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

July 16, 2025

Abstract

This document provides a comprehensive technical manual for the AI-Driven Real World Asset Tokenization Platform, a sophisticated system designed to streamline the verification and tokenization of Real World Assets (RWAs). Leveraging an agentic AI framework, the platform integrates Large Language Models (LLMs) for data extraction, specialized verification agents for asset assessment, and a tokenization module for blockchain integration. This manual details the system architecture, database schema, and Python module functionalities, with accurate code snippets and explanations.

Contents

1 Introduction

The tokenization of Real World Assets (RWAs) transforms physical assets into digital tokens, enhancing liquidity and accessibility. However, traditional processes are manual and inefficient. This platform automates RWA verification and tokenization using advanced AI techniques.

1.1 Challenges in RWA Tokenization

- **Data Extraction and Standardization**: Unstructured asset data requires significant effort to processintrepsitemporal Processing.
- **Verification Complexity**: Ensuring authenticity, value, and compliance demands expertise.
- **Interoperability**: Integrating asset data with blockchain systems is challenging.
- **Scalability**: Manual processes limit throughput.

1.2 Solution Overview

The platform automates the RWA pipeline with:

- **LLM Integration**: Extracts structured data from unstructured inputs.
- **Modular Verification Agents**: Assess asset aspects (e.g., completeness, value, jurisdiction).
- **Tokenization Agent**: Generates mock blockchain tokens.
- **Centralized Database**: Manages user, asset, and transaction data.
- **Flask Backend API**: Orchestrates workflows.

2 System Architecture

The platform features a modular, scalable design:

1. **User Interface (Frontend)**: Web-based asset submission.
2. **Backend API (Flask)**: Manages requests and agent coordination.
3. **Database (SQLite)**: Stores platform data.
4. **AI Agents Layer**:
   - LLM Utility
   - Verification Agents (BasicInfoAgent, ValueAgent, etc.)
   - Tokenization Agent
5. **Blockchain Layer (Mock)**: Simulates blockchain operations.

3 Setup & Prerequisites

3.1 Environment Setup

- **Python**: Virtual environment recommended.
- **Flask**: Web framework.
- **SQLite**: Development database.

3.2 Required Packages

Install via `pip install flask python-dotenv google-generativeai flask-sqlalchemy flask-cors`.

4 Technical Implementation

4.1 Database Schema (`database.py`)

```python
from flask_sqlalchemy import SQLAlchemy
from datetime import datetime
import json

db = SQLAlchemy()

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    wallet_address = db.Column(db.String(42), unique=True, nullable=False)
    email = db.Column(db.String(120), nullable=True)
    kyc_status = db.Column(db.String(20), default='pending')
    jurisdiction = db.Column(db.String(10), nullable=True)
    created_at = db.Column(db.DateTime, default=datetime.utcnow)

class Asset(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=
        False)
    asset_type = db.Column(db.String(50), nullable=False)
    description = db.Column(db.Text, nullable=False)
    estimated_value = db.Column(db.Float, nullable=False)
    location = db.Column(db.String(100), nullable=False)
    verification_status = db.Column(db.String(20), default='pending')
    token_id = db.Column(db.String(64), nullable=True)
    requirements = db.Column(db.Text, default='{}')
    created_at = db.Column(db.DateTime, default=datetime.utcnow)
    updated_at = db.Column(db.DateTime, default=datetime.utcnow, onupdate=
        datetime.utcnow)

class Transaction(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    asset_id = db.Column(db.Integer, db.ForeignKey('asset.id'), nullable=
        False)
    transaction_type = db.Column(db.String(50), nullable=False)
    transaction_hash = db.Column(db.String(64), nullable=True)
    status = db.Column(db.String(20), nullable=False)
    details = db.Column(db.Text, nullable=True)
    created_at = db.Column(db.DateTime, default=datetime.utcnow)
```

4.1.1 Explanation

- **User**: Stores user details (e.g., wallet address). - **Asset**: Holds asset data with verification and tokenization status. - **Transaction**: Logs operations like verification and tokenization.

4.2 LLM Utilities (`llm_utils.py`)

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

load_dotenv()
GENAI_API_KEY = os.getenv("GEMINI_API_KEY")
genai.configure(api_key=GENAI_API_KEY)
model = genai.GenerativeModel("gemini-2.0-flash")

def extract_asset_info_with_llm(user_input: str) -> dict:
    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), no commas or currency symbol
    - location: City or region (e.g., 'Mumbai, India')
    - description: The original user input

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

    Return only valid JSON:
    {{
      "asset_type": "...",
      "estimated_value": ...,
      "location": "...",
      "description": "..."
    }}
    """
    try:
        response = model.generate_content(prompt)
        content = clean_llm_output(response.text.strip())
        data = json.loads(content)
        asset_type = data.get("asset_type", "unknown") or fallback_asset_type(user_input)
        return {
            "asset_type": asset_type,
            "estimated_value": float(data.get("estimated_value", 0)),
            "location": data.get("location", "unknown"),
            "description": data.get("description", user_input)
        }
    except Exception as e:
        asset_type = fallback_asset_type(user_input)
        return {
            "asset_type": asset_type,
            "estimated_value": 0,
            "location": "unknown",
            "description": user_input
        }
```

4.2.1 Explanation

Extracts structured data using Gemini LLM, with fallback logic for asset type categorization.

4.3 Verification Agent (`verification_agent.py`)

```python
from typing import Dict, List
from app.agents.agents_modular import CoordinatorAgent

class VerificationAgent:
    def __init__(self):
        self.verification_threshold = 0.7
        self.coordinator = CoordinatorAgent()

    def verify_asset(self, asset_data: Dict) -> Dict:
        try:
            verification_result = self.coordinator.verify(asset_data)
            result = {
                'overall_score': verification_result.get('overall_score',
                    0.0),
                'status': verification_result.get('status', 'pending'),
                'breakdown': verification_result.get('breakdown', {}),
                'agent_notes': verification_result.get('agent_notes', []),
                'recommendations': self._generate_recommendations(
                    verification_result),
                'next_steps': self._define_next_steps(verification_result.
                    get('status', 'pending')),
                'issues': []
            }
        except Exception as e:
            result = {
                'overall_score': 0.0,
                'status': 'error',
                'breakdown': {},
                'agent_notes': [],
                'recommendations': [],
                'next_steps': [],
                'issues': [f"Verification error: {str(e)}"]
            }
        return result
```

4.3.1 Explanation

Orchestrates verification using the ‘CoordinatorAgent‘, providing scores, status, and recommendations.

4.4 Modular Agents (`agents_modular.py`)

```python
import os
import json
import re
from dotenv import load .dotenv
import google.generativeai as genai

load_dotenv()
GENAI_API_KEY = os.getenv("GEMINI_API_KEY")
genai.configure(api_key=GENAI_API_KEY)
```

```python
llm_model = genai.GenerativeModel("gemini-2.0-flash")

class BasicInfoAgent:
    def assess(self, asset):
        prompt = f"""
        You are an AI agent checking if all basic asset information is present and complete.
        Asset fields:
        - Type: {asset.get('asset_type')}
        - Value: {asset.get('estimated_value')}
        - Location: {asset.get('location')}
        - Description: {asset.get('description')}
        Score 1.0 if all fields are present and detailed, 0.5 if some are missing, 0.0 if mostly missing. Explain.
        Respond as JSON: {{"score": float, "notes": "..."}}
        """
        return call_llm(prompt)

class ValueAgent:
    def assess(self, asset):
        prompt = f"""
        You are an AI agent evaluating if the asset's estimated value is plausible for its type and location.
        Asset fields:
        - Type: {asset.get('asset_type')}
        - Value: {asset.get('estimated_value')}
        - Location: {asset.get('location')}
        - Description: {asset.get('description')}
        Score 1.0 if value is plausible, 0.4 if too low, 0.6 if too high, 0.5 if unknown. Explain.
        Respond as JSON: {{"score": float, "notes": "..."}}
        """
        return call_llm(prompt)

class JurisdictionAgent:
    def assess(self, asset):
        prompt = f"""
        You are an AI agent verifying the jurisdiction/location of the asset.
        Asset fields:
        - Location: {asset.get('location')}
        Score 0.9 if location is specific and recognized (especially any Indian city/state/UT), 0.5 if vague or missing. Explain.
        Respond as JSON: {{"score": float, "notes": "..."}}
        """
        return call_llm(prompt)

class AssetSpecificAgent:
    def assess(self, asset):
        prompt = f"""
        You are an AI agent checking if the asset description contains type-specific details and keywords.
        Asset fields:
        - Type: {asset.get('asset_type')}
        - Description: {asset.get('description')}
        Score 1.0 if many relevant details/keywords, 0.5 if some, 0.0 if none. Explain.
        Respond as JSON: {{"score": float, "notes": "..."}}
        """
        return call_llm(prompt)

class CoordinatorAgent:
    def __init__(self):
        self.agents = [
            ("basic_info", BasicInfoAgent()),
            ("value_assessment", ValueAgent()),
            ("jurisdiction", JurisdictionAgent()),
            ("asset_specific", AssetSpecificAgent())
        ]

    def verify(self, asset):
        results = {}
        explanations = []
        for key, agent in self.agents:
            agent_result = agent.assess(asset)
            results[key] = agent_result.get("score", 0.5)
            explanations.append(f"{key}: {agent_result.get('notes', '')}")
        avg_score = sum(results.values()) / len(results)
        status = "verified" if avg_score >= 0.7 else ("requires_review" if
            avg_score >= 0.5 else "rejected")
        return {
            "overall_score": round(avg_score, 2),
            "status": status,
            "breakdown": results,
            "agent_notes": explanations
        }
```

4.4.1 *Explanation*

- **BasicInfoAgent**: Checks completeness of asset fields. - **ValueAgent**: Assesses value plausibility. - **JurisdictionAgent**: Verifies location specificity. - **AssetSpecificAgent**: Ensures type-specific details. - **CoordinatorAgent**: Aggregates agent results for final status.

4.5 Tokenization Agent (`tokenization_agent.py`)

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

class TokenizationAgent:
    def __init__(self):
        self.token_standard = "RWA-721"
        self.network = "RWA-TestNet"

    def tokenize_asset(self, asset_data: Dict, verification_result: Dict)
        -> Dict:
        if verification_result.get('status') != 'verified':
            return {
                'success': False,
                'error': 'Asset must be verified before tokenization',
                'status': 'failed'
            }
        try:
            token_metadata = self._generate_token_metadata(asset_data,
                verification_result)
            contract_data = self._create_mock_contract(asset_data,
                token_metadata)
            token_id = self._generate_token_id(asset_data)
            transaction_hash = self._generate_transaction_hash(
                contract_data)
            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(),
                'status': 'minted'
            }
        except Exception as e:
            return {
                'success': False,
                'error': f'Tokenization failed: {str(e)}',
                'status': 'failed'
            }
```

4.5.1 Explanation

Generates mock blockchain tokens for verified assets, including metadata and contract details.

4.6 Main Application (`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

sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '..
    ')))
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

app = Flask(__name__, template_folder='../templates', static_folder='../
    static')
app.config['SECRET_KEY'] = 'your-secret-key-change-this'
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///rwa_tokenization.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

db.init_app(app)
CORS(app)
```

```

verification_agent = VerificationAgent()
tokenization_agent = TokenizationAgent()

@app.route('/api/intake', methods=['POST'])
def asset_intake():
    try:
        data = request.get_json()
        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']
        parsed_data = extract_asset_info_with_llm(user_input)
        user = User.query.filter_by(wallet_address=wallet_address).first()
        if not user:
            user = User(wallet_address=wallet_address, email=data.get('
                email'))
            db.session.add(user)
            db.session.commit()
        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',
            requirements=json.dumps({})
        )
        db.session.add(asset)
        db.session.commit()
        return jsonify({'success': True, 'message': 'Asset submitted
            successfully.', 'asset': asset.to_dict()})
    except Exception as e:
        return jsonify({'error': 'Internal server error', 'details': str(e)
            }), 500
```

4.6.1 *Explanation*

The Flask app handles API requests, integrating LLM, verification, and tokenization processes.

5 Conclusion

This manual outlines a robust, AI-driven platform for RWA tokenization, featuring accurate agent implementations and code. Future enhancements could include real blockchain integration and expanded asset types.