TheDrive: Secure Cloud Storage with AI Processing

Overview

TheDrive is a privacy-first cloud storage platform that combines end-to-end encryption with AI-powered document analysis. The system consists of three main components: a client application that handles all encryption and key management, a Django backend that provides zero-knowledge storage and metadata management, and a FastAPI AI node that processes user-authorized content for search and chat capabilities. Users maintain complete control over their data through client-side encryption and can selectively enable AI features while preserving privacy through signed request authentication and session-based processing.

Core Architecture Principles

- **Zero-Knowledge Storage**: The backend never sees plaintext files or encryption keys. All files are encrypted client-side with unique keys before upload, and only encrypted metadata is stored in the database.
- **Self-Sovereign Identity**: Users are identified by ED25519 public keys derived from BIP39 seed phrases using HKDF. Authentication uses signed challenges, and private keys never leave the client device.
- **Per-File Encryption**: Each file is encrypted with a randomly generated AES-256-GCM key, which is then wrapped (encrypted) with the user's master key derived from their seed phrase.
- **Separated Backend Architecture**: The Django backend handles storage, metadata, and user management, while the FastAPI AI node operates independently and authenticates to the backend using its own ED25519 keypair.
- **User-Controlled AI Access**: Users explicitly enable AI processing for specific files or folders. The client decrypts files locally before sending plaintext to the chosen AI node for processing.
- **Session-Based AI Processing**: AI nodes create temporary ChromaDB collections for chat sessions and purge all data when sessions are closed. Processed chunks are encrypted with the AI node's master key before storage.
- **Signed Request Authentication**: All sensitive operations between client and AI node use ED25519 signatures to verify user authorization and prevent unauthorized access.
- **Dual Database Architecture**: The backend uses PostgreSQL for metadata and encrypted chunks, while the AI node uses ChromaDB for temporary session-based vector storage and retrieval.

Technology Stack

Frontend Applications

- **React**: Core UI framework for web application
- **Ionic Framework**: Cross-platform mobile and web UI components
- **Capacitor**: Native mobile app deployment and device API access
- **TypeScript**: Type-safe JavaScript development
- **Vite**: Fast build tool and development server

Backend Service (Django)

- **Django REST Framework**: API development and serialization

- **PostgreSQL**: Primary database for metadata, user data, and encrypted chunks
- **MinIO**: S3-compatible object storage for encrypted files
- **PyNaCl**: ED25519 signature verification and cryptographic operations
- **cryptography**: AES-GCM encryption and key management
- **JWT**: Token-based authentication for API access

AI Node Service (FastAPI)

- **FastAPI**: High-performance API framework for ingestion and chat endpoints
- **ChromaDB**: Vector database for session-based similarity search
- **Sentence Transformers**: Text embedding generation (all-MiniLM-L6-v2 model)
- **Google Generative AI**: Gemini LLM integration for chat responses
- **PyNaCl**: ED25519 signature verification for request authentication
- **cryptography**: AES-GCM encryption for chunk and embedding storage
- **Transformers**: BLIP image captioning and Table Transformer for document processing
- **Tesseract OCR**: Text extraction from images and scanned documents

Client-Side Cryptography

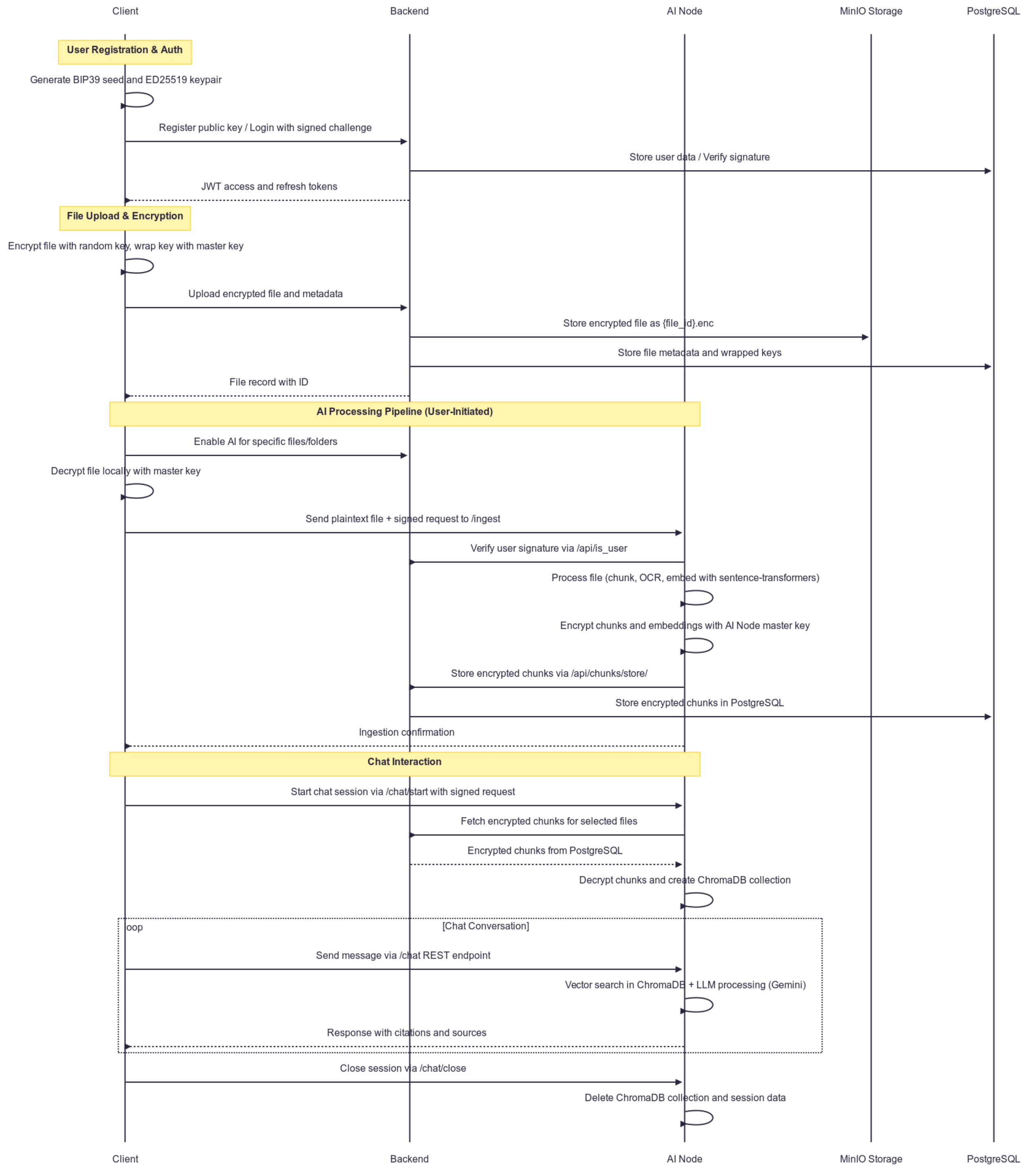
- **Web Crypto API**: AES-256-GCM encryption and decryption operations
- **@scure/bip39**: BIP39 seed phrase generation and validation
- **@noble/ed25519**: ED25519 key pair generation and signing
- **@noble/hashes**: HKDF key derivation from seed phrases

Development and Deployment

- **Docker**: Containerization for backend and AI node services
- **Docker Compose**: Multi-service orchestration for development
- **TLS/HTTPS**: Secure transport layer for production deployments

System Architecture

TheDrive follows a three-tier architecture that separates concerns between client-side encryption, zero-knowledge storage, and AI processing. The system ensures that sensitive operations are performed on trusted clients while maintaining privacy through cryptographic verification.

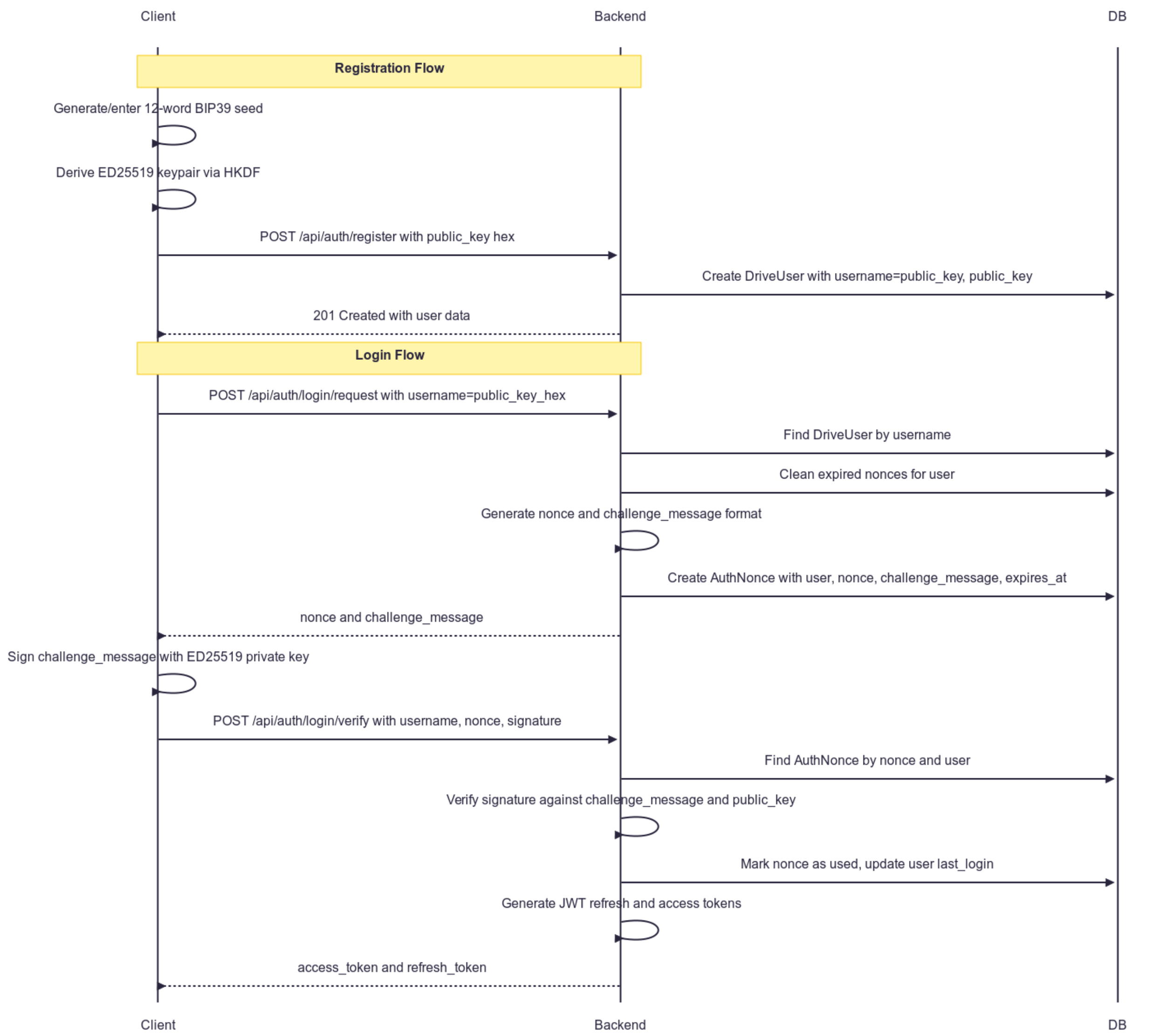


The architecture consists of:

- **Client Layer**: Handles all encryption, key management, and user interactions through React/Ionic applications
- **Backend Layer**: Provides zero-knowledge storage and metadata management via Django REST API with PostgreSQL and MinIO
- **AI Processing Layer**: Offers document analysis and chat capabilities through an independent FastAPI service with ChromaDB

User Authentication and Registration

User identity in TheDrive is based on self-sovereign cryptographic keys derived from BIP39 seed phrases. This approach eliminates the need for traditional username/password combinations while providing secure recovery mechanisms.



Registration Process:

1. Client generates a 12-word BIP39 seed phrase using cryptographically secure randomness
2. The seed phrase is used to derive an ED25519 keypair via HKDF with the salt "auth-ed25519"
3. The public key serves as the user's unique identifier on the platform
4. A master encryption key is derived from the same seed using HKDF with salt "master-encryption"
5. The client registers the public key with the backend, establishing the user account

Login Process:

1. User enters their 12-word seed phrase on any device
2. Client reconstructs the ED25519 keypair and master key from the seed
3. Client requests a challenge nonce from the backend using the public key
4. Client signs the challenge with the private key and sends the signature to the backend
5. Backend verifies the signature against the stored public key
6. Upon successful verification, backend issues JWT access and refresh tokens

7. Client stores tokens securely for subsequent authenticated API requests

Security Features:

- Private keys never leave the client device or are transmitted over the network
- Seed phrases provide portable identity that works across devices
- Challenge-response authentication prevents replay attacks
- JWT tokens have limited lifetimes and can be refreshed securely

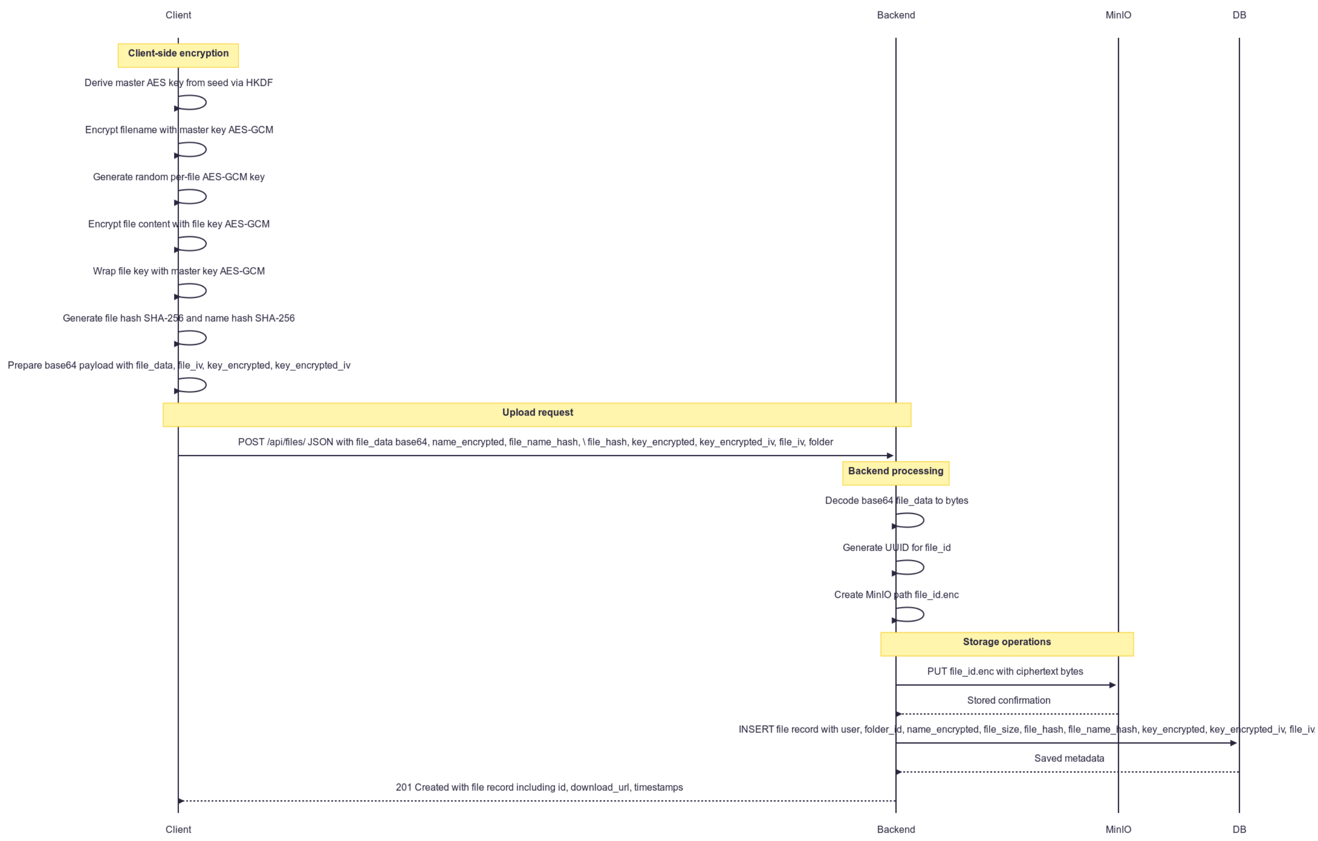
Key Derivation Hierarchy:

```
Seed Phrase (12 words)
├── Authentication Key: HKDF(seed, "auth-ed25519") → ED25519 keypair
└── Drive Master Key: HKDF(seed, "master-encryption") → AES-256-GCM key
    └── File Keys: Random AES-256-GCM keys wrapped by master key
```

Note on Implementation: The current system uses per-file encryption rather than hierarchical folder keys. Each file gets a unique random encryption key that is wrapped (encrypted) with the user's master key. Folder structure is maintained logically in metadata rather than cryptographically.

Storage Architecture

Each file uploaded to TheDrive is encrypted with a unique AES-256-GCM key before transmission to the backend. This approach ensures that even if one file's encryption is compromised, other files remain secure.



File Upload Process

Client-Side Encryption:

1. User selects files for upload through the React/Ionic interface
2. For each file, a random AES-256-GCM key is generated using Web Crypto API
3. The file content is encrypted with this unique key
4. The file key is wrapped (encrypted) with the user's master key derived from their seed phrase
5. The plaintext filename is encrypted with the master key to create `name_encrypted`
6. Encrypted file content, wrapped key, and encrypted metadata are packaged for upload

Backend Processing:

1. Client uploads the encrypted file payload via Django REST API
2. Backend stores the encrypted file in MinIO object storage with the key format `{file_id}.enc`
3. File metadata (including wrapped keys and encrypted filenames) is stored in PostgreSQL
4. No plaintext content or filenames are ever stored on the server
5. Backend returns a file record with unique ID to the client

Encryption Model

File-Level Security:

- Each file uses a unique AES-256-GCM encryption key
- File keys are wrapped with the user's master key (never stored in plaintext)
- Filenames are encrypted to prevent metadata leakage
- Initialization vectors (IVs) are generated per encryption operation

Key Management:

- Master keys are derived from BIP39 seed phrases using HKDF
- File keys are never transmitted in plaintext
- Key wrapping ensures only the user can decrypt their files
- No hierarchical folder encryption - folder structure is logical metadata only

Data Storage Components

MinIO Object Storage:

- Stores encrypted file objects using S3-compatible API
- Object keys follow the pattern `{file_id}.enc` with no plaintext information
- Supports horizontal scaling and redundancy
- Provides durability and availability for encrypted file content

PostgreSQL Database:

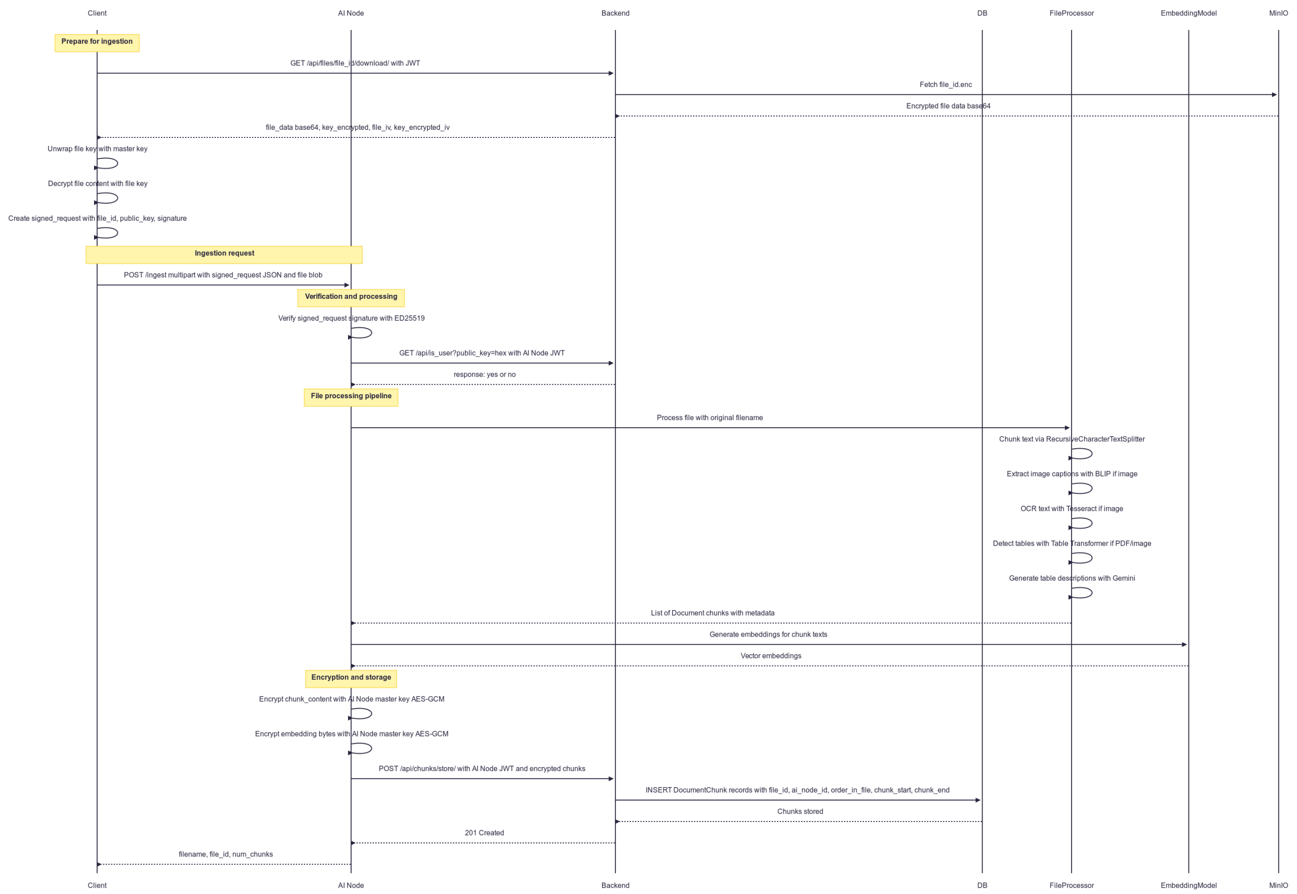
- Stores file and folder metadata with encrypted names
- Contains wrapped encryption keys and initialization vectors
- Manages user accounts and authentication data
- Stores encrypted document chunks for AI-enabled files
- Maintains folder hierarchy as logical relationships, not cryptographic structure

Security Features:

- Zero-knowledge storage: server cannot decrypt any file content
- Encrypted filenames prevent metadata analysis
- Per-file encryption limits blast radius of potential compromise
- Key wrapping ensures master key compromise doesn't expose file keys directly

AI Node Architecture and Ingestion Pipeline

The AI Node operates as an independent FastAPI service that processes user-authorized documents for search and chat capabilities. It maintains its own cryptographic identity and encrypts all processed data before storage.



AI Node Identity and Authentication

AI Node Setup:

1. AI Node generates its own ED25519 keypair and master encryption key using the `ai_node/script.py`
2. The AI Node registers with the backend using its public key as identifier
3. AI Node authenticates to the backend using signed challenge-response, similar to user authentication
4. Backend creates an AINode registry entry linking the node to its cryptographic identity

Request Verification:

- All ingestion requests from clients must be signed with the user's ED25519 private key
- AI Node verifies user signatures by calling the backend `/api/is_user` endpoint
- Only verified users can submit files for processing
- AI Node maintains no persistent user sessions during ingestion

File Ingestion Process

User-Initiated AI Enablement:

1. User explicitly enables AI processing for specific files or folders through the client interface
2. Client downloads the encrypted file from backend storage
3. Client decrypts the file locally using their master key and wrapped file key
4. Client sends the plaintext file content to AI Node `/ingest` endpoint with signed request

AI Node Processing:

1. AI Node verifies the user's signature and authorization via backend API
2. Document content is processed through multiple extraction pipelines:
    - **Text Extraction**: Direct text content and OCR via Tesseract for images
    - **Image Processing**: BLIP model generates captions for embedded images
    - **Table Processing**: Table Transformer detects and extracts table structures
3. Content is chunked into segments (typically 512-1024 tokens each)
4. Text embeddings are generated using Sentence Transformers (all-MiniLM-L6-v2 model)
5. Both chunk text and embeddings are encrypted with the AI Node's master key using AES-GCM
6. Encrypted chunks are sent to backend via `/api/chunks/store/` endpoint
7. Backend stores encrypted chunks in PostgreSQL with associated file metadata

Security During Processing:

- Plaintext content exists only temporarily in AI Node memory during processing
- All stored artifacts (chunks, embeddings) are encrypted with AI Node's master key
- No plaintext content is persisted to disk on the AI Node
- Processing is stateless - no session data retained after ingestion completes

AI Node Data Model

Encryption Architecture:

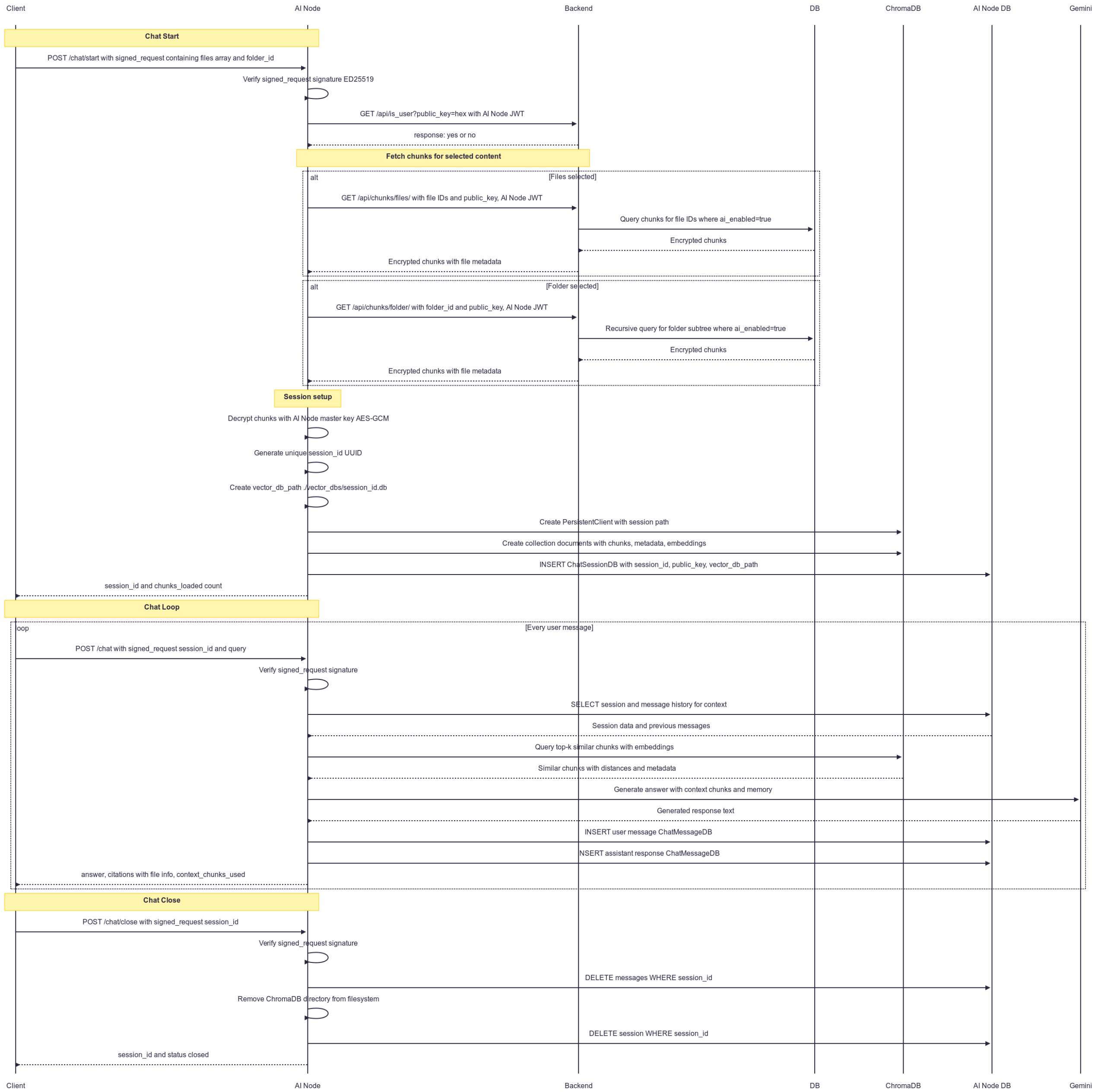
- AI Node maintains its own master key separate from user keys
- Each user's processed data is encrypted with the same AI Node master key
- This allows the AI Node to decrypt and use the data for search/chat while keeping it encrypted at rest
- Users can revoke AI access by disabling AI features, which prevents future access to their content

Storage Integration:

- AI Node has no direct database - all persistent storage goes through the backend
- Encrypted chunks stored in PostgreSQL maintain association with original files
- AI Node can retrieve and decrypt chunks for authorized users during chat sessions
- No user data persists on AI Node between sessions

Chat Session Architecture

The chat functionality provides real-time question-answering capabilities over user-selected documents through session-based vector search and LLM processing.



Session Lifecycle Management

Session Initialization:

1. User initiates a chat session via AI Node `/chat/start` endpoint with signed request
2. Request includes selected files/folders and user's ED25519 signature for authorization
3. AI Node verifies user signature and fetches encrypted chunks from backend for selected content
4. AI Node decrypts chunks using its master key and creates a temporary ChromaDB collection
5. Decrypted chunks and embeddings are loaded into the session-specific ChromaDB collection
6. AI Node returns session ID to client for subsequent chat interactions

Chat Interaction Flow:

1. Client sends questions to AI Node `/chat` endpoint with session ID and signed request
2. AI Node performs vector similarity search in the session's ChromaDB collection
3. Most relevant chunks are retrieved based on semantic similarity to the user's query
4. Retrieved context is combined with the user's question and sent to Gemini LLM

5. LLM generates a response with citations to the source documents
6. AI Node returns the response with document references and chunk citations to the client

Session Termination:

1. User explicitly closes session via `/chat/close` endpoint or session times out
2. AI Node deletes the ChromaDB collection and all session-specific data
3. No trace of user data remains on the AI Node after session closure
4. Session metadata is purged from AI Node memory

Vector Search and Retrieval

ChromaDB Integration:

- Each chat session gets a unique ChromaDB collection created on the AI Node
- Collections contain decrypted text chunks and their corresponding embeddings
- ChromaDB uses its default distance metric (typically squared L2/Euclidean distance)
- Distance scores are converted to similarity scores using the formula: `similarity = 1 / (1 + distance)`
- Search results include both converted similarity scores and source document metadata

Context Assembly:

- Top-K most relevant chunks are selected based on ChromaDB's distance calculation
- Context window management ensures LLM token limits are respected
- Source attribution maintains links between retrieved chunks and original files
- Duplicate or highly similar chunks are filtered to optimize context quality

LLM Processing and Response Generation

Gemini Integration:

- Retrieved document chunks provide context for the user's question
- System prompts guide the LLM to cite sources and maintain accuracy
- Response generation includes confidence indicators and source references
- LLM responses are streamed back to the client for real-time interaction

Security During Chat:

- All chat messages and responses flow through REST endpoints (no WebSocket in current implementation)
- Session data exists only in AI Node memory and temporary ChromaDB collections
- No chat history is persisted beyond the active session
- User authentication is verified on each chat request through signed messages

Session Isolation and Privacy

Data Isolation:

- Each session operates with isolated ChromaDB collections
- Sessions cannot access data from other users or other sessions

- Vector search is limited to documents explicitly selected by the user for that session
- No cross-session data leakage or contamination

Privacy Guarantees:

- Session data is ephemeral and deleted immediately upon session closure
- AI Node retains no memory of conversations after sessions end
- Document access is limited to user-authorized content only
- No training or learning occurs from user conversations

Setup Instructions

Setup Instructions

Follow these steps to set up TheDrive for development or production on Windows, macOS, or Linux.

Prerequisites

- **Docker** and **Docker Compose** installed (Download Docker Desktop)
- **Git** installed (Download Git)
- (Optional) **Python 3.10+** and **Node.js 18+** if you want to run backend/frontend locally without Docker

1. Clone the Repository

```
git clone https://github.com/Parth-2412/TheDrive.git
cd TheDrive
```

2. Environment Configuration

- Copy example environment files (if provided) or create your own `.env` files for backend and AI node as needed.
- Set required secrets (e.g., Django secret key, database credentials, MinIO keys, Gemini API key) in the respective `.env` files.

3. Running with Docker (Recommended)

Windows, macOS, Linux (Docker Compose)

```
docker-compose up --build
```

This will start all services: backend (Django), AI node (FastAPI), PostgreSQL, MinIO, and frontend.

Access the services at:

- Frontend: http://localhost:5173

- Backend API: http://localhost:8000/api/
- AI Node: http://localhost:9000
- MinIO Console: http://localhost:9001

4. Manual Local Development (Optional)

If you want to run services individually (for debugging or development):

Backend (Django)

```
cd backend
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
pip install -r requirements.txt
python manage.py migrate
python manage.py runserver
```

AI Node (FastAPI)

```
cd ai_node
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
pip install -r requirements.txt
python api.py
```

Frontend (React/Ionic)

```
cd frontend
npm install
npm run dev
```

5. Production Deployment

- Set strong secrets and production environment variables in `.env` files.
- Use Docker Compose or orchestrate containers with Kubernetes for scaling.
- Set up HTTPS (TLS) termination (e.g., with Nginx, Caddy, or a cloud load balancer).
- Configure persistent storage for PostgreSQL and MinIO volumes.
- Monitor logs and health endpoints for all services.

6. Additional Notes

- For Windows users, run all commands in PowerShell or Git Bash. For macOS/Linux, use Terminal.
- If you encounter port conflicts, adjust the ports in `docker-compose.yml`.

- For GPU acceleration (optional, for AI node), use a CUDA-enabled Docker image and compatible hardware.
- See `backend_testing.md` for API testing instructions.

For troubleshooting or advanced configuration, refer to the README or open an issue on GitHub.