STACD Migration Guide

Migrating CoreStack Pipelines to STACD Framework

Saharsh Laud

November 2025

Contents

1 Executive Summary

This document provides a comprehensive guide for the CoreStack team to migrate existing geospatial data processing pipelines to the **Spatio-Temporal Asset Catalog with Dependencies (STACD)** framework. STACD extends the STAC specification with algorithm metadata, dependency tracking, and selective recomputation capabilities, enabling reproducible, version-controlled, and lineage-tracked workflows.

1.1 Key Benefits

- **Reproducibility:** Full lineage tracking with UUIDs for DAGs, algorithms, and datasets
- **Flexibility:** Support for API and Docker execution modes with priority-based selection
- **Selective Recomputation:** Recompute only affected nodes when algorithms or datasets change
- **Version Control:** Database-backed tracking of algorithm versions, parameters, and outputs
- **YAML-Driven:** Declarative pipeline definitions with automatic DAG generation

2 System Architecture Overview

The STACD framework consists of four core components:

2.1 Component Architecture

1. **Backend Service** (`stacd_backend/`)
   FastAPI-based REST API and CLI for algorithm execution
2. **Database Layer** (`stacd_database/`)
   SQLite database with SQLAlchemy ORM for metadata and lineage tracking
3. **DAG Generator** (`airflow/new_server/mixed/`)
   Python script that generates Airflow DAGs from YAML specifications
4. **Airflow Orchestrator**
   Apache Airflow for workflow execution and scheduling

[Architecture Diagram: Backend ↔ Database ↔ DAG Generator → Airflow]

Figure 1: STACD System Architecture

3 Current Implementation Status

3.1 Completed Features

3.2 Proven Execution Modes

- **Full Execution (fullexec):** All algorithms executed in dependency order
- **API Mode:** Algorithms called via HTTP POST to backend API
- **Docker Mode:** Algorithms executed in isolated Docker containers
- **Mixed Mode:** Priority-based selection between API and Docker per algorithm

Component	Status	Description
Backend API	Complete	FastAPI with blocking calls
Backend CLI	Complete	Docker-compatible CLI
Database Schema	Complete	UUID-based lineage tracking
Mixed DAG Generator	Complete	API + Docker support
Full Execution Mode	Complete	Execute entire DAG
Selective Recomputation	Planned	Update algo/dataset/DAG
Resume Execution	Planned	Resume from failed nodes
Lineage Plugin	Planned	Enhanced Airflow plugin
Admin Plugin	Planned	GUI for metadata updates

Table 1: Implementation Status

4 Backend Structure

4.1 Directory Layout

```
stacd_backend/
          Dockerfile                    # Container definition
          cli.py                        # Command-line interface
          main.py                       # FastAPI application
          requirements.txt              # Python dependencies
          computing/
              lulc/                     # LULC algorithms
                    lulc_v3_clip_river_basin.py
                    lulc_vector.py
                    cropping_frequency.py
                    misc.py
              terrain/                  # Terrain algorithms
                    terrain_raster.py
                    terrain_clusters.py
                    terrain_utils.py
              lulcxterrain/             # Cross-domain algorithms
                    lulc_on_slope_cluster.py
                    lulc_on_plain_cluster.py
                    terrain_classifier.py
              utils/                    # Shared utilities
                  constants.py
                  gee_utils.py
```

Listing 1: Backend Directory Structure

4.2 Backend API Endpoints

```
@app.post("/api/v1/lulc_v3/")
async def run_lulc_algorithm(
    state: str,
    district: str,
    block: str,
    start_year: int,
    end_year: int,
    execution_id: str
):
    """
```

```
    Execute LULC algorithm with blocking behavior.
    Returns only when computation is complete.
    """
    asset_ids = lulc_river_basin(
        state, district, block,
        start_year, end_year
    )

    return {
        "status": "success",
        "execution_id": execution_id,
        "asset_ids": asset_ids,
        "execution_time": elapsed_time
    }
```

Listing 2: Example FastAPI Endpoint

5 Database Schema

5.1 Core Tables

The database tracks complete lineage using UUIDs:

Table	Primary Key	Purpose
dags	dag_uuid	DAG metadata and structure
algorithm_types	algo_type_id	Algorithm definitions
algorithm_instances	algo_instance_id	Algorithm executions
dataset_types	dataset_type_id	Dataset definitions
dataset_instances	dataset_uuid	Dataset instances

Table 2: Database Schema Overview

5.2 Lineage Tracking

```
# Algorithm execution logging
db.log_algorithm_execution(
    execution_id=uuid.uuid4(),
    algo_type_id=algo_type.algo_type_id,
    dag_uuid=DAG_UUID,
    dag_run_id=context['dag_run'].run_id,
    node_name='execute_LULC_Algorithm',
    status='running',
    parameters={'state': 'jharkhand', ...}
)

# Dataset instance logging
db.log_dataset_instance(
    dataset_uuid=uuid.uuid4(),
    dataset_type_id=dataset_type.dataset_type_id,
    execution_id=execution_id,
    dag_uuid=DAG_UUID,
    asset_ids=['projects/ee-saharshlaud/...'],
    region_params={'state': 'jharkhand', ...}
)
```

Listing 3: Example Database Logging

6 YAML Configuration Format

6.1 DAG Definition (stacd_dag.yaml)

```yaml
---
!DAG
id: new_backend_api_test
name: CoreStack LULC Pipeline
version: "1.0"
description: End-to-end LULC and Terrain processing
alg_type_nodes:
  - LULC_Algorithm
  - Terrain_Algorithm
  - LULC_Vectorization
  - Terrain_Vectorization
  - Terrain_LULC_Slope
  - Terrain_LULC_Plain
dataset_type_nodes:
  - LULC_Raster
  - Terrain_Raster
  - LULC_Vector
  - Terrain_Vector
  - Terrain_LULC_Vector_Slope
  - Terrain_LULC_Vector_Plain
params:
  - state
  - district
  - block
  - start_year
  - end_year
```

Listing 4: DAG Configuration Example

6.2 Algorithm Repository (stacd_algorithm_repo.yaml)

```yaml
---
!Algorithm_Instance
type: LULC_Algorithm
version: "1"
assets:
  code: "https://github.com/your-org/stacd-backend"
date: 2024-11-06 00:00:00
execution_modes:
  api:
    enabled: true
    url: "http://localhost:8000/api/v1/lulc_v3/"
    priority: 1  # API preferred
  docker:
    enabled: true
    image: "yourorg/stacd-backend:latest"
    priority: 2
    volumes:
      - source: "~/.config/earthengine"
        target: "/root/.config/earthengine"
        read_only: true
```

Listing 5: Algorithm Instance Definition

6.3 Algorithm Type Definition

```yaml
---
!Algorithm_Type
id: LULC_Algorithm
name: LULC River Basin Clipper
version: "1.0"
params:
  - name: state
    type: string
  - name: district
    type: string
  - name: block
    type: string
  - name: start_year
    type: integer
  - name: end_year
    type: integer
inputs:
  - state
  - district
  - block
  - start_year
  - end_year
input_datasets: []  # No input datasets (starting node)
outputs:
  - LULC_Raster
```

Listing 6: Algorithm Type Metadata

7 DAG Generator

7.1 Generator Workflow

1. Parse YAML configurations (DAG, algorithm repo, dataset repo)
2. Resolve execution modes using priority system
3. Generate Airflow DAG Python code
4. Insert UUID tracking and database logging
5. Create task dependencies from YAML relationships

7.2 Execution Mode Selection

The generator automatically selects execution mode per algorithm:

```python
def get_execution_mode(self):
    """Select execution mode based on priority"""
    api_config = self.execution_modes.get('api', {})
    docker_config = self.execution_modes.get('docker', {})

    api_enabled = api_config.get('enabled', False)
    docker_enabled = docker_config.get('enabled', False)

    # Only one enabled
    if api_enabled and not docker_enabled:
```

```
        return 'api', api_config
    if docker_enabled and not api_enabled:
        return 'docker', docker_config

    # Both enabled: use priority (lower = higher priority)
    if api_enabled and docker_enabled:
        api_priority = api_config.get('priority', 999)
        docker_priority = docker_config.get('priority', 999)
        if api_priority < docker_priority:
            return 'api', api_config
        else:
            return 'docker', docker_config

    return None, None
```

Listing 7: Priority-Based Mode Selection

7.3 Running the Generator

```
cd ~/airflow/new_server/mixed/dag_generator
python3 stacd_mixed_dag_generator.py

# Output:
# Generated MIXED DAG with dependencies + database
# DAG UUID: e0476807-d8fe-4b7d-8fc7-c5a73ccc9569
# Output: ~/airflow/new_server/mixed/generated_dags/generated_mixed_dag
    .py

# Deploy to Airflow
cp ~/airflow/new_server/mixed/generated_dags/generated_mixed_dag.py \
   ~/airflow/dags/
```

Listing 8: Generate Airflow DAG

8 Migration Requirements for CoreStack

8.1 Critical Backend Requirements

Your existing CoreStack algorithms must be adapted to meet these requirements:

8.1.1 1. Blocking Execution

Required: Algorithms must **block** until computation is complete.

- **Bad:** Return immediately with task ID, require polling
- **Good:** Wait for Google Earth Engine tasks to complete before returning

```
def algorithm_function(params):
    """REQUIRED: Block until computation complete"""

    # Submit tasks to GEE
    tasks = submit_gee_tasks(params)

    # CRITICAL: Wait for completion
    while not all_tasks_complete(tasks):
```

```
        time.sleep(5)
        check_task_status(tasks)

    # Return only when done
    return {
        "status": "success",
        "asset_ids": [task.asset_id for task in tasks],
        "execution_time": elapsed_time
    }
```

Listing 9: Blocking Call Pattern

8.1.2 2. Asset ID Return

Required: Return list of created asset IDs for lineage tracking.

```
{
    "status": "success",
    "execution_id": "ad59386f-fcca-426e-b555-c70f6286a877",
    "asset_ids": [
        "projects/ee-user/assets/lulc_2017_dumka",
        "projects/ee-user/assets/lulc_2018_dumka"
    ],
    "execution_time": 28.7
}
```

Listing 10: Return Format

8.1.3 3. Execution ID Support

Required: Accept and return `execution_id` UUID for tracking.

```
@app.post("/api/v1/your_algorithm/")
async def run_algorithm(
    # ... your parameters ...
    execution_id: str  # REQUIRED parameter
):
    print(f"[{execution_id}] Starting algorithm")
    result = compute(params)
    return {
        "execution_id": execution_id,  # Echo back
        "asset_ids": result.assets,
        ...
    }
```

Listing 11: Execution ID Handling

8.2 Additional Requirements

9 Migration Steps

9.1 Step 1: Adapt Backend API

1. **Add blocking behavior:** Modify algorithms to wait for completion
2. **Standardize return format:** Include status, execution_id, asset_ids
3. **Add execution_id parameter:** Accept UUID in all endpoints

Requirement	Priority	Description
Blocking calls	Critical	Must wait for completion
Asset ID return	Critical	Return list of created assets
Execution ID	Critical	Accept and echo UUID
Status field	High	Return "success" or "failed"
Execution time	High	Return computation duration
Error messages	Medium	Return detailed error on failure
Idempotency	Medium	Same inputs = same outputs
Docker support	Optional	Optionally support CLI execution

Table 3: Backend Migration Requirements

4. **Test with curl:**

```
curl -X POST http://localhost:8000/api/v1/your_algorithm/ \
  -H "Content-Type: application/json" \
  -d '{
    "state": "jharkhand",
    "district": "dumka",
    "block": "masalia",
    "execution_id": "test-uuid-1234"
  }'

# Expected response (blocks until complete):
{
  "status": "success",
  "execution_id": "test-uuid-1234",
  "asset_ids": ["projects/.../asset1", "projects/.../asset2"],
  "execution_time": 45.2
}
```

Listing 12: Test API Endpoint

9.2 Step 2: Create YAML Configurations

1. Define DAG structure in `stacd_dag.yaml`
2. Create algorithm type metadata
3. Create algorithm instance definitions with execution modes
4. Define dataset types

9.3 Step 3: Initialize Database

```
cd ~/stacd_database
python3 init_db.py

# Test database connection
python3 test_db.py
```

Listing 13: Database Setup

9.4 Step 4: Generate DAG

```
cd ~/airflow/new_server/mixed/dag_generator
python3 stacd_mixed_dag_generator.py
cp ../generated_dags/generated_mixed_dag.py ~/airflow/dags/

# Restart Airflow
airflow dags list  # Verify DAG appears
```

Listing 14: Generate and Deploy DAG

9.5 Step 5: Test Execution

```
airflow dags trigger new_backend_api_test_mixed \
  --conf '{
    "state": "jharkhand",
    "district": "dumka",
    "block": "masalia",
    "start_year": 2017,
    "end_year": 2018
  }'

# Monitor execution
airflow dags list-runs -d new_backend_api_test_mixed
```

Listing 15: Trigger DAG Run

10 Docker Support (Optional)

If you want Docker execution mode, create a CLI:

10.1 CLI Script (cli.py)

```
import sys
import json
import argparse
from computing.lulc.lulc_v3_clip_river_basin import lulc_river_basin

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('algorithm', choices=[
        'LULC_Algorithm', 'Terrain_Algorithm', ...
    ])
    parser.add_argument('--state', required=True)
    parser.add_argument('--district', required=True)
    parser.add_argument('--block', required=True)
    parser.add_argument('--execution_id', required=True)

    args = parser.parse_args()

    if args.algorithm == 'LULC_Algorithm':
        result = lulc_river_basin(
            args.state, args.district, args.block,
            args.start_year, args.end_year
        )
```

```
23
24      # Output JSON (last line)
25      print(json.dumps({
26          "status": "success",
27          "execution_id": args.execution_id,
28          "asset_ids": result,
29          "execution_time": elapsed_time
30      }))
31
32  if __name__ == '__main__':
33      main()
```

Listing 16: Docker CLI Example

10.2 Dockerfile

```
1  FROM python:3.10-slim
2  WORKDIR /app
3  COPY requirements.txt .
4  RUN pip install -r requirements.txt
5  COPY . .
6  ENTRYPOINT ["python", "cli.py"]
```

Listing 17: Docker Container

```
1  docker build -t yourorg/stacd-backend:latest .
2
3  docker run --rm \
4    -v ~/.config/earthengine:/root/.config/earthengine:ro \
5    yourorg/stacd-backend:latest \
6    LULC_Algorithm \
7    --state jharkhand \
8    --district dumka \
9    --block masalia \
10   --execution_id test-uuid
```

Listing 18: Build and Test Docker

11 Future Work: Selective Recomputation

11.1 Planned Execution Modes

Mode	Status	Description
`fullexec`	Complete	Execute entire DAG
`update_algo`	Planned	Recompute from updated algorithm node
`update_dataset`	Planned	Recompute from updated dataset node
`update_dag`	Planned	Add new nodes to existing DAG
`resume_exec`	Planned	Resume from last failed node

Table 4: Execution Modes Roadmap

11.2 Update Algorithm Mode (Planned)

When an algorithm version changes, recompute only affected downstream nodes:

```
def determine_execution_path(**context):
    execution_type = context['params']['execution_type']

    if execution_type == 'update_algo':
        updated_algo = context['params']['updated_entity']
        version = context['params']['version']

        # Find all downstream nodes
        downstream_tasks = get_downstream_tasks(
            dag, updated_algo
        )

        # Skip tasks upstream of updated algorithm
        return [updated_algo] + downstream_tasks

    elif execution_type == 'fullexec':
        return all_tasks
```

Listing 19: Update Algorithm Pseudocode

11.3 Resume Execution Mode (Planned)

Store failed node information in database, allow resumption:

```
def determine_execution_path(**context):
    if execution_type == 'resume_exec':
        # Query database for last failed node
        last_run = db.get_last_dag_run(DAG_UUID)
        failed_node = last_run.failed_node

        # Resume from failed node onward
        return get_subgraph_from(dag, failed_node)
```

Listing 20: Resume Execution Pseudocode

12 Troubleshooting

12.1 Common Issues

Issue	Solution
API not blocking	Add polling loop with `time.sleep()`
Missing asset IDs	Return list in `asset_ids` field
Database errors	Run `init_db.py` to create tables
DAG not appearing	Check `/airflow/dags/` and restart scheduler
Docker mount errors	Use absolute paths in volume configuration

Table 5: Common Migration Issues

13 Appendix A: Complete Example

13.1 Full Backend Endpoint

```python
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
import uuid
import time

app = FastAPI()

class AlgorithmRequest(BaseModel):
    state: str
    district: str
    block: str
    start_year: int
    end_year: int
    execution_id: str

@app.post("/api/v1/lulc_v3/")
async def run_lulc_algorithm(request: AlgorithmRequest):
    """
    BLOCKING algorithm execution with full lineage.
    """
    start_time = time.time()

    print(f"[{request.execution_id}] Starting LULC")

    try:
        # Call your algorithm (BLOCKS until complete)
        asset_ids = lulc_river_basin(
            request.state,
            request.district,
            request.block,
            request.start_year,
            request.end_year
        )

        elapsed = time.time() - start_time

        return {
            "status": "success",
            "message": f"Created {len(asset_ids)} assets",
            "execution_id": request.execution_id,
            "asset_ids": asset_ids,
            "execution_time": elapsed
        }

    except Exception as e:
        return {
            "status": "failed",
            "execution_id": request.execution_id,
            "error": str(e)
        }
```

Listing 21: Complete API Endpoint Example

14 Appendix B: Database Queries

14.1 Query Lineage

```python
from db_operations import STACDDatabase

db = STACDDatabase('~/stacd_database/stacd.db')

# Get all executions for a DAG run
executions = db.get_algorithm_executions(
    dag_uuid='e0476807-d8fe-4b7d-8fc7-c5a73ccc9569'
)

for exec in executions:
    print(f"{exec.algo_type_id}: {exec.status} ({exec.execution_time}s)
        ")
    print(f"  Execution ID: {exec.execution_id}")

    # Get output datasets
    datasets = db.get_datasets_by_execution(exec.execution_id)
    for ds in datasets:
        print(f"  -> Dataset: {ds.dataset_type_id}")
        print(f"     Assets: {ds.asset_ids}")
```

Listing 22: Retrieve Execution Lineage

15 Conclusion

The STACD framework provides a robust foundation for reproducible geospatial pipelines. The critical requirements for migration are:

1. **Blocking API calls** that wait for computation completion
2. **Asset ID return** in standardized JSON format
3. **Execution ID support** for lineage tracking

With these modifications, your CoreStack pipelines can benefit from full lineage tracking, selective recomputation (planned), and YAML-driven DAG generation.

Next Steps:

- Implement selective recomputation modes
- Develop Airflow lineage visualization plugin
- Create admin UI for algorithm/dataset metadata management