**# Geospatial Processing Backend – Actinia + GRASS GIS**

**## Executive Summary**

This project implements a scalable geospatial processing backend for smart city applications, focusing on solar radiation simulation and photovoltaic in urban data analysis. The system aims to provide API-driven spatial processing capabilities for raster, vector, and derived time series data in urban contexts.

The backend combines Actinia and GRASS GIS with a FastAPI controller to enable automated workflows for:

- Digital surface model processing

- Solar potential assessment

- Geospatial analytics

- Urban data pre-processing

The architecture is designed as a stateless, containerized system that provides both development flexibility and production reliability.

**## General Architecture**

**### Core Technologies:**

- **\*\*Actinia Core\*\***: A geospatial engine that exposes GRASS GIS functions over REST APIs. It supports operations like slope and aspect computation, solar radiation simulation, zonal statistics, raster algebra, and more. (Actinia is stateless in nature, meaning it does not maintain session contexts and treats each API call as atomic. It supports asynchronous execution via /status endpoints and can dynamically create GRASS locations and mapsets as required by each task.)

- **\*\*GRASS GIS\*\***: The computational backend used by Actinia. It requires structured data management using locations (based on spatial reference systems) and mapsets (user/project-level data grouping). Raster and vector layers are processed here.

- **\*\*FastAPI\*\***: A high-performance Python framework used to wrap and orchestrate Actinia's capabilities. FastAPI provides the logic layer for user authentication, file upload, job coordination, storage consistency, and multi-step workflows. Actinia alone lacks these application-level capabilities.

- **\*\*Database Layer\*\*** (PostgreSQL): Used to store metadata, user sessions, job hashes, simulation parameters, and file inventories.

Actinia uses **PostgreSQL** (a type of relational database) to **remember certain types of information**.

- **\*\*Docker + Volumes\*\***: Both Actinia and PostgreSQL are deployed as Docker containers during development, with persistent volumes mounted to allow long-term storage. In production, all components (including FastAPI) will be containerized.

Actinia does **not remember previous tasks**. Every command you send must include all the information it needs. If a task takes time, it gives you a **status link** to check later. And if your task needs a new working area in GRASS GIS, Actinia can **create it automatically**.

**Metadata**

Data about other data. If you upload a raster file, metadata includes: File name, Date of upload, Size, Resolution, Coordinate system

**User sessions**

Temporary information about users while they are using the system.

**Job hashes:**

Unique IDs for jobs. When you run a GRASS command (e.g., r.slope.aspect), it gets a unique hash like “job\_12345abcde” — used to check the status later.

**Simulation parameters**

Settings or inputs you use when running a simulation or analysis.

**File inventories**

A list or log of files uploaded, created, or processed.

**Docker volumes**

are a way to **store data** that your container creates or uses — **outside** the container itself — so that:

* The data is **not lost** when the container is stopped or deleted.
* You can **reuse, back up, or inspect** the data more easily.

By default, anything inside a Docker container is **temporary**. If the container is removed, its internal data disappears. So we use **volumes** to store data **persistently**.

**### Why FastAPI?**

Actinia exposes raw GRASS functionality but does not handle workflows that require enhanced user experience, file validation, or inter-module orchestration. FastAPI is introduced as a middle layer that performs the following critical functions:

- It implements hybrid authentication: validates user credentials against Actinia while providing JWT tokens for enhanced session management and user experience

- It manages user profiles, preferences, and extended session data while keeping credentials secured in Actinia

- It validates and stores uploaded files, ensuring their compatibility with GRASS and Actinia (e.g., checking CRS, file formats)

- It generates and manages mapsets and locations for each user/job with proper ownership tracking

- It handles business logic, such as synthetic DSM creation when no elevation data is available, suitability filtering, horizon generation, and solar simulation aggregation

- It wraps multi-step workflows, turning them into single API endpoints with proper user authorization and job tracking

- It allows external systems (frontends, schedulers, etc.) to interact with the geospatial engine through a simplified, secure, and documented API layer

**### Visual Architecture Diagram**

```

┌───────────────┐      ┌───────────────┐      ┌───────────────┐

│   Web Client  │      │  External     │      │  Scheduled    │

│   (WebGIS)    │      │  Systems      │      │  Tasks        │

└───────┬───────┘      └───────┬───────┘      └───────┬───────┘

        │                      │                      │

        ▼                      ▼                      ▼

┌─────────────────────────────────────────────────────────────┐

│                      FastAPI Backend                        |

│  ┌─────────────┐  ┌──────────────┐  ┌────────────────────┐  │

│  │ Hybrid Auth │  │ Geospatial   │  │ Job Management &   │  │

│  │ JWT + User  │  │ Processing   │  │ User Profiles      │  │

│  │ Profiles    │  │              │  │                    │  │

│  └─────────────┘  └──────────────┘  └────────────────────┘  │

└───────────────────────────┬─────────────────────────────────┘

             │

       ┌─────────────────┴─────────────────┐

       ▼                                   ▼

┌───────────────────────┐       ┌───────────────────────┐

│   Actinia Core        │       │   PostgreSQL/PostGIS  │

│  ┌─────────────────┐  │       │  ┌─────────────────┐  │

│  │  GRASS GIS      │  │       │  │  User Profiles  │  │

│  │  Processing     │  │       │  │  Sessions       │  │

│  │  + User Auth    │  │       │  │  Job Metadata   │  │

│  └─────────────────┘  │       │  │  GeoJSON Results│  │

└───────────┬───────────┘       └───────────┬───────────┘

            │                               │

            ▼                               ▼

┌───────────────────────┐       ┌───────────────────────┐

│  Persistent Storage   │       │  PostgreSQL Volumes   │

│  - GRASS Database     │       │  - User Profiles      │

│  - Uploaded Files     │       │  - Session Data       │

│  - Result Caches      │       │  - Job Records        │

└───────────────────────┘       └───────────────────────┘

```

**## Database Architecture**

The database layer for this project focuses exclusively on user management, authentication, and job tracking. Geospatial data processing and storage remains the responsibility of Actinia and GRASS GIS, only results will be saved in geojson and then to postgis.

**### Schema Design**

The database schema focuses on user experience enhancement and job management while leveraging Actinia for core authentication:

- **\*\*Users\*\***: User profiles, preferences, and settings (credentials remain in Actinia)

- **\*\*Sessions\*\***: JWT token metadata, device tracking, and session management

- **\*\*Jobs\*\***: Processing task metadata with enhanced user ownership tracking

- **\*\*Files\*\***: File metadata with user-based access control

- **\*\*Results\*\***: Computation results with user-specific storage and sharing permissions

**### Database Selection**

PostgreSQL will be used from the start of development:

**\*\*Advantages\*\***:

- Production-ready database that eliminates migration issues later

- Supports complex queries needed for user management, job tracking, and session handling

- Runs in Docker with minimal configuration

- Enables PostGIS integration for result storage and spatial queries

- Provides robust user profile and session management capabilities

**## User Journey**

**### Example User Flow: Building Solar Potential Assessment**

1. **\*\*Authentication\*\***: User logs in via `/auth/login` endpoint, FastAPI validates against Actinia and returns JWT token

   ```bash

   curl -X POST "http://localhost:8000/auth/login" \

     -H "Content-Type: application/json" \

     -d '{"username": "solar\_analyst", "password": "secure\_password"}'

   # Response includes JWT token and user profile information

   ```

2. **\*\*Profile Management\*\***: User can update preferences and view account information

   ```bash

   curl -X GET "http://localhost:8000/auth/me" \

     -H "Authorization: Bearer {jwt\_token}"

   curl -X PUT "http://localhost:8000/users/profile" \

     -H "Authorization: Bearer {jwt\_token}" \

     -H "Content-Type: application/json" \

     -d '{"preferences": {"default\_crs": "EPSG:4326", "notifications": true}}'

   ```

3. **\*\*Data Upload\*\***: User uploads a Digital Surface Model (DSM) and building footprints

   ```bash

   curl -X POST "http://localhost:8000/upload/raster" \

     -H "Authorization: Bearer {token}" \

     -F "file=@city\_dsm.tif" \

     -F "metadata={\"description\": \"City DSM 1m resolution\"}"

   curl -X POST "http://localhost:8000/upload/vector" \

     -H "Authorization: Bearer {token}" \

     -F "file=@buildings.geojson" \

     -F "metadata={\"description\": \"Building footprints\"}"

   ```

4. **\*\*Preprocessing\*\***: System computes slope and aspect from DSM

   ```bash

   curl -X POST "http://localhost:8000/terrain/analyze" \

     -H "Authorization: Bearer {token}" \

     -H "Content-Type: application/json" \

     -d '{"dsm\_id": "dsm-123", "compute\_slope": true, "compute\_aspect": true}'

   ```

5. **\*\*Horizon Calculation\*\***: User initiates horizon map generation

   ```bash

   curl -X POST "http://localhost:8000/horizon/calculate" \

     -H "Authorization: Bearer {token}" \

     -H "Content-Type: application/json" \

     -d '{"dsm\_id": "dsm-123", "azimuth\_step": 10}'

   ```

6. **\*\*Solar Simulation\*\***: User runs solar radiation simulation for a specific date

   ```bash

   curl -X POST "http://localhost:8000/radiation/run" \

     -H "Authorization: Bearer {token}" \

     -H "Content-Type: application/json" \

     -d '{

       "dsm\_id": "dsm-123",

       "date": "2023-06-21",

       "time\_step": 15,

       "linke": 3.0

     }'

   ```

7. **\*\*Suitability Analysis\*\***: User identifies suitable roof areas

   ```bash

   curl -X POST "http://localhost:8000/site/suitability" \

     -H "Authorization: Bearer {token}" \

     -H "Content-Type: application/json" \

     -d '{

       "slope\_max": 45,

       "aspect\_range": [90, 270],

       "buildings\_id": "buildings-123"

     }'

   ```

8. **\*\*Results Aggregation\*\***: System calculates per-building statistics

   ```bash

   curl -X POST "http://localhost:8000/results/aggregate" \

     -H "Authorization: Bearer {token}" \

     -H "Content-Type: application/json" \

     -d '{

       "radiation\_job\_id": "job-789",

       "buildings\_id": "buildings-123",

       "format": "geojson"

     }'

   ```

9. **\*\*Export Results\*\***: User downloads final data package

   ```bash

   curl -X GET "http://localhost:8000/download/job-789" \

     -H "Authorization: Bearer {token}" \

     --output solar\_results.zip

   ```

**## Docker Strategy**

**### Docker Usage Strategy**

**#### Development Phase:**

- Only pre-built components (Actinia and PostgreSQL) use Docker

- FastAPI runs locally for rapid development and easier debugging

- GRASS data persistence is maintained through Docker volumes

**#### Production Phase:**

- All components get containerized (FastAPI, PostgreSQL, Actinia)

- Inter-service communication uses Docker network names

- Configuration via environment variables with Docker secrets

**### Docker Volume Strategy**

**#### Types of Volumes and Their Usage**

The project employs two distinct volume strategies to balance development flexibility with data persistence:

**##### Bind Mounts for GRASS Data:**

- These bind mounts map local directories directly to container paths, providing:

   - Direct access to GRASS data files from the host system

   - Immediate visibility of processing results without container access

   - Ability to inspect and modify files using host tools (editors, GIS software)

   - Persistence even when containers are removed completely

**##### Named Volumes for Database Data:**

- Named volumes are managed entirely by Docker and provide:

   - Proper isolation for database integrity

   - Automatic permission handling

   - Protection from accidental deletion

   - Better performance for database workloads

**#### Volume Content and Purpose**

- **\*\*grassdb\*\***: Contains GRASS GIS locations (coordinate systems) and mapsets (user workspaces)

- **\*\*resources\*\***: Stores uploaded files, imported datasets, and exported results

- **\*\*userdata\*\***: Maintains user-specific configurations and temporary processing files

- **\*\*postgres\_data\*\***: Stores database files including user accounts, job metadata, and processing logs

**#### Development vs. Production Strategy**

**##### Development Phase:**

- Use bind mounts for GRASS data to enable rapid debugging and inspection

- Use named volumes for PostgreSQL to maintain database integrity

- Keep data directories under version control (excluding large datasets)

- Store sample datasets in separate directory structure

**##### Production Phase:**

- Consider transitioning to named volumes for all data for better isolation

- Implement volume backup strategies

- Configure proper permissions for production security

- Consider network storage options for distributed deployments

**### Docker Transition Checklist**

When moving from development to production, follow these steps:

1. **\*\*Create FastAPI Dockerfile\*\***:

   ```dockerfile

   FROM python:3.9-slim

   WORKDIR /app

   COPY requirements.txt .

   RUN pip install --no-cache-dir -r requirements.txt

   COPY ./app ./app

   CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]

   ```

2. **\*\*Adjust Environment Variables\*\***:

   - Update service references (e.g., change `localhost` to Docker service names)

   - Configure database connection strings

   - Update volume paths for Docker-specific locations

3. **\*\*Test Transition Incrementally\*\***:

   - Run both local FastAPI and dockerized services initially

   - Test containerized FastAPI with existing dockerized services

   - Finally test fully containerized setup

4. **\*\*Update docker-compose.yml\*\***:

   - Uncomment or add the FastAPI service

   - Configure proper networking between services

   - Set up appropriate restart policies

5. **\*\*Verify Database Migrations\*\***:

   - Ensure all database migrations run correctly in the production environment

   - Test database backup and restore procedures

**### Hybrid Development Approach**

An efficient development strategy is to use Actinia and PostgreSQL in Docker while developing the FastAPI application locally:

**#### Benefits:**

- **\*\*Faster Development Cycles\*\***: Make direct code changes to FastAPI without container rebuilds, enabling immediate testing and more efficient debugging.

- **\*\*Simplified Initial Development\*\***: Focus on FastAPI application logic first without dealing with Docker networking complexities.

- **\*\*Gradual Infrastructure Evolution\*\***: Start with just the Actinia and PostgreSQL containers, and Dockerize the FastAPI component once stable.

**#### Implementation Strategy:**

1. **\*\*Run Actinia and PostgreSQL in Docker\*\***: Deploy only these containers with proper volume mounts for GRASS data and database persistence.

2. **\*\*Set Up Local FastAPI Environment\*\***: Create a Python virtual environment and install dependencies locally.

3. **\*\*Configure Communication\*\***: Set up services in your FastAPI app to communicate with the Dockerized instances.

4. **\*\*Use Environment Variables\*\***: Configure connection settings via environment variables to simplify the eventual transition to full containerization.

**## Error Handling and System Monitoring**

**### Error Handling and Recovery Strategy**

**#### Error Classification**

1. **\*\*Input Validation Errors\*\***:

   - Invalid file formats or CRS issues

   - Missing required parameters

   - Response: 400 Bad Request with detailed validation errors

2. **\*\*Processing Errors\*\***:

   - GRASS/Actinia module failures

   - Resource constraints (memory, CPU)

   - Response: 500 Internal Server Error with job details

3. **\*\*Authentication/Authorization Errors\*\***:

   - Invalid/expired tokens

   - Insufficient permissions

   - Response: 401 Unauthorized or 403 Forbidden

**#### Recovery Mechanisms**

1. **\*\*Automated Retry System\*\***:

   - Jobs that fail due to temporary issues automatically retry up to 3 times

   - Exponential backoff between retries (1min, 5min, 15min)

   - Stateful tracking of retry attempts in database

2. **\*\*Partial Result Recovery\*\***:

   - For multi-stage workflows, successful intermediate results are cached

   - Failed jobs can resume from last successful checkpoint

   - Unique hash identifiers for each processing stage enable smart caching

3. **\*\*Error Notification\*\***:

   - Critical failures trigger admin alerts via email/Slack

   - Users receive clear error messages with troubleshooting guidance

   - Job error logs accessible via `/jobs/{job\_id}/logs` endpoint

4. **\*\*Graceful Degradation\*\***:

   - System maintains core functionality during partial outages

   - Non-essential features disabled when resources are constrained

   - Prioritization system for user jobs during high load

**### System Monitoring and Performance**

**#### Monitoring Strategy**

1. **\*\*Application Metrics\*\***:

   - Prometheus integration for real-time metrics collection

   - Key metrics: API request rate, response times, error rates

   - Custom metrics for job throughput and resource utilization

2. **\*\*Infrastructure Monitoring\*\***:

   - Container health checks via Docker health endpoints

   - CPU, memory, disk usage tracking

   - Network traffic and database connection pool stats

3. **\*\*Job Status Tracking\*\***:

   - Real-time dashboard for active and queued jobs

   - Historical job performance analytics

   - Alert thresholds for stuck or long-running jobs

4. **\*\*Log Aggregation\*\***:

   - Centralized logging with structured JSON format

   - Log correlation with request IDs across services

   - Log retention policies aligned with data retention needs

**#### Performance Benchmarks**

| Operation | Target Response Time | Max Resources | Throughput Target |

|-----------|---------------------|---------------|-------------------|

| File Upload (100MB) | < 30 seconds | 2 CPU, 4GB RAM | 10 concurrent |

| DSM Processing | < 2 minutes per km² | 4 CPU, 8GB RAM | 5 concurrent |

| Horizon Calculation | < 5 minutes per km² | 8 CPU, 16GB RAM | 3 concurrent |

| Solar Simulation (daily) | < 10 minutes per km² | 8 CPU, 16GB RAM | 2 concurrent |

| Result Export | < 1 minute | 2 CPU, 4GB RAM | 20 concurrent |

| DB Queries | < 500ms | N/A | 100 per second |

**#### Load Testing Strategy**

1. **\*\*Test Scenarios\*\***:

   - Single user end-to-end workflow

   - Multiple concurrent upload operations

   - Parallel processing jobs with varying resource needs

   - Sudden spike in API requests (burst testing)

2. **\*\*Tools\*\***:

   - Locust for HTTP endpoint load testing

   - Custom scripts for workflow simulation

   - Docker resource constraints for capacity testing

3. **\*\*Performance Analysis\*\***:

   - 95th percentile response times under various loads

   - Resource utilization patterns during peak load

   - Database query performance and connection pool behavior

   - Bottleneck identification through profiling

4. **\*\*Acceptance Criteria\*\***:

   - System handles 25 concurrent users with acceptable performance

   - No degradation after 24 hours of continuous operation

   - Resource utilization remains below 80% during normal operation

   - Recovery time < 5 minutes after resource exhaustion

**## Feature Prioritization**

**### Core Features (Must Have)**

1. User authentication and file upload

2. Terrain analysis (slope/aspect calculation)

3. Solar radiation simulation

4. Basic results aggregation and export

**### High Priority Features**

1. Horizon calculation

2. Suitability analysis based on terrain

3. Zonal statistics per building

4. GeoJSON export with enhanced attributes

**### Nice-to-Have Features**

1. Synthetic DSM generation

2. Advanced filtering options

3. Time series visualization data

4. Automated report generation

**## Troubleshooting and Integration Guide**

**### Common Actinia Issues and Solutions**

1. **\*\*"Permission denied" errors\*\***:

    - Check that Actinia has access to the mounted volumes

    - Verify user permissions in Actinia config

    - Ensure paths are correctly specified in process chains

2. **\*\*"No such location" errors\*\***:

    - Confirm location exists and is accessible

    - Check EPSG code used for location creation

    - Verify path to GRASS database is correct

3. **\*\*GRASS module failures\*\***:

    - Examine stderr output for specific error messages

    - Check input parameter formats and types

    - Verify computational region settings with `g.region -p`

**### Actinia API Integration**

**#### Authentication with Actinia:**

To authenticate with Actinia, you need to obtain a token by providing username and password credentials to the token endpoint. This token is then used in subsequent API calls as a Bearer token in the Authorization header.

**#### Example Actinia Process Chain for DSM Import:**

Process chains in Actinia define a sequence of operations to be executed. For importing a Digital Surface Model (DSM), you would create a process chain that uses the r.import module, specifying input file path and resolution parameters.

**### Debugging Tips:**

1. **\*\*Enable verbose logging\*\***: Configure Python logging at the DEBUG level to get detailed information about operations.

2. **\*\*Inspect Actinia process chain results\*\***: Examine process logs for warnings or errors to identify issues in Actinia execution.

3. **\*\*Test GRASS modules directly in Docker\*\***: Connect to the Actinia container and run GRASS commands directly to verify functionality outside of the API context.

**## Useful Resources**

- [Actinia API Documentation](https://actinia.mundialis.de/api\_docs/)

- [GRASS GIS Manual](https://grass.osgeo.org/grass78/manuals/)

- [FastAPI Documentation](https://fastapi.tiangolo.com/)

- [r.sun Module Documentation](https://grass.osgeo.org/grass78/manuals/r.sun.html)

- [Sample Datasets Repository](https://gitlab.com/mundialis/actinia/actinia-example-data)

- [PostGIS Tutorials](https://postgis.net/workshops/postgis-intro/)

**## Final Assessment Criteria**

The completed project should demonstrate:

1. Correct implementation of the geospatial workflow

2. Proper error handling and validation

3. Efficient resource management

4. Well-documented API with OpenAPI schema

5. Automated tests for critical components

6. Proper authentication and authorization

7. Clean code organization following modern Python practices

8. Reasonable performance for large datasets

9. Comprehensive documentation for users and developers

Remember to maintain regular Git commits with descriptive messages to document your progress and facilitate code review.