StratoPipe

Design Document

By Eric Lalumiere

03/10/2025

Table of Contents

1. Project Overview
2. Scope
3. Requirements
4. Architecture Overview
5. Risk Considerations
6. Security Considerations
7. Unified Error Handling Workflow
8. Project Structure
9. Front-End Implementation
10. Middleware
11. Models
12. Views and Business Logic
13. Serializers
14. URLs and Routing
15. Asynchronous Task Processing
16. Rendering and Thumbnail Generation Implementation
17. AI Functionalities Implementation
18. Data Flow Diagram
19. Main Workflows' Diagrams
20. Deployment and Infrastructure
21. Future Scalability and Maintenance
22. Conclusion

1. Project Overview

StratoPipe is designed to enable remote teams to collaborate effectively on virtual production projects. The platform provides a comprehensive web-based solution for project management, asset management, task tracking, and real-time collaboration among team members. Leveraging Django for a robust backend and React for a responsive frontend, StratoPipe aims to streamline workflows in virtual production environments.
The platform includes features such as user authentication, project management, asset management, real-time collaboration, task tracking, and AI-powered functionalities like image classification for asset categorization, object detection for thumbnail enhancement, and automatic image enhancement.
The proof of concept (PoC) focuses on implementing core functionalities to demonstrate the platform's potential.

2. Scope

The scope of the StratoPipe project encompasses the following key components:

- **User Authentication**
- **Project Management**
- **Asset Management**
- **Rendering and Thumbnail Generation**
- **Real-Time Collaboration**
- **Task Tracking**
- **AI Functionalities**

3. Requirements

User Authentication

- **User Registration**: Users can create accounts with a unique username, email, and password.

- **User Login**: Users can log in with their credentials.
- **Password Security**: User passwords are encrypted and stored securely.
- **User Profile Management**: Users can update their profile information (e.g., name, email, password).

Project Management

- **Project Creation**: Users can create new projects with a project name and description.
- **Project Listing**: Users can view a list of their projects.
- **Project Details**: Users can view project details, including name, description, and creation date.
- **Project Deletion**: Users can delete projects they own.

Asset Management

- **File Upload**: Users can upload files to a project.
- **File Download**: Users can download files from a project.
- **File Categorization**: Users can categorize files (e.g., images, videos, documents, geometry).
- **File Metadata**: Store metadata for each file (e.g., filename, upload date, file type).
- **File Deletion**: Users can delete files they have uploaded.
- **Display of Thumbnails**: The user interface displays the thumbnail as a visual reference for the asset.
- **High-Resolution Render Access**: Users can click on the thumbnail to view the full-sized rendered image.
- **Browse by Category**: Users can browse assets in the user interface (UI) organized by predefined or auto-generated categories.

Rendering and Thumbnail Generation

- **Automatic Rendering**: Upon uploading a geometry asset, an asynchronous rendering process is initiated using an open-source renderer.
- **Thumbnail Generation**: A thumbnail is extracted from the rendered image for quick visual reference.
- **Asset Association**: Rendered images and thumbnails are linked to the corresponding assets in the database.

- **User Interface Integration**: Thumbnails are displayed in the UI; clicking on them opens the high-resolution render.
- **Asynchronous Processing**: Rendering tasks do not block user interaction with the application.

Real-Time Collaboration

- **Commenting System**: Users can add comments to project assets.
- **Real-Time Updates**: Use WebSockets to provide real-time updates for comments and asset rendering notifications.
- **Notifications**: Notify users of new comments and when rendering is complete in real time.

Task Tracking

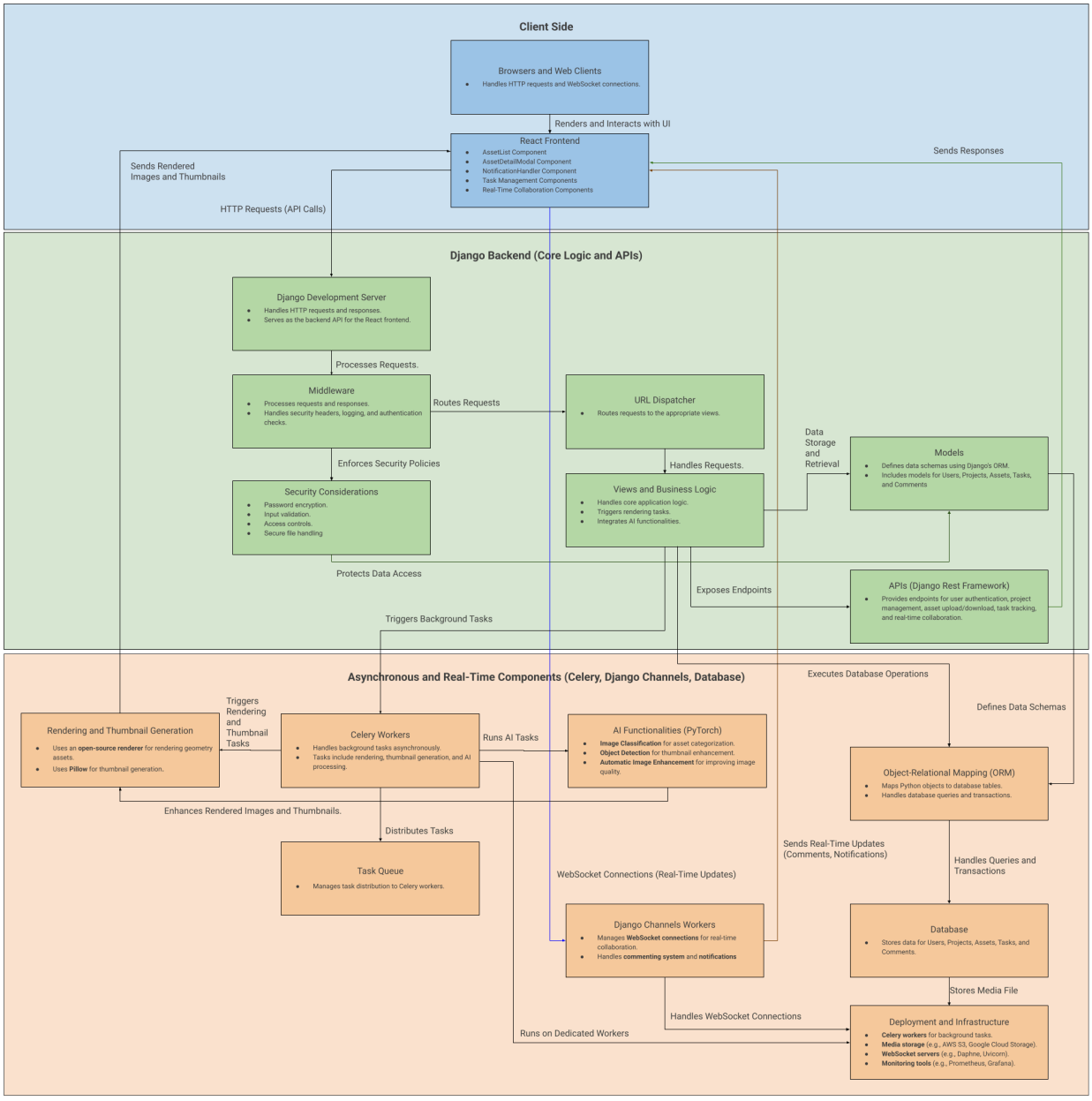
- **Task Creation**: Users can create tasks with a task name, description, and due date.
- **Task Assignment**: Users can assign tasks to team members.
- **Task Status**: Users can update the status of tasks (e.g., open, in progress, completed).
- **Task Listing**: Users can view a list of tasks within a project.
- **Task Deletion**: Users can delete tasks they have created.

AI Functionalities

- **Image Classification for Asset Categorization**: Automatically categorizes assets based on their content using AI models.
- **Object Detection for Thumbnail Enhancement**: Enhances thumbnails by focusing on key objects within images.
- **Automatic Image Enhancement**: Improves the quality of thumbnails and rendered images using AI techniques.

4. Architecture Overview

The inclusion of rendering and thumbnail generation, as well as AI functionalities using PyTorch, introduces additional components to the architecture. These new features are integrated into the existing modular monolithic structure.



5. Risk Considerations

a. Rendering Performance

- **Risk**: Rendering geometry assets may overwhelm the system, especially on free-tier cloud services or a MacBook Pro.
- **Impact**: Slow rendering times may degrade user experience.
- **Mitigation**: Optimize rendering tasks, use lightweight models, and consider offloading rendering to a more powerful machine.

b. Real-Time Collaboration

- **Risk**: Real-time collaboration features (e.g., WebSockets) may not perform well on free-tier cloud services due to resource limitations and sleep policies.
- **Impact**: Real-time updates may be delayed or fail, disrupting collaboration.
- **Mitigation**: Use low-cost paid plans for WebSocket servers and implement fallback mechanisms (e.g., polling).

c. AI Functionalities

- **Risk**: Running AI models (e.g., image classification, object detection) may require significant computational resources, which are not available on free-tier cloud services.
- **Impact**: AI functionalities may be slow or unavailable.
- **Mitigation**: Use pre-trained models, optimize AI tasks, and consider running them locally.

d. Scalability

- **Risk**: The system may not scale well beyond a small number of users or projects due to free-tier resource constraints.
- **Impact**: Performance may degrade as the user base grows.
- **Mitigation**: Design the system with scalability in mind and plan for migration to a paid cloud service if needed.

e. Security

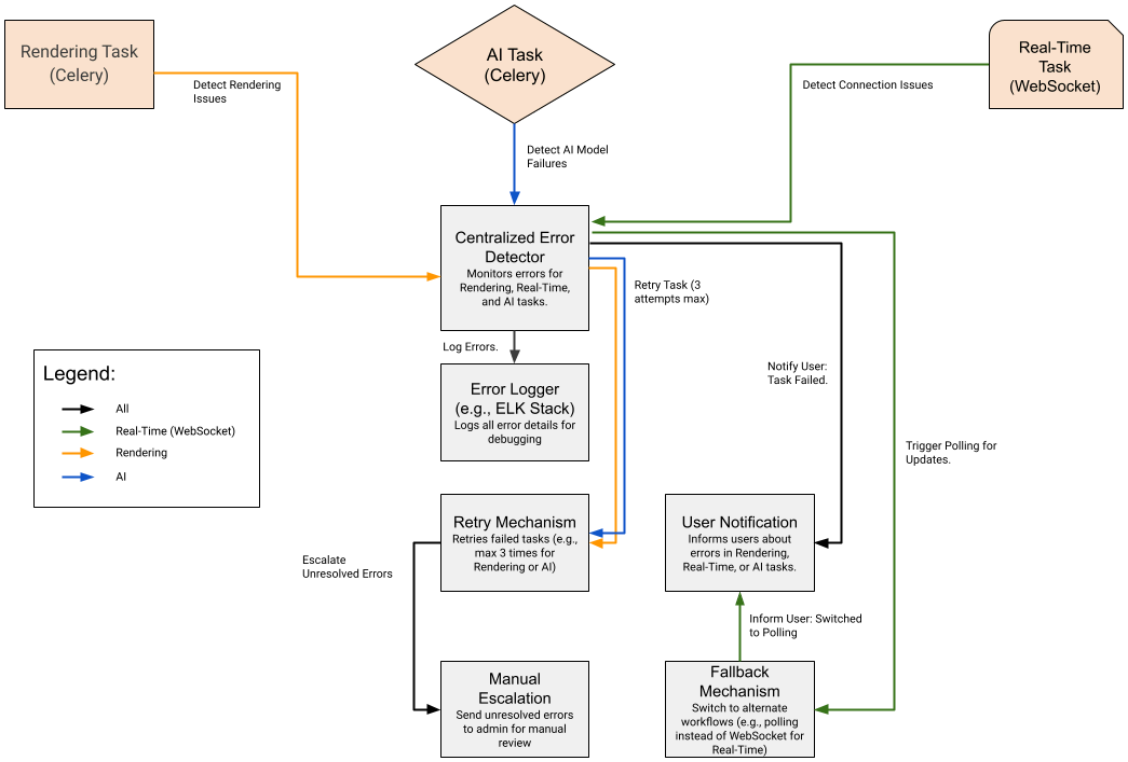
- **Risk**: Malicious file uploads or insecure WebSocket connections may compromise the system.
- **Impact**: Data breaches or system vulnerabilities.
- **Mitigation**: Validate and sanitize file uploads, use secure WebSocket connections, and implement access controls.

---

6. Security Considerations

- **Password Encryption**: Use Django's authentication system, which hashes passwords securely.
- **Input Validation**: Validate all user inputs to prevent injection attacks.
- **Access Controls**: Implement permissions and authentication checks on all views and API endpoints.
- **Data Protection**: Securely store sensitive data and ensure secure transmission over HTTPS.
- **File Handling**:
    - Validate and sanitize files uploaded by users.
    - Use antivirus scanning if necessary.
    - Restrict file types and sizes.
- **Sandboxed Rendering**: Run the rendering process in a sandboxed environment to limit system access in case of malicious geometry files.
- **Protected Media URLs**: Use signed URLs or token-based authentication to protect access to media files.
- **Session Management**: Protect against session hijacking and fixation.

7. Unified Error Handling Workflow



The **Unified Error Handling Workflow** outlines the process by which the system detects, logs, notifies, and resolves errors across three critical functionalities: **Rendering Tasks**, **Real-Time Collaboration**, and **AI Processing Tasks**. This centralized workflow ensures consistent error management, enhances user transparency, and facilitates system resilience by automating retries, escalating unresolved issues, and handling non-recoverable errors effectively.

Core Components of the Workflow

1. **Error Detection**:
    - Centralized error detection monitors failures from different components:
        - **Rendering Tasks**: Issues such as timeouts or crashes during asset rendering.
        - **Real-Time Collaboration**: Problems like WebSocket disconnections or failed updates.
        - **AI Processing Tasks**: Failures including model errors or insufficient resources.
    - Detected errors are immediately sent to the next steps for resolution.
    - **Non-Recoverable Errors**: Certain errors cannot be retried, such as:

        - **Invalid File Formats**: Users are notified immediately via the UI to upload a valid file.
        - **Network Disconnects**: Users are informed of the disconnect, and tasks are paused until reconnection is established.
2. **Error Logging**:
    - All detected errors are logged using a centralized logging mechanism (e.g., an ELK Stack).
    - Logs include comprehensive details (e.g., timestamps, error codes, affected components) for debugging and system audits.
3. **User Notification**:
    - Users are notified in real-time about failures, with specific error details:
        - **Rendering**: Notifications indicate whether rendering has failed or requires retry.
        - **Real-Time Collaboration**: Users are alerted if the system switches to fallback mechanisms like polling.
        - **AI Processing**: Failed tasks trigger notifications, with suggestions to retry or seek support.
4. **Retry Mechanism**:
    - Automated retries are initiated for recoverable errors:
        - **Rendering Tasks**: Retries are limited to a maximum of three attempts.
        - **AI Processing Tasks**: Similarly, retries are capped at three to prevent infinite loops.
    - Each retry is logged for audit purposes.
5. **Fallback Mechanism**:
    - For real-time collaboration errors (e.g., WebSocket disconnections), the system activates additional fallback mechanisms:
        - **Switching to Polling**: HTTP polling ensures functionality continues.
        - **Buffering Unsaved Comments**: Comments are temporarily stored and synced once reconnection is established.
6. **Failure Escalation**:
    - Errors unresolved after retries or fallback mechanisms are escalated to administrators for manual intervention.
    - Escalation ensures that critical issues are promptly addressed, preventing prolonged disruptions.

Flow Summary

The workflow begins with the detection of errors in the respective components (Rendering, Real-Time Collaboration, or AI Processing). Detected errors are logged and categorized. Users are immediately notified, especially for non-recoverable errors like invalid file formats or network disconnects. The system initiates retries for recoverable errors or switches to fallback mechanisms for real-time collaboration. Persistent issues escalate to administrators for manual resolution.

Impact on System Reliability

The **Unified Error Handling Workflow** enhances the platform's reliability and user experience by:

- Minimizing disruptions for end-users through **automated recovery mechanisms**.
- Enhancing transparency with **real-time notifications and detailed error logs**.
- Handling **non-recoverable errors** effectively with immediate feedback.
- Ensuring swift resolution of persistent issues via **failure escalation**.
- Providing additional **fallback mechanisms** like comment buffering during real-time collaboration errors.

By integrating these robust mechanisms, the workflow ensures a seamless user experience and supports the platform’s focus on efficiency, transparency, and resilience.

8. Project Structure

The project structure includes the integration of rendering and thumbnail generation, as well as AI functionalities within existing components.

```
  StratoPipe/
├── manage.py
├── stratopipe/
│   ├── __init__.py
│   ├── asgi.py                     # ASGI configuration for
WebSocket support
│   ├── celery.py                   # Celery configuration for
asynchronous tasks
│   ├── middleware.py               # Custom middleware for security,
logging, etc.
│   ├── settings.py                 # Centralized project settings
│   ├── urls.py                     # Root URL dispatcher
│   ├── wsgi.py
│
├── authentication/                 # Handles user authentication and
profile management
│   ├── __init__.py
│   ├── models.py
│   ├── serializers.py
│   ├── views.py
│   ├── urls.py
│
├── projects/                       # Manages project entities
│   ├── __init__.py
│   ├── models.py
│   ├── serializers.py
│   ├── views.py
│   ├── urls.py
│
├── assets/                         # Core functionalities for asset
handling
│   ├── __init__.py
│   ├── admin.py
```

```
|   ├── apps.py
|   ├── models.py                    # Updated to include rendering
and AI fields
|   ├── serializers.py
|   ├── tasks.py                     # Rendering, thumbnail
generation, and AI processing tasks
|   ├── tests.py
|   ├── views.py                     # Manages rendering triggers and
AI integration
|   ├── urls.py
|
├── collaboration/                   # Handles real-time collaboration
features
|   ├── __init__.py
|   ├── consumers.py                 # WebSocket consumers for
real-time notifications
|   ├── models.py                    # Comment model for collaborative
interactions
|   ├── routing.py                   # WebSocket routing configuration
|   ├── urls.py
|   ├── views.py
|
├── tasks/                           # Centralized management for
background tasks
|   ├── __init__.py
|   ├── tasks.py                     # Includes workflow-specific task
implementations
|
├── frontend/                        # React-based frontend interface
|   ├── src/
|       ├── components/
|           ├── AssetList.js         # Asset gallery with thumbnail
previews
|           ├── AssetDetailModal.js  # Detailed view of assets,
including renders
|       ├── notifications/
|           ├── NotificationHandler.js # Handles real-time updates via
WebSockets
```

```
|       ├── App.js                    # Main application entry point
|   ├── public/
|   ├── package.json
|
```

9. Front-End Implementation

Asset Components

AssetList Component

Purpose:

- The `AssetList` component displays a list of assets with thumbnails and allows users to interact with them.
- It fetches asset data from the backend and renders it in a gallery format.

Key Features:

- **Fetching Assets**: The component uses `axios` to fetch asset data from the backend API (`/api/assets/`).
- **Thumbnail Display**: Each asset's thumbnail is displayed, and users can click on it to view the high-resolution render.
- **Modal Interaction**: When a thumbnail is clicked, the `AssetDetailModal` component is displayed to show detailed information about the asset.

File: `frontend/src/components/AssetList.js`

jsx

```
import React, { useState, useEffect } from 'react';
import axios from 'axios';
import AssetDetailModal from './AssetDetailModal';

function AssetList({ updatedAsset }) {
    const [assets, setAssets] = useState([]);
    const [selectedAsset, setSelectedAsset] = useState(null);

    useEffect(() => {
        const fetchAssets = async () => {
            try {
```

```
                const response = await axios.get('/api/assets/');
                setAssets(response.data);
            } catch (error) {
                console.error('Error fetching assets:', error);
            }
        };
        fetchAssets();
    }, [updatedAsset]);

    const handleThumbnailClick = (asset) => {
        if (asset.is_rendered || asset.rendered_image) {
            setSelectedAsset(asset);
        } else {
            alert('Render is not yet available. Please try again later.');
        }
    };

    const closeModal = () => {
        setSelectedAsset(null);
    };

    return (
        <div>
            <h1>Assets</h1>
            <div className="asset-gallery">
                {assets.map(asset => (
                    <div key={asset.id} className="asset-item">
                        {asset.thumbnail ? (
                            <img
                                src={asset.thumbnail}
                                alt={asset.name}
                                onClick={() => handleThumbnailClick(asset)}
                            />
                        ) : (
                            <div className="placeholder">No Thumbnail</div>
                        )}
                        <p>{asset.name}</p>
                    </div>
                ))}
            </div>
            {selectedAsset && (
                <AssetDetailModal asset={selectedAsset} onClose={closeModal} />
            )}
        </div>
    );
}

export default AssetList;
```

Explanation:

- **useEffect**: Fetches asset data from the backend when the component mounts or when `updatedAsset` changes.
- **handleThumbnailClick**: Handles user interaction when a thumbnail is clicked. If the asset is rendered, it opens the `AssetDetailModal`; otherwise, it shows an alert.
- **AssetDetailModal**: Displays detailed information about the selected asset.

AssetDetailModal Component

Purpose:

- The `AssetDetailModal` component displays detailed information about an asset, including the high-resolution render and AI-enhanced details.
- It is triggered when a user clicks on a thumbnail in the `AssetList` component.

Key Features:

- **High-Resolution Render**: Displays the full-sized rendered image if available.
- **Asset Details**: Shows the asset's name, description, categories, and AI enhancement status.
- **Close Button**: Allows users to close the modal and return to the asset list.
- **Rendering Status**: If the asset is not yet rendered, it displays a "Rendering in progress..." message.

File: `frontend/src/components/AssetDetailModal.js`

jsx

```
import React from 'react';

function AssetDetailModal({ asset, onClose }) {
    return (
        <div className="modal-overlay" onClick={onClose}>
            <div className="modal-content" onClick={e => e.stopPropagation()}>
                {asset.rendered_image ? (
                    <img src={asset.rendered_image} alt={asset.name} />
                ) : (
                    <p>Rendering in progress...</p>
                )}
                <h2>{asset.name}</h2>
                <p>{asset.description}</p>
                {asset.categories && <p>Categories: {asset.categories}</p>}
                {asset.ai_enhanced && <p>Image enhanced using AI</p>}
                <button onClick={onClose}>Close</button>
            </div>
        </div>
```

```
    );
}

export default AssetDetailModal;
```

Explanation:

- **Modal Overlay**: The modal is displayed as an overlay on top of the asset list.
- **Rendered Image**: Displays the high-resolution render if available; otherwise, shows a "Rendering in progress" message.
- **Asset Information**: Displays the asset's name, description, categories, and AI enhancement status.
- **Close Button**: Allows users to close the modal and return to the asset list.

Real-Time Notifications

Real-Time Collaboration Details

The real-time collaboration features in StratoPipe are implemented using Django Channels and WebSockets. These features enable users to interact with project assets in real time, including adding comments and receiving notifications about rendering progress and new comments.

Commenting System

- **Purpose**: Allow users to add comments to project assets and view comments from other team members in real time.
- **Implementation:**
    - Users can add comments to any asset within a project.
    - Comments are broadcast to all users viewing the same asset in real time using WebSockets.
    - Each comment is associated with an asset and includes the author, content, and timestamp.

Comment Model

The Comment model stores comments and their metadata.

Python

```
from django.db import models
from django.contrib.auth.models import User
```

```
from assets.models import Asset

class Comment(models.Model):
    asset = models.ForeignKey(Asset, on_delete=models.CASCADE, related_name='comments')
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    content = models.TextField()
    timestamp = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f"{self.author.username} - {self.asset.name}"
```

Real-Time Notifications

- **Purpose**: Notify users in real time about events such as rendering completion and new comments.
- **Implementation**:
    - **Asset Rendering Notifications**: Users are notified when rendering of an asset is complete.
    - **Comment Notifications**: Users are notified when a new comment is added to an asset they are viewing.
    - Notifications are sent via WebSockets to ensure real-time updates.

NotificationHandler Component (Frontend)

Purpose:

- The `NotificationHandler` component listens for **WebSocket messages** and updates the UI in real time.
- It handles notifications for events like rendering completion and new comments.

Key Features:

- **WebSocket Connection**: Establishes a WebSocket connection to the backend for real-time updates.
- **Real-Time Updates**: Listens for messages (e.g., `asset_rendered`) and triggers UI updates.
- **Cleanup**: Closes the WebSocket connection when the component is unmounted.

File: `frontend/src/notifications/NotificationHandler.js`

jsx

```
import { useEffect } from 'react';

function NotificationHandler({ userId, onAssetRendered, onNewComment }) {
    useEffect(() => {
```

```
        const ws = new WebSocket('ws://localhost:8080/ws/notifications/');

        ws.onopen = () => {
            console.log('WebSocket connected');
        };

        ws.onmessage = (event) => {
            const data = JSON.parse(event.data);
            if (data.type === 'asset_rendered') {
                // Notify the parent component that rendering is complete
                onAssetRendered(data.asset_id);
            } else if (data.type === 'new_comment') {
                // Notify the parent component about a new comment
                onNewComment(data.comment);
            }
        };

        ws.onclose = () => {
            console.log('WebSocket disconnected');
        };

        // Cleanup WebSocket connection on unmount
        return () => {
            ws.close();
        };
    }, [userId, onAssetRendered, onNewComment]);

    return null;
}

export default NotificationHandler;
```

Explanation:

- **useEffect**: Establishes a WebSocket connection when the component mounts and cleans it up when the component unmounts.
- **onmessage**: Listens for WebSocket messages and triggers the `onAssetRendered` callback when rendering is complete.
- **onAssetRendered**: Updates the UI to reflect the new rendering status.
- **onNewComment**: Updates the UI to display the new comment in real time.

WebSocket Consumer for Notifications (Backend)

The NotificationConsumer handles WebSocket connections and broadcasts real-time notifications to users.
File: collaboration/consumers.py

Python

```
from channels.generic.websocket import AsyncWebsocketConsumer
import json

class NotificationConsumer(AsyncWebsocketConsumer):
    async def connect(self):
        self.user = self.scope['user']
        if self.user.is_authenticated:
            # Join a group corresponding to the user
            await self.channel_layer.group_add(f"user_{self.user.id}", self.channel_name)
            await self.accept()
        else:
            await self.close()

    async def disconnect(self, close_code):
        await self.channel_layer.group_discard(f"user_{self.user.id}", self.channel_name)

    async def receive(self, text_data):
        # Handle incoming messages if needed
        pass

    async def send_notification(self, event):
        # Send notification to WebSocket
        await self.send(text_data=json.dumps(event['content']))
```

Routing

WebSocket routing is configured to direct WebSocket connections to the `NotificationConsumer`.
File: `collaboration/routing.py`

Python

```
from django.urls import re_path
from .consumers import NotificationConsumer

websocket_urlpatterns = [
    re_path(r'ws/notifications/$', NotificationConsumer.as_asgi()),
]
```

Usage in Main App Component

Purpose:

- The `App` component is the main entry point for the frontend application.

- It integrates the `AssetList` and `NotificationHandler` components and manages the overall state of the application.

Key Features:

- **State Management**: Manages the `updatedAsset` state, which is used to trigger UI updates when rendering is complete.
- **Component Integration**: Combines the `AssetList` and `NotificationHandler` components to create the full user interface.

File: `frontend/src/App.js`

jsx

```
import React, { useState } from 'react';
import NotificationHandler from './notifications/NotificationHandler';
import AssetList from './components/AssetList';

function App() {
    const [updatedAsset, setUpdatedAsset] = useState(null);
    const [newComment, setNewComment] = useState(null);

    // Handle rendering completion notifications
    const handleAssetRendered = (assetId) => {
        setUpdatedAsset(assetId);
    };

    // Handle new comment notifications
    const handleNewComment = (comment) => {
        setNewComment(comment);
    };

    return (
        <div>
            <NotificationHandler
                onAssetRendered={handleAssetRendered}
                onNewComment={handleNewComment}
            />
            <AssetList updatedAsset={updatedAsset} newComment={newComment} />
        </div>
    );
}

export default App;
```

Explanation:

- useState: Manages the updatedAsset and newComment states, which trigger UI updates when rendering is complete or a new comment is added.

- NotificationHandler: Listens for WebSocket messages and triggers the appropriate callbacks (handleAssetRendered and handleNewComment).
- AssetList: Displays the updated asset list and new comments in real time.

10. Middleware

Custom middleware handles cross-cutting concerns such as:

- **Security Headers**: Protect against common web vulnerabilities.
  - `X-Content-Type-Options`: Prevents browsers from interpreting files as a different MIME type.
  - `X-Frame-Options`: Prevents the page from being embedded in an iframe, protecting against clickjacking attacks.
  - `Content-Security-Policy`: Restricts the sources from which content can be loaded, preventing cross-site scripting (XSS) attacks.
- **Logging and Monitoring**: Track requests and responses for debugging and analytics.
- **Authentication Checks**: Ensure secure access to resources.

File: `stratopipe/middleware.py`

python

```
class SecurityHeadersMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        # Request processing
        response = self.get_response(request)
        # Response processing
        response['X-Content-Type-Options'] = 'nosniff'
        response['X-Frame-Options'] = 'DENY'
        response['Content-Security-Policy'] = "default-src 'self'"
        return response
```

11. Models

User Model

Utilizes Django's built-in `User` model for authentication.

Project Model

- **Purpose**: The `Project` model represents a project in the system.
- **Fields**:
  - `name`: The name of the project (max 100 characters).
  - `description`: A description of the project (optional).
  - `owner`: A foreign key to the `User` model, indicating who owns the project.
  - `created_at`: The date and time the project was created (automatically set).
- **Usage**: This model is used to store and manage project data in the database.

File: `projects/models.py`

python
```
from django.db import models
from django.contrib.auth.models import User

class Project(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField(blank=True)
    owner = models.ForeignKey(User, on_delete=models.CASCADE, related_name='projects')
    created_at = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.name
```

Asset Model (Updated with Rendering and AI Fields)

- **Purpose**: The `Asset` model represents an asset (e.g., image, video, geometry file) in the system.
- **Fields**:
  - `name`: The name of the asset (max 100 characters).
  - `description`: A description of the asset (optional).
  - `asset_type`: The type of asset (e.g., image, video, geometry).
  - `file`: The uploaded file associated with the asset.
  - `owner`: A foreign key to the `User` model, indicating who uploaded the asset.

    - `project`: A foreign key to the `Project` model, indicating which project the asset belongs to.
    - `uploaded_at`: The date and time the asset was uploaded (automatically set).
    - `rendered_image`: The rendered image (for geometry assets).
    - `thumbnail`: The thumbnail image (for quick visual reference).
    - `is_rendered`: A flag indicating whether rendering is complete.
    - `categories`: Categories assigned to the asset (e.g., via AI classification).
    - `ai_enhanced`: A flag indicating whether the image has been enhanced using AI.
- **Usage**: This model is used to store and manage asset data in the database.

File: `assets/models.py`

python

```
from django.db import models
from django.contrib.auth.models import User
from projects.models import Project

class Asset(models.Model):
    ASSET_TYPES = [
        ('image', 'Image'),
        ('video', 'Video'),
        ('document', 'Document'),
        ('geometry', 'Geometry'),
        # Other asset types
    ]

    name = models.CharField(max_length=100)
    description = models.TextField(blank=True)
    asset_type = models.CharField(max_length=20, choices=ASSET_TYPES)
    file = models.FileField(upload_to='assets/')
    owner = models.ForeignKey(User, on_delete=models.CASCADE)
    project = models.ForeignKey(Project, on_delete=models.CASCADE, related_name='assets')
    uploaded_at = models.DateTimeField(auto_now_add=True)
    rendered_image = models.ImageField(upload_to='rendered_images/', null=True, blank=True)
    thumbnail = models.ImageField(upload_to='thumbnails/', null=True, blank=True)
    is_rendered = models.BooleanField(default=False)
    categories = models.CharField(max_length=255, blank=True)  # For image classification
categories
    ai_enhanced = models.BooleanField(default=False)          # Indicates if image enhancement
is applied

    def __str__(self):
        return self.name
```

Task Model

- **Purpose**: The `Task` model represents a task in the system.
- **Fields**:
    - `name`: The name of the task (max 100 characters).
    - `description`: A description of the task.
    - `assigned_to`: A foreign key to the `User` model, indicating who the task is assigned to (optional).
    - `created_by`: A foreign key to the `User` model, indicating who created the task.
    - `project`: A foreign key to the `Project` model, indicating which project the task belongs to.
    - `status`: The status of the task (e.g., open, in progress, completed).
    - `due_date`: The due date of the task (optional).
    - `created_at`: The date and time the task was created (automatically set).
- **Usage**: This model is used to store and manage task data in the database.

File: `tasks/models.py`

python
```
from django.db import models
from django.contrib.auth.models import User
from projects.models import Project

class Task(models.Model):
    STATUS_CHOICES = [
        ('open', 'Open'),
        ('in_progress', 'In Progress'),
        ('completed', 'Completed'),
    ]

    name = models.CharField(max_length=100)
    description = models.TextField()
    assigned_to = models.ForeignKey(User, on_delete=models.SET_NULL, null=True, blank=True,
related_name='assigned_tasks')
    created_by = models.ForeignKey(User, on_delete=models.CASCADE, related_name='created_tasks')
    project = models.ForeignKey(Project, on_delete=models.CASCADE, related_name='tasks')
    status = models.CharField(max_length=20, choices=STATUS_CHOICES, default='open')
    due_date = models.DateField(null=True, blank=True)
    created_at = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.name
```

Comment Model

- **Purpose**: The `Comment` model represents a comment on an asset.
- **Fields**:
    - `asset`: A foreign key to the `Asset` model, indicating which asset the comment belongs to.
    - `author`: A foreign key to the `User` model, indicating who wrote the comment.
    - `content`: The content of the comment.
    - `timestamp`: The date and time the comment was created (automatically set).
- **Usage**: This model is used to store and manage comment data in the database.

File: `collaboration/models.py`

python
```
from django.db import models
from django.contrib.auth.models import User
from assets.models import Asset

class Comment(models.Model):
    asset = models.ForeignKey(Asset, on_delete=models.CASCADE, related_name='comments')
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    content = models.TextField()
    timestamp = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f"{self.author.username} - {self.asset.name}"
```

12. Views and Business Logic

AssetUploadView and AssetDetailView

- **Purpose**: The `AssetUploadView` handles the upload of assets.
- **Functionality**:
    - Authenticated users can upload assets.
    - If the asset is a geometry file, a rendering task is triggered asynchronously using Celery.
    - For other asset types, AI processing tasks are triggered.
- **Usage**: This view is used to upload assets and initiate rendering or AI processing.

Asset Upload and Rendering Trigger

File: `assets/views.py`

python
```
from rest_framework import generics, status
from rest_framework.permissions import IsAuthenticated
from rest_framework.response import Response
from .models import Asset
from .serializers import AssetSerializer
from .tasks import render_asset, process_asset_ai

class AssetUploadView(generics.CreateAPIView):
    queryset = Asset.objects.all()
    serializer_class = AssetSerializer
    permission_classes = [IsAuthenticated]

    def perform_create(self, serializer):
        asset = serializer.save(owner=self.request.user)
        if asset.asset_type == 'geometry':
            # Trigger rendering task
            render_asset.delay(asset.id)
        else:
            # Trigger AI processing tasks for non-geometry assets
            process_asset_ai.delay(asset.id)
```

Asset Detail View

- **Purpose**: The `AssetDetailView` retrieves detailed information about an asset.
- **Functionality**:
    - Authenticated users can view details of their assets.
    - The view filters assets to ensure users can only access their own assets.
- **Usage**: This view is used to display asset details, including rendered images and thumbnails.

python
```
class AssetDetailView(generics.RetrieveAPIView):
    queryset = Asset.objects.all()
    serializer_class = AssetSerializer
    permission_classes = [IsAuthenticated]

    def get_queryset(self):
        return Asset.objects.filter(owner=self.request.user)
```

13. Serializers

Asset Serializer

- **Purpose**: The `AssetSerializer` converts `Asset` model instances to JSON and vice versa.
- **Fields**:
    - Includes all relevant fields from the `Asset` model.
    - Marks certain fields as read-only (e.g., `rendered_image`, `thumbnail`, `is_rendered`).
- **Usage**: This serializer is used in API views to handle asset data.

File: `assets/serializers.py`

python
```
from rest_framework import serializers
from .models import Asset

class AssetSerializer(serializers.ModelSerializer):
    class Meta:
        model = Asset
        fields = [
            'id', 'name', 'description', 'asset_type', 'file',
            'rendered_image', 'thumbnail', 'is_rendered',
            'owner', 'project', 'uploaded_at', 'categories', 'ai_enhanced'
        ]
        read_only_fields = ['rendered_image', 'thumbnail', 'is_rendered', 'owner', 'uploaded_at',
'categories', 'ai_enhanced']
```

14. URLs and Routing

Asset URLs

File: `assets/urls.py`

python
```
from django.urls import path
from .views import AssetUploadView, AssetDetailView, AssetListView

urlpatterns = [
    path('', AssetListView.as_view(), name='asset-list'),
    path('upload/', AssetUploadView.as_view(), name='asset-upload'),
    path('<int:pk>/', AssetDetailView.as_view(), name='asset-detail'),
```

```
]
```

15. Asynchronous Task Processing

Celery Configuration

File: `stratopipe/celery.py`

python
```
import os
from celery import Celery

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'stratopipe.settings')

app = Celery('stratopipe')
app.config_from_object('django.conf:settings', namespace='CELERY')
app.autodiscover_tasks()
```

Rendering and Thumbnail Generation Tasks

- **Purpose**: The `render_asset` task handles the rendering of geometry assets and the generation of thumbnails.
- **Functionality**:
    - Calls an open-source renderer to generate a rendered image from the geometry file.
    - Creates a thumbnail from the rendered image using the Pillow library.
    - Saves the rendered image and thumbnail to the asset.
    - Updates the asset's `is_rendered` flag and notifies the user via WebSocket.
- **Usage**: This task is triggered asynchronously when a geometry asset is uploaded.

File: `assets/tasks.py`

python
```
from celery import shared_task
from .models import Asset
import subprocess
from django.core.files import File
```

```
from django.conf import settings
import os
from PIL import Image

@shared_task
def render_asset(asset_id):
    asset = Asset.objects.get(id=asset_id)
    geometry_file_path = asset.file.path
    rendered_image_path = os.path.join(settings.MEDIA_ROOT, f"rendered_images/{asset.id}.png")
    thumbnail_path = os.path.join(settings.MEDIA_ROOT, f"thumbnails/{asset.id}_thumb.png")

    try:
        # Call the open-source renderer
        subprocess.run(['renderer', '--input', geometry_file_path, '--output',
rendered_image_path], check=True)

        # Open the rendered image and create a thumbnail
        image = Image.open(rendered_image_path)
        image.thumbnail((200, 200))
        image.save(thumbnail_path)

        # Save rendered image and thumbnail to the asset
        with open(rendered_image_path, 'rb') as img_file:
            asset.rendered_image.save(f"{asset.id}.png", File(img_file), save=False)
        with open(thumbnail_path, 'rb') as thumb_file:
            asset.thumbnail.save(f"{asset.id}_thumb.png", File(thumb_file), save=False)

        asset.is_rendered = True
        asset.save()

        # Notify the user via WebSocket
        # Code to send a notification to the client

    except Exception as e:
        # Handle errors
        print(f"Rendering failed for asset {asset.id}: {e}")
        asset.is_rendered = False
        asset.save()
```

16. Rendering and Thumbnail Generation Implementation

Workflow

1. **Asset Upload**:

    - User uploads a geometry asset via the `AssetUploadView`.
    - Asset is saved to the database.
2. **Rendering Task Triggered**:
    - If the asset type is 'geometry', the `render_asset` Celery task is triggered asynchronously.
3. **Rendering Process**:
    - The `render_asset` task uses the open-source renderer to generate a rendered image from the geometry file.
4. **Thumbnail Extraction**:
    - A thumbnail is created from the rendered image using the Pillow (PIL) library.
    - The thumbnail is saved to the asset's `thumbnail` field.
5. **Asset Association**:
    - The rendered image and thumbnail are linked to the corresponding asset in the database.
    - The `is_rendered` flag is set to `True`.
6. **User Interface Integration**:
    - The front-end displays the thumbnail in the asset list or gallery.
    - Clicking on the thumbnail opens the high-resolution rendered image.
7. **Asynchronous Processing**:
    - The rendering task runs in the background, ensuring that the user interface remains responsive.
    - Users are notified in real time when the rendering is complete via WebSockets.

Front-End Interaction

- **Displaying Thumbnails**:
    - The `AssetList` component retrieves asset data, including the thumbnail URL, and displays it.
    - If the `is_rendered` flag is `True`, the thumbnail is clickable.
- **Viewing High-Resolution Render**:
    - When the user clicks on a thumbnail, the `AssetDetailModal` component displays the high-resolution rendered image.
- **Real-Time Notifications**:
    - The `NotificationHandler` component listens for WebSocket messages indicating that rendering is complete.
    - Updates the UI accordingly to reflect the new rendering status.

Error Handling

- If rendering fails, appropriate error messages are logged.
- The `is_rendered` flag remains `False`, and the UI can inform the user that rendering is not available.

17. AI Functionalities Implementation

The **AI Functionalities** in StratoPipe leverage **PyTorch**, a popular deep learning framework, to provide advanced image processing capabilities. These functionalities include:

1. Image Classification for Asset Categorization.
2. Object Detection for Thumbnail Enhancement.
3. Automatic Image Enhancement.

Each of these functionalities is implemented asynchronously using **Celery** to ensure that the user interface remains responsive.

a. Image Classification for Asset Categorization

Purpose:

- Automatically categorize assets (e.g., images, videos) based on their content.
- Simplify asset organization by assigning categories without manual input.

Implementation:

- A pre-trained **Convolutional Neural Network (CNN)** model (e.g., ResNet, VGG) is used for image classification.
- The model analyzes the content of uploaded files and assigns one or more categories (e.g., "landscape", "portrait", "architecture").

Python

```
import torch
from torchvision import models, transforms
from PIL import Image

# Load a pre-trained ResNet model
model = models.resnet50(pretrained=True)
model.eval()

# Define image preprocessing
preprocess = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
```

```
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

def classify_image(image_path):
    # Load and preprocess the image
    image = Image.open(image_path)
    image = preprocess(image).unsqueeze(0)

    # Perform inference
    with torch.no_grad():
        output = model(image)
        _, predicted = torch.max(output, 1)

    # Map the predicted class to a category
    categories = ["landscape", "portrait", "architecture"]  # Example categories
    return categories[predicted.item()]
```

Workflow:

- When a user uploads an image, the `classify_image` function is called asynchronously via Celery.
- The image is preprocessed and passed through the pre-trained model.
- The model predicts the category, which is then saved to the asset's `categories` field in the database.

---

b. Object Detection for Thumbnail Enhancement

Purpose:

- Identify key objects in an image and adjust the thumbnail to focus on these objects.
- Enhance thumbnails by centering or highlighting important elements.

Implementation:

- A pre-trained **object detection model** (e.g., YOLO, Faster R-CNN) is used to detect objects in the image.
- The bounding box of the most prominent object is used to crop and center the thumbnail.

Python

```
from torchvision.models.detection import fasterrcnn_resnet50_fpn
from torchvision.transforms.functional import to_tensor
from PIL import Image
```

```
# Load a pre-trained Faster R-CNN model
model = fasterrcnn_resnet50_fpn(pretrained=True)
model.eval()

def enhance_thumbnail(image_path):
    # Load the image
    image = Image.open(image_path)
    image_tensor = to_tensor(image).unsqueeze(0)

    # Perform object detection
    with torch.no_grad():
        predictions = model(image_tensor)

    # Get the bounding box of the most prominent object
    boxes = predictions[0]['boxes']
    scores = predictions[0]['scores']
    if len(boxes) > 0:
        best_box = boxes[torch.argmax(scores)].tolist()
        # Crop the image to the bounding box
        cropped_image = image.crop(best_box)
        return cropped_image
    return image
```

Workflow:

- When a thumbnail is generated, the `enhance_thumbnail` function is called asynchronously via Celery.
- The object detection model identifies the most prominent object in the image.
- The image is cropped to focus on the detected object, and the enhanced thumbnail is saved.

---

c. Automatic Image Enhancement

Purpose:

- Improve the quality of thumbnails and rendered images using AI.
- Enhance visual appeal by adjusting brightness, contrast, and sharpness.

Implementation:

- A **Generative Adversarial Network (GAN)** or **image enhancement model** (e.g., ESRGAN) is used to enhance image quality.

- The model processes the image and applies enhancements to improve clarity and visual appeal.

python

```
import torch
from torchvision import transforms
from PIL import Image

# Load a pre-trained image enhancement model (e.g., ESRGAN)
# Note: This is a placeholder; the actual model loading code will depend on the model used.
enhancement_model = torch.hub.load('example/esrgan', 'esrgan')

def enhance_image(image_path):
    # Load the image
    image = Image.open(image_path)
    image_tensor = transforms.ToTensor()(image).unsqueeze(0)

    # Perform image enhancement
    with torch.no_grad():
        enhanced_image_tensor = enhancement_model(image_tensor)

    # Convert the enhanced image back to a PIL image
    enhanced_image = transforms.ToPILImage()(enhanced_image_tensor.squeeze(0))
    return enhanced_image
```

Workflow:

- When a rendered image or thumbnail is generated, the `enhance_image` function is called asynchronously via Celery.
- The image enhancement model processes the image and applies improvements.
- The enhanced image is saved and displayed in the user interface.

Integration with Celery

All AI functionalities are implemented as **Celery tasks** to ensure they run asynchronously without blocking the main application. For example:

Python

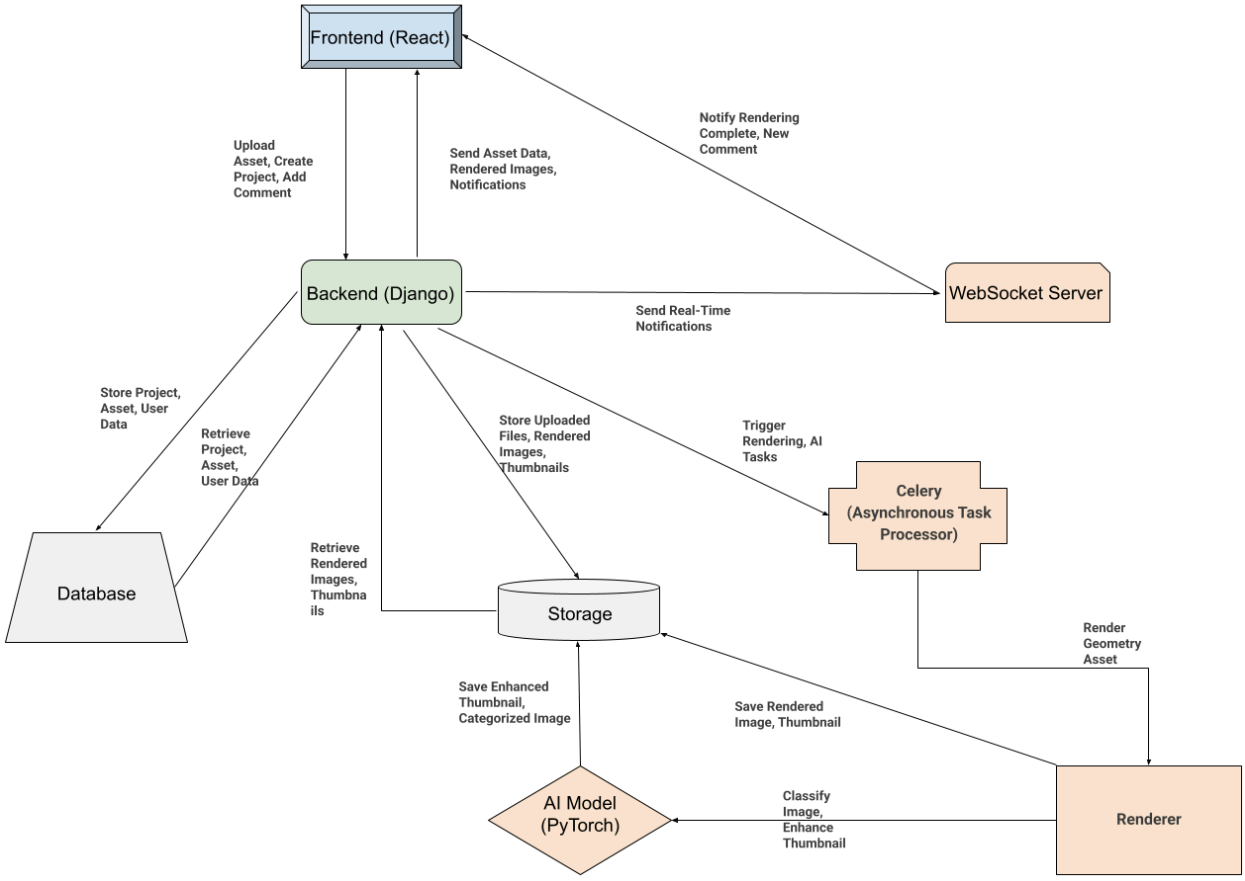
```
from celery import shared_task
from .models import Asset

@shared_task
def process_asset_ai(asset_id):
    asset = Asset.objects.get(id=asset_id)
    if asset.asset_type == 'image':
        # Perform image classification
        categories = classify_image(asset.file.path)
        asset.categories = categories
        asset.save()

        # Enhance the thumbnail
        enhanced_thumbnail = enhance_thumbnail(asset.file.path)
        enhanced_thumbnail.save(asset.thumbnail.path)

        # Enhance the image
        enhanced_image = enhance_image(asset.file.path)
        enhanced_image.save(asset.rendered_image.path)
        asset.ai_enhanced = True
        asset.save()
```

18. Data Flow Diagram



The **Data Flow Diagram (DFD)** in Section 18 provides a comprehensive overview of how data moves through StratoPipe's system, detailing the interactions between core components to illustrate the platform's functionality and efficiency.

Purpose

The DFD offers a system-wide visualization of data exchanges to complement other sections in the design document, such as architecture and workflows. While workflows focus on specific tasks, the DFD emphasizes **overall system interactions**, highlighting how components collaborate to handle user actions and deliver results.

Key Components and Data Interactions

1. **Frontend (React)**:
    - Initiates user actions, such as uploading assets, creating projects, and adding comments.
    - Receives rendered images, thumbnails, real-time notifications, and processed AI results for display.

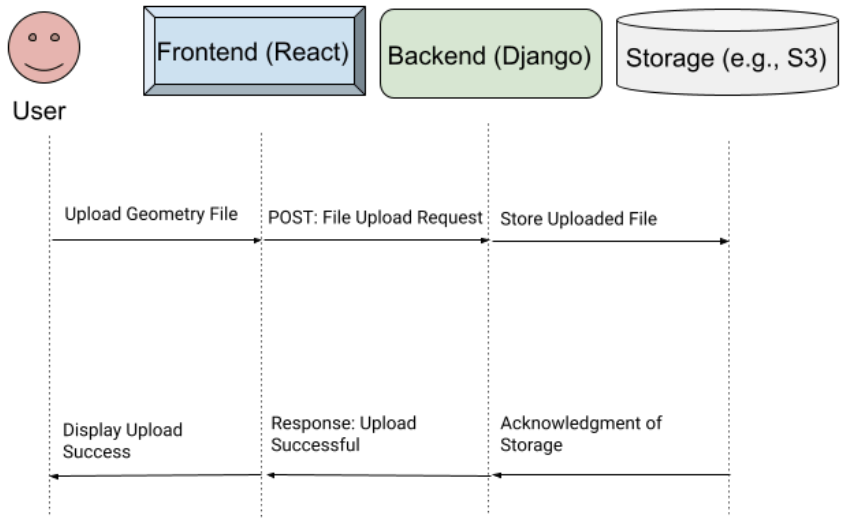
2. **Backend (Django)**:
    - Functions as the central orchestrator, handling API requests from the frontend.
    - Routes tasks to asynchronous processes (e.g., Celery workers) and manages data storage/retrieval from the database and media storage.
3. **WebSocket Server**:
    - Provides real-time capabilities by managing bidirectional communication between the frontend and backend.
    - Ensures users receive live updates (e.g., rendering completion, new comments).
4. **Celery Task Processor**:
    - Executes asynchronous tasks such as rendering, thumbnail generation, and AI-driven functionalities.
    - Interacts with the AI Model and updates backend storage with results.
5. **Database (PostgreSQL)**:
    - Stores critical structured data, including users, projects, assets, tasks, and comments.
    - Facilitates backend queries for retrieving or updating data.
6. **Storage (e.g., AWS S3)**:
    - Handles the storage of uploaded files, rendered images, and enhanced thumbnails.
    - Interfaces with other components to ensure efficient media storage and retrieval.
7. **Renderer and AI Model (PyTorch)**:
    - **Renderer**: Processes geometry files to generate high-quality renders and thumbnails.
    - **AI Model**: Handles tasks such as image classification, object detection, and image enhancement.

Key Data Flows

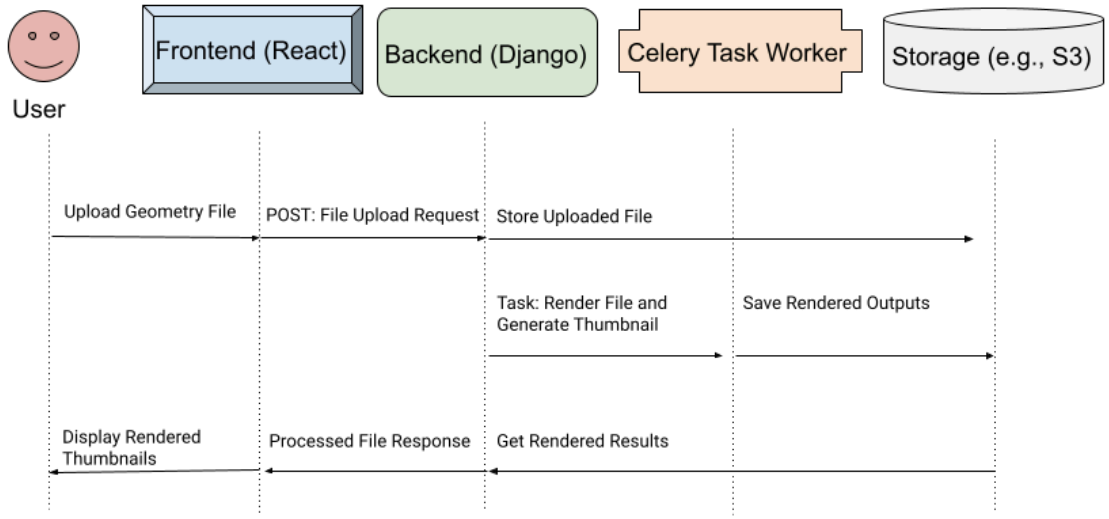
- **From Frontend to Backend**: User requests such as asset uploads and comments are routed via the backend for processing.
- **From Backend to Task Processors**: Asynchronous tasks (e.g., rendering, AI models) are triggered by the backend.
- **From Backend to Database/Storage**: User data, assets, and processed outputs (e.g., renders, thumbnails) are securely saved.
- **From Backend to Frontend**: Processed results and notifications are sent back to the user interface.
- **Real-Time Updates via WebSocket**: Live updates ensure a seamless collaborative experience for users.

19. Main Workflows' Diagrams

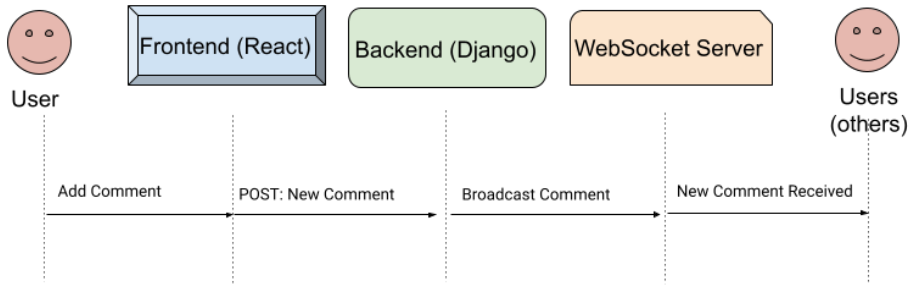
Asset Upload Workflow



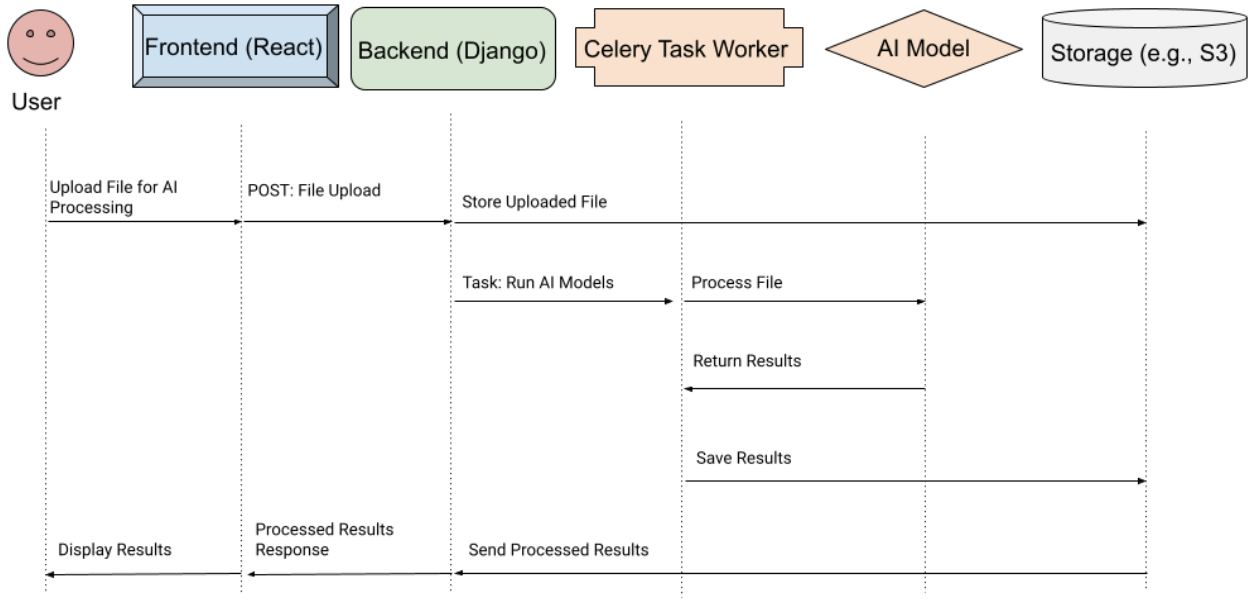
Rendering and Thumbnail Generation



Real-Time Collaboration Workflow

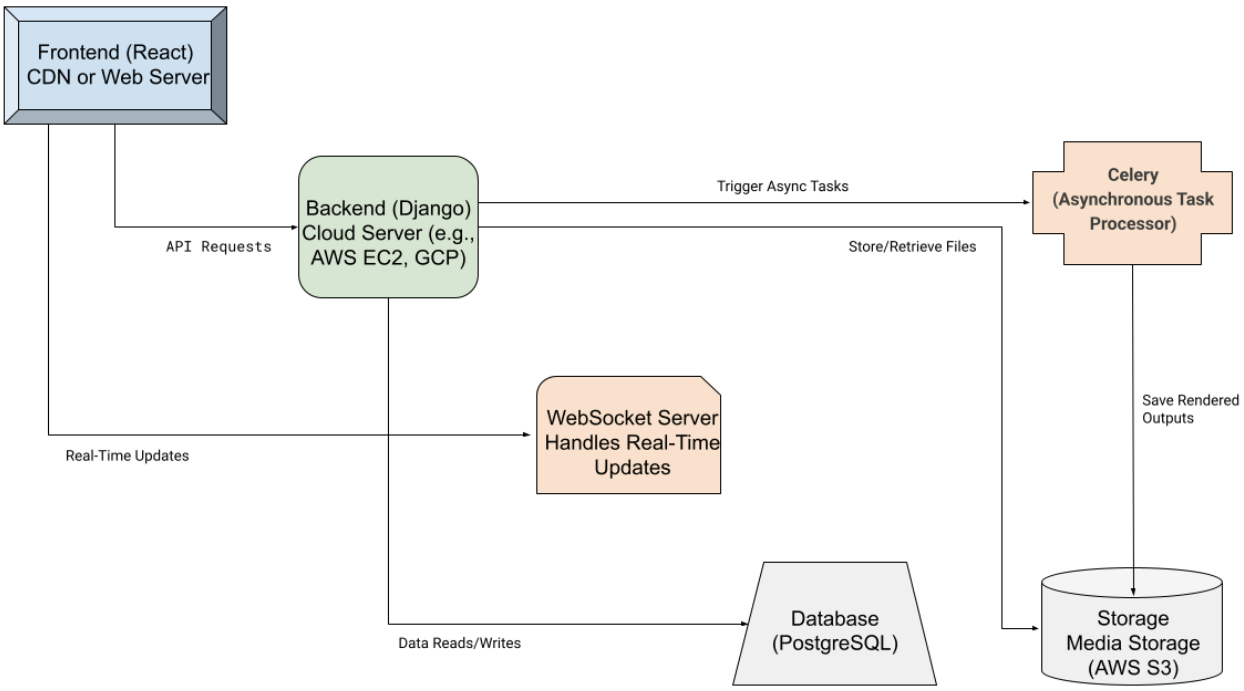


AI Functionalities Workflow Diagram



20. Deployment and Infrastructure

Deployment Diagram for Production Environment



- Open-Source Renderer Installation
  - **Install Renderer**: Ensure that the open-source renderer used for generating images from geometry assets is installed on the server.
  - **Dependencies**: Install any required dependencies for the renderer.
- Hardware Requirements
  - **Rendering Resources**: Rendering tasks may require significant CPU and memory resources.
  - **Scaling**: Consider deploying rendering tasks on servers with higher resources or separate instances to distribute the load.
- Celery Workers
  - **Dedicated Workers**: Run dedicated Celery workers for rendering tasks to manage resource utilization.
  - **Scaling Workers**: Configure auto-scaling for Celery workers based on the rendering workload.
- Media Storage
  - **Static and Media Files**: Configure media storage for rendered images and thumbnails.

        - Use cloud storage solutions (e.g., AWS S3) for production environments.
        - Configure `MEDIA_URL` and `MEDIA_ROOT` in `settings.py`.
- WebSocket Servers
    - **ASGI Server**: Use an ASGI server like Daphne or Uvicorn to handle WebSocket connections in production.
    - **Load Balancing**: Ensure load balancing is properly configured for WebSocket connections if needed.

21. Future Scalability and Maintenance

The long-term success of StratoPipe depends on its ability to scale efficiently, remain maintainable, and provide robust tools for testing and documentation. This section outlines strategies to ensure the platform's performance, adaptability, and usability over time.

Scalability

1. **Modular Codebase**:
    - Maintain a well-organized and modular codebase for easier updates and scalability.
    - Follow best practices such as SOLID principles and clean architecture to minimize interdependencies.
2. **Rendering Task Optimization**:
    - Optimize rendering tasks by improving the efficiency of the open-source renderer and exploring lightweight alternatives for less complex workflows.
    - Distribute rendering workloads across multiple Celery workers or dedicate servers with higher processing power.
3. **Horizontal Scaling**:
    - Scale components such as WebSocket servers, Celery workers, and database instances horizontally to handle increased traffic.
    - Use container orchestration tools like **Kubernetes** for task management and scaling across distributed environments.
4. **Load Testing and Resource Allocation**:
    - Perform regular load testing using tools like **Locust** or **Apache JMeter** to identify bottlenecks under stress.
    - Allocate additional resources for high-demand tasks like AI model processing and rendering during peak usage times.
5. **AI Model Scalability**:
    - Plan for version control and updates to AI models to incorporate advancements without disrupting ongoing workflows.
    - Optimize AI task processing pipelines for resource efficiency.

Maintenance

1. **Centralized Monitoring and Logging**:
    - Implement centralized monitoring systems using tools like **Prometheus** and **Grafana** to track performance metrics such as task latency, server uptime, and rendering efficiency.
    - Use a centralized logging solution like the **ELK Stack** to collect and analyze logs from Celery tasks, WebSocket events, and backend APIs.
2. **Performance Optimization**:
    - Regularly monitor database queries and optimize them with indexing or caching.
    - Utilize tools like **Redis** for caching frequent queries to enhance data retrieval performance.
3. **Security Reviews**:
    - Schedule periodic security audits to identify vulnerabilities in file uploads, WebSocket connections, and data access.
    - Ensure all dependencies remain up-to-date with the latest security patches.
4. **Error Handling Enhancements**:
    - Review and refine the **Unified Error Handling Workflow** periodically to address newly discovered edge cases or evolving user needs.

Testing

1. **Unit Testing**:
    - Focus on error-prone areas such as rendering tasks, AI processing, and asynchronous workflows.
    - Use frameworks like **PyTest** for Python functions and **Mock Library** to simulate rendering and AI tasks during testing.
    - Example tests:
        - Validate file type checks and error detection mechanisms.
        - Simulate task failures to confirm retries and escalations.
2. **Integration and End-to-End Testing**:
    - Test data flow across components (e.g., from asset upload in the frontend to rendering completion in the backend).
    - Confirm the seamless functionality of WebSocket connections and fallback mechanisms.
3. **Performance and Load Testing**:
    - Use tools like **Locust** or **JMeter** to simulate high user traffic scenarios and analyze the system's response times.
    - Validate the scaling behavior of WebSocket servers and Celery workers under concurrent workloads.
4. **Asynchronous Task Testing**:
    - Use **Celery's Test Utilities** to ensure proper task queuing, execution, and retry functionality.
    - Test error handling for long-running Celery tasks and verify appropriate logging.

Documentation

1. **Developer Documentation**:
    - Maintain a comprehensive guide for developers, covering:
        - Environment setup, including Celery workers and WebSocket servers.
        - API documentation using tools like **Swagger** or **Postman** for all backend endpoints.
        - Workflow diagrams for rendering, AI processing, and error handling.
2. **User Documentation**:
    - Create clear and accessible guides for users, focusing on:
        - How to upload assets and manage projects.
        - Troubleshooting real-time collaboration (e.g., fallback from WebSockets to polling).
        - Interpreting error messages and taking corrective actions.
3. **Testing Documentation**:
    - Summarize all testing practices, including frameworks, test coverage, and step-by-step instructions for running tests.
    - Provide a centralized record of test results to track system health over time.
4. **Maintenance Playbook**:
    - Include a playbook for operational procedures such as scaling services, updating AI models, and resolving critical errors.
    - Document how to respond to alerts from monitoring tools, such as task failures or server outages.

22. Conclusion

StratoPipe offers a robust, streamlined solution for virtual production workflows, integrating project management, asset handling, AI-driven functionalities, and real-time collaboration. Its modular monolithic architecture ensures simplicity, scalability, and efficient performance across rendering, AI processing, and collaborative features.

The platform's key components, such as asynchronous task management and fallback mechanisms, focus on reliability and a seamless user experience. Comprehensive testing strategies, centralized monitoring, and detailed documentation further support system scalability and maintainability.

Designed for long-term adaptability, StratoPipe empowers creative teams to collaborate efficiently while addressing future demands, making it a powerful tool for revolutionizing remote workflows in virtual production.