

# **STACD Migration Guide**

Migrating CoreStack Pipelines to STACD Framework

Saharsh Laud

November 2025

## Contents

<b>1</b>	<b>Executive Summary</b>	<b>4</b>
1.1	Key Benefits . . . . .	4
<b>2</b>	<b>System Architecture Overview</b>	<b>4</b>
2.1	Component Architecture . . . . .	4
<b>3</b>	<b>Current Implementation Status</b>	<b>4</b>
3.1	Completed Features . . . . .	4
3.2	Proven Execution Modes . . . . .	4
<b>4</b>	<b>Backend Structure</b>	<b>5</b>
4.1	Directory Layout . . . . .	5
4.2	Backend API Endpoints . . . . .	5
<b>5</b>	<b>Database Schema</b>	<b>6</b>
5.1	Core Tables . . . . .	6
5.2	Lineage Tracking . . . . .	6
<b>6</b>	<b>YAML Configuration Format</b>	<b>7</b>
6.1	DAG Definition (stacd_dag.yaml) . . . . .	7
6.2	Algorithm Repository (stacd_algorithm_repo.yaml) . . . . .	7
6.3	Algorithm Type Definition . . . . .	8
<b>7</b>	<b>DAG Generator</b>	<b>8</b>
7.1	Generator Workflow . . . . .	8
7.2	Execution Mode Selection . . . . .	8
7.3	Running the Generator . . . . .	9
<b>8</b>	<b>Migration Requirements for CoreStack</b>	<b>9</b>
8.1	Critical Backend Requirements . . . . .	9
8.1.1	1. Blocking Execution . . . . .	9
8.1.2	2. Asset ID Return . . . . .	10
8.1.3	3. Execution ID Support . . . . .	10
8.2	Additional Requirements . . . . .	10
<b>9</b>	<b>Migration Steps</b>	<b>10</b>
9.1	Step 1: Adapt Backend API . . . . .	10
9.2	Step 2: Create YAML Configurations . . . . .	11
9.3	Step 3: Initialize Database . . . . .	11
9.4	Step 4: Generate DAG . . . . .	12
9.5	Step 5: Test Execution . . . . .	12
<b>10</b>	<b>Docker Support (Optional)</b>	<b>12</b>
10.1	CLI Script (cli.py) . . . . .	12
10.2	Dockerfile . . . . .	13
<b>11</b>	<b>Future Work: Selective Recomputation</b>	<b>13</b>
11.1	Planned Execution Modes . . . . .	13
11.2	Update Algorithm Mode (Planned) . . . . .	14
11.3	Resume Execution Mode (Planned) . . . . .	14

<b>12 Troubleshooting</b>	<b>14</b>
12.1 Common Issues . . . . .	14
<b>13 Contact and Support</b>	<b>14</b>
<b>14 Appendix A: Complete Example</b>	<b>15</b>
14.1 Full Backend Endpoint . . . . .	15
<b>15 Appendix B: Database Queries</b>	<b>16</b>
15.1 Query Lineage . . . . .	16
<b>16 Conclusion</b>	<b>16</b>

# 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

```

1 stacd_backend/
2     Dockerfile                # Container definition
3     cli.py                    # Command-line interface
4     main.py                   # FastAPI application
5     requirements.txt          # Python dependencies
6     computing/
7         lulc/                  # LULC algorithms
8             lulc_v3_clip_river_basin.py
9             lulc_vector.py
10            cropping_frequency.py
11            misc.py
12        terrain/               # Terrain algorithms
13            terrain_raster.py
14            terrain_clusters.py
15            terrain_utils.py
16        lulcxterrain/          # Cross-domain algorithms
17            lulc_on_slope_cluster.py
18            lulc_on_plain_cluster.py
19            terrain_classifier.py
20        utils/                 # Shared utilities
21            constants.py
22            gee_utils.py

```

Listing 1: Backend Directory Structure

### 4.2 Backend API Endpoints

```

1 @app.post("/api/v1/lulc_v3/")
2 async def run_lulc_algorithm(
3     state: str,
4     district: str,
5     block: str,
6     start_year: int,
7     end_year: int,
8     execution_id: str
9 ):
10     """

```

```

11     Execute LULC algorithm with blocking behavior.
12     Returns only when computation is complete.
13     """
14     asset_ids = lulc_river_basin(
15         state, district, block,
16         start_year, end_year
17     )
18
19     return {
20         "status": "success",
21         "execution_id": execution_id,
22         "asset_ids": asset_ids,
23         "execution_time": elapsed_time
24     }

```

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

```

1  # Algorithm execution logging
2  db.log_algorithm_execution(
3      execution_id=uuid.uuid4(),
4      algo_type_id=algo_type.algo_type_id,
5      dag_uuid=DAG_UUID,
6      dag_run_id=context['dag_run'].run_id,
7      node_name='execute_LULC_Algorithm',
8      status='running',
9      parameters={'state': 'jharkhand', ...}
10 )
11
12 # Dataset instance logging
13 db.log_dataset_instance(
14     dataset_uuid=uuid.uuid4(),
15     dataset_type_id=dataset_type.dataset_type_id,
16     execution_id=execution_id,
17     dag_uuid=DAG_UUID,
18     asset_ids=['projects/ee-saharshlaud/...'],
19     region_params={'state': 'jharkhand', ...}
20 )

```

Listing 3: Example Database Logging

## 6 YAML Configuration Format

### 6.1 DAG Definition (stacd\_dag.yaml)

```

1 ---
2 !DAG
3 id: new_backend_api_test
4 name: CoreStack LULC Pipeline
5 version: "1.0"
6 description: End-to-end LULC and Terrain processing
7 alg_type_nodes:
8   - LULC_Algorithm
9   - Terrain_Algorithm
10  - LULC_Vectorization
11  - Terrain_Vectorization
12  - Terrain_LULC_Slope
13  - Terrain_LULC_Plain
14 dataset_type_nodes:
15   - LULC_Raster
16   - Terrain_Raster
17   - LULC_Vector
18   - Terrain_Vector
19   - Terrain_LULC_Vector_Slope
20   - Terrain_LULC_Vector_Plain
21 params:
22   - state
23   - district
24   - block
25   - start_year
26   - end_year

```

Listing 4: DAG Configuration Example

### 6.2 Algorithm Repository (stacd\_algorithm\_repo.yaml)

```

1 ---
2 !Algorithm_Instance
3 type: LULC_Algorithm
4 version: "1"
5 assets:
6   code: "https://github.com/your-org/stacd-backend"
7 date: 2024-11-06 00:00:00
8 execution_modes:
9   api:
10     enabled: true
11     url: "http://localhost:8000/api/v1/lulc_v3/"
12     priority: 1 # API preferred
13   docker:
14     enabled: true
15     image: "yourorg/stacd-backend:latest"
16     priority: 2
17     volumes:
18       - source: "~/.config/earthengine"
19         target: "/root/.config/earthengine"
20         read_only: true

```

Listing 5: Algorithm Instance Definition

## 6.3 Algorithm Type Definition

```

1  ---
2  !Algorithm_Type
3  id: LULC_Algorithm
4  name: LULC River Basin Clipper
5  version: "1.0"
6  params:
7    - name: state
8      type: string
9    - name: district
10     type: string
11    - name: block
12     type: string
13    - name: start_year
14     type: integer
15    - name: end_year
16     type: integer
17  inputs:
18    - state
19    - district
20    - block
21    - start_year
22    - end_year
23  input_datasets: [] # No input datasets (starting node)
24  outputs:
25    - 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:

```

1  def get_execution_mode(self):
2      """Select execution mode based on priority"""
3      api_config = self.execution_modes.get('api', {})
4      docker_config = self.execution_modes.get('docker', {})
5
6      api_enabled = api_config.get('enabled', False)
7      docker_enabled = docker_config.get('enabled', False)
8
9      # Only one enabled
10     if api_enabled and not docker_enabled:

```



```

11         return 'api', api_config
12     if docker_enabled and not api_enabled:
13         return 'docker', docker_config
14
15     # Both enabled: use priority (lower = higher priority)
16     if api_enabled and docker_enabled:
17         api_priority = api_config.get('priority', 999)
18         docker_priority = docker_config.get('priority', 999)
19         if api_priority < docker_priority:
20             return 'api', api_config
21         else:
22             return 'docker', docker_config
23
24     return None, None

```

Listing 7: Priority-Based Mode Selection

## 7.3 Running the Generator

```

1 cd ~/airflow/new_server/mixed/dag_generator
2 python3 stacd_mixed_dag_generator.py
3
4 # Output:
5 # Generated MIXED DAG with dependencies + database
6 # DAG UUID: e0476807-d8fe-4b7d-8fc7-c5a73ccc9569
7 # Output: ~/airflow/new_server/mixed/generated_dags/generated_mixed_dag
8   .py
9
10 # Deploy to Airflow
11 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

```

1 def algorithm_function(params):
2     """REQUIRED: Block until computation complete"""
3
4     # Submit tasks to GEE
5     tasks = submit_gee_tasks(params)
6
7     # CRITICAL: Wait for completion
8     while not all_tasks_complete(tasks):

```

```

9         time.sleep(5)
10        check_task_status(tasks)
11
12        # Return only when done
13    return {
14        "status": "success",
15        "asset_ids": [task.asset_id for task in tasks],
16        "execution_time": elapsed_time
17    }

```

Listing 9: Blocking Call Pattern

### 8.1.2 2. Asset ID Return

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

```

1  {
2      "status": "success",
3      "execution_id": "ad59386f-fcca-426e-b555-c70f6286a877",
4      "asset_ids": [
5          "projects/ee-user/assets/lulc_2017_dumka",
6          "projects/ee-user/assets/lulc_2018_dumka"
7      ],
8      "execution_time": 28.7
9  }

```

Listing 10: Return Format

### 8.1.3 3. Execution ID Support

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

```

1  @app.post("/api/v1/your_algorithm/")
2  async def run_algorithm(
3      # ... your parameters ...
4      execution_id: str # REQUIRED parameter
5  ):
6      print(f"[{execution_id}] Starting algorithm")
7      result = compute(params)
8      return {
9          "execution_id": execution_id, # Echo back
10         "asset_ids": result.assets,
11         ...
12     }

```

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:

```

1 curl -X POST http://localhost:8000/api/v1/your_algorithm/ \
2   -H "Content-Type: application/json" \
3   -d '{
4     "state": "jharkhand",
5     "district": "dumka",
6     "block": "masalia",
7     "execution_id": "test-uuid-1234"
8   },'
9
10 # Expected response (blocks until complete):
11 {
12   "status": "success",
13   "execution_id": "test-uuid-1234",
14   "asset_ids": ["projects/.../asset1", "projects/.../asset2"],
15   "execution_time": 45.2
16 }
```

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

```

1 cd ~/stacd_database
2 python3 init_db.py
3
4 # Test database connection
5 python3 test_db.py
```

Listing 13: Database Setup

## 9.4 Step 4: Generate DAG

```

1 cd ~/airflow/new_server/mixed/dag_generator
2 python3 stacd_mixed_dag_generator.py
3 cp ../generated_dags/generated_mixed_dag.py ~/airflow/dags/
4
5 # Restart Airflow
6 airflow dags list # Verify DAG appears

```

Listing 14: Generate and Deploy DAG

## 9.5 Step 5: Test Execution

```

1 airflow dags trigger new_backend_api_test_mixed \
2   --conf '{
3     "state": "jharkhand",
4     "district": "dumka",
5     "block": "masalia",
6     "start_year": 2017,
7     "end_year": 2018
8   },'
9
10 # Monitor execution
11 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)

```

1 import sys
2 import json
3 import argparse
4 from computing.lulc.lulc_v3_clip_river_basin import lulc_river_basin
5
6 def main():
7     parser = argparse.ArgumentParser()
8     parser.add_argument('algorithm', choices=[
9         'LULC_Algorithm', 'Terrain_Algorithm', ...
10    ])
11     parser.add_argument('--state', required=True)
12     parser.add_argument('--district', required=True)
13     parser.add_argument('--block', required=True)
14     parser.add_argument('--execution_id', required=True)
15
16     args = parser.parse_args()
17
18     if args.algorithm == 'LULC_Algorithm':
19         result = lulc_river_basin(
20             args.state, args.district, args.block,
21             args.start_year, args.end_year
22         )

```

```

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:

```

1 def determine_execution_path(**context):
2     execution_type = context['params']['execution_type']
3
4     if execution_type == 'update_algo':
5         updated_algo = context['params']['updated_entity']
6         version = context['params']['version']
7
8         # Find all downstream nodes
9         downstream_tasks = get_downstream_tasks(
10             dag, updated_algo
11         )
12
13         # Skip tasks upstream of updated algorithm
14         return [updated_algo] + downstream_tasks
15
16     elif execution_type == 'fullexec':
17         return all_tasks

```

Listing 19: Update Algorithm Pseudocode

## 11.3 Resume Execution Mode (Planned)

Store failed node information in database, allow resumption:

```

1 def determine_execution_path(**context):
2     if execution_type == 'resume_exec':
3         # Query database for last failed node
4         last_run = db.get_last_dag_run(DAG_UUID)
5         failed_node = last_run.failed_node
6
7         # Resume from failed node onward
8         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 <code>time.sleep()</code>
Missing asset IDs	Return list in <code>asset_ids</code> field
Database errors	Run <code>init_db.py</code> to create tables
DAG not appearing	Check <code>/airflow/dags/</code> and restart scheduler
Docker mount errors	Use absolute paths in volume configuration

Table 5: Common Migration Issues

# 13 Contact and Support

For questions or issues during migration:

- **Technical Lead:** Saharsh Laud ([saharsh.laud@iitd.ac.in](mailto:saharsh.laud@iitd.ac.in))
- **Research Supervisor:** Prof. Mudit Verma (IIT Delhi)
- **Documentation:** <https://core-stack.org/stacd>
- **GitHub:** <https://github.com/your-org/stacd-framework>

## 14 Appendix A: Complete Example

### 14.1 Full Backend Endpoint

```

1 from fastapi import FastAPI, HTTPException
2 from pydantic import BaseModel
3 import uuid
4 import time
5
6 app = FastAPI()
7
8 class AlgorithmRequest(BaseModel):
9     state: str
10    district: str
11    block: str
12    start_year: int
13    end_year: int
14    execution_id: str
15
16 @app.post("/api/v1/lulc_v3/")
17 async def run_lulc_algorithm(request: AlgorithmRequest):
18     """
19     BLOCKING algorithm execution with full lineage.
20     """
21     start_time = time.time()
22
23     print(f"[{request.execution_id}] Starting LULC")
24
25     try:
26         # Call your algorithm (BLOCKS until complete)
27         asset_ids = lulc_river_basin(
28             request.state,
29             request.district,
30             request.block,
31             request.start_year,
32             request.end_year
33         )
34
35         elapsed = time.time() - start_time
36
37         return {
38             "status": "success",
39             "message": f"Created {len(asset_ids)} assets",
40             "execution_id": request.execution_id,
41             "asset_ids": asset_ids,
42             "execution_time": elapsed
43         }
44
45     except Exception as e:

```

```

46         return {
47             "status": "failed",
48             "execution_id": request.execution_id,
49             "error": str(e)
50         }

```

Listing 21: Complete API Endpoint Example

## 15 Appendix B: Database Queries

### 15.1 Query Lineage

```

1  from db_operations import STACDDatabase
2
3  db = STACDDatabase('~/stacd_database/stacd.db')
4
5  # Get all executions for a DAG run
6  executions = db.get_algorithm_executions(
7      dag_uuid='e0476807-d8fe-4b7d-8fc7-c5a73ccc9569'
8  )
9
10 for exec in executions:
11     print(f"{exec.algo_type_id}: {exec.status} ({exec.execution_time}s)
12         ")
13     print(f"    Execution ID: {exec.execution_id}")
14
15     # Get output datasets
16     datasets = db.get_datasets_by_execution(exec.execution_id)
17     for ds in datasets:
18         print(f"        -> Dataset: {ds.dataset_type_id}")
19         print(f"        Assets: {ds.asset_ids}")

```

Listing 22: Retrieve Execution Lineage

## 16 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