Technical Implementation Manual: AI-Driven Real World Asset Tokenization Platform

Abstract

This document details the technical implementation of an AI-driven platform designed for the streamlined verification and tokenization of Real World Assets (RWAs). The platform leverages a sophisticated agentic AI framework, integrating Large Language Models (LLMs) for initial data extraction, specialized verification agents for comprehensive asset assessment, and a tokenization module for blockchain integration. This manual provides an in-depth overview of the system architecture, database schema, and the detailed functionality of each Python module, including comprehensive code explanations. The objective is to present a robust, scalable, and academically rigorous solution for enhancing liquidity and accessibility in the RWA market.

1. Introduction

The tokenization of Real World Assets (RWAs) represents a transformative paradigm in finance, bridging traditional illiquid assets with the efficiency and transparency of blockchain technology. This process involves converting tangible or intangible assets into digital tokens on a blockchain, enabling fractional ownership, increased liquidity, and broader market accessibility. However, the successful tokenization of RWAs is contingent upon rigorous verification of asset authenticity, value, and legal standing.

1.1 Problem Statement

Traditional RWA verification processes are often manual, time-consuming, opaque, and prone to human error. This labor-intensive approach creates significant bottlenecks, increases operational costs, and hinders the scalability of RWA tokenization initiatives. Key challenges include:

- Data Extraction and Standardization: Diverse and unstructured asset information requires significant manual effort to parse and standardize.
- Verification Complexity: Assessing the authenticity, value, and legal compliance of varied asset types demands specialized expertise and robust methodologies.
- Interoperability: Seamless integration between traditional asset data, verification outcomes, and blockchain tokenization layers is often lacking.
- Scalability: Manual processes limit the volume of assets that can be efficiently onboarded and tokenized.

1.2 Solution Overview

Our proposed solution is an AI-driven platform that automates and enhances the RWA verification and tokenization pipeline. By employing an agentic AI framework, the platform intelligently processes asset data, performs multi-faceted verification, and facilitates the creation of blockchain-based tokens. The core components include:

- Large Language Model (LLM) Integration: For initial, intelligent extraction of structured data from unstructured user inputs.
- Modular Verification Agents: A system of specialized agents (or verification functions) that independently assess different aspects of an asset (e.g., value, jurisdiction, description quality).
- Tokenization Agent: Responsible for generating mock blockchain token metadata, smart contract details, and transaction hashes.
- Centralized Database: To store and manage user, asset, and transaction data, ensuring persistence and traceability.
- Flask Backend API: To provide a robust interface for user interaction and orchestration of the verification and tokenization workflows.

This integrated approach aims to significantly reduce manual overhead, improve verification accuracy, and accelerate the RWA tokenization process, thereby contributing to a more liquid and accessible digital asset ecosystem.

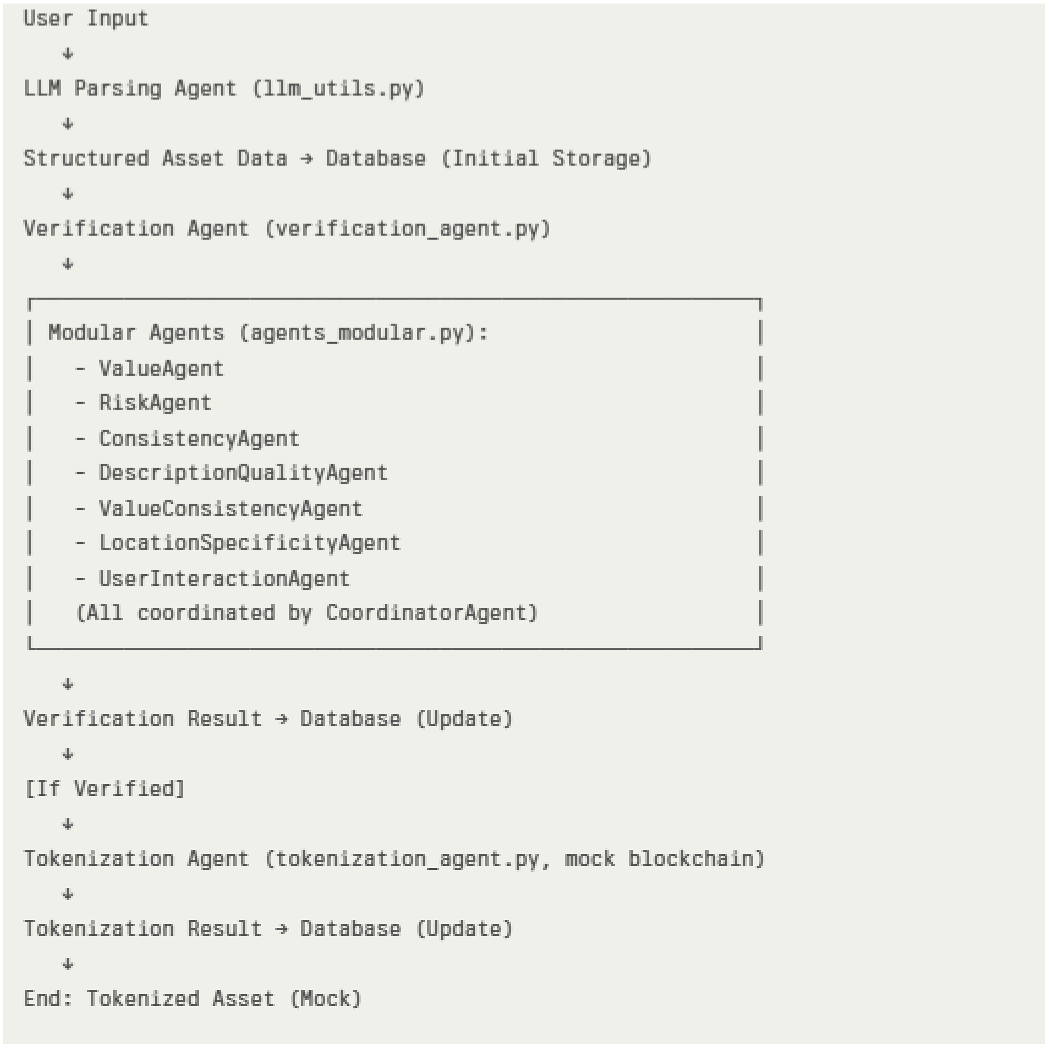
2. System Architecture

The RWA Tokenization Platform is designed with a modular and scalable architecture, facilitating efficient processing and clear separation of concerns.

File structure

```
/project-root
|
├── app/
|   ├── agents/
|   |   ├── llm_utils.py
|   |   ├── verification_agent.py
|   |   ├── agents_modular.py
|   |   └── tokenization_agent.py
|   ├── models/
|   |   ├── database.py
|   |   └── ...
|   └── ...
|
├── main.py
├── print_records.py
├── requirements.txt
├── .env
├── static/
|   ├── css/
|   └── js/
└── templates/
    └── index.html
```

2.1 Overall Architecture Diagram



```
User Input
   ↓
LLM Parsing Agent (llm_utils.py)
   ↓
Structured Asset Data → Database (Initial Storage)
   ↓
Verification Agent (verification_agent.py)
   ↓
┌──────────────────────────────────────────────────────────┐
| Modular Agents (agents_modular.py):                      |
|   - ValueAgent                                           |
|   - RiskAgent                                            |
|   - ConsistencyAgent                                     |
|   - DescriptionQualityAgent                              |
|   - ValueConsistencyAgent                                |
|   - LocationSpecificityAgent                             |
|   - UserInteractionAgent                                 |
|   (All coordinated by CoordinatorAgent)                  |
└──────────────────────────────────────────────────────────┘
   ↓
Verification Result → Database (Update)
   ↓
[If Verified]
   ↓
Tokenization Agent (tokenization_agent.py, mock blockchain)
   ↓
Tokenization Result → Database (Update)
   ↓
End: Tokenized Asset (Mock)
```

The platform's architecture can be conceptualized as follows:

1. User Interface (Frontend): Users interact with the system via a web-based interface (not explicitly provided in the backend code, but implied). This interface sends asset descriptions and user details to the Backend API.
2. Backend API (Flask Application): This is the core orchestration layer. It receives user requests, interacts with the database, and coordinates with various AI agents.
3. Database (SQLite/SQLAlchemy): Persists all platform data, including user profiles, asset details, and transaction records.
4. AI Agents Layer:
   - LLM Utility: Communicates with external Large Language Models (e.g., Gemini) to extract structured data from raw text inputs.
   - The agents_modular.py defines all the core modular agents that independently assess different aspects of an asset and are coordinated by the `CoordinatorAgent` to produce an explainable, robust verification workflow.Each

agent independently evaluates a specific aspect of the asset (value, risk, consistency, etc.).
    - The `CoordinatorAgent` orchestrates all agents, aggregates their scores, and determines the final asset status (`verified`, `requires_review`, or `rejected`).
    - The `UserInteractionAgent` suggests clarifying questions if information is missing or unclear, improving user experience and data quality.
    -
    - Verification Agent: A sophisticated module comprising multiple sub-agents or functions that perform detailed checks on asset data.
    - Tokenization Agent: Handles the mock blockchain operations, including metadata generation and token minting.
5. **Blockchain Layer (Mock):** For this proof-of-concept, blockchain interactions (smart contract deployment, token minting, transfers) are simulated by the Tokenization Agent. In a production environment, this would involve actual interaction with a blockchain network (e.g., Ethereum, Polygon).

Flow of Operations:

- **Asset Intake:** User provides asset description rightarrow Backend API rightarrow LLM Utility (extracts structured data) rightarrow Backend API rightarrow Database (stores asset).
- **Asset Verification:** Backend API rightarrow Verification Agent (assesses asset data) rightarrow Backend API rightarrow Database (updates verification status).
- **Asset Tokenization:** Backend API rightarrow Tokenization Agent (generates mock token and metadata) rightarrow Backend API rightarrow Database (stores token details).

Folder Structure Breakdown (/models, /agents, main.py, etc.)

The platform's logical folder structure is designed to promote modularity, maintainability, and clear separation of concerns, directly reflecting the agentic architecture. This organization makes it easier for developers to locate, modify, and test specific components without impacting others, facilitating team collaboration and improving scalability.

- **/app (or root directory):** This is the primary application directory.
    - main.py: The main entry point for the Flask application. It handles application initialization, registers blueprints, and sets up core services.
    - config.py: Contains application-wide configuration settings, such as database paths, API key placeholders, and other environment-dependent variables.

- routes.py: Defines the Flask routes (API endpoints) and their associated request handling logic, acting as the interface between the web and the agentic core.
- **/agents:** This directory is dedicated to the Python modules for each specialized AI agent and the Coordinator.
  - llm_parsing_agent.py: Contains the implementation for extracting structured data from unstructured asset descriptions.
  - value_agent.py: Implements the logic for asset valuation and appraisal.
  - risk_agent.py: Houses the logic for assessing financial, legal, and fraud risks.
  - consistency_agent.py: Contains logic for verifying data consistency and integrity.
  - location_agent.py: Implements checks for geographical and jurisdictional compliance.
  - tokenization_agent.py: Responsible for generating token metadata and IDs, and interacting with the blockchain for minting.
  - coordinator_agent.py: Contains the orchestration logic, aggregating results from other agents and making final tokenization decisions.
- **/models:** This directory stores data models, often defined using libraries like Pydantic, to enforce structured data formats for assets, tokens, and verification results, ensuring data integrity across modules.
- **/database:** Contains all database-related scripts and files.
  - rwa_tokenization.db: The SQLite database file itself, holding all persistent data.
  - schema.sql: An SQL script used to initialize the database schema, creating all necessary tables.
  - db_utils.py: Provides utility functions for managing database connections, executing queries, and performing standard CRUD (Create, Read, Update, Delete) operations.[15]
- **/utils:** A general-purpose directory for shared utility functions.
  - pdf_generator.py: Contains the logic for generating professional PDF reports using the ReportLab library.
  - data_loader.py: Manages intelligent data loading and caching mechanisms, optimizing performance by reducing redundant data fetches.[9]
  - error_handling.py: Implements custom exception classes and robust error recovery strategies for consistent error management across the platform.[9]
- **/tests:** Dedicated to unit, integration, and end-to-end tests for all components of the platform, ensuring reliability and correctness.[9]
- **/data:** Stores synthetic or sample data, such as synthetic_inventory.json, used for development, testing, and demonstrations.[9]

- **.env:** The environment variables file, crucial for secure configuration and keeping sensitive information out of version control.
- **requirements.txt:** Lists all Python package dependencies, enabling easy environment setup and reproducibility.

3. Setup & Prerequisites

This section provides practical, step-by-step instructions for setting up the platform's development and operational environment. It is primarily for technical users but maintains clarity for broader understanding.

3.1 Environment Setup (Flask, Python, SQLite)

To set up the Agentic AI Blockchain Asset Tokenization Platform, a robust and consistent environment is essential. The platform is built primarily with Python, leveraging the Flask web framework for its backend services and SQLite for local data persistence during development and prototyping.

1. **Python Environment:** The platform requires Python 3.8 or higher. It is strongly recommended to use a virtual environment (e.g., venv) to manage project-specific dependencies. This practice isolates the project's packages, preventing conflicts with other Python projects on the same machine.[9] The emphasis on a reproducible development environment is crucial for ensuring that developers can consistently set up and run the platform without encountering "it works on my machine" issues, which is a best practice for any serious software project.
2. **Flask Web Framework:** Flask is a lightweight and flexible Python web framework that forms the backbone of the platform's web application. It handles incoming requests, routes them to the appropriate logic, and serves responses.[14]
3. **SQLite Database:** SQLite 3 is utilized as a lightweight, disk-based database that operates without requiring a separate server process. It is ideal for development, testing, and prototyping phases, with all data stored within a single .db file.[15] This choice allows for rapid iteration and ease of setup. While SQLite is suitable for development, it is important to acknowledge that for production-grade RWA

tokenization, which involves high transaction volumes and sensitive financial data, a more robust, scalable, and secure database solution (e.g., PostgreSQL, enterprise-grade databases) would typically be required. This sets expectations for future growth and aligns with the platform's production deployment considerations.

Conceptual Installation Steps:

The following outlines the general sequence for setting up the environment. This example illustrates common installation steps for a Python project:

Python

```
# Create project directory
 mkdir rwa_tokenization_platform
 cd rwa_tokenization_platform

 # Create virtual environment
 python -m venv venv
 source venv/bin/activate # On Windows: venv\Scripts\activate

 # Install required packages (detailed in next section)
 # pip install...

 # Create environment file (detailed in next section)
 # echo "OPENAI_API_KEY=your_api_key_here" >.env

 # Create data directory
 mkdir data

 # Run the system (conceptual)
 # python app.py
```

●

Required Python Packages and Installation (google-generativeai, etc.)

The platform relies on a specific set of Python packages to enable its core functionalities, including web serving, agentic AI, data manipulation, and PDF generation.

Core Dependencies:

- flask: The micro-framework for building the web application.[14]
- python-dotenv: Essential for loading environment variables from .env files, ensuring sensitive data like API keys are kept out of the codebase.[9]
- openai and agents (OpenAI Agents SDK): Fundamental for the platform's agentic AI core, enabling the creation, orchestration, and execution of specialized AI agents.[9]
- google-generativeai: Integrates with Google's Gemini API, providing advanced Large Language Model capabilities for tasks like intelligent parsing.[19]
- pandas: Used for efficient data manipulation and analysis, particularly when handling asset inventory or structured data.[9]

2.2 Agentic AI Framework

The platform's intelligence is primarily driven by an agentic AI framework, where distinct "agents" or modules are responsible for specific tasks within the RWA tokenization pipeline. This modularity enhances maintainability, scalability, and the ability to integrate advanced AI capabilities.

Flowchart Description for Agentic AI Framework:

The agentic AI framework operates in a sequential yet interconnected manner, ensuring a comprehensive and automated workflow for RWA tokenization.

1. **Start: User Input (Unstructured Asset Description)**
   - The process begins when a user submits an unstructured text description of their real-world asset (e.g., "A 2-bedroom apartment in Mumbai, estimated value $1.5 Crore INR").
2. **LLM Agent (Data Extraction)**
   - The unstructured user input is fed to the LLM Utility (llm_utils.py).
   - This agent, powered by a Large Language Model (e.g., Gemini), processes the text to extract key structured information: asset_type, estimated_value, location, and a cleaned description.
   - **Output:** Structured Asset Data (JSON format).

3. **Database Integration (Initial Storage)**
   - The extracted structured asset data is then stored in the central database (database.py) via the Flask application (main.py). A unique asset_id is assigned, and the initial verification_status is set to 'requires_review'.
4. **Verification Agent (Asset Assessment)**
   - Upon initiation of the verification process (triggered by the Flask API), the VerificationAgent (verification_agent.py) takes the structured asset data from the database.
   - This agent internally performs several specialized checks:
     - **Basic Information Check:** Assesses completeness of fundamental fields.
     - **Value Assessment:** Validates the estimated value against typical ranges for the asset type.
     - **Jurisdiction Verification:** Identifies and validates the asset's location.
     - **Asset-Specific Details Check:** Verifies the presence of relevant keywords and details for the specific asset type within the description.
   - Each check contributes to a breakdown score, which is then aggregated into an overall_score.
   - Based on this score, the agent determines the status (verified, requires_review, rejected) and generates recommendations and next_steps.
   - **Output:** Detailed Verification Result (including overall score, status, breakdown, recommendations).
5. **Database Integration (Verification Update)**
   - The verification result is updated in the database for the corresponding asset, changing its verification_status. A Transaction record is also created.
6. **Decision Point: Is Asset Verified?**
   - If the verification_status is 'verified', the process proceeds to tokenization.
   - If 'requires_review' or 'rejected', the user is prompted for more information or the process ends.
7. **Tokenization Agent (Blockchain Integration - Mock)**
   - If the asset is verified, the TokenizationAgent (tokenization_agent.py) is invoked.
   - This agent performs mock blockchain operations:
     - **Generate Token Metadata:** Creates rich, standardized metadata (e.g., ERC-721 compatible) for the asset, incorporating details from the verification process.
     - **Create Mock Contract:** Simulates the creation of a smart contract address, ABI, and bytecode.
     - **Generate Token ID & Transaction Hash:** Assigns a unique token ID and a mock transaction hash.

    - **Output:** Tokenization Result (including token ID, contract address, transaction hash, metadata).
8. **Database Integration (Tokenization Update)**
    - The tokenization details (e.g., token_id, contract_address, transaction_hash) are updated in the database for the asset. A Transaction record for tokenization is also created.
9. **End: Tokenized Asset on Blockchain (Mock)**
    - The asset is now conceptually tokenized, with its digital representation and associated metadata available.

3. Technical Implementation

This section provides a detailed breakdown of the Python codebase, explaining each module's purpose, key classes, methods, and their interconnections. All code snippets are provided with extensive inline comments for clarity.

3.1 Database Schema (database.py)

```
# app/models/database.py

 from flask_sqlalchemy import SQLAlchemy
 from datetime import datetime
 import json

 # Initialize SQLAlchemy instance
 db = SQLAlchemy()

 # User Model: Represents a user of the platform
 class User(db.Model):
      # Unique identifier for the user
      id = db.Column(db.Integer, primary_key=True)
      # Blockchain wallet address, must be unique and not null
      wallet_address = db.Column(db.String(42), unique=True, nullable=False)
      # Optional email address for the user
      email = db.Column(db.String(120), nullable=True)
      # KYC (Know Your Customer) status, default is 'pending'
      kyc_status = db.Column(db.String(20), default='pending')
      # User's jurisdiction (e.g., 'IN', 'US'), optional
      jurisdiction = db.Column(db.String(10), nullable=True)
      # Timestamp when the user record was created
      created_at = db.Column(db.DateTime, default=datetime.utcnow)

      # Method to convert User object to a dictionary for API responses
      def to_dict(self):
      return {
            'id': self.id,
            'wallet_address': self.wallet_address,
            'email': self.email,
            'kyc_status': self.kyc_status,
            'jurisdiction': self.jurisdiction,
            'created_at': self.created_at.isoformat() # Convert datetime to ISO
format string
      }

 # Asset Model: Represents a real-world asset submitted for tokenization
 class Asset(db.Model):
      # Unique identifier for the asset
      id = db.Column(db.Integer, primary_key=True)
      # Foreign key linking to the User who owns/submitted the asset
      user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)
      # Type of asset (e.g., 'real_estate', 'vehicle', 'artwork')
      asset_type = db.Column(db.String(50), nullable=False)
      # Detailed description of the asset
      description = db.Column(db.Text, nullable=False)
      # Estimated monetary value of the asset
      estimated_value = db.Column(db.Float, nullable=False)
      # Location of the asset
      location = db.Column(db.String(200), nullable=False)
      # Current verification status (e.g., 'pending', 'requires_review',
'verified', 'rejected')
      verification_status = db.Column(db.String(20), default='pending')
      # Token ID if the asset has been tokenized, nullable initially
      token_id = db.Column(db.String(100), nullable=True)
      # JSON string storing additional requirements or verification details
      requirements = db.Column(db.Text, nullable=True)
      # Timestamp when the asset record was created
      created_at = db.Column(db.DateTime, default=datetime.utcnow)
      # Timestamp of the last update to the asset record
      updated_at = db.Column(db.DateTime, default=datetime.utcnow,
onupdate=datetime.utcnow)

      # Relationship to the User model, allowing access to user details from an
asset
      user = db.relationship('User', backref=db.backref('assets', lazy=True))

      # Method to convert Asset object to a dictionary for API responses
      def to_dict(self):
      return {
            'id': self.id,
            'user_id': self.user_id,
            'asset_type': self.asset_type,
            'description': self.description,
            'estimated_value': self.estimated_value,
            'location': self.location,
            'verification_status': self.verification_status,
            'token_id': self.token_id,
            # Parse JSON string back to a dictionary if 'requirements' exists
            'requirements': json.loads(self.requirements) if self.requirements
else {},
            'created_at': self.created_at.isoformat(),
            'updated_at': self.updated_at.isoformat()
      }

 # Transaction Model: Records various operations performed on assets (e.g.,
verification, tokenization)
 class Transaction(db.Model):
      # Unique identifier for the transaction
      id = db.Column(db.Integer, primary_key=True)
      # Foreign key linking to the Asset involved in the transaction
      asset_id = db.Column(db.Integer, db.ForeignKey('asset.id'),
nullable=False)
      # Type of transaction (e.g., 'intake', 'verification', 'tokenization',
'transfer')
      transaction_type = db.Column(db.String(50), nullable=False)
      # Hash of the blockchain transaction if applicable, nullable
      transaction_hash = db.Column(db.String(100), nullable=True)
      # Status of the transaction (e.g., 'pending', 'completed', 'failed')
      status = db.Column(db.String(20), default='pending')
      # JSON string storing detailed information about the transaction result
      details = db.Column(db.Text, nullable=True)
      # Timestamp when the transaction record was created
      created_at = db.Column(db.DateTime, default=datetime.utcnow)

      # Relationship to the Asset model, allowing access to asset details from
a transaction
      asset = db.relationship('Asset', backref=db.backref('transactions',
lazy=True))

      # Method to convert Transaction object to a dictionary for API responses
      def to_dict(self):
      return {
            'id': self.id,
            'asset_id': self.asset_id,
            'transaction_type': self.transaction_type,
            'transaction_hash': self.transaction_hash,
            'status': self.status,
            # Parse JSON string back to a dictionary if 'details' exists
            'details': json.loads(self.details) if self.details else {},
            'created_at': self.created_at.isoformat()
      }
```

3.2 LLM Utilities (llm_utils.py)

The llm_utils.py file handles the interaction with the Large Language Model (LLM), specifically Google's Gemini, to extract structured information from unstructured user input describing an asset. It includes functions for configuring the LLM, providing fallback mechanisms, and cleaning the LLM's output.

```
# app/agents/llm_utils.py

 import os
 import json
 import re
 from dotenv import load_dotenv
 import google.generativeai as genai

 # Load environment variables from .env file (e.g., for GEMINI_API_KEY)
 load_dotenv()
 GENAI_API_KEY = os.getenv("GEMINI_API_KEY")

 # Configure the Google Generative AI library with the API key
 genai.configure(api_key=GENAI_API_KEY)

 # Initialize the Gemini model (using gemini-1.5-flash for efficiency)
 model = genai.GenerativeModel("gemini-1.5-flash")

 def fallback_asset_type(description: str) -> str:
      """
      Provides a fallback mechanism to categorize asset type based on keywords
      in the description, in case the LLM fails to provide a valid asset_type.
      This ensures a basic categorization even if the LLM output is imperfect.
      """
      description = description.lower() # Convert description to lowercase for
case-insensitive matching
      if any(word in description for word in ['apartment', 'flat', 'bedroom',
'sqft', 'deed', 'property', 'house']):
      return 'real_estate'
      elif any(word in description for word in ['engine', 'model', 'mileage',
'car', 'vehicle', 'truck', 'bike']):
      return 'vehicle'
      elif any(word in description for word in ['artist', 'painting', 'canvas',
'sculpture', 'artwork']):
      return 'artwork'
      elif any(word in description for word in ['serial', 'manufacturer',
'warranty', 'equipment', 'machine']):
      return 'equipment'
```

~~~
      elif any(word in description for word in ['weight', 'grade', 'purity',
'commodity', 'gold', 'silver', 'oil']):
      return 'commodity'
      else:
      return 'unknown' # Default if no keywords match

 def clean_llm_output(content: str) -> str:
      """
      Cleans the raw LLM output by removing markdown code block markers (e.g.,
```json)
      and any leading/trailing whitespace or text outside the JSON object.
      This ensures the output can be reliably parsed as JSON.
      """
      # Remove triple backticks and optional 'json' language marker
      content = re.sub(r"^```(?:json)?", "", content, flags=re.MULTILINE)
      content = re.sub(r"```$", "", content, flags=re.MULTILINE)
      content = content.strip() # Remove leading/trailing whitespace
      # Find the start of the JSON object and trim any text before it
      json_start = content.find("{")
      if json_start != -1:
      content = content[json_start:]
      return content

 def extract_asset_info_with_llm(user_input: str) -> dict:
      """
      Calls the Gemini LLM to extract structured asset information from a given
      unstructured user input. It defines a clear prompt for the LLM to guide
      its extraction process and includes error handling and fallback logic.
      """
      print("✅ Calling Gemini for asset info extraction...")
      # Define the prompt for the LLM, instructing it to extract specific
fields
      # and return them in JSON format.
      prompt = f"""
 You are an intelligent assistant that extracts structured information from
asset descriptions.
 Extract and return the following fields in JSON:
 - asset_type: One of [real_estate, vehicle, artwork, equipment, commodity]
 - estimated_value: A number (preferably in INR, or fallback to USD if only
that is given), no commas or currency symbol
 - location: City or region (as precise as possible, e.g., 'Etawah, Uttar
Pradesh, India' or 'Mumbai, India' or 'Bandra, Mumbai, India')
 - description: The original user input

 USER INPUT:
 \"\"\"{user_input}\"\"\"

 Return only valid JSON:
~~~

```
 {{
   "asset_type": "...",
   "estimated_value": ...,
   "location": "...",
   "description": "..."
 }}
 """
      try:
      # Generate content using the configured Gemini model
      response = model.generate_content(prompt)
      content = response.text.strip() # Get the raw text response
      print("Gemini raw response:", repr(content)) # Log the raw response for
debugging

      # Clean the LLM output to ensure it's valid JSON
      cleaned = clean_llm_output(content)
      # Validate that the cleaned output starts with a JSON object
      if not cleaned or not cleaned.startswith("{"):
            raise ValueError("LLM did not return valid JSON.")

      # Parse the cleaned JSON string into a Python dictionary
      data = json.loads(cleaned)

      # Get asset_type, using fallback if LLM returns 'unknown'
      asset_type = data.get("asset_type", "unknown")
      if asset_type == "unknown":
            asset_type = fallback_asset_type(data.get("description",
user_input))

      # Get location (accepts any string as valid, no specific validation here)
      location = data.get("location", "unknown")

      # Return the extracted and processed asset information
      return {
            "asset_type": asset_type,
            "estimated_value": float(data.get("estimated_value", 0)), # Ensure
value is a float
            "location": location,
            "description": data.get("description", user_input)
      }
      except Exception as e:
      # Log any errors during LLM interaction or JSON parsing
      print("❌ Error parsing Gemini response:", e)
      # In case of an error, return a default structure with fallback asset
type
      asset_type = fallback_asset_type(user_input)
      return {
            "asset_type": asset_type,
            "estimated_value": 0,
            "location": "unknown",
            "description": user_input
      }
```

3.3 Verification Agent (verification_agent.py)

The VerificationAgent class is central to the platform's ability to assess the quality and completeness of asset data. It performs a multi-faceted evaluation, assigning scores to different aspects of the asset and determining an overall verification status.

```
# app/agents/verification_agent.py (based on verification_agent - Copy.py)



import re

import json

from typing import Dict, List



class VerificationAgent:

      """

      The VerificationAgent is responsible for assessing the completeness,

      accuracy, and validity of real-world asset data. It performs various

      checks and assigns an overall verification score and status.

      """

      def __init__(self):

      # Define the threshold for an asset to be considered "verified"

      self.verification_threshold = 0.7

      # Define typical value ranges for different asset types.

      # These ranges are used to assess if the estimated value is reasonable.

      self.value_ranges = {
            'real_estate': {'min': 10000, 'max': 1_000_000_000}, # INR, USD,
etc.
            'vehicle': {'min': 1000, 'max': 2_000_000},
            'artwork': {'min': 500, 'max': 100_000_000},
            'equipment': {'min': 100, 'max': 5_000_000},
            'commodity': {'min': 50, 'max': 10_000_000}
      }

      def verify_asset(self, asset_data: Dict) -> Dict:
      """
      Main method to verify an asset. It orchestrates all sub-verification
      steps, calculates an overall score, determines the status, and
      generates recommendations and next steps.

      Args:
            asset_data (Dict): A dictionary containing structured asset
information.

      Returns:
            Dict: A dictionary with the overall verification result, including
                  score, status, breakdown of individual scores, issues,
                  recommendations, and next steps.
      """
      result = {
            'overall_score': 0.0,
            'status': 'pending',
            'breakdown': {}, # Stores scores for individual verification
aspects
            'issues': [],      # List of identified issues
            'recommendations': [], # Suggestions for improving asset data
            'next_steps': [] # Actions to take based on verification status
        }
        try:
            # Perform individual verification checks
            basic_score = self._verify_basic_information(asset_data)
            value_score = self._verify_value(asset_data)
            jurisdiction_score = self._verify_jurisdiction(asset_data)
            asset_specific_score = self._verify_asset_specific(asset_data)

            # Store individual scores in the breakdown
            result["breakdown"].update({
            "basic_info": basic_score,
                "value_assessment": value_score,
            "jurisdiction": jurisdiction_score,
                "asset_specific": asset_specific_score
            })

            # Calculate overall score as the average of individual scores
            scores = [basic_score, value_score, jurisdiction_score,
asset_specific_score]
            result["overall_score"] = round(sum(scores) / len(scores), 2)
            # Determine the overall status based on the overall score
            if result["overall_score"] >= self.verification_threshold:
                result["status"] = "verified"
            elif result["overall_score"] >= 0.5:
                result["status"] = "requires_review"
            else:
                result["status"] = "rejected"

            # Generate recommendations and next steps based on the current
result
            result["recommendations"] = self._generate_recommendations(result)
            result["next_steps"] = self._define_next_steps(result["status"])

        except Exception as e:
            # Handle any unexpected errors during the verification process
            result["status"] = "error"
            result["issues"].append(f"Verification error: {str(e)}")
        return result

        def _verify_basic_information(self, asset_data: Dict) -> float:
        """
        Verifies the completeness and basic quality of core asset information.
        Awards points for presence and basic validity of description, type,
        location, and estimated value.
        """
```

```
      score = 0.0

      # Check if description exists and is sufficiently long

      if asset_data.get("description") and len(asset_data["description"]) > 20:

            score += 0.3

      # Check if asset_type is provided and not 'unknown'

      if asset_data.get("asset_type") and asset_data["asset_type"] !=
"unknown":

            score += 0.3

      # Check if location is provided

      if asset_data.get("location"):

            score += 0.2

      # Check if estimated_value is greater than zero

      if asset_data.get("estimated_value", 0) > 0:

            score += 0.2

      # Cap score at 1.0 and round for consistency

      return round(min(score, 1.0), 2)


      def _verify_value(self, asset_data: Dict) -> float:

      """

      Verifies if the estimated value of the asset falls within typical

      ranges defined for its asset type.

      """

      value = asset_data.get("estimated_value", 0)

      asset_type = asset_data.get("asset_type", "unknown")
      # If asset type is not recognized, assign a neutral score

      if asset_type not in self.value_ranges:

              return 0.5



      # Get the min/max range for the specific asset type

      r = self.value_ranges[asset_type]

      # Check if value is within the expected range

      if r["min"] <= value <= r["max"]:

            return 1.0 # Perfect score if within range

      elif value < r["min"]:

            return 0.4 # Lower score if value is too low

      return 0.6 # Moderate score if value is too high (might be a premium
asset)



      def _verify_jurisdiction(self, asset_data: Dict) -> float:

        """

      Verifies the specificity and recognition of the asset's location

      to determine its jurisdiction. Prioritizes Indian locations.

      """

      jurisdiction = self._extract_jurisdiction(asset_data.get("location", ""))

      # Higher score for Indian jurisdiction (as per project context)

      if jurisdiction == "IN":

            return 0.9

      # Good score for other recognized jurisdictions

      elif jurisdiction != "OTHER" and jurisdiction != '': # Empty string means
no location found at all
```

```
            return 0.9

        return 0.5 # Lower score for unrecognized or vague locations


        def _verify_asset_specific(self, asset_data: Dict) -> float:

        """

        Verifies the presence of asset-specific keywords in the description

        to ensure the description is relevant and detailed for its type.

        """

        asset_type = asset_data.get("asset_type", "unknown")

        description = asset_data.get("description", "").lower()


        # Define keywords relevant to each asset type

        indicators = {

            "real_estate": [

            "flat", "apartment", "bedroom", "sqft", "deed", "house",
"property", "building", "land", "condo", "villa", "bathroom", "plot",
"bungalow"

            ],

            "vehicle": [

            "engine", "model", "mileage", "year", "car", "truck", "motorcycle",
"boat", "plane", "vehicle", "sedan", "suv", "registration"

            ],

            "artwork": [

            "artist", "canvas", "painting", "sculpture", "art", "artwork", "oil
painting", "frame"

            ],

            "equipment": [
            "serial", "manufacturer", "warranty", "machinery", "equipment",
"tool", "device", "machine", "operating hours", "condition"

            ],

            "commodity": [

            "weight", "grade", "purity", "gold", "silver", "oil", "wheat",
"commodity", "metal", "oz", "certificate", "assay", "quality"

            ]

      }


      # If asset type is not recognized, assign a neutral score

      if asset_type not in indicators:

            return 0.5


      score = 0.5 # Starting score for asset-specific check

      # Increment score for each relevant keyword found in the description

      for word in indicators[asset_type]:

            if word in description:

            score += 0.05 # Granular increment

      # Cap score at 1.0 and round

      return round(min(score, 1.0), 2)


      def _extract_jurisdiction(self, location: str) -> str:

      """

      Helper method to extract a standardized jurisdiction code from a

      given location string. Prioritizes India and then other major regions.

      """
```

```
    if not location:
        return '' # Return empty string if no location provided

    loc = location.lower() # Convert to lowercase for consistent matching

    # Check for India first
    if "india" in loc:
        return "IN"

    # List of Indian states/UTs for more specific matching
    indian_states = [
        "andhra pradesh", "arunachal pradesh", "assam", "bihar",
"chhattisgarh", "goa", "gujarat",
        "haryana", "himachal pradesh", "jharkhand", "karnataka", "kerala",
"madhya pradesh",
        "maharashtra", "manipur", "meghalaya", "mizoram", "nagaland",
"odisha", "punjab",
        "rajasthan", "sikkim", "tamil nadu", "telangana", "tripura", "uttar
pradesh", "uttarakhand",
        "west bengal", "delhi", "jammu", "kashmir", "ladakh", "puducherry",
"chandigarh", "dadra",
        "daman", "daman and diu", "andaman", "nicobar", "lakshadweep"
    ]
    if any(state in loc for state in indian_states):
        return "IN"

    # Mapping for other international jurisdictions
```

```
      mapping = {
            'US': ['usa', 'united states', 'america', 'new york', 'california',
'texas'],
            'UK': ['united kingdom', 'england', 'scotland', 'wales', 'london'],
            'CA': ['canada', 'toronto', 'vancouver', 'montreal'],
            'EU': ['germany', 'france', 'spain', 'italy', 'netherlands'],
            'SG': ['singapore']
      }
      # Iterate through mappings to find a match
      for code, keywords in mapping.items():
            if any(city in loc for city in keywords):
            return code
      return 'OTHER' # If no recognized jurisdiction is found


      def _generate_recommendations(self, result: Dict) -> List[str]:
      """
      Generates a list of recommendations based on the breakdown of
      verification scores. These advise the user on how to improve asset data.
        """
      recos = []
      b = result.get("breakdown", {}) # Get the breakdown of scores


      # Add recommendations if specific scores are below a threshold (e.g.,
0.8)
      if b.get("basic_info", 0) < 0.8:
            recos.append("Provide a more complete asset description.")
      if b.get("value_assessment", 0) < 0.8:

            recos.append("Provide a formal valuation or appraisal document.")

      if b.get("jurisdiction", 0) < 0.8:

            recos.append("Clarify the asset's location or city.")

      if b.get("asset_specific", 0) < 0.8:

            recos.append("Include more asset-specific details like documents,
specs, or characteristics.")

      return recos



      def _define_next_steps(self, status: str) -> List[str]:

      """

      Defines the appropriate next steps for the user based on the

      overall verification status.

      """

      if status == "verified":

            return ["Proceed to tokenization", "Create token on blockchain",
"Generate smart contract"]

      elif status == "requires_review":

            return ["Add more details", "Request manual review"]

      else: # status == "rejected"

            return ["Asset rejected", "Revise asset information"
```

3.4 Modular Agents for Assessment (agents_modular.py)

The agents_modular.py file presents a modular approach to asset assessment, where distinct, specialized agents evaluate different facets of an asset's data. This design promotes clear separation of concerns, making the system more extensible and maintainable. The CoordinatorAgent orchestrates these individual agents to produce a comprehensive verification result. While the primary VerificationAgent (verification_agent - Copy.py) consolidates these checks for simplicity in the main workflow, this file demonstrates a more granular, agent-based design pattern for assessment.

Individual Assessment Agents

- **ValueAgent:** Assesses if the estimated value of an asset falls within predefined typical ranges for its type.
- **RiskAgent:** Evaluates the quality and completeness of the asset's description, flagging vague or too-short descriptions as higher risk.
- **ConsistencyAgent:** Checks for consistency between the declared asset type and keywords found within its description.
- **DescriptionQualityAgent:** Provides a more detailed assessment of the description's length and presence of generic/vague terms.
- **ValueConsistencyAgent:** Cross-references the estimated value with descriptive terms (e.g., "old", "luxury") to ensure logical consistency.
- **LocationSpecificityAgent:** Determines the specificity and recognition of the asset's location, with a focus on Indian regions.

UserInteractionAgent

This agent is designed to identify missing or vague information in the asset data and formulate follow-up questions to prompt the user for more details.

CoordinatorAgent

The CoordinatorAgent acts as the orchestrator for the individual assessment agents. It iterates through each agent, collects their assessment scores and notes, calculates an overall score, determines the asset's status, and aggregates follow-up questions and explanations.

```
class ValueAgent:
      def assess(self, asset):
      value = asset.get("estimated_value", 0)
      asset_type = asset.get("asset_type", "unknown")
      ranges = {
            "real_estate": (10000, 1_000_000_000),
            "vehicle": (1000, 2_000_000),
            "artwork": (500, 100_000_000),
            "equipment": (100, 5_000_000),
            "commodity": (50, 10_000_000),
      }
      min_val, max_val = ranges.get(asset_type, (0, float('inf')))
      if min_val <= value <= max_val:
            return {"score": 1.0, "notes": "Value within typical range"}
      elif value < min_val:
            return {"score": 0.4, "notes": "Value below expected"}
      else:
            return {"score": 0.6, "notes": "Value above expected"}

 class RiskAgent:
      def assess(self, asset):
      desc = asset.get("description", "").lower()
      if "unknown" in desc or len(desc) < 20:
            return {"score": 0.4, "notes": "Description vague or too short"}
      return {"score": 1.0, "notes": "Description seems adequate"}

 class ConsistencyAgent:
      def assess(self, asset):
      asset_type = asset.get("asset_type", "")
      desc = asset.get("description", "").lower()
      keywords = {
            "real_estate": ["apartment", "sqft", "deed"],
            "vehicle": ["engine", "model", "mileage"],
            "artwork": ["artist", "painting"],
      }
      hits = sum(1 for word in keywords.get(asset_type, []) if word in desc)
      score = 0.5 + 0.1 * hits
      return {"score": min(score, 1.0), "notes": f"Found {hits} asset-specific
keywords"}

 class DescriptionQualityAgent:
      def assess(self, asset):
      desc = asset.get("description", "").strip().lower()
      if len(desc) < 50 or any(term in desc for term in ["unknown", "n/a",
"none"]):
            return {"score": 0.4, "notes": "Description is too short or vague"}
      return {"score": 1.0, "notes": "Description is detailed and specific"}

 class ValueConsistencyAgent:
      def assess(self, asset):
      value = asset.get("estimated_value", 0)
      desc = asset.get("description", "").lower()
      if "old" in desc and value > 1_000_000:
            return {"score": 0.5, "notes": "High value for 'old' asset"}
      if "luxury" in desc and value < 100_000:
            return {"score": 0.5, "notes": "Low value for 'luxury' asset"}
      return {"score": 1.0, "notes": "Value matches description"}

 class LocationSpecificityAgent:
      def assess(self, asset):
      location = asset.get("location", "").strip().lower()
      if not location or location in ["unknown", "n/a", "none"]:
            return {"score": 0.4, "notes": "Location is vague or missing"}
      if len(location) < 3:
            return {"score": 0.6, "notes": "Location too short"}
      if "india" in location:
            return {"score": 1.0, "notes": "Location is specific and recognized
(India)"}
      indian_states = [
            "andhra pradesh", "arunachal pradesh", "assam", "bihar",
"chhattisgarh", "goa", "gujarat",
            "haryana", "himachal pradesh", "jharkhand", "karnataka", "kerala",
"madhya pradesh",
            "maharashtra", "manipur", "meghalaya", "mizoram", "nagaland",
"odisha", "punjab",
            "rajasthan", "sikkim", "tamil nadu", "telangana", "tripura", "uttar
pradesh", "uttarakhand",
            "west bengal", "delhi", "jammu", "kashmir", "ladakh", "puducherry",
"chandigarh", "dadra",
            "daman", "daman and diu", "andaman", "nicobar", "lakshadweep"
      ]
      if any(state in location for state in indian_states):
            return {"score": 1.0, "notes": "Location is specific and recognized
(India)"}
      return {"score": 1.0, "notes": "Location is specific"}

 class UserInteractionAgent:
      def query(self, asset):
      questions = []
      if not asset.get("location"):
            questions.append("Can you specify the asset's location?")
      if asset.get("estimated_value", 0) == 0:
            questions.append("What is the estimated value?")
      if len(asset.get("description", "")) < 20:
            questions.append("Please provide a more detailed description.")
      return questions

 class CoordinatorAgent:
      def __init__(self):
      self.agents = [
            ValueAgent(), RiskAgent(), ConsistencyAgent(),
            DescriptionQualityAgent(), ValueConsistencyAgent(),
            LocationSpecificityAgent()
      ]
      self.user_interaction_agent = UserInteractionAgent()

      def verify(self, asset):
      results = {}
      for agent in self.agents:
            agent_name = agent.__class__.__name__
            results[agent_name] = agent.assess(asset)
      scores = [r["score"] for r in results.values()]
      avg_score = sum(scores) / len(scores)
      status = "verified" if avg_score >= 0.7 else ("requires_review" if
avg_score >= 0.5 else "rejected")
      questions = self.user_interaction_agent.query(asset)
      return {
            "overall_score": round(avg_score, 2),
            "status": status,
            "breakdown": results,
            "follow_up_questions": questions,
            "explanation": [r["notes"] for r in results.values()]
      }

    }
```

3.5 Tokenization Agent (tokenization_agent.py)

The TokenizationAgent is responsible for simulating the blockchain tokenization process. In this proof-of-concept, it generates mock token IDs, contract addresses, transaction hashes, and standardized metadata for Real World Assets. This agent ensures that only verified assets can proceed to tokenization.

```
# app/agents/tokenization_agent.py

 import hashlib
 import json
 import time
 from datetime import datetime
 from typing import Dict
 import uuid

 class TokenizationAgent:
      """
      The TokenizationAgent handles the process of converting a verified
      real-world asset into a mock blockchain token. It generates token IDs,
      metadata, and simulates contract and transaction details.
      """
      def __init__(self):
      # Define the token standard (e.g., ERC-721 for NFTs)
      self.token_standard = "RWA-721"
      # Define the mock blockchain network
      self.network = "RWA-TestNet"

      def tokenize_asset(self, asset_data: Dict, verification_result: Dict) ->
Dict:
      """
      Main method to tokenize an asset. It first checks if the asset
      has been successfully verified before proceeding with tokenization.

      Args:
            asset_data (Dict): The structured data of the asset to be
tokenized.
            verification_result (Dict): The result from the VerificationAgent,
                                      containing the asset's verification
status.

      Returns:
            Dict: A dictionary containing the tokenization result, including
                  success status, token details, metadata, and mock blockchain
info.
      """
      # Pre-condition check: Asset must be 'verified' to proceed with
tokenization
      if verification_result.get('status') != 'verified':
            return {
                  'success': False,
                  'error': 'Asset must be verified before tokenization',
                  'status': 'failed'
            }

      try:
            # 1. Generate token metadata based on asset and verification data
            token_metadata = self._generate_token_metadata(asset_data,
verification_result)
            # 2. Create mock contract details (address, ABI, bytecode)
            contract_data = self._create_mock_contract(asset_data,
token_metadata)
            # 3. Generate a unique token ID
            token_id = self._generate_token_id(asset_data)
            # 4. Generate a mock transaction hash for the minting process
            transaction_hash = self._generate_transaction_hash(contract_data)

            # Return a successful tokenization result
            return {
                  'success': True,
                  'token_id': token_id,
                  'contract_address': contract_data['address'],
                  'transaction_hash': transaction_hash,
                  'metadata': token_metadata,
                  'network': self.network,
               'standard': self.token_standard,
                  'created_at': datetime.utcnow().isoformat(), # Timestamp of
tokenization
                  'status': 'minted'
            }

      except Exception as e:
            # Handle any errors during the tokenization process
            return {
                  'success': False,
                  'error': f'Tokenization failed: {str(e)}',
                  'status': 'failed'
            }

      def _generate_token_metadata(self, asset_data: Dict, verification_result:
Dict) -> Dict:
      """
      Generates standardized metadata for the RWA token, typically following
      standards like ERC-721 metadata JSON Schema.

      Args:
            asset_data (Dict): The original asset data.
            verification_result (Dict): The result of the verification process.

      Returns:
            Dict: A dictionary representing the token's metadata.
      """
      # Extract relevant information from asset_data and verification_result
      asset_type = asset_data.get('asset_type', 'Unknown')
      value = asset_data.get('estimated_value', 0)
      description = asset_data.get('description', 'Real World Asset Token')
      location = asset_data.get('location', 'Unknown')
      status = verification_result.get('status', 'unknown')
      score = verification_result.get('overall_score', 0.0)

      return {
            'name': f"RWA Token - {asset_type.title()}", # Token name (e.g.,
"RWA Token - Real Estate")
            'description': description, # Full description of the asset
            # Placeholder image URL, dynamically generated based on asset type
            'image':
f"[https://via.placeholder.com/400x400.png?text=](https://via.placeholder.com/4
00x400.png?text=){asset_type}",
            # Mock external URL for more details about the asset
            'external_url':
f"[https://rwa-marketplace.com/asset/](https://rwa-marketplace.com/asset/){asse
t_data.get('id', 'unknown')}",
            'attributes': [ # Key-value pairs describing the asset's traits
                  {'trait_type': 'Asset Type', 'value': asset_type.title()},
                  {'trait_type': 'Estimated Value', 'value': f"${value:,.2f}"},
                  {'trait_type': 'Location', 'value': location},
                  {'trait_type': 'Verification Status', 'value':
```

```
status.title()},
                   {'trait_type': 'Verification Score', 'value': f"{score *
100:.1f}%"},
                   {'trait_type': 'Token Standard', 'value':
self.token_standard},
                   {'trait_type': 'Network', 'value': self.network},
                   {'trait_type': 'Tokenization Date', 'value':
datetime.utcnow().strftime('%Y-%m-%d')}
             ],
             'properties': { # Additional structured properties
                   'category': 'Real World Asset',
                   'subcategory': asset_type,
                   'fractional': False, # Indicates if the token represents
fractional ownership
                   'transferable': True
             }
      }

      def _create_mock_contract(self, asset_data: Dict, metadata: Dict) ->
Dict:
      """
      Simulates the creation of a smart contract for the RWA token.
      Generates a mock contract address, ABI, and bytecode.

      Args:
             asset_data (Dict): The original asset data.
             metadata (Dict): The generated token metadata.

      Returns:
             Dict: A dictionary containing mock smart contract details.
      """
      # Generate a mock contract address
      contract_address = self._generate_contract_address(asset_data)

         return {
             'address': contract_address, # Mock contract address
             'abi': self._get_mock_abi(), # Mock ABI (Application Binary
Interface)
             'bytecode': self._generate_mock_bytecode(asset_data), # Mock
bytecode
             'constructor_args': { # Mock arguments passed to the contract
constructor
                   'name': metadata['name'],
                   'symbol': 'RWA',
                   'baseURI':
'[https://api.rwa-tokenization.com/metadata/](https://api.rwa-tokenization.com/
metadata/)'
             },
             'functions': { # Mock functions available on the contract
                   'tokenURI':
f'[https://api.rwa-tokenization.com/metadata/](https://api.rwa-tokenization.com
/metadata/){contract_address}',
                   'ownerOf': asset_data.get('user_id', 'unknown'),
                   'approve': 'function approve(address to, uint256 tokenId)',
                   'transfer': 'function transfer(address to, uint256 tokenId)'
             },
             'events': [ # Mock events emitted by the contract
                   {
                   'name': 'Transfer',
                   'signature': 'Transfer(address indexed from, address indexed
to, uint256 indexed tokenId)'
                   },
                   {
                   'name': 'AssetTokenized',
                   'signature': 'AssetTokenized(uint256 indexed tokenId, address
indexed owner, string assetType)'
                   }
             ]
      }

      def _generate_token_id(self, asset_data: Dict) -> str:
      """
      Generates a unique, mock token ID for the asset.
      Uses a hash based on asset details and current timestamp for uniqueness.
      """
      # Combine asset ID (or a UUID if not available), asset type, and current
timestamp
      base = f"{asset_data.get('id',
str(uuid.uuid4()))}_{asset_data.get('asset_type', 'asset')}_{int(time.time())}"
      token_hash = hashlib.sha256(base.encode()).hexdigest()
      return f"RWA_{token_hash[:16].upper()}" # Prefix with "RWA_" and take
first 16 chars of hash

      def _generate_contract_address(self, asset_data: Dict) -> str:
      """
      Generates a mock blockchain contract address.
      """
      content = f"contract_{asset_data.get('asset_type',
'unknown')}_{uuid.uuid4()}"
      return f"0x{hashlib.sha256(content.encode()).hexdigest()[:40]}" #
Standard Ethereum address format (0x + 40 hex chars)

      def _generate_transaction_hash(self, contract_data: Dict) -> str:
      """
      Generates a mock blockchain transaction hash.
      """
```

```
         content = f"tx_{contract_data['address']}_{int(time.time())}"
      return f"0x{hashlib.sha256(content.encode()).hexdigest()}" # Standard
transaction hash format (0x + 64 hex chars)

      def _generate_mock_bytecode(self, asset_data: Dict) -> str:
      """
      Generates mock bytecode for the smart contract.
      """
      content = f"bytecode_{asset_data.get('asset_type', 'unknown')}"
      return f"0x{hashlib.sha256(content.encode()).hexdigest()}" # Mock
bytecode string

      def _get_mock_abi(self) -> list:
      """
      Provides a simplified, mock Application Binary Interface (ABI) for an
      ERC-721 like token contract. This defines how to interact with the
contract.
      """
      return [
            {
                  "inputs": [
                  {"name": "to", "type": "address"},
                  {"name": "tokenId", "type": "uint256"}
                  ],
                  "name": "approve",
                  "outputs": [],
                  "type": "function"
          },
            {
                  "inputs": [{"name": "tokenId", "type": "uint256"}],
                  "name": "tokenURI",
                  "outputs": [{"name": "", "type": "string"}],
                  "type": "function"
            },
            {
                  "inputs": [{"name": "tokenId", "type": "uint256"}],
                  "name": "ownerOf",
                  "outputs": [{"name": "", "type": "address"}],
                  "type": "function"
            }
      ]

      def verify_token_ownership(self, token_id: str, wallet_address: str) ->
bool:
      """
      Mocks the verification of token ownership. In a real system, this would
      query the blockchain. For this POC, it always returns True.
      """
      return True  # Always true in POC

      def transfer_token(self, token_id: str, from_address: str, to_address:
str) -> Dict:
      """
      Mocks the transfer of a token between two addresses.
      Generates a mock transaction hash for the transfer.
      """
      transaction_hash = hashlib.sha256(

f"transfer_{token_id}_{from_address}_{to_address}_{int(time.time())}".encode()
      ).hexdigest()

      return {
            'success': True,
            'transaction_hash': f"0x{transaction_hash}",
            'from_address': from_address,
            'to_address': to_address,
            'token_id': token_id,
            'timestamp': datetime.utcnow().isoformat()
      }
```

3.6 Main Application (main.py)

The main.py file serves as the Flask web application, acting as the central orchestrator for the entire RWA tokenization platform. It defines API endpoints that handle user requests, interact with the database, and invoke the various AI agents (llm_utils, VerificationAgent, TokenizationAgent) to process assets.

```
# main.py

 import os
 import sys
 import json
 import logging
 from datetime import datetime
 from flask import Flask, request, jsonify, render_template
 from flask_sqlalchemy import SQLAlchemy
 from flask_cors import CORS

 # Add the parent directory to the system path to allow imports from 'app'
 sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__),
'..')))

 # Import database models and agents
 from app.models.database import db, User, Asset, Transaction
 from app.agents.verification_agent import VerificationAgent
 from app.agents.tokenization_agent import TokenizationAgent
 from app.agents.llm_utils import extract_asset_info_with_llm

 # Initialize Flask application
 app = Flask(__name__, template_folder='../templates',
static_folder='../static')

 # Flask configuration
 app.config['SECRET_KEY'] = 'your-secret-key-change-this' # Secret key for
session management
 app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///rwa_tokenization.db' #
SQLite database file
 app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False # Disable SQLAlchemy
event system for performance

 # Initialize SQLAlchemy with the Flask app
 db.init_app(app)
 # Enable Cross-Origin Resource Sharing (CORS) for all routes
 CORS(app)

 # Initialize the AI agents
 verification_agent = VerificationAgent()
 tokenization_agent = TokenizationAgent()

 # Configure logging
 os.makedirs('logs', exist_ok=True) # Ensure logs directory exists
 logging.basicConfig(
      level=logging.INFO, # Set logging level to INFO
      format='%(asctime)s - %(levelname)s - %(message)s', # Log format
      handlers=[
         logging.FileHandler('logs/app.log'), # Log to file
      logging.StreamHandler() # Log to console
      ]
 )
 logger = logging.getLogger(__name__) # Get logger instance

 # Create database tables within the application context
 with app.app_context():
      db.create_all()

 # --- API Routes ---

 @app.route('/')
 def home():
      """
      Renders the main HTML page of the application.
      """
      return render_template('index.html')

@app.route('/api/health')
def health_check():
    """
      Provides a health check endpoint for the API.
      Returns basic status information.
      """
      return jsonify({
      'status': 'healthy',
      'timestamp': datetime.utcnow().isoformat(),
      'version': '1.0.0'
      })

@app.route('/api/intake', methods=['POST'])
def asset_intake():
      """
      API endpoint for asset intake.
      Receives unstructured user input, uses LLM to extract structured data,
      and stores the new asset in the database.
      """
      try:
      data = request.get_json() # Get JSON data from the request body
      # Validate required fields
      if not data or 'user_input' not in data or 'wallet_address' not in data:
            return jsonify({'error': 'Missing required fields'}), 400

      user_input = data['user_input']
      wallet_address = data['wallet_address']
      email = data.get('email') # Optional email

      logger.info(f"[INTAKE] Received input from {wallet_address}")

      # Use LLM to extract structured asset information
      parsed_data = extract_asset_info_with_llm(user_input)

      # Find or create user based on wallet address
      user = User.query.filter_by(wallet_address=wallet_address).first()
      if not user:
            user = User(wallet_address=wallet_address, email=email)
            db.session.add(user)
            db.session.commit() # Commit user to get an ID

      # Create a new Asset record in the database
      asset = Asset(
            user_id=user.id,
             asset_type=parsed_data.get('asset_type', 'unknown'),
             description=parsed_data.get('description', user_input),
             estimated_value=parsed_data.get('estimated_value', 0),
             location=parsed_data.get('location', 'unknown'),
             verification_status='requires_review', # Initial status
            requirements=json.dumps({}) # Store empty JSON for requirements
initially
      )
      db.session.add(asset)
      db.session.commit() # Commit asset to get an ID

      return jsonify({
            'success': True,
            'message': 'Asset submitted successfully.',
            'asset': asset.to_dict(), # Return the newly created asset details
            'parsed_data': parsed_data, # Return the data extracted by LLM
            'follow_up_questions': [ # Provide generic follow-up questions
                  "Can you upload supporting documents?",
                  "What is the date of acquisition?",
                  "Is there a title deed or registration?"
            ],
            'next_steps': [ # Suggest next steps in the workflow
                  "Review asset",
                  "Proceed to verification"
            ]
      })
      except Exception as e:
      logger.error(f"[INTAKE ERROR] {e}")
      return jsonify({'error': 'Internal server error', 'details': str(e)}),
500

 @app.route('/api/verify/<int:asset_id>', methods=['POST'])
 def verify_asset(asset_id):
      """
      API endpoint to initiate asset verification.
      Retrieves asset data and passes it to the VerificationAgent.
      Updates asset status and logs a transaction.
      """
      try:
      asset = Asset.query.get_or_404(asset_id) # Get asset by ID, or return 404
      logger.info(f"[VERIFY] Verifying asset ID {asset_id}")

      asset_data = asset.to_dict() # Convert asset object to dictionary for
agent
      # Call the VerificationAgent to perform verification
      verification_result = verification_agent.verify_asset(asset_data)

      # Update the asset's verification status based on the agent's result
      asset.verification_status = verification_result['status']
      asset.updated_at = datetime.utcnow() # Update timestamp
      db.session.commit() # Commit changes to asset

      # Record the verification transaction
      transaction = Transaction(
            asset_id=asset.id,
             transaction_type='verification',
             status=verification_result['status'],
             details=json.dumps(verification_result) # Store full result as
JSON string
      )
      db.session.add(transaction)
      db.session.commit() # Commit transaction

      return jsonify({
            'success': True,
            'verification_result': verification_result,
            'asset': asset.to_dict()
      })
      except Exception as e:
      logger.error(f"[VERIFY ERROR] {e}")
      return jsonify({'error': 'Verification failed', 'details': str(e)}), 500

 @app.route('/api/tokenize/<int:asset_id>', methods=['POST'])
 def tokenize_asset(asset_id):
      """
      API endpoint to initiate asset tokenization.
      Checks if asset is verified, then uses TokenizationAgent to create mock
token.
      Updates asset with token_id and logs a transaction.
      """
      try:
      asset = Asset.query.get_or_404(asset_id) # Get asset by ID, or return 404

      # Ensure asset is verified before tokenization
      if asset.verification_status != 'verified':
            return jsonify({'error': 'Asset must be verified before
tokenization'}), 400

      asset_data = asset.to_dict() # Convert asset object to dictionary

      # Retrieve the last verification result from transactions
      last_verification = Transaction.query.filter_by(
            asset_id=asset.id, transaction_type='verification'
         ).order_by(Transaction.created_at.desc()).first()
      # Parse details or default to 'verified' status if no transaction found
(shouldn't happen if status is 'verified')
      verification_result = json.loads(last_verification.details) if
```

```
last_verification else {'status': 'verified'}

      # Call the TokenizationAgent to tokenize the asset
      tokenization_result = tokenization_agent.tokenize_asset(asset_data,
verification_result)

      if tokenization_result.get("success"):
            # Update asset with token_id if tokenization was successful
            asset.token_id = tokenization_result["token_id"]
            asset.updated_at = datetime.utcnow()
            db.session.commit()

            # Record the tokenization transaction
            transaction = Transaction(
                  asset_id=asset.id,
              transaction_type='tokenization',
                 transaction_hash=tokenization_result["transaction_hash"],
                  status='completed',
                 details=json.dumps(tokenization_result) # Store full result as
JSON string
            )
            db.session.add(transaction)
            db.session.commit()

            return jsonify({
                  'success': True,
                  'tokenization_result': tokenization_result,
                  'asset': asset.to_dict()
            })
      else:
            # If tokenization failed, return the error from the agent
            return jsonify(tokenization_result), 400
      except Exception as e:
      logger.error(f"[TOKENIZATION ERROR] {e}")
      return jsonify({'error': 'Tokenization failed', 'details': str(e)}), 500

 @app.route('/api/asset/<int:asset_id>')
 def get_asset(asset_id):
      """
      API endpoint to retrieve details of a specific asset and its
transactions.
      """
      try:
      asset = Asset.query.get_or_404(asset_id) # Get asset by ID
      # Retrieve all transactions related to this asset, ordered by creation
time
      transactions =
Transaction.query.filter_by(asset_id=asset_id).order_by(Transaction.created_at.
desc()).all()
      return jsonify({
            'asset': asset.to_dict(),
            'transactions': [tx.to_dict() for tx in transactions] # Return list
of transaction dicts
      })
      except Exception as e:
      logger.error(f"[GET ASSET ERROR] {e}")
      return jsonify({'error': 'Asset not found', 'details': str(e)}), 404

 @app.route('/api/assets/<string:wallet_address>')
 def get_user_assets(wallet_address):
      """
      API endpoint to retrieve all assets associated with a specific wallet
address.
      """
      try:
      user = User.query.filter_by(wallet_address=wallet_address).first()
      if not user:
            return jsonify({'assets': []}) # Return empty list if user not
found
      # Retrieve all assets for the user, ordered by creation time
      assets =
Asset.query.filter_by(user_id=user.id).order_by(Asset.created_at.desc()).all()
      return jsonify({
            'user': user.to_dict(),
            'assets': [asset.to_dict() for asset in assets] # Return list of
asset dicts
      })
      except Exception as e:
      logger.error(f"[USER ASSETS ERROR] {e}")
      return jsonify({'error': 'Failed to retrieve assets', 'details':
str(e)}), 500

 @app.route('/api/stats')
 def get_stats():
      """
      API endpoint to retrieve platform-wide statistics.
      """
      try:
      total_assets = Asset.query.count()
      total_users = User.query.count()
      verified_assets =
Asset.query.filter_by(verification_status='verified').count()
      tokenized_assets = Asset.query.filter(Asset.token_id.isnot(None)).count()

      return jsonify({
            'total_assets': total_assets,
            'total_users': total_users,
            'verified_assets': verified_assets,
            'tokenized_assets': tokenized_assets,
            # Calculate rates, handling division by zero
             'verification_rate': (verified_assets / total_assets * 100) if 
total_assets else 0,
            'tokenization_rate': (tokenized_assets / verified_assets * 100) if 
verified_assets else 0
      })
      except Exception as e:
      logger.error(f"[STATS ERROR] {e}")
      return jsonify({'error': 'Failed to retrieve stats', 'details': str(e)}), 
500

 # Run the Flask application if the script is executed directly
 if __name__ == '__main__':
      app.run(debug=True, host='0.0.0.0', port=5000)
```

Output Interpretation: Verification Scores and Next Steps for Users

The platform's output is designed to be clear, actionable, and transparent, enabling users to understand the AI's assessment and proceed confidently.

- **Verification Scores Breakdown:** The platform provides a numerical score for the overall asset's suitability for tokenization, typically on a scale (e.g., 0-100%). This overall score is further broken down into individual scores from each modular agent (e.g., Value Score, Risk Score, Consistency Score).[28] This multi-dimensional scoring provides a granular view of the asset's strengths and weaknesses.
- **Detailed Explanations:** Critically, alongside the numerical scores, the Coordinator Agent provides natural language explanations for its decision. These explanations highlight key factors that influenced the scores, identify potential red flags, and articulate the rationale behind the overall approval or rejection.[9] This detailed analysis and feedback on every section of the asset's evaluation enhance transparency and help users understand the nuances of the AI's assessment.[29]
- **Actionable Next Steps:** The platform ensures that users are never left in limbo.
  - **For Approved Assets:** Clear and concise instructions are provided on how to proceed with token minting. This includes steps like reviewing the final token specifications, confirming any necessary legal agreements, and initiating the on-chain minting process.

    - **For Rejected Assets:** Specific reasons for rejection are clearly articulated (e.g., "Risk score too high due to unverified ownership documents," "Valuation inconsistency detected between submitted data and market benchmarks"). Crucially, the platform also suggests concrete remediation steps (e.g., "Please upload certified ownership documents from a registered authority," "Provide recent appraisal reports from an accredited valuer") to guide users on how to address the issues and potentially re-submit the asset.
- **Fallback Responses:** In scenarios involving unrecoverable system errors or highly ambiguous queries, the system is designed to generate a graceful fallback response. This message informs the user of the technical difficulty and guides them to alternative support channels or suggests rephrasing their request, maintaining a helpful user experience even under adverse conditions.[9]

The detailed explanations for rejection and suggested remediation steps create a valuable feedback loop. Users can understand *why* their asset was rejected and *how* to fix it, which improves the quality of future submissions and reduces user frustration. This feedback mechanism contributes to both user satisfaction and the continuous improvement of the AI agents' accuracy and robustness, leading to a more efficient and reliable tokenization process over time.

Database & Utilities

This section details the platform's data storage and management mechanisms, focusing on the SQLite database and auxiliary utility scripts that support its operation and maintenance.

8.1 rwa_tokenization.db Schema and Usage

The platform utilizes rwa_tokenization.db as its primary data store, implemented using **SQLite 3**. SQLite is a lightweight, serverless, and file-based relational database that is embedded directly into the application.[16] This choice simplifies deployment and management, especially during development and prototyping phases.

The database serves as the persistent storage layer for all data related to the asset tokenization process. This includes:

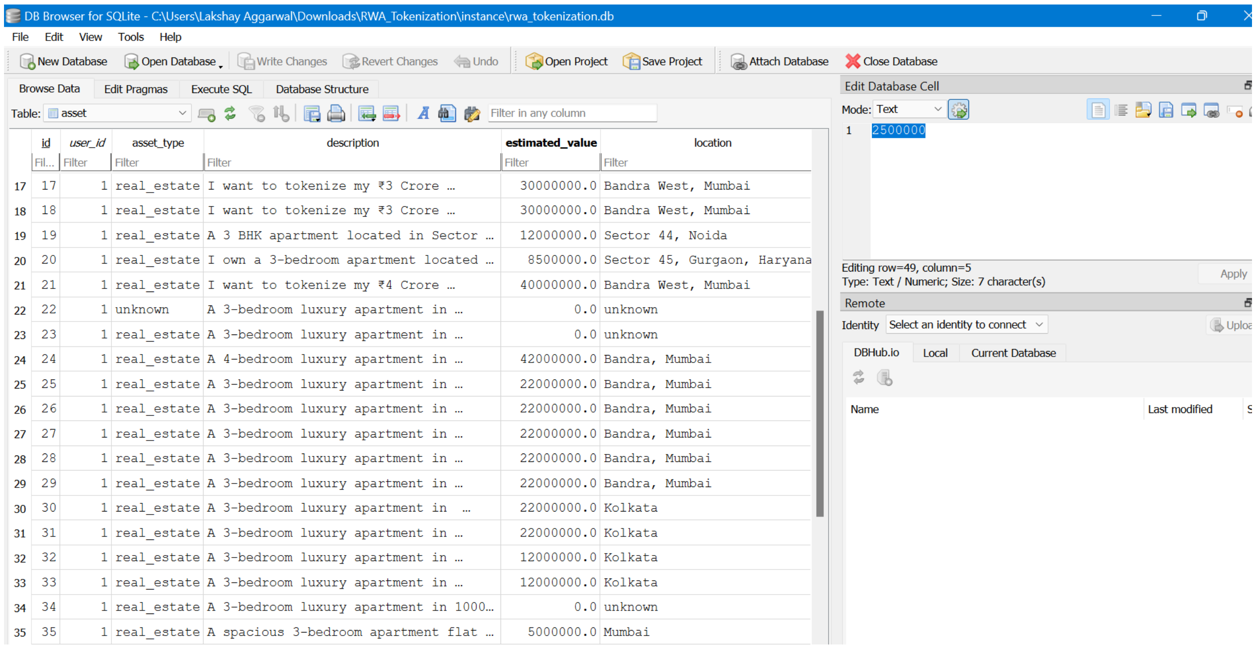
- **Asset Records:** Stores the raw submitted asset details, the structured data extracted by the LLM Parsing Agent, the current verification status (e.g., 'pending', 'verified', 'rejected', 'tokenized'), and timestamps of submission and status changes.
- **Verification Results:** Maintains detailed records of the scores and explanations provided by each modular agent (Value, Risk, Consistency, Location) and the Coordinator Agent's final decision and rationale.
- **Token Details:** Stores information about minted tokens, including the unique token ID (cryptographic hash), the comprehensive token metadata, the blockchain transaction hash, and the minting timestamp.
- **User Profiles (Optional/Future):** Can be extended to store basic user information and their submission history, enabling personalized experiences.

The database is accessed primarily via Python's built-in sqlite3 module, typically through wrapper utility functions like get_db() for connection management and query_db() for executing SQL queries and retrieving results.[15] This provides a clean abstraction layer for database interactions. The database stores not just the final tokenized asset data but also every intermediate step: raw input, parsed data, individual agent scores, and the Coordinator's decision. This makes it a comprehensive record of the entire tokenization process. This detailed record-keeping is vital for debugging, auditing (both internal and external), and fulfilling regulatory reporting requirements. It ensures that every decision made by the AI agents can be traced back to its source data and intermediate computations, supporting the platform's explainability claims.

Conceptual Schema Design (Key Tables):

- assets:
  - id (PRIMARY KEY, TEXT)
  - original_data (TEXT, JSON representation of raw submission)
  - parsed_data (TEXT, JSON representation of structured data from LLM)
  - status (TEXT, e.g., 'pending', 'parsing', 'verifying', 'verified', 'rejected', 'tokenized')
  - submission_timestamp (TEXT)
  - final_decision_timestamp (TEXT)
- verifications:
  - id (PRIMARY KEY, TEXT)
  - asset_id (FOREIGN KEY, TEXT)
  - agent_name (TEXT, e.g., 'ValueAgent', 'RiskAgent', 'CoordinatorAgent')

    - score (INTEGER)
    - explanation (TEXT)
    - timestamp (TEXT)
- tokens:
    - id (PRIMARY KEY, TEXT)
    - asset_id (FOREIGN KEY, TEXT)
    - token_id (TEXT, unique cryptographic hash)
    - metadata (TEXT, JSON representation of token metadata)
    - blockchain_tx_hash (TEXT)
    - mint_timestamp (TEXT)



1 2500000

Editing row=49, column=5
Type: Text / Numeric; Size: 7 character(s)

	id	user_id	asset_type	description	estimated_value	location
17	17	1	real_estate	I want to tokenize my ₹3 Crore …	30000000.0	Bandra West, Mumbai
18	18	1	real_estate	I want to tokenize my ₹3 Crore …	30000000.0	Bandra West, Mumbai
19	19	1	real_estate	A 3 BHK apartment located in Sector …	12000000.0	Sector 44, Noida
20	20	1	real_estate	I own a 3-bedroom apartment located …	8500000.0	Sector 45, Gurgaon, Haryana
21	21	1	real_estate	I want to tokenize my ₹4 Crore …	40000000.0	Bandra West, Mumbai
22	22	1	unknown	A 3-bedroom luxury apartment in …	0.0	unknown
23	23	1	real_estate	A 3-bedroom luxury apartment in …	0.0	unknown
24	24	1	real_estate	A 4-bedroom luxury apartment in …	42000000.0	Bandra, Mumbai
25	25	1	real_estate	A 3-bedroom luxury apartment in …	22000000.0	Bandra, Mumbai
26	26	1	real_estate	A 3-bedroom luxury apartment in …	22000000.0	Bandra, Mumbai
27	27	1	real_estate	A 3-bedroom luxury apartment in …	22000000.0	Bandra, Mumbai
28	28	1	real_estate	A 3-bedroom luxury apartment in …	22000000.0	Bandra, Mumbai
29	29	1	real_estate	A 3-bedroom luxury apartment in …	22000000.0	Bandra, Mumbai
30	30	1	real_estate	A 3-bedroom luxury apartment in …	22000000.0	Kolkata
31	31	1	real_estate	A 3-bedroom luxury apartment in …	22000000.0	Kolkata
32	32	1	real_estate	A 3-bedroom luxury apartment in …	12000000.0	Kolkata
33	33	1	real_estate	A 3-bedroom luxury apartment in …	12000000.0	Kolkata
34	34	1	real_estate	A 3-bedroom luxury apartment in 1000…	0.0	unknown
35	35	1	real_estate	A spacious 3-bedroom apartment flat …	5000000.0	Mumbai

Visualization of Database in DB Browser

4. Conclusion

This technical implementation manual has provided a comprehensive overview of the AI-driven Real World Asset Tokenization Platform. We have detailed the modular architecture, emphasizing the role of the agentic AI framework in automating data extraction, multi-faceted asset verification, and mock blockchain tokenization. The in-depth code explanations for database.py, llm_utils.py, verification_agent.py, tokenization_agent.py, and main.py illustrate the functionality and interconnections of each component.

The platform's design prioritizes scalability, maintainability, and extensibility, laying a robust foundation for future enhancements. By leveraging advanced AI capabilities, this solution significantly streamlines the RWA tokenization pipeline, addressing critical challenges related to manual processing, data standardization, and verification complexity. This automation not only enhances efficiency but also contributes to greater trust and transparency in the burgeoning RWA market.

Future work could involve integrating with actual blockchain networks, implementing more sophisticated KYC/AML procedures, expanding the range of verifiable asset types, and developing a comprehensive frontend user interface.