**ChainNotary**

**Project Architecture**

**📋 Architecture Overview**

ChainNotary is a **decentralized document verification and AI analytics platform** built on the **Internet Computer Protocol (ICP)**. It follows a **modern full-stack architecture** with a React frontend and a Rust-based smart contract backend.

**System Architecture Layers**

**1. Frontend Layer (React + TypeScript)**

**Technology Stack:**

* **Framework**: React 19.1.0 with TypeScript
* **Build Tool**: Vite 5.4.11
* **UI Framework**: Ant Design 5.26.7
* **Styling**: Tailwind CSS 4.1.11
* **Icons**: Lucide React
* **Routing**: React Router 7.7.1
* **ICP Integration**: @dfinity/\* packages

**Architecture Pattern:**

text

frontend/src/

├ pages/ # Route-based page components

├components/ # Reusable UI components

├ services/ # API communication layer

├ utils/ # Helper functions

├ hooks/ # Custom React hooks

└── router.ts # Application routing

**Key Components:**

* **Pages**: Home, CreateDocument, DocumentDetails, DocumentAnalytics, QueryDocument
* **Services**: Document operations, Analytics API calls
* **Utils**: File processing, hash calculation, form validation

**2. ICP Blockchain Layer**

**Configuration (dfx.json):**

json

{

"canisters": {

"backend": {

"type": "custom",

"wasm": "target/wasm32-unknown-unknown/release/backend.wasm",

"candid": "backend/backend.did"

},

"frontend": {

"type": "assets",

"dependencies": ["backend"]

}

}

}

**Key Features:**

* **Local Development**: Ephemeral network on 127.0.0.1:8080
* **WASM Compilation**: Rust → WebAssembly for blockchain execution
* **Candid Interface**: Type-safe communication between frontend/backend
* **Asset Hosting**: Frontend served as static assets on ICP

**3. Backend Canister (Rust + WASM)**

**Technology Stack:**

* **Language**: Rust 2021 Edition
* **Target**: wasm32-unknown-unknown (WebAssembly)
* **Framework**: IC-CDK 0.18.3
* **Storage**: IC-Stable-Structures 0.6.7
* **Serialization**: Serde + Candid
* **PDF Processing**: lopdf 0.32
* **Cryptography**: sha2, hex

**Module Architecture:**

rust

backend/src/

├ lib.rs # Main entry point & exports

├ functions/ # Business logic modules

│ ├ document.rs # Document operations

│ ├ analytics.rs # AI analytics

│ ├ collection.rs # Document collections

│ ├ institution.rs # Institution management

│ └\*\_queries.rs # Query functions

├ types/ # Data models & types

│ └── models.rs # Core data structures

├ storage/ # Data persistence

│ └── memory.rs # IC memory management

└── utils/ # Helper functions

└── helpers.rs # Utility functions

**Core Data Models:**

rust

*// Primary entities*

struct Document {

document\_id: String,

owner: Principal,

file\_data: Vec<u8>, *// PDF binary data*

document\_data: DocumentType,

*// ... metadata fields*

}

struct Institution {

institution\_id: String,

name: String,

collections: Vec<String>,

}

struct CollectionMetadata {

collection\_id: String,

documents: Vec<String>,

category: CollectionCategory,

}

**4. AI Integration Layer**

**Analytics Engine (analytics.rs):**

rust

*// AI-powered document analysis*

#[update]

pub async fn analyze\_document\_data(

request: AnalyticsRequest

) -> AnalyticsResponse {

*// 1. Extract PDF content*

*// 2. Create analysis prompt*

*// 3. Call Gemini API*

*// 4. Parse and return results*

}

**Features:**

* **PDF Text Extraction**: lopdf library processes binary data
* **AI API Integration**: HTTP outcalls to Gemini 2.5 Pro
* **Multi-Analysis Types**: Financial summary, investment insights, charts
* **Smart Prompting**: Context-aware AI instructions

**5. Data Storage Architecture**

**IC Stable Structures:**

rust

*// Persistent storage on ICP*

thread\_local! {

static DOCUMENTS: RefCell<StableBTreeMap<String, Vec<u8>, Memory>> =

RefCell::new(StableBTreeMap::init(MEMORY\_MANAGER.with(|m| m.borrow().get(MemoryId::new(0)))));

static COLLECTIONS: RefCell<StableBTreeMap<String, Vec<u8>, Memory>> =

RefCell::new(StableBTreeMap::init(MEMORY\_MANAGER.with(|m| m.borrow().get(MemoryId::new(1)))));

}

**Storage Strategy:**

* **Stable Memory**: Survives canister upgrades
* **BTreeMap**: Efficient key-value storage
* **Serialization**: Candid + Serde for type safety
* **Memory Management**: Separate memory regions per data type

**Data Flow Architecture**

**Document Upload Flow:**

text

1. User selects PDF → Frontend

2. Convert to Uint8Array → getUint8Array()

3. Calculate hash → computeFileHash()

4. Form submission → createDocumentService()

5. Canister call → upload\_file\_and\_publish\_document()

6. Store in IC memory → DOCUMENTS.insert()

7. Return document ID → Frontend redirect

**AI Analytics Flow:**

text

1. User clicks "AI Analytics" → DocumentDetails

2. Navigate to analytics page → DocumentAnalytics

3. Select analysis type → Dropdown

4. Backend call → analyze\_document\_data()

5. Extract PDF text → lopdf processing

6. HTTP outcall → Gemini API

7. Parse response → JSON extraction

8. Return analysis → Frontend display

**Security Architecture**

**Authentication & Authorization:**

* **Principal-based**: ICP identity system
* **Owner validation**: Document access control
* **Canister security**: Update/query function permissions

**Data Integrity:**

* **Hash verification**: SHA-256 for file integrity
* **Immutable storage**: Blockchain-based persistence
* **Type safety**: Rust + Candid type system

**External API Security:**

* **HTTPS only**: Secure HTTP outcalls
* **API key management**: (TODO: Secure in production)
* **Response validation**: JSON parsing with error handling

**Deployment Architecture**

**Development Environment:**

bash

*# Local ICP network*

dfx start --clean --host 127.0.0.1:8080

*# Build & deploy*

dfx deploy --network local

**Production Considerations:**

* **Mainnet deployment**: IC mainnet canisters
* **Cycles management**: Computation resource allocation
* **Canister upgrades**: Stable memory preservation
* **Asset optimization**: Gzip + shrink enabled

**📊 Performance Characteristics**

**Scalability:**

* **Horizontal scaling**: Multiple canister instances
* **Memory efficiency**: Stable structures optimization
* **Query performance**: BTreeMap O(log n) operations

**Reliability:**

* **Fault tolerance**: ICP consensus mechanism
* **Data persistence**: Stable memory across upgrades
* **Error handling**: Comprehensive Result<T, E> patterns

**Development Workflow**

**Build Process:**

1. **Rust compilation**: cargo build --target wasm32-unknown-unknown --release
2. **Candid generation**: candid-extractor for type definitions
3. **Frontend build**: vite build for optimized assets
4. **Deployment**: dfx deploy to ICP network

**Key Development Features:**

* **Hot reload**: Vite dev server for frontend
* **Type safety**: End-to-end TypeScript + Rust + Candid
* **Testing**: Comprehensive test suites (TESTING.md)
* **Documentation**: Detailed API documentation

**Architectural Strengths**

**✅ Decentralization:**

* No single point of failure
* Censorship-resistant document storage
* Transparent and verifiable operations

**✅ Type Safety:**

* Rust's memory safety
* TypeScript frontend validation
* Candid interface contracts

**✅ Modern Stack:**

* Latest React patterns
* Efficient Rust performance
* WebAssembly compilation

**✅ AI Integration:**

* External API capabilities
* Consensus-based HTTP outcalls
* Structured AI responses

This architecture provides a solid foundation for a decentralized document verification platform with advanced AI capabilities, combining the security of blockchain with the power of modern web technologies and artificial intelligence.