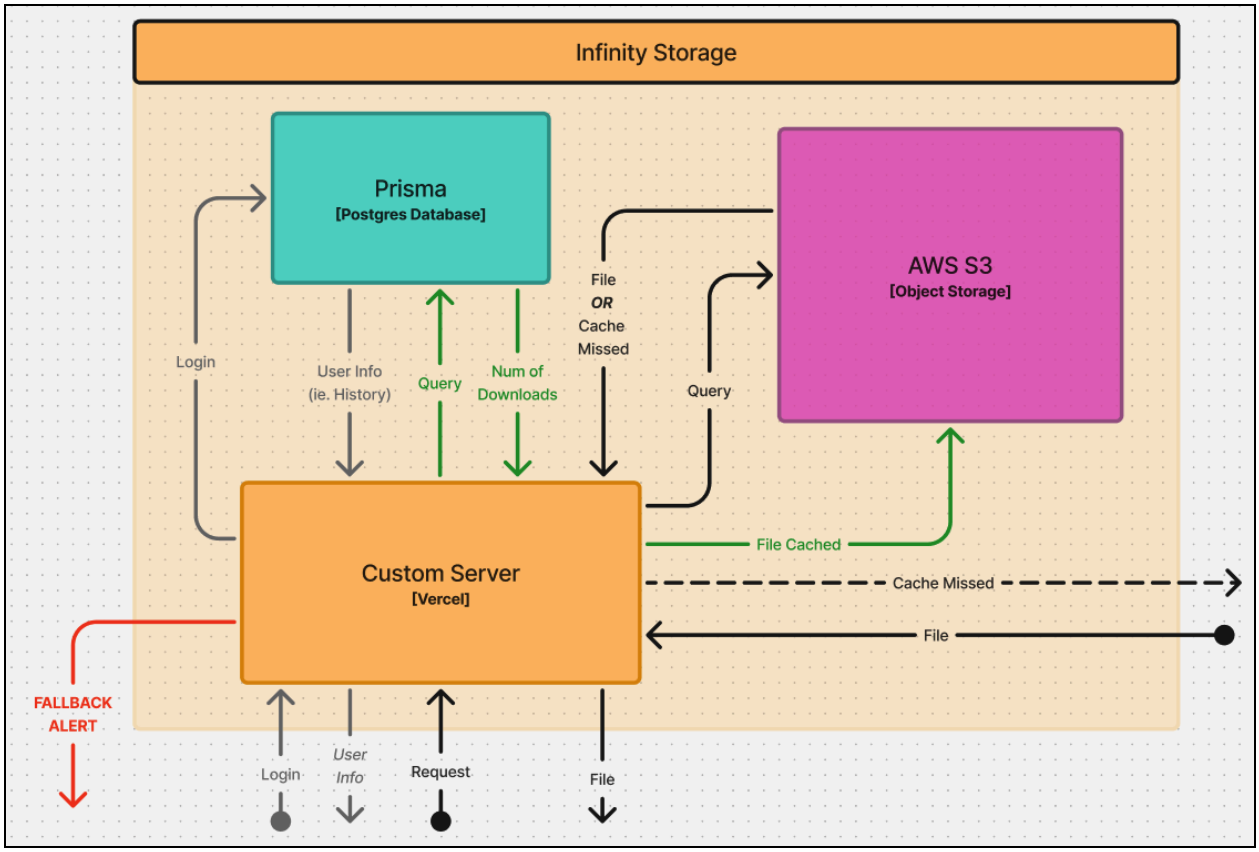
If there aren't enough funds, we forward a response to the user stating there's an insufficient amount in their account, and from there, the user can redirect themselves to the balance page to add more funds.

If there are sufficient funds, we want to then ensure that this isn't a duplicate upload, so our custom server will query our PostgreSQL database (hosted on Prisma) to search for the hashed string, generated by hashing the file's contents (separate from walrus's Blob ID as that's generated after a transaction).



If there is no history of the file, we will proceed with uploading to the Walrus network, and then forward the outputted Blob ID to the user. If there is a history of the file, the query will output the corresponding blobID that was stored within the database's tables, and that will be returned to the user. All of this is done to (1) save on time uploading, as uploading to the Walrus network can be slow, and (2) to save resources, such as funds, for the user.

2.3.2 Handling Downloads



Similarly to Section 2.3.1, we assume the user has provided their credentials and logged in – in the case of downloads, this is important in order to populate their history tab, as users can also download from that page. Additionally, any of the following procedures will only occur if a FALLBACK ALERT has not been forwarded to the client-side from our custom server – this will only occur if any of our centralized services/providers are down, and therefore we will proceed with the centralized fallback system (not yet implemented).

Once a download request has been provided, alongside the Blob ID that corresponds to the file being requested, we will first query/search our S3 object storage (hosted on AWS) to see if the file has been previously cached. For some context, a file will be cached to S3 if it's been deemed a popular file, which (at the moment) is when a file has been downloaded more than two times. From this, they will have a limited lifetime on our S3 service, and this allows for faster downloads than downloading directly from the Walrus network – and once that lifetime runs out, the file will be removed from cache (thus having to be deemed 'popular' once again). All of this involves the green arrows above, which involve querying our Postgres database for how many times it had been downloaded (plus the current request for download), and if the number of recent downloads is greater than 2, we cache the file/reset the lifetime.

With that in mind, if the file had been previously cached, our custom server will receive the file from S3, and that will be returned to the user. Otherwise, we will get a cache-missed alert, and we will have to proceed to downloading directly from the Walrus network, at a cost of taking more time. Once the file has been fetched, it will be sent to the user and downloaded automatically.

Requirements Specification

3.1 Functional Requirements

- The system should encrypt files and decrypt them after download using a symmetric encryption scheme (e.g. AES-GCM).
- The system should handle lazy upload so the user perceives the upload as happening much faster than it actually is.
- The system must ensure that user files and wallet identifiers remain private and inaccessible to developers or third parties.
- The user should be able to upload their data as a blob and retrieve it with a linked decryption key.
- The user should be able to access their data even if the server fails as a property of decentralized storage systems.

3.2 Non-Functional Requirements

- The user should experience a smooth and responsive file upload experience, ensuring that users perceive upload times as comparable to standard cloud storage services (ie.y Google Cloud).
- Users should only be able to access data blobs if they have a linked decryption key to maintain privacy.
- Users should be able to upload large file sizes up to 5GB without experiencing large lag times or errors.
- Encrypted blobs should maintain 100% bit integrity(won't be corrupted).
- There shouldn't be any logging of user data or metadata beyond operational necessity.

3.3 User Stories

- As a user, I want to download my files securely even if the centralized system fails, so that my data remains accessible under all circumstances.
- As a user, I want to view my SUI and WAL balances so that I can confirm I have enough tokens to store new files.
- As a user, I want to view, manage, and delete my uploaded files from a single interface so that I can easily organize and control my stored data.
- As a user, I want my files to persist on Walrus so that I can still access them if the centralized cache or provider shuts down.
- As a user, I want to be able to upload multiple files at once, so I don't have to waste time uploading files one at a time.
- As a user, I want to be able to keep my encryption keys in a safe and accessible location so that they don't get lost and I can easily use them to access data.
- As a user, I want to verify that my stored data has not been corrupted, so that I can trust the integrity and safety of my files.

- As a user, I want to access my data at any time, regardless of the state of the service provider, so that I retain full control over my information.
- As a user, I want to see clear, upfront costs for uploading and storing data, so that I can make informed decisions about how I use storage resources.
- As a user, I want to pay for storage using traditional payment methods (e.g., credit card) so that I don't need to manage cryptocurrency wallets or additional blockchain tools.
- As a user, I want to be able to see my history of uploaded files, so I can have quick access to files that I've uploaded before.
- As a user, I want to be able to share my files so that another user is able to access them with my permission.
- As a user, I want to be able to download my files through a simple command or interface so I don't have to understand the underlying architecture of the app.
- As a user, I want to be able to automatically renew my file upload limit, so I don't have to worry about data being lost past a certain time.
- As a user, I want to be able to view the status of my files, their time periods, and storage usage so that I can identify any issues beforehand.
- As a user, I want to be able to view my files from multiple devices, so that I don't have to rely on a single device to manage my files.
- As a user, I want to be able to organize my files into folders or a directory structure, so that I can easily keep track of my files.

Appendix

4.1 Technologies

- Programming languages, frameworks, libraries
  - **Languages**: TypeScript, JavaScript, Python
  - **Frameworks**: React, Next.js, Vite
  - **Libraries**: Tailwind CSS, Axios, Crypto libraries
- APIs, SDKs, databases, and cloud services
  - **APIs:** Walrus API
  - **SDK:** Vercel SDK
- Version control, deployment, and CI/CD tools
  - **Version Control:** Git
  - **Deployment:** Vercel, Netlify
  - **CI/CD:** GitHub Actions
- Testing tools and methodologies
  - **Testing tools:** Jest

Glossary

- **AES-GCM:** Advanced Encryption Standard with Galois/Counter Mode. A symmetric encryption algorithm provides both confidentiality and data integrity.
- **Backend:** The server-side components responsible for logic, data management, and communication with decentralized networks.
- **Blob ID:** A unique identifier or hash assigned to each uploaded file to verify its authenticity and integrity.
- **Caching Layer:** A temporary storage component that stores frequently accessed data or recently uploaded/downloaded files to reduce latency and improve system responsiveness. Examples include in-memory caches such as Redis or local browser caches.
- **CLI:** Abbreviation for Command Line Interface
- **Decentralized Storage:** A storage model in which files are distributed across multiple nodes, eliminating single points of failure and enhancing data sovereignty.
- **Encryption:** The process of converting data into a coded format that can only be read or decrypted by authorized parties.
- **Frontend:** What our end-users interact with, and houses all the client-side operations.
- **Lazy Upload:** A technique where files are uploaded asynchronously, allowing users to continue interacting with the system before the upload fully completes.
- **Sui:** A Layer-1 blockchain designed for fast, secure, and scalable transactions, used in this system for managing wallet balances and transaction validation.
- **S3:** Amazon S3 (Simple Storage Service) is an object storage service offered by Amazon Web Services (AWS). It provides a highly scalable, durable, and secure solution for storing and retrieving any amount of data from anywhere on the web.
- **Symmetric Encryption:** A method of encryption that uses the same key for both encryption and decryption operations.
- **Wallet:** A digital account or cryptographic key pair used to manage tokens or digital assets on the Sui blockchain.
- **Walrus:** A decentralized data storage framework built by Mysten Labs that provides secure and distributed file persistence.