

# Source: CREDENTIAL\_VENDING\_DOCS\_UPDATE.md

## Documentation Update Summary - Credential Vending

### Files Requiring Updates

#### 1. `/docs/features/security\_vending.md` ⚠ HIGH PRIORITY

**Current State:** Only documents S3/AWS credential vending

**Required Updates:**

Add section for Azure ADLS Gen2 credential vending

OAuth2 mode (adls.token)

Account key mode (adls.account-name, adls.account-key)

Add section for GCP credential vending

OAuth2 mode (gcp-oauth-token)

Service account mode (gcp-project-id)

Update Pylceberg integration examples with multi-cloud

Add warehouse configuration examples for Azure and GCP

**Suggested New Sections:**

```
## Azure ADLS Gen2 Credential Vending

### OAuth2 Mode
```python
# Warehouse configuration
{
  "type": "azure",
  "account_name": "mystorageaccount",
  "container": "data",
  "tenant_id": "...",
  "client_id": "...",
  "client_secret": "..."
}
```

### Account Key Mode

```
# Warehouse configuration
{
  "type": "azure",
```

```
"account_name": "mystorageaccount",
"container": "data",
"account_key": "...
}
```

## GCP Credential Vending

...

```
---

### 2. `/docs/architecture/signer-trait.md` ⚠ MEDIUM PRIORITY
**Current State:** Documents old Signer trait, outdated structure
**Required Updates:**
- Replace with new `CredentialSigner` trait documentation
- Update `VendedCredentials` structure
- Document PyIceberg-compatible property names:
  - S3: `s3.access-key-id`, `s3.secret-access-key`, `s3.session-token`
  - Azure: `adls.token`, `adls.account-name`, `adls.account-key`
  - GCP: `gcp-oauth-token`, `gcp-project-id`
- Update implementation examples for all three cloud providers

**New Trait Definition:**
```rust
#[async_trait]
pub trait CredentialSigner: Send + Sync {
    async fn generate_credentials(
        &self,
        resource_path: &str,
        permissions: &[String],
        duration: Duration,
    ) -> Result<VendedCredentials>;

    fn storage_type(&self) -> &str;
}

pub struct VendedCredentials {
    pub prefix: String,
    pub config: HashMap<String, String>, // PyIceberg-compatible properties
    pub expires_at: Option<DateTime<Utc>>,
}
```

## 3. `/docs/pyiceberg/multi\_cloud.md` ⚠ HIGH PRIORITY

**Current State:** Mentions vending but uses incorrect property names

**Required Updates:**

Update Azure properties section:

-

`adls.token` (for OAuth2 vended credentials)

`adls.account-name` (account name)

`adls.account-key` (for account key mode)

Remove outdated `adls.connection-string` (not used in vending)

Update GCP properties section:

`gcp-oauth-token` (for vended OAuth2 tokens)

`gcp-project-id` (project ID)

Remove `gcs.oauth2.token` (use `gcp-oauth-token`)

Add complete credential vending examples

### Updated Azure Example:

```
catalog = load_catalog(  
    "azure_catalog",  
    **{  
        "type": "rest",  
        "uri": "http://localhost:8080/v1/azure_catalog",  
        "token": "YOUR_JWT_TOKEN",  
        # Pangolin vends credentials automatically!  
        # Properties vended: adls.token, adls.account-name  
    }  
)
```

## 4. `/docs/features/warehouse_management.md` LOW PRIORITY

**Current State:** May only show S3 warehouse examples

**Required Updates:**

Add Azure warehouse creation example

Add GCP warehouse creation example

Document `use\_sts` behavior for each cloud provider

## 5. `/docs/pyiceberg/auth_vended_creds.md` MEDIUM PRIORITY

**Current State:** Unknown (need to review)

**Likely Updates:**

Add multi-cloud vended credentials examples

Update property names to PyIceberg-compatible format

## Property Name Reference (PyIceberg Compatible)

## S3 / AWS

Property	Description	Example
-----	-----	-----
`s3.access-key-id`	AWS access key	`AKIAIOSFODNN7EXAMPLE`
`s3.secret-access-key`	AWS secret key	`wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY`
`s3.session-token`	STS session token	`FwoGZXIvYXdzEBYaD...`
`s3.endpoint`	S3 endpoint URL	`http://localhost:9000`
`s3.region`	AWS region	`us-east-1`

## Azure ADLS Gen2

Property	Description	Example
-----	-----	-----
`adls.token`	OAuth2 access token	`eyJ0eXAiOiJKV1QiLCJhbGc...`
`adls.account-name`	Storage account name	`mystorageaccount`
`adls.account-key`	Storage account key	`Eby8vdM02xNOcqFlqUwJPLlmEtlCDXJ1OUzFT50uSRZ6IFsuFq2UVERCz4I6tq/K1SZFPTOtr/KBHBeksoGMGw==`
`adls.container`	Container name	`data`

## GCP Cloud Storage

Property	Description	Example
-----	-----	-----
`gcp-oauth-token`	OAuth2 access token	`ya29.a0AfH6SMBx...`
`gcp-project-id`	GCP project ID	`my-project-12345`

## Testing Status

### Tested and Working

**S3 Static Credentials:** Tested with MinIO

**S3 STS Foundation:** MinIO IAM configured

**Azure Account Key:** Tested with Azurite

**GCP Service Account:** Tested with fake-gcs-server

⚠ Code Ready, Needs Real Testing

**Azure OAuth2:** Code complete, needs real Azure AD testing

**S3 STS AssumeRole:** Needs AWS STS feature flag enabled

## Implementation Details

## Files Modified:

`pangolin\_api/src/credential\_signers/azure\_signer.rs` - Pylceberg properties

`pangolin\_api/src/credential\_signers/gcp\_signer.rs` - Pylceberg properties

`pangolin\_api/src/credential\_signers/s3\_signer.rs` - Pylceberg properties

`pangolin\_api/src/credential\_vending.rs` - Factory and helpers

`pangolin\_api/src/signing\_handlers.rs` - Refactored (-170 lines)

**Tests:** 28/28 passing

4 unit tests

15 integration tests

5 end-to-end tests

4 live emulator tests

## Recommended Documentation Update Priority

**HIGH:** `security\_vending.md` - Add Azure/GCP sections

**HIGH:** `multi\_cloud.md` - Fix property names

**MEDIUM:** `signer-trait.md` - Update trait documentation

**MEDIUM:** `auth\_vended\_creds.md` - Add multi-cloud examples

**LOW:** `warehouse\_management.md` - Add Azure/GCP warehouse examples

## Quick Reference for Documentation Writers

### When documenting Azure credential vending:

Use `adls.token` (not `azure-oauth-token`)

Use `adls.account-name` (not `azure-account-name`)

Use `adls.account-key` (not `azure-account-key`)

### When documenting GCP credential vending:

Use `gcp-oauth-token` (not `gcs.oauth2.token`)

Use `gcp-project-id` (not `gcs.project-id`)

**All property names must match Pylceberg expectations for compatibility.**

Source:  
DEPLOYMENT\_GUIDE\_AUDIT\_LOGGING.md

# Enhanced Audit Logging - Deployment Guide

**Version:** 1.0.0

**Date:** December 18, 2025

**Status:** Production Ready

## Table of Contents

Pre-Deployment Checklist

Database Migrations

API Integration

Testing

Production Deployment

Monitoring & Verification

Rollback Procedures

Troubleshooting

## Pre-Deployment Checklist

Before deploying the enhanced audit logging system, ensure:

- [ ] All code changes are committed and reviewed
- [ ] Database backup is created
- [ ] Test environment is available
- [ ] Deployment window is scheduled (recommended: off-peak hours)
- [ ] Rollback plan is documented
- [ ] Team is notified of deployment

## Database Migrations

### Step 1: Backup Current Database

#### PostgreSQL:

```
pg_dump -U postgres -d pangolin > pangolin_backup_$(date +%Y%m%d_%H%M%S).sql
```

#### SQLite:

```
cp pangolin.db pangolin_backup_$(date +%Y%m%d_%H%M%S).db
```

## MongoDB:

```
mongodump --db pangolin --out=pangolin_backup_$(date +%Y%m%d_%H%M%S)
```

## Step 2: Run Migrations

### PostgreSQL

```
# Navigate to migrations directory
cd migrations/postgres

# Review the migration script
cat 001_enhanced_audit_logging.sql

# Run the migration
psql -U postgres -d pangolin < 001_enhanced_audit_logging.sql

# Verify the migration
psql -U postgres -d pangolin -c "SELECT COUNT(*) FROM audit_logs;"
psql -U postgres -d pangolin -c "SELECT action, COUNT(*) FROM audit_logs GROUP BY action LIMIT 10;"
```

### SQLite

```
# Navigate to migrations directory
cd migrations/sqlite

# Review the migration script
cat 001_enhanced_audit_logging.sql

# Run the migration
sqlite3 pangolin.db < 001_enhanced_audit_logging.sql

# Verify the migration
sqlite3 pangolin.db "SELECT COUNT(*) FROM audit_logs;"
sqlite3 pangolin.db "SELECT action, COUNT(*) FROM audit_logs GROUP BY action LIMIT 10;"
```

## MongoDB

No migration needed - the enhanced schema is automatically applied when new audit events are logged.

## Step 3: Verify Migration Success

Run these verification queries:

### PostgreSQL/SQLite:

```
-- Check table structure
SELECT column_name, data_type
FROM information_schema.columns
WHERE table_name = 'audit_logs';

-- Check for NULL values in required fields
SELECT
    COUNT(*) FILTER (WHERE username IS NULL) as null_username,
    COUNT(*) FILTER (WHERE action IS NULL) as null_action,
    COUNT(*) FILTER (WHERE resource_name IS NULL) as null_resource_name,
    COUNT(*) FILTER (WHERE result IS NULL) as null_result
FROM audit_logs;

-- Check index creation
SELECT indexname FROM pg_indexes WHERE tablename = 'audit_logs';
```

### Expected Results:

All required fields should have 0 NULL values

At least 7 indexes should be created

Old data should be migrated successfully

## API Integration

### Step 1: Verify Code Changes

The following files have been updated:

**`pangolin\_api/src/lib.rs`:**

Added `pub mod audit\_handlers;`

Updated audit routes to use new handlers

**`pangolin\_api/src/audit\_handlers.rs`:**

New file with 3 endpoint handlers

Supports filtering, pagination, and counting

### Step 2: Compile and Test

```
# Compile the API
cargo build --release --manifest-path pangolin/Cargo.toml -p pangolin_api

# Run unit tests
cargo test --manifest-path pangolin/Cargo.toml -p pangolin_store audit

# Run integration tests (if DATABASE_URL is set)
export DATABASE_URL="postgres://postgres:postgres@localhost/pangolin"
```

```
cargo test --manifest-path pangolin/Cargo.toml -p pangolin_store postgres_audit
```

## Step 3: Start the API Server

```
# Set environment variables
export DATABASE_URL="postgresql://postgres:postgres@localhost/pangolin"
export PANGOLIN_STORAGE_TYPE="postgres" # or "mongodb", "sqlite"

# Start the server
cargo run --release --manifest-path pangolin/Cargo.toml -p pangolin_api
```

## Testing

### Manual API Testing

#### 1. List Audit Events (No Filter)

```
curl -H "Authorization: Bearer YOUR_TOKEN" \
-H "X-Pangolin-Tenant: YOUR_TENANT_ID" \
"http://localhost:8080/api/v1/audit"
```

**Expected Response:** JSON array of audit events

#### 2. Filter by Action

```
curl -H "Authorization: Bearer YOUR_TOKEN" \
-H "X-Pangolin-Tenant: YOUR_TENANT_ID" \
"http://localhost:8080/api/v1/audit?action=create_table&limit=10"
```

**Expected Response:** JSON array of CreateTable audit events (max 10)

#### 3. Filter by Result

```
curl -H "Authorization: Bearer YOUR_TOKEN" \
-H "X-Pangolin-Tenant: YOUR_TENANT_ID" \
"http://localhost:8080/api/v1/audit?result=failure"
```

**Expected Response:** JSON array of failed audit events

#### 4. Filter by Time Range

```
START_TIME=$(date -u -d '7 days ago' +%Y-%m-%dT%H:%M:%SZ)
END_TIME=$(date -u +%Y-%m-%dT%H:%M:%SZ)
```

```
curl -H "Authorization: Bearer YOUR_TOKEN" \
-H "X-Pangolin-Tenant: YOUR_TENANT_ID" \
"http://localhost:8080/api/v1/audit?start_time=$START_TIME&end_time=$END_TIME"
```

**Expected Response:** JSON array of audit events from the last 7 days

## 5. Pagination

```
# First page
curl -H "Authorization: Bearer YOUR_TOKEN" \
-H "X-Pangolin-Tenant: YOUR_TENANT_ID" \
"http://localhost:8080/api/v1/audit?limit=50&offset=0"

# Second page
curl -H "Authorization: Bearer YOUR_TOKEN" \
-H "X-Pangolin-Tenant: YOUR_TENANT_ID" \
"http://localhost:8080/api/v1/audit?limit=50&offset=50"
```

**Expected Response:** Different sets of 50 audit events

## 6. Count Events

```
curl -H "Authorization: Bearer YOUR_TOKEN" \
-H "X-Pangolin-Tenant: YOUR_TENANT_ID" \
"http://localhost:8080/api/v1/audit/count?action=create_table"
```

**Expected Response:** `{"count": N}` where N is the number of CreateTable events

## 7. Get Specific Event

```
curl -H "Authorization: Bearer YOUR_TOKEN" \
-H "X-Pangolin-Tenant: YOUR_TENANT_ID" \
"http://localhost:8080/api/v1/audit/EVENT_UUID"
```

**Expected Response:** Single audit event object or 404 if not found

## Automated Testing

Run the live test script:

```
cd scripts
chmod +x test_audit_logging.sh
./test_audit_logging.sh
```

**Expected Output:** All test scenarios should pass

# Production Deployment

## Deployment Steps

### Stop the API Server:

```
systemctl stop pangolin-api  
# or  
pkill -f pangolin_api
```

### Run Database Migrations:

```
# Follow steps in "Database Migrations" section above
```

### Deploy New Code:

```
# Pull latest code  
git pull origin main  
  
# Build release binary  
cargo build --release --manifest-path pangolin/Cargo.toml -p pangolin_api  
  
# Copy binary to deployment location  
cp target/release/pangolin_api /usr/local/bin/
```

### Start the API Server:

```
systemctl start pangolin-api  
# or  
/usr/local/bin/pangolin_api &
```

### Verify Deployment:

```
# Check server is running  
curl http://localhost:8080/health  
  
# Test audit endpoint  
curl -H "Authorization: Bearer TOKEN" \  
      -H "X-Pangolin-Tenant: TENANT_ID" \  
      "http://localhost:8080/api/v1/audit?limit=1"
```

## Environment Variables

Ensure these environment variables are set:

```
# Database connection  
export DATABASE_URL="postgresql://user:pass@host:5432/pangolin"
```

```
# Storage type
export PANGOLIN_STORAGE_TYPE="postgres" # or "mongodb", "sqlite"

# Optional: Logging level
export RUST_LOG="info,pangolin_api=debug"
```

## Monitoring & Verification

### Health Checks

#### API Health:

```
curl http://localhost:8080/health
```

Expected: `OK`

#### Database Connectivity:

```
psql -U postgres -d pangolin -c "SELECT 1;"
```

Expected: `1`

#### Audit Logging:

```
# Create a test event (e.g., create a catalog)
curl -X POST -H "Authorization: Bearer TOKEN" \
  -H "Content-Type: application/json" \
  -H "X-Pangolin-Tenant: TENANT_ID" \
  -d '{"name":"test_catalog"}' \
  "http://localhost:8080/api/v1/catalogs"

# Verify it was logged
curl -H "Authorization: Bearer TOKEN" \
  -H "X-Pangolin-Tenant: TENANT_ID" \
  "http://localhost:8080/api/v1/audit?action=create_catalog&limit=1"
```

Expected: Audit event for catalog creation

### Metrics to Monitor

**Audit Event Count:** Should increase with system activity

**Query Performance:** Audit queries should complete in <100ms

**Error Rate:** Should be <0.1% for audit operations

**Database Size:** Monitor audit\_logs table growth

# Performance Benchmarks

Expected performance (with proper indexes):

List audit events (100 records): <50ms

Count audit events: <20ms

Get specific event: <10ms

Insert audit event: <5ms

## Rollback Procedures

If issues are encountered, follow these rollback steps:

### Step 1: Stop the API Server

```
systemctl stop pangolin-api
```

### Step 2: Restore Database

#### **PostgreSQL:**

```
psql -U postgres -d pangolin < pangolin_backup_TIMESTAMP.sql
```

#### **SQLite:**

```
cp pangolin_backup_TIMESTAMP.db pangolin.db
```

#### **MongoDB:**

```
mongorestore --db pangolin pangolin_backup_TIMESTAMP/pangolin
```

### Step 3: Revert Code Changes

```
git revert HEAD # or specific commit  
cargo build --release --manifest-path pangolin/Cargo.toml -p pangolin_api
```

### Step 4: Restart API Server

```
systemctl start pangolin-api
```

### Step 5: Verify Rollback

```
curl http://localhost:8080/health
curl -H "Authorization: Bearer TOKEN" \
-H "X-Pangolin-Tenant: TENANT_ID" \
"http://localhost:8080/api/v1/audit?limit=1"
```

## Troubleshooting

### Issue: Migration Fails with Foreign Key Error

**Cause:** Referenced tables (tenants, users) don't exist

**Solution:**

```
-- Temporarily disable foreign key checks
SET CONSTRAINTS ALL DEFERRED; -- PostgreSQL
PRAGMA foreign_keys = OFF;    -- SQLite

-- Run migration
-- ...

-- Re-enable foreign key checks
SET CONSTRAINTS ALL IMMEDIATE; -- PostgreSQL
PRAGMA foreign_keys = ON;      -- SQLite
```

### Issue: API Returns 500 Error

**Cause:** Database connection issue or missing indexes

**Solution:**

Check database connectivity:

```
psql -U postgres -d pangolin -c "SELECT 1;"
```

Verify indexes exist:

```
SELECT indexname FROM pg_indexes WHERE tablename = 'audit_logs';
```

Check API logs:

```
journalctl -u pangolin-api -n 100
```

### Issue: Slow Query Performance

**Cause:** Missing indexes or large dataset

## Solution:

Verify indexes:

```
EXPLAIN ANALYZE SELECT * FROM audit_logs WHERE tenant_id = 'UUID' ORDER BY timestamp  
DESC LIMIT 100;
```

Add missing indexes if needed

Consider partitioning for very large tables (>10M rows)

## Issue: Type Inference Errors in Tests

**Cause:** Missing type annotations

**Solution:** Add explicit type annotations:

```
let store: PostgresStore = PostgresStore::new(&connection_string).await?;
```

## Additional Resources

**API Documentation:** <http://localhost:8080/swagger-ui>

**Migration Scripts:** ``/migrations/``

**Test Scripts:** ``/scripts/test_audit_logging.sh``

**Implementation Guide:** ``./gemini/brain/.../final_summary.md``

## Post-Deployment Checklist

After deployment, verify:

- [ ] Database migration completed successfully
- [ ] API server is running
- [ ] Health check passes
- [ ] Audit endpoints return data
- [ ] Filtering works correctly
- [ ] Pagination works correctly
- [ ] Performance is acceptable (<100ms for queries)
- [ ] No errors in logs
- [ ] Monitoring is active
- [ ] Team is notified of successful deployment

# Success Criteria

Deployment is successful when:

- ~All database migrations complete without errors
- ~API compiles and starts successfully
- ~All audit endpoints return expected responses
- ~Query performance meets benchmarks
- ~No errors in application logs
- ~Existing functionality remains unaffected

**Deployment completed successfully!**

For questions or issues, contact the development team.

Source: README.md

## Pangolin Documentation

Welcome to the comprehensive documentation for **Pangolin**, the cloud-native Apache Iceberg REST Catalog. Use the categories below to navigate the guides, feature deep-dives, and tool references.

### 1. Getting Started

\*Quickest path from zero to a running lakehouse.\*

[Onboarding Index](#) - **Start Here!**

[Installation Guide](#) - Run Pangolin in 5 minutes.

[Evaluating Pangolin](#) - Rapid local testing with `NO\_AUTH` mode.

[Deployment Guide](#) - Local, Docker, and Production setup.

[Environment Variables](#) - Complete system configuration reference.

### 2. Core Infrastructure

\*Managing the foundations: storage and metadata.\*

[Infrastructure Features](#) - Index of all platform capabilities.

[Warehouse Management](#) - Configuring S3, Azure, and GCS storage.

[Metadata Backends](#) - Memory, Postgres, MongoDB, and SQLite.

[Asset Management](#) - Tables, Views, and CRUD operations.

Federated Catalogs - Proxying external REST catalogs.

Known Issues - Documented limitations and active bugs (e.g., SQL backend quirks).

## 🛡️ 3. Governance & Security

\*Multi-tenancy, RBAC, and auditing.\*

Security Concepts - Identity and Credential Vending principles.

Credential Vending (IAM Roles) - Scoped cloud access (STS, SAS, Downscoped).

Permission System - Understanding RBAC and granular grants.

Service Users - Programmatic access and API key management.

Audit Logging - Global action tracking and compliance.

## 4. Data Life Cycle

\*Git-for-Data and maintenance workflows.\*

Branch Management - Working with isolated data environments.

Merge Operations - The 3-way merge workflow.

Merge Conflicts - Theory and resolution strategies.

Business Metadata & Discovery - Search, tags, and access requests.

Maintenance Utilities - Snapshot expiration and compaction.

## 5. Interfaces & Integration

\*Connecting tools and using our management layers.\*

Management UI - Visual guide to the administration portal.

PyPangolin SDK (Official) - Rich Python client with Git-like operations and types.

Pylceberg Integration - Native Python client configuration.

CLI Reference - Documentation for `pangolin-admin` and `pangolin-user`.

API Reference - Iceberg REST and Management API specs.

## 6. Architecture & Internals

\*Deep-dives for developers and contributors.\*

Architecture Overview - System design and component interaction.

Technical Logic Deep-Dive - Caching, Branching (Git-for-Data), and Trait details.

-

[Data Models](#) - Understanding the internal schema.

[CatalogStore Trait](#) - Extending Pangolin storage.

[Developer Utilities](#) - Tools for contributors (e.g. OpenAPI generation).

## 7. Best Practices

\*Production guides and operational wisdom.\*

[Best Practices Index](#) - Complete guide to operating Pangolin.

[Deployment & Security](#) - Production checklists.

[Scalability](#) - Tuning for high performance.

[Iceberg Tuning](#) - Optimizing table layout and compaction.

**Last Updated:** December 2025

**Project Status:** Alpha

Source: README.md

## API Documentation

REST API documentation for Pangolin catalog.

### Overview

Pangolin implements the [Apache Iceberg REST Catalog specification](#) with additional extensions for Git-like branching and multi-tenancy.

### Core API

#### API Overview

Complete REST API reference covering:

Namespace operations

Table operations

Catalog configuration

Maintenance endpoints

#### Authentication

Authentication methods:

-

Bearer token authentication

NO\_AUTH mode (development)

Custom headers (X-Pangolin-Tenant)

## API Categories

### Iceberg REST Catalog (Standard)

#### **Namespace Operations:**

`GET /v1/{prefix}/namespaces` - List namespaces

`POST /v1/{prefix}/namespaces` - Create namespace

`DELETE /v1/{prefix}/namespaces/{namespace}` - Delete namespace

#### **Table Operations:**

`GET /v1/{prefix}/namespaces/{namespace}/tables` - List tables

`POST /v1/{prefix}/namespaces/{namespace}/tables` - Create table

`GET /v1/{prefix}/namespaces/{namespace}/tables/{table}` - Load table

`POST /v1/{prefix}/namespaces/{namespace}/tables/{table}` - Update table

`DELETE /v1/{prefix}/namespaces/{namespace}/tables/{table}` - Delete table

#### **Credential Vending:**

`GET /v1/{prefix}/namespaces/{namespace}/tables/{table}/credentials` - Get credentials

## Pangolin Extensions

#### **Branch Management:**

`GET /api/v1/branches` - List branches

`POST /api/v1/branches` - Create branch

`POST /api/v1/branches/merge` - Merge branches

#### **Tag Management:**

`GET /api/v1/tags` - List tags

`POST /api/v1/tags` - Create tag

`DELETE /api/v1/tags/{name}` - Delete tag

#### **Merge Operations:**

`GET /api/v1/catalogs/{catalog}/merge-operations` - List merge operations

GET /api/v1/merge-operations/{id}` - Get merge operation  
GET /api/v1/merge-operations/{id}/conflicts` - List conflicts  
POST /api/v1/conflicts/{id}/resolve` - Resolve conflict  
POST /api/v1/merge-operations/{id}/complete` - Complete merge  
POST /api/v1/merge-operations/{id}/abort` - Abort merge

### **Business Metadata & Access Requests:**

GET/POST/DELETE /api/v1/assets/{id}/metadata` - Manage business metadata  
GET /api/v1/search` - Unified search (Catalogs, Namespaces, Assets)  
GET /api/v1/search/assets` - Search assets by name  
GET /api/v1/assets/{id}` - Get asset details  
POST /api/v1/assets/{id}/request-access` - Request access  
GET /api/v1/access-requests` - List access requests  
GET/PUT /api/v1/access-requests/{id}` - Get/update access request

### **Audit Logs:**

GET /api/v1/audit` - List audit events (with filtering)  
GET /api/v1/audit/count` - Get audit event counts  
GET /api/v1/audit/{id}` - Get specific audit event

### **Warehouse Management:**

GET /api/v1/warehouses` - List warehouses  
POST /api/v1/warehouses` - Create warehouse  
GET /api/v1/warehouses/{name}` - Get warehouse

### **Catalog Management:**

GET /api/v1/catalogs` - List catalogs  
POST /api/v1/catalogs` - Create catalog  
GET /api/v1/catalogs/{name}` - Get catalog  
GET /api/v1/catalogs/{prefix}/namespaces/tree` - Get tree structure

### **Service Users & API Keys:**

POST /api/v1/service-users` - Create service user  
GET /api/v1/service-users` - List service users  
POST /api/v1/service-users/{id}/rotate` - Rotate API key

# Interactive Documentation

Pangolin provides an interactive Swagger UI for live API exploration:

**Swagger UI:** ``http://localhost:8080/swagger-ui``

**OpenAPI JSON:** ``http://localhost:8080/api-docs/openapi.json``

## Contents

Document	Description
<a href="#">api_overview.md</a>	Complete REST API reference
<a href="#">authentication.md</a>	Authentication methods and setup

## Quick Examples

### Create Namespace

```
curl -X POST http://localhost:8080/v1/analytics/namespaces \
  -H "Content-Type: application/json" \
  -d '{"namespace": ["my_namespace"]}'
```

### Create Table

```
curl -X POST http://localhost:8080/v1/analytics/namespaces/my_ns/tables \
  -H "Content-Type: application/json" \
  -d '{
    "name": "my_table",
    "schema": {...}
  }'
```

### Create Warehouse

```
curl -X POST http://localhost:8080/api/v1/warehouses \
  -H "Content-Type: application/json" \
  -d '{
    "name": "my_warehouse",
    "storage_config": {
      "type": "s3",
      "bucket": "my-bucket",
      "access_key_id": "...",
      "secret_access_key": "..."
    }
  }'
```

# See Also

[Getting Started](#) - Setup and configuration

[Features](#) - Advanced features

[Storage](#) - Storage configuration

Source: `api_overview.md`

## API Reference Overview

Pangolin exposes a multi-tenant REST API split into three core functional areas. All list endpoints support standard pagination via `?limit=N&pageToken=...` (or `offset` for management APIs).

### Authentication & Identity

\*Base Path: `/api/v1/``\*

Endpoint	Method	Description
<code>users/login`</code>	POST	Authenticate and receive a JWT session.
<code>tokens`</code>	POST	Generate a long-lived JWT token for a specific user.
<code>auth/revoke`</code>	POST	Invalidate current user token.
<code>service-users`</code>	GET/POST	Manage programmatic API identities.
<code>service-users/{id}`</code>	GET/PUT/DELETE	Manage specific service user.
<code>service-users/{id}/rotate`</code>	POST	Rotate API key for a service user.

### Standard Iceberg API (REST Catalog)

\*Base Path: `/v1/``\*

Pangolin is 100% compliant with the [Apache Iceberg REST Specification](#).

Endpoint	Method	Description
<code>config`</code>	GET	Get client configuration.
<code>{prefix}/namespaces`</code>	GET/POST	Manage catalog namespaces.
<code>{prefix}/namespaces/{ns}/tables`</code>	GET/POST	Manage tables (Standard Iceberg).
<code>{prefix}/namespaces/{ns}/tables/{table}`</code>	GET/POST	Table metadata and schema evolution.
<code>{prefix}/namespaces/{ns}/tables/{table}/credentials`</code>	GET	<b>Credential Vending:</b> Get temporary S3/Azure/GCS tokens.
<code>{prefix}/namespaces/{ns}/tables/{table}/presign`</code>	GET	<b>Presigning:</b> Get pre-signed URL for table data/metadata.

[!TIP]

**Branching:** Use the `@`` suffix in the table name (e.g., `my_table@dev``) to redirect Iceberg operations to a specific Pangolin branch.

# Management & Data Governance

\*Base Path: `/api/v1/\*`

## 1. Multi-Tenancy (Root Only)

Endpoint	Method	Use Case
---   ---   ---		
`tenants`	GET/POST/PUT	Onboard new organizations to the platform.
`config/settings`	GET/PUT	Manage global platform defaults.

## 2. Infrastructure & Data (Tenant Admin)

Endpoint	Method	Use Case
---   ---   ---		
`warehouses`	GET/POST/PUT	Configure cloud storage containers (S3/GCS/Azure).
`catalogs`	GET/POST/PUT	Create local or federated Iceberg catalogs.
`audit`	GET/POST	Query standard or count audit logs.
`maintenance`	POST	Trigger snapshot expiration or orphan file removal.

## 3. Branching & Merging

Endpoint	Method	Use Case
---   ---   ---		
`branches`	GET/POST	List or create feature/ingest branches.
`branches/merge`	POST	Initiate a Git-like merge between two branches.
`merge-operations`	GET	Track progress of active or past merges.
`conflicts/{id}/resolve`	POST	Apply conflict resolution strategies (Source Wins/Target Wins).

## 4. Federated Catalogs

Endpoint	Method	Use Case
---   ---   ---		
`federated-catalogs`	GET/POST	Register external Iceberg REST catalogs.
`federated-catalogs/{name}/test`	POST	Test connection connectivity.
`federated-catalogs/{name}/sync`	POST	Manually sync external catalog metadata.
---		
`federated-catalogs/{name}/stats`	GET	View cache and connection statistics.

## 5. Search & Optimization

Endpoint	Method	Use Case
---   ---   ---		
`search`	GET	Unified search across Catalogs, Namespaces, Tables, and Branches.
`search/assets`	GET	Optimized lookup for specific tables/views by name with permission filtering.
`validate/names`	POST	Check if a catalog or warehouse name is available.
`bulk/assets/delete`	POST	Bulk deletion of assets (up to 100).
`dashboard/stats`	GET	Global platform statistics.

| `catalogs/{name}/summary` | GET | Catalog-specific statistics. |

## OAuth 2.0 Flow

Pangolin supports Google, GitHub, and Microsoft OAuth.

**Initiate:** `GET /oauth/authorize/{provider}`

**Callback:** `GET /oauth/callback/{provider}`

## Quick Integration

To use the API, ensure you provide the `Authorization` header and the `X-Pangolin-Tenant` header (unless in `NO\_AUTH` mode).

```
curl -X GET http://localhost:8080/api/v1/catalogs \
-H "Authorization: Bearer <JWT_TOKEN>" \
-H "X-Pangolin-Tenant: <TENANT_UUID>"
```

Source: authentication.md

## Authentication & Security

Pangolin implements a secure authentication system based on JSON Web Tokens (JWT) and Role-Based Access Control (RBAC).

## Authentication Flow

**Login:** Users authenticate against the `/api/v1/users/login` endpoint using their credentials.

**Root Login:** Omit `tenant-id` or set to `null`

**Tenant-Scoped Login:** Include `tenant-id` with tenant UUID

**Token Issuance:** Upon successful authentication, the server returns a signed JWT.

**API Key Authentication (Service Users):** Machine accounts use a static API key passed in the `X-API-Key` header.

```
X-API-Key: <your-api-key>
```

## Login Examples

**Root Login:**

```
curl -X POST http://localhost:8080/api/v1/users/login \
-H "Content-Type: application/json" \
-d '{"username":"admin","password":"password","tenant-id":null}'
```

**Tenant-Scoped Login** (for users with duplicate usernames across tenants):

```
curl -X POST http://localhost:8080/api/v1/users/login \
-H "Content-Type: application/json" \
-d '{"username":"user","password":"pass123","tenant-id":"<tenant-uuid>"}'
```

[!IMPORTANT]

Use ``tenant-id`` (kebab-case), not ``tenant_id`` (underscore).

## Root User

A "Root User" is configured via environment variables to bootstrap the system. This user has full administrative privileges.

``PANGOLIN_ROOT_USER``: Username for the root user.

``PANGOLIN_ROOT_PASSWORD``: Password for the root user.

## Roles & Permissions

Pangolin supports the following roles:

**Root:** Full system access. Can manage tenants, users, and system configuration.

**TenantAdmin:** Tenant-level administration. Can manage warehouses, catalogs, and users within a tenant.

**TenantUser:** Standard access. Can read/write data based on catalog permissions.

[!NOTE]

When using the API directly (e.g., creating users), these roles must be specified in kebab-case: ``root``, ``tenant-admin``, ``tenant-user``.

## API Key Authentication (Service Users)

Service users are intended for machine-to-machine communication (CI/CD, automated scripts). They do not use JWT tokens; instead, they use a persistent API key.

```
curl http://localhost:8080/api/v1/catalogs \
-H "X-API-Key: pgl_key_abc123..." \
-H "X-Pangolin-Tenant: <tenant-uuid>"
```

[!TIP]

API keys are only displayed once upon creation. If lost, the key must be rotated via the ``/api/v1/service-users/{id}/rotate`` endpoint.

## Token Generation (Users)

For temporary programmatic access by human users, you can generate long-lived JWT tokens:

# Token Revocation

Tokens can be revoked for security:

**Self-revoke:** ``POST /api/v1/auth/revoke`` - Revoke your current token

**Admin-revoke:** ``POST /api/v1/auth/revoke/{token_id}`` - Revoke any token (admin only)

## OAuth Authentication

Pangolin supports OAuth providers (Google, GitHub, etc.):

Navigate to ``/oauth/authorize/{provider}`` (e.g., ``/oauth/authorize/google``)

Complete OAuth flow with provider

Receive JWT token upon successful authentication

## Middleware

The API is protected by authentication middleware that:

Validates the JWT signature.

Extracts user claims (ID, Role, Tenant).

Enforces role-based access policies for specific endpoints.

## S3 Credential Vending

For direct S3 access (e.g., from engines like Spark or Trino), Pangolin provides a credential vending mechanism. See [Security & Vending](#) for details.

Source: [curl\\_examples.md](#)

## Pangolin API - Curl Examples

This document provides practical curl examples for all Pangolin API endpoints.

**Note:** For Iceberg REST API endpoints (``/v1/*``), refer to the [Apache Iceberg REST Catalog specification](#).

## Table of Contents

Authentication

Tenant Management

Warehouse Management

Catalog Management

User Management

Credential Vending

Federated Catalogs

Service Users

Roles & Permissions

OAuth

Branch Operations

Tag Operations

Merge Operations & Conflict Resolution

Business Metadata & Access Requests

Audit Logs

Token Management (Admin)

System Configuration (Admin)

Federated Catalog Operations

Data Explorer

Search & Optimization

## Authentication

### No Auth Mode (Development)

```
# Set environment variable
export PANGOLIN_NO_AUTH=true

# No authentication headers needed
curl http://localhost:8080/api/v1/tenants
```

### JWT Authentication

```
# Login to get JWT token
curl -X POST http://localhost:8080/api/v1/users/login \
  -H "Content-Type: application/json" \
  -d '{
    "username": "admin",
    "password": "password"
  }'
```

```
# Use token in subsequent requests
export TOKEN="eyJhbGciOiJIUzI1NiIs..."
curl http://localhost:8080/api/v1/tenants \
  -H "Authorization: Bearer $TOKEN"
```

## Generate Long-Lived Token

```
# Generate token for automation/scripts
curl -X POST http://localhost:8080/api/v1/tokens \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "tenant_id": "550e8400-e29b-41d4-a716-446655440000",
    "username": "automation-user",
    "expires_in_hours": 720
  }'
```

### Response:

```
{
  "token": "eyJhbGciOiJIUzI1NiIs...",
  "expires_at": "2026-01-17T10:00:00Z"
}
```

## Revoke Token

```
# Revoke your current token
curl -X POST http://localhost:8080/api/v1/auth/revoke \
  -H "Authorization: Bearer $TOKEN"
```

## API Key Authentication (Service Users)

```
# Use API key header
export API_KEY="pgl_key_abc123..."
curl http://localhost:8080/api/v1/catalogs \
  -H "X-API-Key: $API_KEY"
```

## Tenant Management

### List All Tenants

```
curl http://localhost:8080/api/v1/tenants \
  -H "Authorization: Bearer $TOKEN"
```

## Response:

```
[
  {
    "id": "550e8400-e29b-41d4-a716-446655440000",
    "name": "acme-corp",
    "properties": {},
    "created_at": "2025-12-14T10:00:00Z"
  }
]
```

## Create Tenant

```
curl -X POST http://localhost:8080/api/v1/tenants \
-H "Authorization: Bearer $TOKEN" \
-H "Content-Type: application/json" \
-d '{
  "name": "new-tenant",
  "properties": {
    "department": "analytics",
    "owner": "data-team"
  }
}'
```

## Get Tenant

```
curl http://localhost:8080/api/v1/tenants/550e8400-e29b-41d4-a716-446655440000 \
-H "Authorization: Bearer $TOKEN"
```

## Update Tenant

```
curl -X PUT http://localhost:8080/api/v1/tenants/550e8400-e29b-41d4-a716-446655440000 \
-H "Authorization: Bearer $TOKEN" \
-H "Content-Type: application/json" \
-d '{
  "name": "updated-tenant-name",
  "properties": {
    "department": "data-engineering"
  }
}'
```

## Delete Tenant

```
curl -X DELETE http://localhost:8080/api/v1/tenants/550e8400-e29b-41d4-a716-446655440000 \
-H "Authorization: Bearer $TOKEN"
```

# Warehouse Management

## List Warehouses

```
curl http://localhost:8080/api/v1/warehouses \  
-H "Authorization: Bearer $TOKEN"
```

## Create S3 Warehouse (Static Credentials)

```
curl -X POST http://localhost:8080/api/v1/warehouses \  
-H "Authorization: Bearer $TOKEN" \  
-H "Content-Type: application/json" \  
-d '{  
  "name": "s3-warehouse",  
  "use_sts": false,  
  "storage_config": {  
    "type": "s3",  
    "bucket": "my-data-bucket",  
    "region": "us-east-1",  
    "access_key_id": "AKIAIOSFODNN7EXAMPLE",  
    "secret_access_key": "wJalrXUtnFEMI/K7MDENG/bPxrFiCYEXAMPLEKEY",  
    "endpoint": "http://localhost:9000"  
  }  
'
```

## Create S3 Warehouse (AWS STS)

```
curl -X POST http://localhost:8080/api/v1/warehouses \  
-H "Authorization: Bearer $TOKEN" \  
-H "Content-Type: application/json" \  
-d '{  
  "name": "s3-sts-warehouse",  
  "use_sts": true,  
  "storage_config": {  
    "type": "s3",  
    "bucket": "my-data-bucket",  
    "region": "us-east-1",  
    "role_arn": "arn:aws:iam::123456789:role/DataAccess",  
    "external_id": "optional-external-id"  
  }  
'
```

## Create Azure Warehouse (OAuth2)

```
curl -X POST http://localhost:8080/api/v1/warehouses \  
-H "Authorization: Bearer $TOKEN" \  
-H "Content-Type: application/json" \  

```

```
-d '{
  "name": "azure-warehouse",
  "use_sts": true,
  "storage_config": {
    "type": "azure",
    "tenant_id": "your-azure-tenant-id",
    "client_id": "your-azure-client-id",
    "client_secret": "your-azure-client-secret",
    "account_name": "mystorageaccount",
    "container": "data"
  }
}'
```

## Create GCP Warehouse (Service Account)

```
curl -X POST http://localhost:8080/api/v1/warehouses \
-H "Authorization: Bearer $TOKEN" \
-H "Content-Type: application/json" \
-d '{
  "name": "gcp-warehouse",
  "use_sts": true,
  "storage_config": {
    "type": "gcs",
    "service_account_key":
"{\"type\": \"service_account\", \"project_id\": \"my-project\", ...}\",
    "project_id": "my-project",
    "bucket": "my-bucket"
  }
}'
```

## Get Warehouse

```
curl http://localhost:8080/api/v1/warehouses/s3-warehouse \
-H "Authorization: Bearer $TOKEN"
```

## Update Warehouse

```
curl -X PUT http://localhost:8080/api/v1/warehouses/s3-warehouse \
-H "Authorization: Bearer $TOKEN" \
-H "Content-Type: application/json" \
-d '{
  "use_sts": true,
  "storage_config": {
    "role_arn": "arn:aws:iam::123456789:role/NewRole"
  }
}'
```

# Delete Warehouse

```
curl -X DELETE http://localhost:8080/api/v1/warehouses/s3-warehouse \  
-H "Authorization: Bearer $TOKEN"
```

## Catalog Management

### List Catalogs

```
curl http://localhost:8080/api/v1/catalogs \  
-H "Authorization: Bearer $TOKEN"
```

### Create Catalog

```
curl -X POST http://localhost:8080/api/v1/catalogs \  
-H "Authorization: Bearer $TOKEN" \  
-H "Content-Type: application/json" \  
-d '{  
  "name": "production",  
  "warehouse_name": "s3-warehouse",  
  "storage_location": "s3://my-bucket/warehouse",  
  "properties": {  
    "owner": "data-team"  
  }  
'
```

### Get Catalog

```
curl http://localhost:8080/api/v1/catalogs/production \  
-H "Authorization: Bearer $TOKEN"
```

### Update Catalog

```
curl -X PUT http://localhost:8080/api/v1/catalogs/production \  
-H "Authorization: Bearer $TOKEN" \  
-H "Content-Type: application/json" \  
-d '{  
  "warehouse_name": "new-warehouse",  
  "properties": {  
    "environment": "production"  
  }  
'
```

# Delete Catalog

```
curl -X DELETE http://localhost:8080/api/v1/catalogs/production \  
-H "Authorization: Bearer $TOKEN"
```

## User Management

### List Users

```
curl http://localhost:8080/api/v1/users \  
-H "Authorization: Bearer $TOKEN"
```

### Create User

```
curl -X POST http://localhost:8080/api/v1/users \  
-H "Authorization: Bearer $TOKEN" \  
-H "Content-Type: application/json" \  
-d '{  
  "username": "john.doe",  
  "email": "john@example.com",  
  "password": "secure-password",  
  "tenant_id": "550e8400-e29b-41d4-a716-446655440000"  
'
```

### Get User

```
curl http://localhost:8080/api/v1/users/USER_ID \  
-H "Authorization: Bearer $TOKEN"
```

### Update User

```
curl -X PUT http://localhost:8080/api/v1/users/USER_ID \  
-H "Authorization: Bearer $TOKEN" \  
-H "Content-Type: application/json" \  
-d '{  
  "email": "newemail@example.com"  
'
```

### Delete User

```
curl -X DELETE http://localhost:8080/api/v1/users/USER_ID \  
-H "Authorization: Bearer $TOKEN"
```

# Credential Vending

## Get Table Credentials (AWS STS)

```
curl http://localhost:8080/v1/production/namespaces/db/tables/users/credentials \
-H "Authorization: Bearer $TOKEN"
```

### Response:

```
{
  "storage-credentials": [
    {
      "prefix": "s3://my-bucket/warehouse/",
      "config": {
        "access-key": "ASIA...",
        "secret-key": "...",
        "session-token": "...",
        "expiration": "2025-12-14T17:00:00Z",
        "s3.region": "us-east-1"
      }
    }
  ]
}
```

## Get Table Credentials (Azure OAuth2)

```
curl http://localhost:8080/v1/production/namespaces/db/tables/users/credentials \
-H "Authorization: Bearer $TOKEN"
```

### Response:

```
{
  "storage-credentials": [
    {
      "prefix": "abfss://data@mystorageaccount.dfs.core.windows.net/",
      "config": {
        "adls.auth.type": "OAuth2",
        "adls.oauth2.token": "eyJ0eXAiOiJKV1QiLCJhbGciOi..."
      }
    }
  ]
}
```

# Federated Catalogs

## List Federated Catalogs

```
curl http://localhost:8080/api/v1/federated-catalogs \
-H "Authorization: Bearer $TOKEN"
```

## Create Federated Catalog

```
curl -X POST http://localhost:8080/api/v1/federated-catalogs \
-H "Authorization: Bearer $TOKEN" \
-H "Content-Type: application/json" \
-d '{
  "name": "partner-catalog",
  "config": {
    "properties": {
      "uri": "https://partner.example.com",
      "token": "partner-api-token"
    }
  }
}'
```

## Get Federated Catalog

```
curl http://localhost:8080/api/v1/federated-catalogs/partner-catalog \
-H "Authorization: Bearer $TOKEN"
```

## Test Federated Catalog Connection

```
curl -X POST http://localhost:8080/api/v1/federated-catalogs/partner-catalog/test \
-H "Authorization: Bearer $TOKEN"
```

## Delete Federated Catalog

```
curl -X DELETE http://localhost:8080/api/v1/federated-catalogs/partner-catalog \
-H "Authorization: Bearer $TOKEN"
```

## Service Users (API Keys)

### Create Service User

```
curl -X POST http://localhost:8080/api/v1/service-users \
-H "Authorization: Bearer $TOKEN" \
-H "Content-Type: application/json" \
-d '{
  "name": "ci-cd-pipeline",
  "description": "Service user for CI/CD automation",
  "token": "ci-cd-pipeline-token"
}'
```

```
"role": "TenantUser",  
"expires_in_days": 30  
'}
```

## Response:

```
{  
  "id": "...",  
  "name": "ci-cd-pipeline",  
  "api_key": "pgl_key_abc123...",  
  "created_at": "2025-12-14T10:00:00Z",  
  "expires_at": "2026-12-31T23:59:59Z"  
}
```

**Important:** Save the `api\_key` - it's only shown once!

## List Service Users

```
curl http://localhost:8080/api/v1/service-users \  
-H "Authorization: Bearer $TOKEN"
```

## Delete Service User

```
curl -X DELETE http://localhost:8080/api/v1/service-users/SERVICE_USER_ID \  
-H "Authorization: Bearer $TOKEN"
```

## Roles & Permissions

### List Roles

```
curl http://localhost:8080/api/v1/roles \  
-H "Authorization: Bearer $TOKEN"
```

### Create Role

```
curl -X POST http://localhost:8080/api/v1/roles \  
-H "Authorization: Bearer $TOKEN" \  
-H "Content-Type: application/json" \  
-d '{  
  "name": "DataEngineer",  
  "description": "Can manage tables and schemas",  
  "permissions": ["read", "write", "create"]  
}'
```

## List Permissions

```
curl http://localhost:8080/api/v1/permissions \
-H "Authorization: Bearer $TOKEN"
```

## Grant Permission

```
curl -X POST http://localhost:8080/api/v1/permissions \
-H "Authorization: Bearer $TOKEN" \
-H "Content-Type: application/json" \
-d '{
  "user-id": "USER_ID",
  "scope": {
    "type": "catalog",
    "catalog-id": "550e8400-e29b-41d4-a716-446655440000"
  },
  "actions": ["Read", "Write"]
}'
```

## Revoke Permission

```
curl -X DELETE http://localhost:8080/api/v1/permissions/PERMISSION_ID \
-H "Authorization: Bearer $TOKEN"
```

## OAuth

### Initiate OAuth Flow (Google)

```
# Open in browser or redirect
curl http://localhost:8080/oauth/authorize/google
```

### Initiate OAuth Flow (GitHub)

```
# Open in browser or redirect
curl http://localhost:8080/oauth/authorize/github
```

**Note:** OAuth flows require browser interaction. The callback endpoint `/oauth/callback/{provider}` is handled automatically.

## Complete Workflow Example

### 1. Create Complete Environment

```

# Set base URL and token
export BASE_URL="http://localhost:8080"
export TOKEN="your-jwt-token"

# Create tenant
TENANT_RESPONSE=$(curl -s -X POST $BASE_URL/api/v1/tenants \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"name":"acme-corp"}')

TENANT_ID=$(echo $TENANT_RESPONSE | jq -r '.id')

# Create warehouse
curl -X POST $BASE_URL/api/v1/warehouses \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "name":"s3-warehouse",
    "use_sts":true,
    "storage_config":{
      "type":"s3",
      "bucket":"my-bucket",
      "region":"us-east-1",
      "role_arn":"arn:aws:iam::123:role/DataAccess"
    }
  }'

# Create catalog
curl -X POST $BASE_URL/api/v1/catalogs \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "name":"production",
    "warehouse_name":"s3-warehouse"
  }'

# Create namespace (Iceberg REST API)
curl -X POST $BASE_URL/v1/production/namespaces \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "namespace":["analytics"]
  }'

echo " Environment created successfully!"

```

## Error Handling

### Common Error Responses

#### 401 Unauthorized:

```
{
  "error": "Unauthorized",
  "message": "Invalid or missing authentication token"
}
```

### 404 Not Found:

```
{
  "error": "Not Found",
  "message": "Warehouse 'nonexistent' not found"
}
```

### 500 Internal Server Error:

```
{
  "error": "Internal Server Error",
  "message": "Failed to connect to database"
}
```

## Tips & Best Practices

### Use jq for JSON parsing:

```
curl http://localhost:8080/api/v1/catalogs | jq '.[] | .name'
```

### Save tokens in environment variables:

```
export TOKEN=$(curl -s -X POST http://localhost:8080/api/v1/auth/login \
  -H "Content-Type: application/json" \
  -d '{"username":"admin","password":"password"}' | jq -r '.token')
```

### Use `-v` for debugging:

```
curl -v http://localhost:8080/api/v1/tenants
```

### Pretty print JSON responses:

```
curl http://localhost:8080/api/v1/catalogs | jq '.'
```

## Related Documentation

[OpenAPI Specification](#) - Complete API schema

[Apache Iceberg REST Spec](#) - Iceberg endpoints

[Authentication Guide](#) - Authentication setup

## Branch Operations

### List Branches

```
curl http://localhost:8080/api/v1/branches \
-H "Authorization: Bearer $TOKEN"
```

### Create Branch

```
curl -X POST http://localhost:8080/api/v1/branches \
-H "Authorization: Bearer $TOKEN" \
-H "Content-Type: application/json" \
-d '{
  "name": "dev",
  "catalog_name": "production",
  "source_branch": "main"
}'
```

### Get Branch

```
curl http://localhost:8080/api/v1/branches/dev \
-H "Authorization: Bearer $TOKEN"
```

### List Branch Commits

```
curl http://localhost:8080/api/v1/branches/dev/commits \
-H "Authorization: Bearer $TOKEN"
```

### Merge Branch

```
curl -X POST http://localhost:8080/api/v1/branches/merge \
-H "Authorization: Bearer $TOKEN" \
-H "Content-Type: application/json" \
-d '{
  "source_branch": "dev",
  "target_branch": "main",
  "catalog_name": "production"
}'
```

## Tag Operations

## List Tags

```
curl http://localhost:8080/api/v1/tags \  
-H "Authorization: Bearer $TOKEN"
```

## Create Tag

```
curl -X POST http://localhost:8080/api/v1/tags \  
-H "Authorization: Bearer $TOKEN" \  
-H "Content-Type: application/json" \  
-d '{  
  "name": "v1.0.0",  
  "catalog_name": "production",  
  "commit_id": "abc123..."  
}
```

## Delete Tag

```
curl -X DELETE http://localhost:8080/api/v1/tags/v1.0.0 \  
-H "Authorization: Bearer $TOKEN"
```

# Merge Operations & Conflict Resolution

## List Merge Operations

```
curl http://localhost:8080/api/v1/catalogs/production/merge-operations \  
-H "Authorization: Bearer $TOKEN"
```

## Get Merge Operation

```
curl http://localhost:8080/api/v1/merge-operations/OPERATION_ID \  
-H "Authorization: Bearer $TOKEN"
```

## List Merge Conflicts

```
curl http://localhost:8080/api/v1/merge-operations/OPERATION_ID/conflicts \  
-H "Authorization: Bearer $TOKEN"
```

## Resolve Conflict

```
curl -X POST http://localhost:8080/api/v1/conflicts/CONFLICT_ID/resolve \  
-H "Authorization: Bearer $TOKEN"
```

```
-H "Authorization: Bearer $TOKEN" \  
-H "Content-Type: application/json" \  
-d '{  
  "strategy": "TakeSource"  
'
```

## Complete Merge

```
curl -X POST http://localhost:8080/api/v1/merge-operations/OPERATION_ID/complete \  
-H "Authorization: Bearer $TOKEN"
```

## Abort Merge

```
curl -X POST http://localhost:8080/api/v1/merge-operations/OPERATION_ID/abort \  
-H "Authorization: Bearer $TOKEN"
```

## Business Metadata & Access Requests

### Add Business Metadata

```
curl -X POST http://localhost:8080/api/v1/assets/ASSET_ID/metadata \  
-H "Authorization: Bearer $TOKEN" \  
-H "Content-Type: application/json" \  
-d '{  
  "owner": "data-team",  
  "classification": "PII",  
  "tags": ["customer", "sensitive"]  
'
```

### Get Business Metadata

```
curl http://localhost:8080/api/v1/assets/ASSET_ID/metadata \  
-H "Authorization: Bearer $TOKEN"
```

### Search Assets

```
curl http://localhost:8080/api/v1/assets/search?q=customer \  
-H "Authorization: Bearer $TOKEN"
```

### Request Access to Asset

```
curl -X POST http://localhost:8080/api/v1/assets/ASSET_ID/request-access \
-H "Authorization: Bearer $TOKEN" \
-H "Content-Type: application/json" \
-d '{
  "reason": "Need access for analytics project"
}'
```

## List Access Requests

```
curl http://localhost:8080/api/v1/access-requests \
-H "Authorization: Bearer $TOKEN"
```

## Update Access Request

```
curl -X PUT http://localhost:8080/api/v1/access-requests/REQUEST_ID \
-H "Authorization: Bearer $TOKEN" \
-H "Content-Type: application/json" \
-d '{
  "status": "approved"
}'
```

## Audit Logs

### List Audit Events

```
curl http://localhost:8080/api/v1/audit \
-H "Authorization: Bearer $TOKEN"
```

#### Response:

```
[
  {
    "id": "...",
    "timestamp": "2025-12-18T10:00:00Z",
    "user_id": "...",
    "action": "table.create",
    "resource": "production.analytics.users",
    "details": {}
  }
]
```

## Token Management (Admin)

## List User Tokens

```
# List all tokens for a specific user (admin only)
curl http://localhost:8080/api/v1/users/USER_ID/tokens \
-H "Authorization: Bearer $TOKEN"
```

### Response:

```
[
  {
    "id": "token-uuid-123",
    "user_id": "user-uuid-456",
    "created_at": "2025-12-19T10:00:00Z",
    "expires_at": "2026-01-19T10:00:00Z",
    "is_valid": true
  }
]
```

## Delete Token

```
# Delete a specific token by ID (admin only)
curl -X DELETE http://localhost:8080/api/v1/tokens/TOKEN_ID \
-H "Authorization: Bearer $TOKEN"
```

## System Configuration (Admin)

### Get System Settings

```
# Get all system configuration settings (admin only)
curl http://localhost:8080/api/v1/config/settings \
-H "Authorization: Bearer $TOKEN"
```

### Response:

```
{
  "allow_public_signup": false,
  "default_retention_days": 30,
  "default_warehouse_bucket": "my-default-bucket",
  "smtp_host": "smtp.example.com",
  "smtp_port": 587,
  "smtp_user": "noreply@example.com",
  "smtp_password": "****"
}
```

## Update System Settings

```
# Update system configuration settings (admin only)
curl -X PUT http://localhost:8080/api/v1/config/settings \
-H "Authorization: Bearer $TOKEN" \
-H "Content-Type: application/json" \
-d '{
  "allow_public_signup": true,
  "default_retention_days": 90,
  "smtp_host": "smtp.newprovider.com"
}'
```

## Federated Catalog Operations

### Sync Federated Catalog

```
# Trigger immediate metadata sync for federated catalog
curl -X POST http://localhost:8080/api/v1/federated-catalogs/partner-catalog/sync \
-H "Authorization: Bearer $TOKEN"
```

#### Response:

```
{
  "status": "Sync triggered",
  "catalog_name": "partner-catalog",
  "triggered_at": "2025-12-19T10:00:00Z"
}
```

### Get Federated Catalog Stats

```
# Get sync statistics and status for federated catalog
curl http://localhost:8080/api/v1/federated-catalogs/partner-catalog/stats \
-H "Authorization: Bearer $TOKEN"
```

#### Response:

```
{
  "catalog_name": "partner-catalog",
  "last_synced_at": "2025-12-19T09:00:00Z",
  "sync_status": "success",
  "tables_synced": 42,
  "namespaces_synced": 5,
  "last_error": null,
  "next_scheduled_sync": "2025-12-19T12:00:00Z"
}
```

## Data Explorer

# Get Namespace Tree

```
# Get hierarchical namespace tree structure for a catalog
curl http://localhost:8080/api/v1/catalogs/production/namespaces/tree \
-H "Authorization: Bearer $TOKEN"
```

## Response:

```
{
  "root": [
    {
      "name": ["analytics"],
      "children": [
        {
          "name": ["analytics", "sales"],
          "children": []
        },
        {
          "name": ["analytics", "marketing"],
          "children": []
        }
      ]
    },
    {
      "name": ["staging"],
      "children": []
    }
  ]
}
```

# Complete Examples

## Token Management Workflow

```
# 1. List all tokens for a user (admin)
curl http://localhost:8080/api/v1/users/$USER_ID/tokens \
-H "Authorization: Bearer $TOKEN"

# 2. Delete a specific token
curl -X DELETE http://localhost:8080/api/v1/tokens/$TOKEN_ID \
-H "Authorization: Bearer $TOKEN"
```

# System Configuration Workflow

```
# 1. Get current settings
SETTINGS=$(curl -s http://localhost:8080/api/v1/config/settings \
-H "Authorization: Bearer $TOKEN")
```

```
echo $SETTINGS | jq '.'

# 2. Update specific settings
curl -X PUT http://localhost:8080/api/v1/config/settings \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "allow_public_signup": true,
    "default_retention_days": 60
  }'
```

## Federated Catalog Monitoring

```
# 1. Check sync stats
curl http://localhost:8080/api/v1/federated-catalogs/partner-catalog/stats \
  -H "Authorization: Bearer $TOKEN" | jq '.'

# 2. Trigger manual sync if needed
curl -X POST http://localhost:8080/api/v1/federated-catalogs/partner-catalog/sync \
  -H "Authorization: Bearer $TOKEN"

# 3. Wait and check stats again
sleep 30
curl http://localhost:8080/api/v1/federated-catalogs/partner-catalog/stats \
  -H "Authorization: Bearer $TOKEN" | jq '.last_synced_at'
```

## Search & Optimization

### Unified Search

Search across catalogs, namespaces, and tables.

```
curl
"http://localhost:8080/api/v1/search?q=sales&tenant_id=550e8400-e29b-41d4-a716-446655440
000" \
  -H "Authorization: Bearer $TOKEN"
```

### Validate Table Name

Check if a name is valid and available.

```
curl -X POST http://localhost:8080/api/v1/validate/names \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "resource_type": "catalog",
    "names": ["new_catalog_v2"]
  }'
```

```

}'

### Bulk Asset Deletion
```bash
curl -X POST http://localhost:8080/api/v1/bulk/assets/delete \
-H "Authorization: Bearer $TOKEN" \
-H "Content-Type: application/json" \
-d '{
  "asset_ids": ["uuid-1", "uuid-2"]
}'

```

```

---

# Source: README.md

# Architecture Documentation

This directory contains detailed technical documentation for the Pangolin architecture.

## Core Structure
- [High-Level Architecture](./architecture.md): Overall system design, component interaction, and multi-tenant isolation.
- [API Handlers](./handlers.md): Map of API endpoints categorized by functional domain (Iceberg, Versioning, Admin).
- [Models](./models.md): Comprehensive guide to core system structs (Tenant, Asset, Merge, User).
- [Enums](./enums.md): Exhaustive list of system enumerations and their serialized values.

## Interfaces & Logic
- [System Traits](./traits.md): In-depth look at `CatalogStore` and `Signer` interfaces.
- [Branching & Merging](./branching.md): Operational details of the "Git-for-Data" versioning model.
- [Caching Strategy](./caching.md): multi-layered performance optimizations for metadata and cloud backends.

## Security & Operations
- [Authentication](./authentication.md): Deep dive into JWT, Service User API Keys, and RBAC.
- [Storage & Connectivity](./storage_and_connectivity.md): Cloud connectivity, modular store structure, and credential vending.
- [Dependencies](./dependencies.md): Final list of technology stack and library versions.

---

## Status
All documents in this directory were audited and updated in December 2025 to reflect the modularized backend and enhanced security features.

```

```

---

# Source: architecture.md

# Pangolin Architecture

## Overview
Pangolin is a Rust-based, multi-tenant, branch-aware lakehouse catalog. It is fully compatible with the Apache Iceberg REST API while providing enterprise-grade extensions for Git-like branching, unified discovery, and cross-catalog federation.

## Core Components

### 1. API Layer (`pangolin_api`)
- Framework: Axum (Async Rust).
- Core Engine: Handles HTTP routing, JSON (de)serialization, and OpenAPI schema generation via `utoipa`.
- Key Modules:
    - `iceberg_handlers`: Faithful implementation of the Iceberg REST specification (Namespaces, Tables, Scans).
    - `branch_handlers`: Logic for Git-like workflows (Branching, Tagging, Merging).
    - `business_metadata_handlers`: Business catalog features (Search, Tags, Access Requests).
    - `auth_handlers`: Multi-mode authentication logic (JWT, API Keys, OAuth2).
    - `management_handlers`: Administrative CRUD for Tenants, Warehouses, and Users.
    - `federated_handlers`: Proxy logic for external REST catalogs.

### 2. Core Domain (`pangolin_core`)
- Responsibility: Defines the system's "Source of Truth" models and validation logic.
- Key Models:
    - `Tenant`: The root multi-tenancy unit; all data is isolated by Tenant ID.
    - `Asset`: Unified representation of `IcebergTable`, `View`, and other data resources.
    - `Branch`: References to commit chains (`Ingest` vs `Experimental` types).
    - `Commit`: Immutable snapshots of catalog state tracking `Put` and `Delete` operations.
    - `PermissionScope`: Granular target definitions (Tenant, Catalog, Namespace, Asset, Tag).

### 3. Storage Layer (`pangolin_store`)
- Metadata Persistence: Abstracted via the `CatalogStore` trait.
    - Modular Backends: All backends are refactored into focused submodules (e.g., `tenants.rs`, `warehouses.rs`, `assets.rs`) for better maintainability.
    - `MemoryStore`: Concurrent in-memory store for rapid development/testing.
    - `PostgresStore`: SQL backend using `sqlx` for production scale.
    - `MongoStore`: Document backend for high-availability deployments.
    - `SqliteStore`: Embedded backend for local dev and edge use cases.
- Performance: Direct `assets_by_id` lookup for O(1) authorization checks.
- Data Storage: Object storage (S3/GCS/Azure) via the `object_store` crate.
- Credential Vending: Integrated `Signer` trait to vend temporary tokens (AWS STS, Azure SAS, GCP Downscoped).

```

### ### 4. Security & Isolation

- **Authentication**:
  - **JWT**: Standard for UI and corporate identity access.
  - **API Keys**: Managed via **Service Users** for machine-to-machine/CI-CD access. Includes automatic rotation and usage tracking.
  - **OAuth 2.0 / OIDC**: Native integration with Google, Microsoft, GitHub, and custom providers.
- **Authorization**:
  - **RBAC**: Role-based access control with 3 default tiers (Root, TenantAdmin, TenantUser).
  - **TBAC**: Tag-based access control allowing permissions to flow to assets with specific business labels.
  - **Access Requests**: Integrated workflow for users to request access to restricted assets via the UI.
- **Tenant Isolation**: Strictly enforced at the middleware layer; all store queries are scoped by `tenant\_id`.

### ### 5. Git-like Data Lifecycle

- **Branching Engine**: Supports full and partial catalog branching.
- **Fork-on-Write**: Writes to a branch create new commits without affecting the parent branch until merged.
- **3-Way Merging**: Automated conflict detection using common ancestor (base commit) analysis.
- **Conflict Types**: Detects Schema, Data (partition overlap), Metadata, and Deletion conflicts.

### ### 6. Catalog Federation

- **REST Proxy**: Pangolin acts as a unified entry point, proxying requests to external Iceberg REST catalogs.
- **Auth Translation**: Translates Pangolin credentials to the authentication required by remote catalogs (Basic, Bearer, ApiKey).
- **Global Governance**: Apply Pangolin RBAC and Audit policies even to external federated data.

## ## Management Interfaces

### ### 1. Management UI (`pangolin\_ui`)

- **Stack**: SvelteKit + Vanilla CSS (modern, glassmorphic design).
- **Features**: Visual management of Users, Permissions, Audit Logs, and the Data Discovery portal.

### ### 2. CLI Tools

- **pangolin-admin**: High-level system administration (Tenants, Warehouses, Roles).
- **pangolin-user**: Engineering workflow tool (Branching, Code generation, Search).

## ## Request Flow

1. **Auth Middleware**: Resolves user identity from Header. Validates JWT or API Key.
2. **Tenant Middleware**: Resolves `X-Pangolin-Tenant` or extracts it from User session.
3. **Handler**: Executes business logic. Interacts with `CatalogStore` for metadata.
4. **Vending (Optional)**: If requesting a table load, vends temporary S3/Cloud credentials.
5. **Audit Middleware**: Asynchronously logs the operation result (Success/Failure) to the audit store.

```

## Testing Strategy
- **Store Tests**: Generic test suite run against all 4 backends (Memory, PG, Mongo, SQLite).
- **API Tests**: Axum integration tests covering all RBAC permutations.
- **Live Verification**: End-to-end scripts using PyIceberg, Spark, and the Pangolin CLI.

---

# Source: authentication.md

# Authentication & Authorization Architecture

## Overview

Pangolin implements a multi-layered authentication and authorization system supporting multiple authentication methods and role-based access control (RBAC). This document provides a comprehensive guide to how authentication works across all components.

## Authentication Methods

### 1. Basic Authentication (Root Users Only)

**Location**:
[auth_middleware.rs:L174-210](file:///home/alexmerced/development/personal/Personal/2026/pangolin/pangolin/pangolin_api/src/auth_middleware.rs#L174-210)

Basic Auth is **ONLY** supported for Root-level administrative access via environment variables.

#### Configuration
```bash
export PANGOLIN_ROOT_USER=admin
export PANGOLIN_ROOT_PASSWORD=your_secure_password

```

## Usage

```
curl -u admin:your_secure_password http://localhost:8080/api/v1/tenants
```

## How It Works

Client sends `Authorization: Basic <base64(username:password)>` header

Middleware decodes the credentials

Compares against `PANGOLIN\_ROOT\_USER` and `PANGOLIN\_ROOT\_PASSWORD` environment variables

[!IMPORTANT]

### Root vs Tenant Admin:

``PANGOLIN_ROOT_USER``: Required for **Global Operations** (e.g., creating new tenants). Logs in without a ``tenant_id``.

``PANGOLIN_ADMIN_USER``: Seeds a **Tenant Admin** for the default tenant. Can only manage resources *\*within\** that default tenant.

If you need to bootstrap a new deployment (create tenants), you **MUST** set ``PANGOLIN_ROOT_USER``.

If match: Creates a Root session with ``UserRole::Root``

If no match: Falls through to Bearer token authentication

\u003e [!IMPORTANT]

\u003e **Basic Auth is NOT supported for tenant users.** Tenant admins and users MUST use Bearer tokens obtained via the ``/api/v1/users/login`` endpoint.

## 2. Bearer Token Authentication (All Users)

**Location:** [auth\\_middleware.rs:L213-232](#)

Bearer tokens are JWT tokens used for all tenant-level authentication.

### Obtaining a Token

**Login Endpoint:** ``POST /api/v1/users/login``

```
curl -X POST http://localhost:8080/api/v1/users/login \\  
-H "Content-Type: application/json" \\  
-d '{  
  "username": "tenant_admin",  
  "password": "password123"  
}
```

### Response:

```
{  
  "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9...",  
  "user": {  
    "id": "...",  
    "username": "tenant_admin",  
    "role": "TenantAdmin",  
    "tenant_id": "..."  
  }  
}
```

### Tenant-Scoped Login

For multi-tenant deployments, users can have the same username across different tenants. To resolve username collisions, include the `tenant-id` field in login requests:

### Root Login (no tenant context):

```
curl -X POST http://localhost:8080/api/v1/users/login \
-H "Content-Type: application/json" \
-d '{
  "username": "admin",
  "password": "password",
  "tenant-id": null
}'
```

### Tenant-Scoped Login (with tenant context):

```
curl -X POST http://localhost:8080/api/v1/users/login \
-H "Content-Type: application/json" \
-d '{
  "username": "user",
  "password": "pass123",
  "tenant-id": "<tenant-uuid>"
}'
```

### [!IMPORTANT]

The field name is `tenant-id` (kebab-case), not `tenant\_id` (underscore). This is due to the `LoginRequest` struct using `#[serde(rename\_all = "kebab-case")]`.

## Usage

```
curl -H "Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9..." \\
http://localhost:8080/api/v1/catalogs
```

## Token Structure

Tokens are signed JWTs containing:

`sub`: User ID

`jti`: Token ID (for revocation)

`username`: Username

`tenant\_id`: Tenant ID (null for Root users)

`role`: User role (`Root`, `TenantAdmin`, `TenantUser`)

`exp`: Expiration timestamp

`iat`: Issued at timestamp

## Token Verification Flow

Extract token from `Authorization: Bearer <token>` header

Verify JWT signature using `PANGOLIN\_JWT\_SECRET` (default: `default\_secret\_for\_dev`)

Check if token is revoked (via `is\_token\_revoked` store method)

Extract claims and create session

Insert session into request extensions

### 3. API Key Authentication (Service Users Only)

**Location:** [auth\\_middleware.rs:L135-154](#)

API keys provide long-lived authentication specifically for **Service Users** (machine accounts). Regular human users cannot use API keys; they use Bearer tokens.

#### Header Format

```
X-API-Key: <api_key_value>
```

#### How It Works

Extract API key from `X-API-Key` header.

Hash the key and look up the matching record in the store via `get\_service\_user\_by\_api\_key\_hash`.

Verify the service account is active and not expired.

Create a session with the service user's ID, tenant context, and role.

Track usage via `update\_service\_user\_last\_used`.

[!TIP]

Use Service Users for CI/CD pipelines, Spark clusters, or any non-interactive automation.

### Granular Permissions (RBAC)

**Location:** [permission.rs](#)

Beyond roles, Pangolin supports fine-grained permission grants at the Asset, Namespace, or Catalog level.

#### Permission Structure

**Scope:** Where the permission applies (Tenant, Catalog, Namespace, Asset, or Tag).

**Actions:** What is allowed (`Read`, `Write`, `Create`, `Delete`, `ManageDiscovery`, etc.).

# Multi-Tenant Isolation

The auth middleware automatically extracts the tenant context from the user's session or the `X-Pangolin-Tenant` header (for Root users). All subsequent storage operations are strictly scoped to this tenant ID.

## Authentication Flow

```
graph TD
    A[Incoming Request] --> B{Public Endpoint?}
    B -->|Yes| C[Allow Request]
    B -->|No| D{X-API-Key Header?}
    D -->|Yes| E[Verify API Key]
    E -->|Valid| F[Create Session from Key]
    E -->|Invalid| G[401 Unauthorized]
    D -->|No| H{Authorization Header?}
    H -->|No| G
    H -->|Yes| I{Basic or Bearer?}
    I -->|Basic| J{Root Credentials?}
    J -->|Yes| K[Create Root Session]
    J -->|No| L{Bearer Token?}
    L -->|Bearer| M[Verify JWT]
    L -->|No| G
    M -->|Valid| N{Token Revoked?}
    N -->|No| O[Create Session from Claims]
    N -->|Yes| G
    O --> P[Inject Session into Request]
    P --> Q[Continue to Handler]
```

## User Roles & Permissions

### Role Hierarchy

**Root** (`UserRole::Root`)

Full system access

Can manage all tenants

Can override tenant context via `X-Pangolin-Tenant` header

Authenticated via Basic Auth only

**TenantAdmin** (`UserRole::TenantAdmin`)

Full access within their tenant

Can manage users, warehouses, catalogs

Cannot access other tenants

Authenticated via Bearer tokens

**TenantUser** (`UserRole::TenantUser`)

Limited access based on permissions

Can read/write data based on grants

Cannot manage tenant resources

Authenticated via Bearer tokens

## Authorization Checks

**Location:** [authz.rs](#)

Authorization is enforced through:

**Role-based checks:** Verify user role meets minimum requirement

**Tenant isolation:** Ensure users can only access their tenant's resources

**Permission grants:** Fine-grained access control via RBAC

## Session Management

### Session Structure

```
pub struct UserSession {  
    pub user_id: Uuid,  
    pub username: String,  
    pub tenant_id: Option<Uuid>,  
    pub role: UserRole,  
    pub expires_at: i64,  
}
```

### Session Lifecycle

**Creation:** Session created during authentication

**Injection:** Inserted into request extensions via middleware

**Extraction:** Handlers extract session from `Extension<UserSession>`

**Expiration:** Tokens expire based on `exp` claim (default: 1 hour for Root, configurable for others)

## Public Endpoints

The following endpoints bypass authentication:

`/health` - Health check

`/api/v1/users/login` - User login

`/api/v1/app-config` - Application configuration

`/v1/config` - Iceberg REST config

`\*/config` - Any config endpoint

`/oauth/authorize/\*` - OAuth authorization flow

`/oauth/callback/\*` - OAuth callback handling

**Location:** [auth\\_middleware.rs:L158-167](#)

## Admin User Provisioning

### Automatic Seeding

When `PANGOLIN\_SEED\_ADMIN=true` or `PANGOLIN\_NO\_AUTH=true`:

```
export PANGOLIN_SEED_ADMIN=true
export PANGOLIN_ADMIN_USER=admin
export PANGOLIN_ADMIN_PASSWORD=password
```

**Location:** [main.rs:L62-127](#)

### What Happens

Creates default tenant (`00000000-0000-0000-0000-000000000000`)

Hashes the admin password

Creates a `TenantAdmin` user with ID `00000000-0000-0000-0000-000000000000`

Generates a long-lived JWT token (365 days)

Prints configuration banner with credentials and token

\u003e [!WARNING]

\u003e **The seeded admin user is a TENANT ADMIN, not a ROOT user.** They can only authenticate via Bearer tokens, not Basic Auth, unless you also set `PANGOLIN\_ROOT\_USER=admin` and `PANGOLIN\_ROOT\_PASSWORD=password`.

## Common Pitfalls & Troubleshooting

Issue: "Missing or invalid authorization header"

**Symptoms:** All requests return 401 with this message

**Common Causes:**

### Using Basic Auth for tenant users

```
# WRONG - Tenant users cannot use Basic Auth
curl -u tenant_admin:password http://localhost:8080/api/v1/catalogs

# CORRECT - Use Bearer token
TOKEN=$(curl -X POST http://localhost:8080/api/v1/users/login \
  -H "Content-Type: application/json" \
  -d '{"username":"tenant_admin","password":"password"}' \
  | jq -r '.token')
curl -H "Authorization: Bearer $TOKEN" http://localhost:8080/api/v1/catalogs
```

### Root credentials not set

```
# WRONG - No root credentials configured
curl -u admin:password http://localhost:8080/api/v1/tenants

# CORRECT - Set root credentials first
export PANGOLIN_ROOT_USER=admin
export PANGOLIN_ROOT_PASSWORD=password
# Restart API
curl -u admin:password http://localhost:8080/api/v1/tenants
```

### Seeded admin vs Root user confusion

```
# When PANGOLIN_SEED_ADMIN=true creates a user, it's a TENANT ADMIN
# They need Bearer tokens, NOT Basic Auth

# To use Basic Auth, you need BOTH:
export PANGOLIN_SEED_ADMIN=true
export PANGOLIN_ADMIN_USER=admin
export PANGOLIN_ADMIN_PASSWORD=password
export PANGOLIN_ROOT_USER=admin          # ← Also needed for Basic Auth
export PANGOLIN_ROOT_PASSWORD=password  # ← Also needed for Basic Auth
```

## Issue: Token expired

**Symptoms:** 401 with "Invalid token: token expired"

**Solution:** Obtain a new token via `/api/v1/users/login`

## Issue: Token revoked

**Symptoms:** 401 with "Token has been revoked"

**Solution:** Token was explicitly revoked. Obtain a new token.

# Issue: Different behavior across backends

**Symptoms:** Authentication works with MemoryStore but fails with PostgresStore

**Root Cause:** Environment variables not consistently set across test runs

**Solution:** Ensure all required environment variables are set for each backend:

```
# Required for ALL backends if using Basic Auth
export PANGOLIN_ROOT_USER=admin
export PANGOLIN_ROOT_PASSWORD=password

# Required for persistent backends (Postgres, Mongo, SQLite)
export PANGOLIN_SEED_ADMIN=true
export PANGOLIN_ADMIN_USER=admin
export PANGOLIN_ADMIN_PASSWORD=password
```

## Environment Variables Reference

Variable	Purpose	Default	Required For
`PANGOLIN_ROOT_USER`	Root username for Basic Auth	(empty)	Basic Auth as Root
`PANGOLIN_ROOT_PASSWORD`	Root password for Basic Auth	(empty)	Basic Auth as Root
`PANGOLIN_SEED_ADMIN`	Auto-create tenant admin on startup	`false`	Admin provisioning
`PANGOLIN_ADMIN_USER`	Seeded admin username	`tenant_admin`	Admin provisioning
`PANGOLIN_ADMIN_PASSWORD`	Seeded admin password	`password123`	Admin provisioning
`PANGOLIN_JWT_SECRET`	JWT signing secret	`default_secret_for_dev`	Token verification
`PANGOLIN_NO_AUTH`	Disable auth (dev only)	`false`	Development

## Security Best Practices

### Production Deployment

- Set strong `PANGOLIN\_JWT\_SECRET`
- Use HTTPS for all API communication
- Rotate Root credentials regularly
- Never use `PANGOLIN\_NO\_AUTH=true` in production

### Token Management

- Set appropriate token expiration times
- Implement token refresh mechanism

Revoke tokens on logout

Store tokens securely on client side

## Password Security

Passwords are hashed using bcrypt (cost factor 10)

Never log or expose password hashes

Enforce strong password policies

## API Keys

Treat API keys like passwords

Set expiration dates

Revoke unused keys

Audit key usage regularly

# Testing Authentication

## Integration Test Setup

For Pylceberg integration tests or other automated testing:

```
# Start API with Root credentials for test setup
export PANGOLIN_ROOT_USER=admin
export PANGOLIN_ROOT_PASSWORD=password
export PANGOLIN_SEED_ADMIN=true
export PANGOLIN_ADMIN_USER=admin
export PANGOLIN_ADMIN_PASSWORD=password

# Run API
cargo run --bin pangolin_api

# Tests can now use Basic Auth for setup
curl -u admin:password http://localhost:8080/api/v1/tenants

# Or obtain Bearer token for tenant operations
TOKEN=$(curl -X POST http://localhost:8080/api/v1/users/login \\\
  -H "Content-Type: application/json" \\\
  -d '{"username":"admin","password":"password"}' \\\
  | jq -r '.token')
```

## Related Documentation

[User Management](#)

[RBAC System](#)

Source: branching.md

## Branching & Merging (Git-for-Data)

Pangolin implements a "Git-for-Data" model similar to Project Nessie, allowing users to isolate changes, experiment with data, and safely merge updates.

### Core Concepts

#### 1. Branches

A **Branch** is a named pointer to a specific commit hash in the history.

**main**: The default branch (protected).

**dev / feature-\*\*\***: **Temporary branches for ingestion or experimentation.**

Just like Git, creating a branch is cheap ( $O(1)$ ) as it is just a reference copy.

#### 2. Commits

A **Commit** is an immutable record of a state change.

**Operations**: Each commit contains a list of ``Put`` (add/update table) or ``Delete`` (remove table) operations.

**Parent**: Links to the previous commit, forming a Directed Acyclic Graph (DAG).

**Snapshot**: In Iceberg terms, a commit points to a specific ``metadata.json`` location for a table.

#### 3. Tags

A **Tag** is an immutable reference to a specific commit. Useful for marking releases (e.g., ``Q1_REPORT_FINAL``).

### Merge Lifecycle

Merging is managed via the ``MergeOperation`` model, which tracks the transition from initiation to completion.

#### 1. Initiation

A merge is started between a ``source`` and ``target`` branch. Pangolin identifies the **Base Commit** (common ancestor) to perform a 3-way analysis.

## 2. Conflict Detection

Pangolin automatically detects:

**Schema Conflict:** Incompatible evolution on both branches.

**Data Conflict:** Concurrent writes to overlapping partitions.

**Metadata Conflict:** Conflicting table property changes.

## 3. Resolution

**Auto-Merge:** Orthogonal changes are applied automatically, and the operation moves to ``Completed``.

**Manual Intervention:** If conflicts are found, the operation enters ``Conflicted`` status. Users must use the API to resolve each conflict using a ``ResolutionStrategy`` (``TakeSource``, ``TakeTarget``, or ``ThreeWayMerge``).

## 4. Completion

Once all conflicts are resolved, the merge is finalized, creating a new commit on the target branch.

Source: [caching.md](#)

## Caching Architecture

To achieve high performance and low latency, particularly for object store operations, Pangolin implements a multi-layered caching strategy.

### 1. Metadata Cache (``MetadataCache``)

**Location:** ``pangolin_store/src/metadata_cache.rs``

**Backend:** ``moka`` (High-performance LRU cache)

Reading Iceberg metadata files (snapshots, manifests) from S3/GCS is a high-latency operation. The ``MetadataCache`` stores the binary content of these files in memory to avoid repeated network calls.

## Configuration

**Max Capacity:** 10,000 entries (default).

**TTL (Time-To-Live):** 5 minutes (default).

**Key:** Absolute location string (e.g., ``s3://warehouse/db/table/metadata/v1.json``).

**Value:** Byte vector (``Vec<u8>``).

# Usage

Used by all ``CatalogStore`` implementations (``MemoryStore``, ``SqliteStore``, ``PostgresStore``, ``MongoStore``).

Shared across all modular sub-backends (e.g., ``postgres/assets.rs`` and ``postgres/tenants.rs`` both leverage the same cache instance).

Implements a ``get_or_fetch`` pattern to handle cache misses transparently.

```
// logical flow
let data = metadata_cache.get_or_fetch(location, || async {
    // Only executed on cache miss
    object_store.get(location).await
}).await?;
```

## 2. Object Store Cache (``ObjectStoreCache``)

**Location:** ``pangolin_store/src/object_store_cache.rs``

**Backend:** ``DashMap`` (Concurrent HashMap)

Creating new connections to object storage (S3 clients, Azure clients) is expensive due to potential TLS handshakes and authentication checks. This cache reuses existing client instances for the same configuration.

### Logic

**Key:** Hash of (Endpoint + Bucket + AccessKey + Region).

**Value:** ``Arc<dyn ObjectStore>`` (Thread-safe shared client).

**Eviction:** None (lives for the application lifecycle), but lightweight.

### Benefits

drastically reduces connection overhead for frequently accessed warehouses.

Crucial for multi-tenant environments where many warehouses exist.

## 3. Database Connection Pooling

While not a custom cache, all persistent stores utilize connection pooling:

**Postgres:** ``sqlx::PgPool``

**SQLite:** ``sqlx::SqlitePool``

**MongoDB:** ``mongodb::Client`` internal pooling

## 4. Wrapper Caching (``CachedCatalogStore``)

**Location:** ``pangolin_api/src/cached_store.rs``

**Backend:** ``moka``

**Wrapper Pattern:** Implements the ``CatalogStore`` trait and wraps another store instance.

To reduce database load for high-frequency "Read Configuration" paths, the ``CachedCatalogStore`` wraps the primary storage backend.

## Caching Logic

**Scope:** Currently caches ``get_warehouse`` calls (critical for every STS/Signer request).

**Key:** ``(TenantId, WarehouseName)`` tuple.

**Strategy:** Read-Through / Write-Update / Delete-Invalidate.

**Read:** Check cache -> Return if hit -> Fetch from inner DB -> Insert to cache -> Return.

**Update:** Update inner DB -> Update cache with new value immediately.

**Delete:** Invalidate cache key -> Delete from inner DB.

## Future Expansion

This pattern establishes the architectural precedent for caching ``get_tenant``, ``get_user``, and other high-read, low-write entities.

## Future Caching Plans

**Manifest Caching:** Parsing Avro manifest files is CPU intensive. Future optimizations may cache the parsed ``ManifestFile`` structs rather than just the raw bytes.

**Distributed Cache:** For horizontal scaling, replacing in-memory ``moka`` with Redis.

Source: `catalog-store-trait.md`

## CatalogStore Trait

### Overview

The ``CatalogStore`` trait is the fundamental abstraction layer for Pangolin's metadata and state persistence. It enforces a strict multi-tenant interface, ensuring that data isolation is handled at the storage-engine level.

**Source:** ``pangolin_store/src/lib.rs``

## Core Infrastructure Operations

### 1. Tenant Management

Handles the lifecycle of the highest-level isolation units.

```
`create_tenant(tenant: Tenant) -> Result<()>`
```

```
`get_tenant(id: Uuid) -> Result<Option<Tenant>>`
```

```
`list_tenants() -> Result<Vec<Tenant>>`
```

```
`update_tenant(id: Uuid, updates: TenantUpdate) -> Result<Tenant>`
```

```
`delete_tenant(id: Uuid) -> Result<()>`
```

## 2. Warehouse & Catalog Setup

Defines the storage and catalog configurations.

```
`create_warehouse(tenant_id: Uuid, warehouse: Warehouse) -> Result<()>`
```

```
`create_catalog(tenant_id: Uuid, catalog: Catalog) -> Result<()>`
```

```
`get_catalog(tenant_id: Uuid, name: String) -> Result<Option<Catalog>>`
```

```
`list_catalogs(tenant_id: Uuid) -> Result<Vec<Catalog>>`
```

## Data & Object Operations

### 3. Namespace & Asset Management

Pangolin uses hierarchical namespaces and versioned assets.

```
`create_namespace(tenant_id: Uuid, catalog: &str, namespace: Namespace) ->  
Result<()>`
```

```
`list_namespaces(tenant_id: Uuid, catalog: &str, parent: Option<String>) ->  
Result<Vec<Namespace>>`
```

```
`create_asset(tenant_id: Uuid, catalog: &str, branch: Option<String>, namespace:  
Vec<String>, asset: Asset) -> Result<()>`
```

```
`get_asset(tenant_id: Uuid, catalog: &str, ..., name: String) -> Result<Option<Asset>>`
```

```
`get_asset_by_id(tenant_id: Uuid, asset_id: Uuid) -> Result<Option<(Asset, String,  
Vec<String>)>>`
```

### 4. Branching & Tagging

Git-like operations for data versioning.

```
`create_branch(tenant_id: Uuid, catalog: &str, branch: Branch) -> Result<()>`
```

```
`merge_branch(tenant_id: Uuid, catalog: &str, source: String, target: String) ->  
Result<()>`
```

```
`create_tag(tenant_id: Uuid, catalog: &str, tag: Tag) -> Result<()>`
```

```
`delete_tag(tenant_id: Uuid, catalog: &str, name: String) -> Result<()>`
```

## Governance & Security

### 5. IAM (Users, Roles, Permissions)

Granular access control management.

```
`create_user(_user: User) -> Result<()>`
```

```
`list_users(_tenant_id: Option<Uuid>) -> Result<Vec<User>>`
```

```
`create_role(_role: Role) -> Result<()>`
```

```
`assign_role(_user_role: UserRole) -> Result<()>`
```

```
`list_user_permissions(_user_id: Uuid) -> Result<Vec<Permission>>`
```

### 6. Service Users

Programmatic access management.

```
`create_service_user(_service_user: ServiceUser) -> Result<()>`
```

```
`get_service_user_by_api_key_hash(_hash: &str) -> Result<Option<ServiceUser>>`
```

### 7. Audit Logging

System-wide activity tracking.

```
`log_audit_event(tenant_id: Uuid, event: AuditLogEntry) -> Result<()>`
```

```
`list_audit_events(tenant_id: Uuid, filter: Option<AuditLogFilter>) ->  
Result<Vec<AuditLogEntry>>`
```

```
`count_audit_events(tenant_id: Uuid, filter: Option<AuditLogFilter>) -> Result<usize>`
```

## Maintenance & System

### 8. Table Maintenance

Low-level object storage cleanup.

```
`expire_snapshots(tenant_id: Uuid, ..., table: String, retention_ms: i64) -> Result<()>`
```

```
`remove_orphan_files(tenant_id: Uuid, ..., table: String, older_than_ms: i64) ->  
Result<()>`
```

### 9. System Configuration

```
`get_system_settings(tenant_id: Uuid) -> Result<SystemSettings>`
```

```
`update_system_settings(tenant_id: Uuid, settings: SystemSettings) -> Result<SystemSettings>`
```

## 10. Federated Catalog Operations

```
`sync_federated_catalog(tenant_id: Uuid, catalog_name: &str) -> Result<()>`
```

```
`get_federated_catalog_stats(tenant_id: Uuid, catalog_name: &str) -> Result<SyncStats>`
```

## 11. Token Lifecycle

```
`revoke_token(token_id: Uuid, expires_at: DateTime<Utc>, reason: Option<String>) -> Result<()>`
```

```
`is_token_revoked(token_id: Uuid) -> Result<bool>`
```

```
`cleanup_expired_tokens() -> Result<usize>`
```

```
`list_active_tokens(tenant_id: Uuid, user_id: Option<Uuid>, pagination: Option<PaginationParams>) -> Result<Vec<TokenInfo>>`
```

```
`store_token(token: TokenInfo) -> Result<()>`
```

```
`validate_token(token: &str) -> Result<Option<TokenInfo>>`
```

## 12. Permissions & Roles

```
`create_permission(permission: Permission) -> Result<()>`
```

```
`revoke_permission(permission_id: Uuid) -> Result<()>`
```

```
`list_permissions(tenant_id: Uuid, pagination: Option<PaginationParams>) -> Result<Vec<Permission>>`
```

```
`list_user_permissions(user_id: Uuid, pagination: Option<PaginationParams>) -> Result<Vec<Permission>>`
```

## Implementation Requirements

Every ``CatalogStore`` implementation must:

**Enforce Multi-tenancy:** Every method (where applicable) requires a ``tenant_id``. Implementations must never leak data across tenants.

**Thread Safety:** Derive ``Send + Sync`` to support concurrent API requests.

**Async/Await:** Use ``async_trait`` for all persistence logic.

**Error Handling:** Return ``anyhow::Result`` for consistent error propagation up to the API layer.

# Project Dependencies & Tech Stack

## Overview

Pangolin is built as a high-performance Rust backend (``pangolin_api``) paired with a modern SvelteKit frontend (``pangolin_ui``). This document outlines the key libraries and tools that power the system.

### Backend (Rust)

The backend is organized as a Cargo workspace with multiple crates.

#### core Frameworks

Dependency	Version	Purpose
<code>axum</code>	<code>`0.7`</code>	High-performance, ergonomic async web framework. Handles HTTP routing, extractors, and middleware.
<code>tokio</code>	<code>`1.0`</code>	Asynchronous runtime. Provides the event loop, async TCP, and task scheduling.
<code>utoipa</code>	<code>`4.2`</code>	OpenAPI documentation generator (used for Swagger UI).

#### Data & Serialization

Dependency	Version	Purpose
<code>serde</code>	<code>`1.0`</code>	Serialization/Deserialization framework.
<code>sqlx</code>	<code>`0.8`</code>	(In <code>`pangolin_store`</code> ) Async SQL toolkit for PostgreSQL and SQLite.
<code>mongodb</code>	<code>`3.4`</code>	(In <code>`pangolin_store`</code> ) Official async driver for MongoDB.
<code>chrono</code>	<code>`0.4`</code>	Date and Time handling.

#### Storage & Cloud

Dependency	Version	Purpose
<code>object_store</code>	<code>`0.11`</code>	Unified interface for S3, Azure Blob, GCS, and local files.
<code>aws-sdk-s3</code>	<code>`1.0`</code>	Native AWS SDK for S3 and STS.

#### Utility & Logging

Dependency	Version	Purpose
<code>tracing</code>	<code>`0.1`</code>	Instrumentation and structural logging.
<code>thiserror</code>	<code>`1.0`</code>	Ergonomic custom error type derivation.
<code>anyhow</code>	<code>`1.0`</code>	Flexible error handling for application code.
<code>dashmap</code>	<code>`6.0`</code>	Concurrent HashMap. Used in <code>`MemoryStore`</code> for thread-safe

in-memory state. |  
| [dotenv](#) | `0.15` | Loads environment variables from `.env` files. |  
| [clap](#) | `4.4` | (In CLIs) Command Line Argument Parser. Powers `pangolin-admin` and `pangolin-user`. |

## Frontend (SvelteKit)

The management UI (`pangolin\_ui`) is a Single Page Application (SPA) built with SvelteKit.

## Core Frameworks

Dependency	Purpose
[SvelteKit](#)	The meta-framework. Handles routing, SSR/CSR, and build optimization.
[Svelte 5](#)	The UI component framework.
[Vite](#)	Next-generation frontend tooling. Powers the dev server and build process.

## Styling & Components

Dependency	Purpose
[TailwindCSS](#)	Utility-first CSS framework for layout and spacing.
[SMUI](#)	Svelte Material UI. implementations of Material Design components (Buttons, Inputs, Tables).
[Material Icons](#)	Icon set used throughout the UI.

## Utilities

Dependency	Purpose
[marked](#)	Markdown parser. Used for rendering descriptions and documentation.
[TypeScript](#)	Static typing for JavaScript.

## Dev Tools & Testing

Tool	Purpose
**Docker & Docker Compose**	Containerization for consistent development and deployment environments.
**MinIO**	S3-compatible object storage for local testing.
**Pylceberg**	Python client for Apache Iceberg. Used for end-to-end integration testing.
**Vitest**	Unit testing framework for the generic frontend logic.
**Playwright**	End-to-end testing for the frontend.

## Source: enums.md

## Pangolin Enums

This document details the core enumerations defined in ``pangolin_core``. These define the discrete states and types used across the system.

## Data & Storage

### AssetType

Defines the format and nature of a data asset.

``SCREAMING_SNAKE_CASE`` serialization.

``IcebergTable``, ``DeltaTable``, ``HudiTable``, ``ParquetTable``, ``CsvTable``, ``JsonTable``.

``View``, ``MIModel``, ``ApachePaimon``, ``Vortex``, ``Lance``, ``Nimble``.

``Directory``, ``VideoFile``, ``ImageFile``, ``DbConnString``, ``Other``.

### CatalogType

Defines the nature of a catalog backend.

``Local``: Native Pangolin catalog with full versioning.

``Federated``: Proxy to an external Iceberg REST catalog.

### BranchType

Defines the purpose and behavior of a branch.

``Ingest``: For transient data ingestion; can be merged.

``Experimental``: For sandbox testing; limited merge capabilities.

### VendingStrategy

Configuration for how cloud credentials are vended.

``AwsSts``: AWS STS AssumeRole with external ID support.

``AwsStatic``: Static IAM credentials.

``AzureSas``: Azure Blob SAS tokens.

``GcpDownscoped``: GCP Downscoped access tokens.

``None``: No vending; client provides own credentials.

## Federated Catalogs

### SyncStats

Statistics returned after a catalog sync operation.

``last_synced_at``: Timestamp of last sync.

``sync_status``: "Success", "Failed", "Syncing".

``tables_synced``: Count of tables synchronized.

``namespaces_synced``: Count of namespaces synchronized.

``error_message``: Optional error details.

## Identity & Access

### UserRole

Defines the high-level authorization tier for a user.

``Root``: Global system administrator.

``TenantAdmin``: Administrative control over a specific tenant.

``TenantUser``: Standard user within a tenant with granular permissions.

### OAuthProvider

Supported identity providers for OIDC login.

``Google``, ``Microsoft``, ``GitHub``, ``Okta``.

### PermissionScope

Defines the target boundary for a permission grant.

``Tenant``: Applies to all assets in the tenant.

``Catalog``: Scoped to a specific catalog ID.

``Namespace``: Scoped to a catalog + namespace path.

``Asset``: Scoped to a specific asset ID.

``Tag``: Applies to all assets with a matching business tag.

### Action

Atomic actions that can be permitted.

``Read``, ``Write``, ``Delete``, ``Create``, ``Update``, ``List``, ``All``.

``IngestBranching``, ``ExperimentalBranching``.

``ManageDiscovery``: Ability to edit business metadata.

# Conflict & Versioning

## ConflictType

Reasons for merge failure.

`SchemaChange`, `DataOverlap`, `MetadataConflict`, `DeletionConflict`.

## ResolutionStrategy

Strategies for resolving detected conflicts.

`AutoMerge`, `TakeSource`, `TakeTarget`, `Manual`, `ThreeWayMerge`.

## MergeStatus

Lifecycle of a merge operation.

`Pending` (Detection) -> `Conflicted` / `Ready` -> `Resolving` -> `Completed` / `Aborted`.

## Source: handlers.md

## Pangolin API Handlers

This document lists the handler modules responsible for the API implementation, grouped by their domain/functionality.

## Iceberg Core

**File:** `iceberg\_handlers.rs` (Refactor to `iceberg/` in progress)

**Purpose:** Implements the Apache Iceberg REST Catalog Specification.

`get\_iceberg\_catalog\_config\_handler`: Catalog configuration endpoint.

`list\_namespaces` / `create\_namespace` / `delete\_namespace`: Namespace management.

`list\_tables` / `create\_table`: Table lifecycle.

`load\_table` / `update\_table` / `delete\_table`: Table operations.

`report\_metrics`: Metrics reporting.

**Planned Refactor:** See [Iceberg Modularization Plan](#).

## Tenant & Storage Management

**Files:**

``tenant_handlers.rs``: Create, list, details, delete tenants.

``warehouse_handlers.rs``: Configure S3/GCS/Azure storage buckets and vending strategies.

**New in v0.5.0:** ``get_warehouse_credentials`` for direct integration testing of STS/SAS flows.

``asset_handlers.rs``: Generic asset management (Tables, Views, etc.) outside the Iceberg scope.

## Catalog Management

### Files:

``pangolin_handlers.rs``: Core catalog, branch, and tag management.

``federated_catalog_handlers.rs``: Sync, status, and config for external catalogs.

``federated_proxy.rs``: Proxy logic for forwarding requests to federated backends.

## Access Control & Authentication

### Files:

``auth_middleware.rs``: JWT and API key validation layer.

``oauth_handlers.rs``: Google/GitHub/Microsoft OIDC flows.

``token_handlers.rs``: Token generation, rotation, and revocation.

``permission_handlers.rs``: RBAC management (Role CRUD, assignments, grants).

``user_handlers.rs``: User profile management and basic login.

``service_user_handlers.rs``: Machine-to-machine account management and API key rotation.

## Data Versioning (Git-for-Data)

### Files:

``merge_handlers.rs``: Merge operation lifecycle (Start, Complete, Abort).

``conflict_detector.rs``: Logic for detecting schema and data conflicts between branches.

``pangolin_handlers.rs`` (Versioning parts): Branching and tagging logic.

## Governance & Metadata

### Files:

``audit_handlers.rs``: Enhanced audit logging with filtering and count endpoints (Isolated by user/resource).

``business_metadata_handlers.rs``: Business tagging, descriptions, access requests, and ownership.

``signing_handlers.rs``: Endpoint for vending temporary cloud credentials.

## Optimization & Search

### Files:

``optimization_handlers.rs``: Unified search, bulk operations, and name validation.

``dashboard_handlers.rs``: Aggregated statistics for the UI.

## System & Background

### Files:

``system_config_handlers.rs``: Global settings (SMTP, defaults).

``cleanup_job.rs``: Background worker for expired tokens.

## CLI Admin Handlers

**File:** ``pangolin_cli_admin/src/handlers.rs`` (Refactor to ``handlers/`` in progress)

**Planned Refactor:** See [CLI Modularization Plan](#).

## Source: models.md

## Pangolin Core Models

This document details the core data structures (Structs) defined in ``pangolin_core``.

## Identity & Storage

### Tenant

The root multi-tenancy isolation boundary.

``id``: ``Uuid``

``name``: ``String``

``properties``: ``HashMap<String, String>``

### Warehouse

Physical storage backend configuration (S3, GCS, ADLS).

``id`: `Uuid``  
``name`: `String` (Unique per tenant)`  
``tenant_id`: `Uuid``  
``storage_config`: `HashMap<String, String>``  
``vending_strategy`: `Option<VendingStrategy>``  
``use_sts`: `bool` (Legacy flag)`

## Catalog

Logical grouping for assets, maps to an Iceberg Catalog.

``id`: `Uuid` (For granular permissions)`  
``name`: `String``  
``catalog_type`: `CatalogType` (Local or Federated)`  
``warehouse_name`: `Option<String>``  
``storage_location`: `Option<String>``  
``federated_config`: `Option<FederatedCatalogConfig>``  
``properties`: `HashMap<String, String>``

## Asset Management

### Namespace

Hierarchical organization unit (``db.schema``).

``name`: `Vec<String>` (Path parts)`  
``properties`: `HashMap<String, String>``

### Asset

Managed data artifact.

``id`: `Uuid``  
``name`: `String``  
``kind`: `AssetType``  
``location`: `String` (Object store URI)`  
``properties`: `HashMap<String, String>``

# Version Control

## Branch

Pointer to a commit history chain.

```
`name`: `String`  
`head_commit_id`: `Option<Uuid>`  
`branch_type`: `BranchType`  
`assets`: `Vec<String>` (Visible asset names)
```

## Tag

Named immutable reference to a commit.

```
`name`: `String`  
`commit_id`: `Uuid`
```

## Commit

Immutable record of atomic changes.

```
`id`: `Uuid`  
`parent_id`: `Option<Uuid>`  
`timestamp`: `i64` (Epoch)  
`author`: `String`  
`message`: `String`  
`operations`: `Vec<CommitOperation>` (Put or Delete)
```

# Security & Identity

## User

Human user profile.

```
`id`: `Uuid`  
`username`: `String`  
`email`: `String`  
`password_hash`: `Option<String>` (BCrypt)  
`oauth_provider`: `Option<OAuthProvider>`
```

``tenant_id`: `Option<Uuid>``

``role`: `UserRole``

``active`: `bool``

## ServiceUser

Machine account for API key access.

``id`: `Uuid``

``name`: `String``

``tenant_id`: `Uuid``

``api_key_hash`: `String``

``role`: `UserRole``

``last_used`: `Option<DateTime<Utc>>``

``active`: `bool``

## Role

A named group of permissions within a tenant.

``id`: `Uuid``

``name`: `String``

``tenant_id`: `Uuid``

``permissions`: `Vec<PermissionGrant>``

## Permission

Directly assigned permission grant.

``id`: `Uuid``

``user_id`: `Uuid``

``scope`: `PermissionScope``

``actions`: `HashSet<Action>``

## PermissionGrant

Permission definition stored within a Role.

``scope`: `PermissionScope``

-

`actions`: `HashSet<Action>`

## Merge & Discovery

### MergeOperation

State tracking for branch reconciliation.

`id`: `Uuid`

`source\_branch`: `String`

`target\_branch`: `String`

`base\_commit\_id`: `Option<Uuid>`

`status`: `MergeStatus`

`conflicts`: `Vec<Uuid>`

### BusinessMetadata

Governance metadata for an asset.

`asset\_id`: `Uuid`

`owner`: `String`

`tags`: `Vec<String>`

`description`: `Option<String>`

### AccessRequest

Workflow record for requesting asset access.

`id`: `Uuid`

`asset\_id`: `Uuid`

`requester\_id`: `Uuid`

`status`: `String` (Pending/Approved/Rejected)

## System

### SystemSettings

Global configuration for a tenant.

`allow\_public\_signup`: `Option<bool>`

``default_warehouse_bucket`: `Option<String>``

``default_retention_days`: `Option<i32>``

``smtp_host`: `Option<String>`, etc.`

## SyncStats

Performance metrics for federated catalogs.

``last_synced_at`: `Option<DateTime<Utc>>``

``sync_status`: `String``

``tables_synced`: `i32``

Source: `performance_and_scaling.md`

## Performance & Scaling Architecture

Pangolin is designed for high-throughput, low-latency metadata operations. This document details the key architectural patterns used to achieve performance at scale.

### 1. Database Indexing (P1 Optimization)

To support efficient querying for high-cardinality entities, strategic indexes are applied to the backend databases (specifically Postgres).

Key Indexes (``20251227000000_add_perf_indexes.sql``)

``commits.parent_id``: Optimizes ``get_commit_ancestry`` recursion. Crucial for traversing long commit histories (e.g., Iceberg snapshots).

``active_tokens.user_id``: Optimizes ``list_active_tokens`` and token validation lookups, essential for high-concurrency API authentication.

### 2. Pagination Strategy

All list operations in the ``CatalogStore`` trait enforce pagination to prevent memory exhaustion and ensure predictable response times.

``PaginationParams`` Struct

**Limit:** Maximum number of items to return (default 100, max 1000).

**Offset:** Number of items to skip.

## Implementation

**API Layer:** Extracts ``page`` and ``page_size`` query params, converts to ``PaginationParams``.

**Store Layer:** Applies ``LIMIT`` / ``OFFSET`` (SQL) or cursor-based skipping (Mongo).

**Client SDKs:** Paginators are provided to abstract this logic for users.

## 3. Asynchronous I/O

The entire backend stack is fully asynchronous, built on ``tokio`` and ``axum``.

**Non-blocking DB Calls:** Uses ``sqlx`` (Postgres/SQLite) and ``mongodb::Client`` in async mode.

**Concurrent Object Store Access:** ``object_store`` crate usage allows parallel fetching of metadata files.

**Background Tasks:** Tasks like ``cleanup_expired_tokens`` run in detached Tokio tasks to avoid blocking request paths.

## 4. Caching Layers

Performance is further augmented by a multi-tier caching strategy explained in [Caching Architecture](#).

**Level 1:** Wrapper Cache (``CachedCatalogStore``) - Configuration data.

**Level 2:** Metadata Cache (``moka``) - Iceberg Manifest bytes.

**Level 3:** Connection Pools (``ObjectStoreCache``, DB Pools).

Source: `signer-trait.md`

## Signer Trait

### Overview

The ``Signer`` trait provides a secure mechanism for Pangolin to vend temporary, downscoped credentials or presigned URLs to clients (like Pylceberg or Spark) for direct access to object storage.

**Source:** ``pangolin_store/src/signer.rs``

## Credentials Enum

The ``Credentials`` enum encapsulates provider-specific access tokens.

```
#[derive(Debug, Serialize, Deserialize, Clone)]
#[serde(tag = "type")]
pub enum Credentials {
    // AWS S3 credentials (STS or Static)
```

```

    Aws {
        access_key_id: String,
        secret_access_key: String,
        session_token: Option<String>,
        expiration: Option<DateTime<Utc>>,
    },
    // Azure SAS token
    Azure {
        sas_token: String,
        account_name: String,
        expiration: DateTime<Utc>,
    },
    // GCP OAuth2 token
    Gcp {
        access_token: String,
        expiration: DateTime<Utc>,
    },
}

```

## The Signer Trait

Implementations of this trait interact with cloud IAM services to generate temporary tokens.

```

#[async_trait]
pub trait Signer: Send + Sync {
    /// Get temporary credentials for accessing a specific table location.
    /// Implementation should scope permissions to the provided location prefix.
    async fn get_table_credentials(&self, location: &str) -> Result<Credentials>;

    /// Generate a presigned URL for a specific file location.
    async fn presign_get(&self, location: &str) -> Result<String>;
}

```

## Key Behaviors

**Downscoping:** Credentials vended via ``get_table_credentials`` are typically scoped to the specific S3/GCS prefix of the Iceberg table to prevent unauthorized access to other data in the bucket.

**Expiration handling:** Clients are expected to refresh credentials based on the ``expiration`` field provided in the response.

**Transport Security:** Vended credentials should only be transmitted over HTTPS.

## SignerImpl

A default implementation of the ``Signer`` trait is provided for basic setups, though full cloud integrations typically require the ``aws-sts``, ``azure-oauth``, or ``gcp-oauth`` feature

flags.

``prefix``: S3 prefix to scope credentials to (e.g., "warehouse1/catalog1/")

**Returns:** ``Result<Credentials>``

**Usage:**

```
let creds = store.get_table_credentials(  
    "s3://warehouse1/catalog1/table1"  
).await?;
```

## Credential Vending Flow

1. Client requests table metadata
2. API validates permissions
3. Signer generates scoped credentials
4. Credentials returned via X-Iceberg-Access-Delegation header
5. Client uses credentials for direct S3 access

## Security Considerations

### Scope Limitation

Credentials should be scoped to the minimum necessary prefix:

```
// Good - scoped to specific table  
let prefix = format!("{}/{}/{}", warehouse, catalog, namespace);  
  
// Bad - too broad  
let prefix = "/";
```

### Expiration

Set appropriate expiration times:

```
// Short-lived for metadata reads  
let expiration = 3600; // 1 hour  
  
// Longer for large data operations  
let expiration = 14400; // 4 hours
```

### Audit Logging

Log all credential vending operations:

```
store.log_audit_event(tenant_id, AuditLogEntry {
```

```

    action: "vend_credentials".to_string(),
    resource: format!("{}", catalog, table),
    // ...
  }).await?;

```

## AWS S3 Implementation

The default `SignerImpl` uses AWS STS to assume roles:

```

pub struct SignerImpl {
    s3_client: aws_sdk_s3::Client,
    sts_client: aws_sdk_sts::Client,
}

```

### Configuration:

```

[warehouse.storage_config]
s3.endpoint = "https://s3.amazonaws.com"
s3.region = "us-east-1"
s3.role-arn = "arn:aws:iam::123456789012:role/IcebergAccess"

```

## Azure Blob Implementation

Azure uses SAS tokens for temporary access:

```

pub struct AzureSigner {
    account_name: String,
    account_key: String,
}

```

### Configuration:

```

[warehouse.storage_config]
azure.account-name = "mystorageaccount"
azure.account-key = "..."

```

## GCP Storage Implementation

GCP uses service account impersonation:

```

pub struct GcpSigner {
    service_account_email: String,
    private_key: String,
}

```

### Configuration:

```
[warehouse.storage_config]
gcs.project-id = "my-project"
gcs.service-account = "iceberg@my-project.iam.gserviceaccount.com"
```

## Error Handling

All methods return `Result<T>` where errors can include:

**CredentialError:** Failed to generate credentials

**ConfigurationError:** Invalid warehouse configuration

**PermissionError:** Insufficient permissions to assume role

**NetworkError:** Failed to communicate with cloud provider

## Testing

Mock signer for testing:

```
pub struct MockSigner;

#[async_trait]
impl Signer for MockSigner {
    async fn generate_presigned_url(
        &self,
        bucket: &str,
        key: &str,
        _expiration_secs: u64,
    ) -> Result<String> {
        Ok(format!("https://{}.s3.amazonaws.com/{}", bucket, key))
    }

    async fn vend_credentials(
        &self,
        _warehouse_config: &HashMap<String, String>,
        _prefix: &str,
    ) -> Result<VendedCredentials> {
        Ok(VendedCredentials {
            access_key_id: "MOCK_KEY".to_string(),
            secret_access_key: "MOCK_SECRET".to_string(),
            session_token: Some("MOCK_TOKEN".to_string()),
            expiration: Some(Utc::now() + chrono::Duration::hours(1)),
        })
    }
}
```

## Best Practices

**Minimal Scope:** Always scope credentials to the narrowest prefix possible

**Short Expiration:** Use short expiration times (1-4 hours)

**Audit Everything:** Log all credential vending operations

**Validate Config:** Verify warehouse configuration before vending

**Handle Errors:** Gracefully handle cloud provider errors

## Integration with CatalogStore

The `CatalogStore` trait extends `Signer`, making signing capabilities available to all store implementations:

```
#[async_trait]
pub trait CatalogStore: Send + Sync + Signer {
    // ... catalog operations ...
}
```

This allows handlers to vend credentials directly:

```
let creds = store.vend_credentials(&warehouse.storage_config, &prefix).await?;
```

## See Also

[CatalogStore Trait](#)

[Credential Vending Guide](#)

[Storage Configuration](#)

Source: `storage_and_connectivity.md`

# Storage Backend Connectivity & Integration Guide

## Overview

This document explains how Pangolin's storage backends connect to object storage (S3/MinIO), how credential vending works, and common pitfalls to avoid. It consolidates lessons learned from debugging PostgresStore integration issues.

## Architecture Overview

```
graph TD
    A[PyIceberg Client] -->|1. Request Table| B[Iceberg REST API]
    B -->|2. Authenticate| C[Auth Middleware]
    C -->|3. Load Metadata| D[Storage Backend]
    D -->|4. Read from S3| E[Object Store]
```

```
B -->|5. Vend Credentials| F[Warehouse Config]
F -->|6. Return Creds| A
A -->|7. Write Data| E
```

## Key Components

**Storage Backend** (`CatalogStore` trait): PostgreSQL, MongoDB, SQLite, or Memory. All backends follow a modular directory structure (e.g., `src/postgres/tenants.rs`) to prevent monolithic file growth.

**Object Store Factory:** Creates S3/Azure/GCS clients from warehouse config

**Credential Vending:** Returns temporary credentials to Pylceberg

**Warehouse Configuration:** Stores S3 endpoint, bucket, and access credentials

## Modular Storage Architecture

As of December 2025, all Pangolin storage backends have been refactored into a modular pattern. See the [Backend Architecture Audit](#) for detailed status.

Each backend now resides in its own directory:

```
`pangolin_store/src/postgres/`
```

```
`pangolin_store/src/mongo/`
```

```
`pangolin_store/src/sqlite/`
```

```
`pangolin_store/src/memory/`
```

Within these directories, logic is subdivided by functional domain (e.g., `assets.rs`, `namespaces.rs`, `auth.rs`), making the codebase much easier to audit and extend.

## Critical Configuration Requirements

### 1. Environment Variables for Testing

For PostgresStore (and other backends) to work with Pylceberg integration tests:

```
# Database Connection
export DATABASE_URL=postgres://pangolin:password@localhost:5432/pangolin
export PANGOLIN_STORAGE_TYPE=postgres

# Authentication (REQUIRED for Basic Auth)
export PANGOLIN_ROOT_USER=admin
export PANGOLIN_ROOT_PASSWORD=password

# S3/MinIO Configuration
export AWS_ACCESS_KEY_ID=minioadmin
export AWS_SECRET_ACCESS_KEY=minioadmin
```

```
export AWS_REGION=us-east-1
export AWS_ENDPOINT_URL=http://localhost:9000
export S3_ENDPOINT=http://localhost:9000
```

[!IMPORTANT]

**Basic Auth requires BOTH ``PANGOLIN_ROOT_USER`` and ``PANGOLIN_ROOT_PASSWORD``.** Without these, the auth middleware will reject all Basic Auth requests, even if a tenant admin user exists in the database.

## 2. Warehouse Configuration Format

Warehouse ``storage_config`` must use specific key names:

```
{
  "type": "s3",
  "s3.bucket": "postgres-bucket",
  "s3.region": "us-east-1",
  "s3.endpoint": "http://localhost:9000",
  "s3.access-key-id": "minioadmin",
  "s3.secret-access-key": "minioadmin"
}
```

[!CAUTION]

**Key names MUST match exactly:** ``s3.access-key-id`` and ``s3.secret-access-key``, NOT ``access_key_id`` or ``AWS_ACCESS_KEY_ID``. The code extracts credentials using these exact keys.

## 3. Bucket Naming Convention

Each backend should use its own bucket in MinIO:

**MemoryStore:** ``memory-bucket``

**SqliteStore:** ``sqlite-bucket``

**MongoStore:** ``mongo-bucket``

**PostgresStore:** ``postgres-bucket``

[!WARNING]

Do NOT use a generic ``test-bucket`` unless it actually exists in MinIO. The test script's ``BUCKET_NAME`` environment variable must match an existing bucket.

## Common Issues & Solutions

### Issue 1: "Missing or invalid authorization header"

**Symptoms:** All API requests return 401, even with valid credentials

**Root Causes:**

`PANGOLIN\_ROOT\_USER` and `PANGOLIN\_ROOT\_PASSWORD` not set

Using Basic Auth for tenant users (only works for Root)

Tenant users must use Bearer tokens from `/api/v1/users/login`

### Solution:

```
# For Root user Basic Auth
export PANGOLIN_ROOT_USER=admin
export PANGOLIN_ROOT_PASSWORD=password

# For tenant users, get a token
TOKEN=$(curl -X POST http://localhost:8080/api/v1/users/login \
  -H "Content-Type: application/json" \
  -d '{"username":"tenant_admin","password":"password123"}' \
  | jq -r '.token')

curl -H "Authorization: Bearer $TOKEN" http://localhost:8080/api/v1/catalogs
```

**Code Reference:** [auth\\_middleware.rs:L174-210](#)

## Issue 2: "NoSuchBucket" errors

**Symptoms:** S3 operations fail with "The specified bucket does not exist"

### Root Causes:

Bucket name mismatch between test config and MinIO

`object\_store\_factory` parsing bucket from S3 path instead of using config

Bucket not created in MinIO

### Solution:

```
# Verify bucket exists
docker exec pangolin-minio-1 mc ls minio/

# Create bucket if needed
docker exec pangolin-minio-1 mc mb minio/postgres-bucket --ignore-existing

# Ensure test uses correct bucket name
export BUCKET_NAME=postgres-bucket
```

<b>Code</b>	<b>Fix</b>	<b>Applied:</b>	object_store_factory.rs:L25-36	=
/pangolin/pangolin_store/src/object_store_factory.rs#L25-36				

```
// CORRECT: Use bucket from config first
let bucket = config
    .get("s3.bucket")
    .map(|s| s.as_str())
    .or_else(|| {
```

```
// Fallback: parse from location
location.strip_prefix("s3://")
  .and_then(|rest| rest.split_once('/'))
  .map(|(b, _)| b)
})
.ok_or_else(|| anyhow::anyhow!("Cannot determine S3 bucket"))?;

// WRONG: Always parsing from path
let (bucket, _) = location.strip_prefix("s3://").unwrap().split_once('/').unwrap();
```

## Issue 3: "INVALID\_ACCESS\_KEY\_ID" when Pylceberg writes data

**Symptoms:** Table creation succeeds, but data writes fail with invalid credentials

**Root Cause:** Credential vending code looks for wrong key names in warehouse config

**Solution:** Fix key extraction in `iceberg\_handlers.rs`

```
// CORRECT: Match warehouse config keys
let access_key = warehouse.storage_config.get("s3.access-key-id").cloned();
let secret_key = warehouse.storage_config.get("s3.secret-access-key").cloned();

// WRONG: These keys don't exist in config
let access_key = warehouse.storage_config.get("access_key_id").cloned();
let secret_key = warehouse.storage_config.get("secret_access_key").cloned();
```

**Code Reference:** [iceberg\\_handlers.rs:L981-982](#)

## Issue 4: "Invalid JSON: expected value at line 1 column 1"

**Symptoms:** Metadata read operations fail

**Root Causes:**

Metadata file was never written to S3 (write failed silently)

S3 connectivity issues (wrong endpoint, credentials)

Empty response from S3 being parsed as JSON

**Debugging Steps:**

```
# 1. Check if file exists in MinIO
docker exec pangolin-minio-1 mc ls minio/postgres-bucket/cat_vended/ns1/table1/metadata/

# 2. Try to read the file
docker exec pangolin-minio-1 mc cat minio/postgres-bucket/cat_vended/ns1/table1/metadata/00000-*.metadata.json

# 3. Test S3 connectivity from Python
python3 -c "
```

```
import boto3
s3 = boto3.client('s3',
    endpoint_url='http://localhost:9000',
    aws_access_key_id='minioadmin',
    aws_secret_access_key='minioadmin')
print('Buckets:', [b['Name'] for b in s3.list_buckets()['Buckets']])
"
```

# 4. Check API logs for write errors  
 grep -i "Failed to write\|ERROR" api\_postgres.log

## Issue 5: PostgreSQL Permission Errors

**Symptoms:** API fails to start with "permission denied for schema public"

**Root Cause:** The `pangolin` database user lacks permissions to create tables

**Solution:**

```
docker exec pangolin-postgres psql -U postgres -d pangolin -c \
  "GRANT ALL ON SCHEMA public TO pangolin; \
  GRANT ALL ON ALL TABLES IN SCHEMA public TO pangolin; \
  GRANT ALL ON ALL SEQUENCES IN SCHEMA public TO pangolin;"
```

## Credential Vending Flow

### How It Works

**Pylceberg sends `X-Iceberg-Access-Delegation: vended-credentials` header**

**API's `load\_table` endpoint:**

Retrieves warehouse config for the catalog

Extracts `s3.access-key-id` and `s3.secret-access-key`

Returns credentials in `TableResponse.config`

**Pylceberg uses vended credentials** to read/write data directly to S3

### Code Flow

```
// 1. Extract credentials from warehouse config
let credentials = match store.get_catalog(tenant_id, catalog_name).await {
  Ok(Some(catalog)) => {
    if let Some(warehouse_name) = catalog.warehouse_name {
      match store.get_warehouse(tenant_id, warehouse_name).await {
        Ok(Some(warehouse)) => {
          let access_key = warehouse.storage_config
            .get("s3.access-key-id").cloned();
```

```

        let secret_key = warehouse.storage_config
            .get("s3.secret-access-key").cloned();

        if let (Some(ak), Some(sk)) = (access_key, secret_key) {
            Some((ak, sk))
        } else {
            None
        }
    },
    _ => None
}
} else {
    None
}
},
_ => None
};

// 2. Return credentials in response
TableResponse::with_credentials(location, metadata, credentials)

```

## Vending Strategy Configuration

The warehouse's `vending\_strategy` field determines how credentials are generated:

```

{
  "vending_strategy": {
    "AwsStatic": {
      "access_key_id": "minioadmin",
      "secret_access_key": "minioadmin"
    }
  }
}

```

### [!NOTE]

For MinIO/local testing, `AwsStatic` is sufficient. Production should use `AwsSts` for temporary credentials.

## Testing Checklist

Before running Pylceberg integration tests against a new backend:

- [ ] MinIO container is running (`docker ps | grep minio`)
- [ ] Bucket exists (`docker exec pangolin-minio-1 mc ls minio/`)
- [ ] Database is accessible (for PostgreSQL/MongoDB)
- [ ] Database user has proper permissions
- [ ] `PANGOLIN\_ROOT\_USER` and `PANGOLIN\_ROOT\_PASSWORD` are set

- [ ] S3 environment variables are set (`AWS\_ACCESS\_KEY\_ID`, `S3\_ENDPOINT`)
- [ ] Warehouse config uses correct key names (`s3.access-key-id`)
- [ ] `BUCKET\_NAME` environment variable matches existing bucket
- [ ] API starts successfully and listens on port 8080

## Network Considerations

### Host vs Docker Network

**API running on host:** Use `localhost:9000` for MinIO endpoint

**API running in Docker:** Use `minio:9000` (Docker network hostname)

**Pylceberg on host:** Always uses `localhost:9000`

[!TIP]

For local development, run the API on the host (not in Docker) to avoid network complexity. Use `localhost` for all endpoints.

## Debugging Workflow

When a backend fails Pylceberg tests:

### Verify Authentication

```
curl -u admin:password http://localhost:8080/api/v1/tenants
```

If fails: Check `PANGOLIN\_ROOT\_USER`/`PASSWORD` env vars

### Verify S3 Connectivity

```
docker exec pangolin-minio-1 mc ls minio/
```

If fails: Start MinIO container

### Verify Bucket Exists

```
docker exec pangolin-minio-1 mc mb minio/postgres-bucket --ignore-existing
```

### Check Warehouse Config

```
SELECT storage_config FROM warehouses WHERE name='test-warehouse';
```

Verify keys are `s3.access-key-id` not `access\_key\_id`

### Test Credential Vending

```
curl -H "Authorization: Bearer $TOKEN" \
"http://localhost:8080/v1/cat_vended/v1/config" | jq '.defaults'
```

Should include `X-Iceberg-Access-Delegation: vended-credentials`

## Check API Logs

```
grep -E "ERROR|Failed|panic" api_postgres.log
```

## Key Files Reference

File	Purpose	Common Issues
<a href="#">auth_middleware.rs</a>	Authentication logic	Basic Auth only works for Root users
<a href="#">iceberg_handlers.rs</a>	Iceberg REST endpoints	Credential vending key names
<a href="#">object_store_factory.rs</a>	S3 client creation	Bucket extraction logic
<a href="#">postgres.rs</a>	PostgreSQL backend	`write_file`/`read_file` implementations
<a href="#">test_pyiceberg_matrix.py</a>	Integration tests	Environment variable configuration

## Lessons Learned

### 1. Configuration Key Consistency

**Problem:** Different parts of the codebase used different key names for the same configuration.

**Solution:** Standardize on warehouse config format (`s3.access-key-id`) and update all code to use these exact keys.

[!NOTE]

**Current Limitation:** The credential vending logic in `iceberg\_handlers.rs` currently only extracts S3 credentials (`s3.access-key-id` and `s3.secret-access-key`). While `object\_store\_factory.rs` supports Azure (`azure.account-name`, `azure.account-key`) and GCP (`gcp.service-account`), the `load\_table` and `create\_table` endpoints don't yet extract and vend these credentials. This means:

⚠️ S3/MinIO: Full credential vending support

⚠️ Azure/GCP: Object store factory works, but credentials must be client-provided (not vended)

To add Azure/GCP credential vending, update `iceberg\_handlers.rs:L976-997` to detect storage type and extract appropriate credentials.

### 2. Silent Errors

**Problem:** Errors in `create\_user` and S3 operations were swallowed with `Err(\_) => log::debug!()`.

**Solution:** Always log errors at appropriate levels and propagate them when they

indicate real failures.

### 3. Environment Variable Dependencies

**Problem:** Tests worked for some backends but not others due to missing environment variables.

**Solution:** Document ALL required environment variables and create a comprehensive testing checklist.

### 4. Network Hostname Confusion

**Problem:** Using `localhost` vs Docker network hostnames caused connectivity issues.

**Solution:** For local development, run everything on the host network to avoid Docker networking complexity.

### 5. Bucket Naming Assumptions

**Problem:** Tests assumed a `test-bucket` existed, but each backend had its own bucket.

**Solution:** Use backend-specific bucket names and make `BUCKET\_NAME` configurable via environment variable.

## Related Documentation

[Authentication Architecture](#)

[Warehouse Configuration](#)

[Pylceberg Integration](#)

[Security & Vending](#)

Source: authentication.md

## Authentication & Access Control

Pangolin supports secure authentication via **JWT Tokens**, **API Keys (Service Users)**, and **OAuth 2.0 / OIDC** (Google, GitHub, Microsoft, Okta).

### Authentication Modes

Pangolin can be configured in three primary authentication modes via environment variables:

#### 1. No-Auth Mode (Development Only)

Disables all authentication checks. All requests are treated as being made by the `root` superadmin.

**Environment Variable:** `PANGOLIN\_NO\_AUTH=true`

**Use Case:** Local development, quick prototyping, automated integration tests.

[!CAUTION]

Never use No-Auth mode in production or any environment accessible from the internet.

## 2. JWT Authentication (Standard)

The default mode for secure, stateless session management.

**Root Credentials:** Initial setup uses `PANGOLIN\_ROOT\_USER` (default: `admin`) and `PANGOLIN\_ROOT\_PASSWORD` (default: `password`).

**Secret Key:** You **must** set `PANGOLIN\_JWT\_SECRET` to a strong random string (at least 32 characters).

**Flow:** POST credentials to `/api/v1/users/login` -> Receive JWT -> Include in header: `Authorization: Bearer <jwt>`.

### Tenant-Scoped Login

For multi-tenant deployments, users can have the same username across different tenants. To resolve username collisions, include the `tenant-id` field in login requests:

**Root Login** (no tenant context):

```
curl -X POST http://localhost:8080/api/v1/users/login \
  -H "Content-Type: application/json" \
  -d '{"username":"admin","password":"password","tenant-id":null}'
```

**Tenant-Scoped Login** (with tenant context):

```
curl -X POST http://localhost:8080/api/v1/users/login \
  -H "Content-Type: application/json" \
  -d '{"username":"user","password":"pass123","tenant-id":"<tenant-uuid>}'
```

[!IMPORTANT]

The field name is `tenant-id` (kebab-case), not `tenant\_id` (underscore). This is due to the `LoginRequest` struct using `#[serde(rename\_all = "kebab-case")]`.

## 3. OAuth 2.0 / OIDC (Enterprise SSO)

Allows users to sign in with external identity providers.

**Supported Providers:** Google, Microsoft (Entra ID), GitHub, Okta.

**Configuration:** Requires `FRONTEND\_URL` and provider-specific Client IDs/Secrets.

See Provider Setup below.

## Service Users & API Keys

For programmatic access (CI/CD, ETL, automation), use **Service Users**.

Service users use **X-API-Key** authentication rather than JWTs.

They are managed via the UI or the `pangolin-admin service-users` CLI.

See the [Service Users Guide](#) for details.

## Provider Setup (OAuth)

### Google

Create a project in [Google Cloud Console](#).

Setup OAuth 2.0 Client ID for a Web Application.

Redirect URI: `https://<api-domain>/oauth/callback/google`.

**Env:**           `OAUTH\_GOOGLE\_CLIENT\_ID`,           `OAUTH\_GOOGLE\_CLIENT\_SECRET`,  
`OAUTH\_GOOGLE\_REDIRECT\_URI`.

### Microsoft (Azure AD)

Register an app in the [Azure Portal](#).

Add a Web platform and Redirect URI: `https://<api-domain>/oauth/callback/microsoft`.

**Env:**       `OAUTH\_MICROSOFT\_CLIENT\_ID`,       `OAUTH\_MICROSOFT\_CLIENT\_SECRET`,  
`OAUTH\_MICROSOFT\_TENANT\_ID`, `OAUTH\_MICROSOFT\_REDIRECT\_URI`.

### GitHub

Create an OAuth App in **Settings > Developer settings**.

Callback URL: `https://<api-domain>/oauth/callback/github`.

**Env:**           `OAUTH\_GITHUB\_CLIENT\_ID`,           `OAUTH\_GITHUB\_CLIENT\_SECRET`,  
`OAUTH\_GITHUB\_REDIRECT\_URI`.

## Connecting with Pylceberg

Option    A:   Standard    OAuth2    Credential    Flow  
(Recommended)

Use a Service User's `client\_id` (UUID) and `client\_secret` (API Key) for automated authentication.

```

from pyiceberg.catalog import load_catalog

catalog = load_catalog(
    "pangolin",
    **{
        "uri": "http://localhost:8080/v1/default/",
        "type": "rest",
        "credential": "<service_user_uuid>:<api_key>",
        "oauth2-server-uri": "http://localhost:8080/v1/default/v1/oauth/tokens",
        "scope": "catalog"
    }
)

```

#### [!NOTE]

The `oauth2-server-uri` must be the full path. Pangolin supports paths starting with both `/v1/...` and `/api/v1/iceberg/...`.

## Option B: Manual Token (Access Token)

If you have a JWT from a previous login:

```

from pyiceberg.catalog import load_catalog

catalog = load_catalog(
    "pangolin",
    **{
        "uri": "http://localhost:8080/v1/default/",
        "token": "YOUR_JWT_TOKEN",
    }
)

```

## Option B: Credential Vending (Recommended)

Pangolin automatically vends S3/Cloud credentials if the Warehouse is configured with a role. You only need to provide the Pangolin token.

See [Credential Vending](#) for warehouse setup.

## Troubleshooting

### 401 Unauthorized

**Cause:** Token missing, expired, or invalid secret.

**Fix:** Verify `PANGOLIN\_JWT\_SECRET` matches across all instances. Regenerate token via login.

### 403 Forbidden

**Cause:** User lacks RBAC permissions for the resource (Catalog/Namespace/Asset).

**Fix:** Verify roles in the Management UI or via `pangolin-admin permissions`.

Source: README.md

## Backend Storage Options

Pangolin supports multiple backend storage options for persisting catalog metadata. Choose the backend that best fits your deployment requirements.

### Overview

Backend storage is where Pangolin stores **catalog metadata** including:

Tenant information

Warehouse configurations

Catalog definitions

Namespace hierarchies

Asset (table/view) metadata

Branch and tag information

Audit logs

**Note:** Backend storage is separate from **warehouse storage** (S3, Azure, GCS) which stores the actual data files.

### Performance Features

All backends (including Memory/SQLite) now feature:

**Metadata Cache:** In-memory LRU cache (`moka`) for high-latency Iceberg metadata files (manifests, snapshots), defaulting to 5-minute TTL.

**Object Store Connection Pooling:** Reuses S3/GCS/Azure connections to reduce handshake overhead.

**Unified Search:** Optimized full-text search across Catalogs, Namespaces, and Tables regardless of the backing store.

### Quick Comparison

<b>Scalability</b>	Low	Low	High	Very High	
<b>Transactions</b>	ACID	ACID	ACID	ACID	
<b>Foreign Keys</b>	Yes	Yes	Yes	No	
<b>Schema</b>	Strict	Strict SQL	Strict SQL	Flexible	
<b>Replication</b>	No	Manual	Built-in	Built-in	

**Cloud Managed**	No	No	RDS, Azure, GCP	Atlas, Azure, GCP
**Resource Usage**	Very Low	Very Low	Medium	Medium
**Persistence**	No	Yes	Yes	Yes
**Multi-Tenant Isolation**	Excellent	Excellent	Excellent	Excellent

## Choosing a Backend

### Use In-Memory When:

- ~You're developing locally
- ~You're running tests (unit or integration)
- ~You need instant setup with zero configuration
- ~You're prototyping or learning
- ~Data persistence is not required
- ~You're running in CI/CD pipelines

### Use SQLite When:

- ~You're developing locally and need persistence
- ~You need embedded database
- ~You're deploying to edge/IoT devices
- ~You want zero configuration with persistence
- ~You have low concurrent write needs
- ~You want minimal resource usage

### Use PostgreSQL When:

- ~You need a proven, battle-tested SQL database
- ~You want strong consistency and ACID guarantees
- ~You're deploying to traditional infrastructure
- ~You need complex queries and joins
- ~You want managed cloud options (RDS, Azure Database, Cloud SQL)

### Use MongoDB When:

- ~You prefer document-based storage
- ~You need horizontal scalability

- ~You're building cloud-native applications
- ~You want flexible schema evolution
- ~You're already using MongoDB in your stack

## Configuration

Set the backend using the `DATABASE\_URL` environment variable:

```
# In-Memory (default - no DATABASE_URL needed)
# Just don't set DATABASE_URL

# SQLite
DATABASE_URL=sqlite:///path/to/pangolin.db

# PostgreSQL
DATABASE_URL=postgresql://user:password@localhost:5432/pangolin

# MongoDB
DATABASE_URL=mongodb://user:password@localhost:27017/pangolin
```

## Next Steps

[In-Memory Setup Guide](#)

[SQLite Setup Guide](#)

[PostgreSQL Setup Guide](#)

[MongoDB Setup Guide](#)

[Detailed Comparison](#)

## Migration Between Backends

Currently, Pangolin does not provide automated migration tools between backends. If you need to migrate:

Export metadata from source backend (custom script)

Transform to target backend format

Import into target backend

**Tip:** Start with SQLite for development, then migrate to PostgreSQL or MongoDB for production.

Source: [comparison.md](#)

## Backend Storage Comparison

Detailed comparison of PostgreSQL, MongoDB, and SQLite for Pangolin metadata storage.

## Quick Reference

Feature	PostgreSQL	MongoDB	SQLite
<b>Type</b>	Relational (SQL)	Document (NoSQL)	Relational (SQL)
<b>Setup</b>	Medium	Medium	None
<b>Production Ready</b>	Yes	Yes	Yes (small scale)
<b>Persistence</b>	Catalog: No / Data: Yes	Yes	Yes
<b>Scalability</b>	Vertical	Horizontal	Single machine
<b>Concurrent Writes</b>	High	Very High	Low
<b>Concurrent Reads</b>	Very High	Very High	Very High
<b>ACID</b>	Full	Full	Full
<b>Transactions</b>	Yes	Yes	Yes
<b>Foreign Keys</b>	Yes	No	Yes
<b>Indexes</b>	Advanced	Advanced	Basic
<b>Full-Text Search</b>	Built-in	Built-in	FTS5
<b>JSON Support</b>	JSONB	Native	JSON1
<b>Branch Isolation</b>	Yes	Yes (Native)	Yes
<b>Replication</b>	Built-in	Built-in	Manual
<b>Sharding</b>	⚠ Extensions	Built-in	No
<b>Backup</b>	pg_dump	mongodump	File copy
<b>Point-in-Time Recovery</b>	Yes	Yes (Atlas)	No
<b>Cloud Managed</b>	RDS, Azure, GCP	Atlas, Azure, GCP	No
<b>Resource Usage</b>	Medium	Medium	Very Low
<b>Memory Footprint</b>	~100MB+	~100MB+	~10MB
<b>Disk Space</b>	Medium	Medium	Low
<b>Network Required</b>	Yes	Yes	No

## Performance Comparison

### Read Performance

Operation	PostgreSQL	MongoDB	SQLite
Single row by ID	Excellent	Excellent	Excellent
Range queries	Excellent	Excellent	Excellent
Complex joins	Excellent	Good	Good
Aggregations	Excellent	Excellent	Good
Full table scan	Good	Good	Excellent (small DBs)

### Write Performance

Operation	PostgreSQL	MongoDB	SQLite
Single insert	Excellent	Excellent	Excellent
Bulk insert	Excellent	Excellent	Good
Concurrent writes	Excellent	Excellent	Poor
Updates	Excellent	Excellent	Good

| Deletes | Excellent | Excellent | Good |

# Scalability

## PostgreSQL

**Vertical Scaling:** Scale up to very large servers (1TB+ RAM)

**Read Replicas:** Multiple read-only replicas

**Connection Pooling:** PgBouncer for thousands of connections

**Partitioning:** Table partitioning for large datasets

**Limitations:** Single-master writes

## MongoDB

**Horizontal Scaling:** Shard across multiple servers

**Replica Sets:** 3-50 members

**Auto-Sharding:** Automatic data distribution

**Geographic Distribution:** Multi-region clusters

**Limitations:** Complexity increases with sharding

## SQLite

**Single Machine:** Limited to one server

**File Size:** Tested up to 281TB (theoretical)

**Practical Limit:** ~1TB with good performance

**Limitations:** No built-in replication or clustering

# Cost Analysis

## Development/Testing

Backend	Setup Cost	Ongoing Cost	Complexity
PostgreSQL	Medium	\$0 (local)	Medium
MongoDB	Medium	\$0 (local)	Medium
SQLite	None	\$0	None

## Production (Small - <100 users)

Backend	Monthly Cost	Setup Time	Maintenance
PostgreSQL (RDS t3.small)	~\$30	1 hour	Low
MongoDB (Atlas M10)	~\$60	30 min	Very Low
SQLite (EC2 t3.micro)	~\$10	30 min	Medium

## Production (Medium - 100-1000 users)

Backend	Monthly Cost	Setup Time	Maintenance
PostgreSQL (RDS t3.medium)	~\$120	2 hours	Low
MongoDB (Atlas M30)	~\$250	1 hour	Very Low
SQLite	Not recommended	-	-

## Production (Large - 1000+ users)

Backend	Monthly Cost	Setup Time	Maintenance
PostgreSQL (RDS r5.xlarge)	~\$500+	4 hours	Medium
MongoDB (Atlas M60+)	~\$800+	2 hours	Low
SQLite	Not recommended	-	-

## Feature Support

### Data Integrity

Feature	PostgreSQL	MongoDB	SQLite
Foreign Keys	Enforced	Application-level	Enforced
Unique Constraints	Yes	Yes	Yes
Check Constraints	Yes	△ Limited	Yes
NOT NULL	Yes	△ Schema validation	Yes
Triggers	Yes	Yes	Yes

### Query Capabilities

Feature	PostgreSQL	MongoDB	SQLite
SQL Support	Full SQL	MQL only	Full SQL
Joins	All types	△ \$lookup	All types
Subqueries	Yes	△ Limited	Yes
Window Functions	Yes	△ Limited	Yes
CTEs	Yes	No	Yes
Aggregations	Yes	Pipeline	Yes

### Advanced Features

Feature	PostgreSQL	MongoDB	SQLite
---------	------------	---------	--------

Full-Text Search	tsvector	Text indexes	FTS5
Geospatial	PostGIS	Built-in	△ Extension
Multi-Tenancy	SQL Isolation	Collection Level	SQL Isolation
RBAC / TBAC	Built-in (Migs)	Supported	Built-in (Schema)
Audit Logging	Automated	Supported	Automated
Federated Catalogs	Supported	Supported	Supported
JSON Queries	JSONB operators	Native	JSON1

## Schema Management

Feature	PostgreSQL	MongoDB	SQLite
-----	-----	-----	-----
Auto-Initialization	Yes (sqlx migs)	△ Partial (coll only)	Yes (embedded)
Auto-Migrations	Yes (sqlx)	Manual	Manual (reset req)
Manual Indexing	Not required	Essential	Not required

## Decision Matrix

### Choose PostgreSQL If:

- ~You need proven, enterprise-grade reliability
- ~You want strong data integrity (foreign keys)
- ~You're familiar with SQL
- ~You need complex queries and joins
- ~You want managed cloud options (RDS)
- ~You're building a traditional web application
- ~You need excellent tooling and ecosystem

### Choose MongoDB If:

- ~You need horizontal scalability
- ~You prefer document-based data model
- ~You want flexible schema evolution
- ~You're building cloud-native applications
- ~You need multi-region deployments
- ~You're already using MongoDB
- ~You want excellent managed service (Atlas)

### Choose SQLite If:

-

- ~You're developing locally
- ~You need embedded database
- ~You're deploying to edge/IoT devices
- ~You want zero configuration
- ~You have low concurrent write needs
- ~You want minimal resource usage
- ~You're prototyping or testing

## Migration Paths

### SQLite → PostgreSQL

**Difficulty:** Medium  
**Tools:** Custom scripts, pgloader  
**Downtime:** Required  
**Use Case:** Moving from development to production

### SQLite → MongoDB

**Difficulty:** Hard  
**Tools:** Custom scripts  
**Downtime:** Required  
**Use Case:** Rare

### PostgreSQL ↔ MongoDB

**Difficulty:** Hard  
**Tools:** Custom ETL  
**Downtime:** Required  
**Use Case:** Architectural change

## Test Coverage

All backends have comprehensive test coverage:

Backend	Tests	Status
PostgreSQL	Shared Regression Suite + Integration	All passing
MongoDB	Shared Regression Suite + Integration	All passing
SQLite	Shared Regression Suite + Integration	All passing

## Recommendations by Use Case

### Startup/MVP

**Recommended:** SQLite → PostgreSQL

Start with SQLite for rapid development

Migrate to PostgreSQL when you have users

## Enterprise

**Recommended:** PostgreSQL

Proven reliability

Excellent tooling

Strong consistency

## Cloud-Native SaaS

**Recommended:** MongoDB

Horizontal scalability

Managed service (Atlas)

Multi-region support

## Edge/IoT

**Recommended:** SQLite

Embedded

Low resources

No network required

## Multi-Tenant SaaS

**Recommended:** PostgreSQL or MongoDB

Both provide excellent tenant isolation

PostgreSQL: Better for complex queries

MongoDB: Better for horizontal scaling

## Conclusion

All three backends are production-ready and fully tested. Your choice should be based on:

**Deployment Environment:** Cloud, on-prem, edge?

-

**Scale Requirements:** How many users/requests?

**Team Expertise:** SQL vs NoSQL familiarity?

**Budget:** Managed vs self-hosted costs?

**Data Model:** Relational vs document fit?

**Default Recommendation:** Start with **SQLite** for development, use **PostgreSQL** for production unless you specifically need MongoDB's horizontal scaling.

## Next Steps

[PostgreSQL Setup](#)

[MongoDB Setup](#)

[SQLite Setup](#)

[Warehouse Storage](#)

Source: [memory.md](#)

## In-Memory Store Backend

The in-memory store is a high-performance, zero-configuration backend for development, testing, and temporary deployments.

### Overview

The in-memory store uses ``DashMap`` (concurrent HashMap) to store all catalog metadata in RAM with:

**Zero configuration** - No database setup required

**Blazing fast** - All operations in memory

**Thread-safe** - Concurrent access via DashMap

**Ephemeral** - Data lost on restart

**Perfect for testing** - Instant setup and teardown

## Pros and Cons

### Advantages

**Zero Setup:** No database installation or configuration

**Maximum Performance:** All operations in RAM

**Instant Startup:** No connection delays

**Perfect for CI/CD:** Fast, isolated test runs

**No Dependencies:** Built into Pangolin

**Thread-Safe:** Concurrent access without locks

**Deterministic:** Fresh state on every restart

## ⚠ Considerations

**Catalog Metadata Loss on Restart:** Tenants, Warehouses, and Catalog configurations are lost.

**Improved Data Safety:** Iceberg tables' `metadata.json` and data files **are persisted** if a persistent Warehouse is configured.

**Memory Usage:** All catalog metadata (not data files) is stored in RAM.

**Single Instance:** No clustering or replication.

**Not for Production:** The catalog structure itself is ephemeral.

## Use Cases

### Best For:

Local development

Unit and integration testing

CI/CD pipelines

Demos and prototyping

Temporary environments

Learning and experimentation

### Not For:

Production deployments

Data that needs to persist

Multi-instance deployments

Large catalogs (>10GB metadata)

## Configuration

## Environment Variables

```
# No DATABASE_URL needed - in-memory is the default!  
# Just don't set DATABASE_URL and Pangolin uses in-memory store
```

```
# Optional: Set log level
RUST_LOG=info
```

## Explicit Configuration

If you want to be explicit, or if `DATABASE\_URL` is set globally and you want to override it to use in-memory:

```
# Set storage type explicitly
PANGOLIN_STORAGE_TYPE=memory
```

[!NOTE]

If `DATABASE\_URL` is completely omitted, Pangolin defaults to the In-Memory store.

## Usage

### Starting Pangolin with In-Memory Store

```
# Default - uses in-memory store
cd pangolin
cargo run --bin pangolin_api

# Or with explicit log level
RUST_LOG=debug cargo run --bin pangolin_api
```

## Docker

```
# Run without DATABASE_URL
docker run -p 8080:8080 pangolin/pangolin-api

# Or with environment file (omit DATABASE_URL)
docker run -p 8080:8080 --env-file .env pangolin/pangolin-api
```

## Testing

```
# Run tests with in-memory store (default)
cargo test

# Run specific test
cargo test test_catalog_crud

# Run with output
cargo test -- --nocapture
```

# Performance Characteristics

## Speed

**Create operations:** ~1-10 microseconds

**Read operations:** ~1-5 microseconds

**List operations:** ~10-100 microseconds (depends on size)

**Delete operations:** ~1-10 microseconds

## Memory Usage

Approximate memory per entity:

**Tenant:** ~500 bytes

**Warehouse:** ~1 KB

**Catalog:** ~1 KB

**Namespace:** ~500 bytes

**Asset:** ~2 KB (including properties)

**Example:** 1000 tables = ~2 MB of metadata

## Concurrency

**Thread-safe:** Multiple concurrent requests

**Lock-free reads:** DashMap allows concurrent reads

**Write performance:** Excellent for moderate concurrency

**No connection pool:** No connection overhead

## Development Workflow

### Quick Start

```
# 1. Start Pangolin (uses in-memory by default)
cargo run --bin pangolin_api

# 2. Create a tenant
curl -X POST http://localhost:8080/api/v1/tenants \
  -H "Content-Type: application/json" \
  -d '{"name": "dev-tenant"}'

# 3. Create a catalog
curl -X POST http://localhost:8080/api/v1/catalogs \
```

```
-H "X-Pangolin-Tenant: dev-tenant" \  
-H "Content-Type: application/json" \  
-d '{  
  "name": "dev-catalog",  
  "type": "local"  
'
```

```
# 4. Use with PyIceberg  
python your_script.py
```

```
# 5. Restart to reset - all data cleared!  
# Ctrl+C and restart  
cargo run --bin pangolin_api
```

## Testing Pattern

```
#[tokio::test]  
async fn test_my_feature() {  
  // In-memory store is created automatically in tests  
  let store = MemoryStore::new();  
  
  // Create test data  
  let tenant = Tenant {  
    id: Uuid::new_v4(),  
    name: "test".to_string(),  
    properties: HashMap::new(),  
  };  
  
  store.create_tenant(tenant.clone()).await.unwrap();  
  
  // Test your feature  
  let result = store.get_tenant(tenant.id).await.unwrap();  
  assert!(result.is_some());  
  
  // No cleanup needed - test isolation automatic  
}
```

## CI/CD Integration

```
# GitHub Actions example  
name: Test  
on: [push]  
jobs:  
  test:  
    runs-on: ubuntu-latest  
    steps:  
      - uses: actions/checkout@v2  
      - uses: actions-rs/toolchain@v1  
        with:  
          toolchain: stable
```

```
- name: Run tests
  run: cargo test
  # Uses in-memory store by default - no DB setup needed!
```

## Switching to Persistent Storage

When you're ready for production:

### To PostgreSQL

```
# 1. Set DATABASE_URL
export DATABASE_URL=postgresql://user:password@localhost:5432/pangolin

# 2. Restart Pangolin
cargo run --bin pangolin_api

# Data from in-memory store is NOT migrated
# You'll start with a fresh database
```

### To MongoDB

```
# 1. Set DATABASE_URL
export DATABASE_URL=mongodb://user:password@localhost:27017/pangolin

# 2. Restart Pangolin
cargo run --bin pangolin_api
```

### To SQLite

```
# 1. Set DATABASE_URL
export DATABASE_URL=sqlite:///./pangolin.db

# 2. Restart Pangolin
cargo run --bin pangolin_api
```

## Limitations

### Data Persistence

- ~No persistence across restarts
- ~No backup/restore
- ~No point-in-time recovery

~Data lost on crash

## Scalability

~Single instance only

~No replication

~No clustering

~Limited by RAM

## Production Features

~No audit trail persistence

~No disaster recovery

~No multi-region support

## Troubleshooting

### Out of Memory

Error: Cannot allocate memory

**Solution:** Switch to persistent backend or increase available RAM

### Data Lost After Restart

**This is expected behavior!** In-memory store is ephemeral.

**Solution:** Use PostgreSQL, MongoDB, or SQLite for persistence

### Slow Performance with Large Datasets

**Solution:** In-memory should be fast. If slow:

Check available RAM

Monitor memory usage

Consider persistent backend for large datasets

## Monitoring

### Memory Usage

```
# Check Pangolin process memory
ps aux | grep pangolin_api

# Or use top
top -p $(pgrep pangolin_api)
```

## Rust Logging

```
# Enable debug logging
RUST_LOG=debug cargo run --bin pangolin_api

# See memory-related logs
RUST_LOG=pangolin_store=debug cargo run --bin pangolin_api
```

## Best Practices

### Development

- ˘Use in-memory for local development
- ˘Restart to reset state during testing
- ˘Use scripts to seed test data
- ˘Keep test datasets small

### Testing

- ˘Use in-memory for unit tests
- ˘Use in-memory for integration tests
- ˘Each test gets isolated state
- ˘No cleanup needed between tests

### CI/CD

- ˘Use in-memory for fast CI runs
- ˘No database setup required
- ˘Parallel test execution safe
- ˘Deterministic test results

## Hybrid Persistence (Development Mode)

Starting with version 0.1.0, the In-Memory store supports a "Hybrid" mode when paired with a persistent Warehouse.

## How it works:

**Ephemeral Catalog:** Your tenants, warehouses, and catalog settings exist only in RAM.

**Persistent Iceberg Metadata:** When you create a table, Pangolin writes the Iceberg `metadata.json` directly to your configured object storage (S3/Azure/GCP).

**Resilience:** If you restart the API, you must recreate the Warehouse and Catalog with the same settings. Once recreated, you can "load" existing tables from storage because the `metadata.json` was never lost.

## Configuration for Hybrid Mode:

Ensure your `Warehouse` is configured with a real bucket/container:

```
# Create Warehouse with real S3 bucket
curl -X POST http://localhost:8080/api/v1/warehouses \
  -d '{
    "name": "prod-s3",
    "storage_config": { "s3.bucket": "my-persistent-bucket" },
    "vending_strategy": { ... }
  }'
```

## Comparison with Other Backends

Feature	In-Memory	SQLite	PostgreSQL	MongoDB
Setup Time	0 seconds	0 seconds	Minutes	Minutes
Persistence	Catalog: No / Data: Yes	Yes	Yes	Yes
Performance	Excellent	Excellent	Very Good	Very Good
Memory Usage	High	Low	Medium	Medium
Production Ready	No	Yes	Yes	Yes
Testing	Perfect	Good	⚠ Requires setup	⚠ Requires setup

## Migration Path

**Recommended Development Flow:**

**Start:** In-memory for rapid development

**Local Testing:** SQLite for persistent local testing

**Staging:** PostgreSQL or MongoDB

**Production:** PostgreSQL or MongoDB

## Additional Resources

[DashMap Documentation](#)

[Backend Storage Comparison](#)

[PostgreSQL Backend](#)

[MongoDB Backend](#)

[SQLite Backend](#)

## Next Steps

[PostgreSQL Backend](#) - For production deployments

[MongoDB Backend](#) - For cloud-native deployments

[SQLite Backend](#) - For embedded deployments

[Backend Comparison](#) - Choose the right backend

Source: [mongodb.md](#)

## MongoDB Backend Storage

MongoDB is a popular NoSQL document database that provides excellent scalability and flexibility for Pangolin's metadata storage.

### Overview

MongoDB stores Pangolin catalog metadata as JSON documents with:

Flexible schema evolution

Horizontal scalability through sharding

Built-in replication

Rich query capabilities

Cloud-native architecture

### Pros and Cons

#### Advantages

**Horizontal Scalability:** Shard across multiple servers

**Flexible Schema:** Easy to add new fields without migrations

**Cloud-Native:** Excellent managed options (MongoDB Atlas)

-

**JSON-Native:** Natural fit for metadata storage

**Aggregation Framework:** Powerful data processing

**Change Streams:** Real-time data monitoring

**Geographic Distribution:** Multi-region deployments

## ⚠ Considerations

**No Foreign Keys:** Application-level referential integrity

**Memory Usage:** Keeps working set in RAM

**Learning Curve:** Different from SQL databases

**Cost:** Atlas pricing can be higher than RDS

## Use Cases

### **Best For:**

Cloud-native deployments

Applications requiring horizontal scaling

Flexible schema requirements

Multi-region deployments

When you're already using MongoDB

### **Not Ideal For:**

When you need strict foreign key constraints

Traditional SQL-heavy applications

Embedded deployments

When you need complex joins

## Prerequisites

MongoDB 5.0+ installed

Database created for Pangolin

User with readWrite permissions

## Installation

### Local Development (Docker)

```
# Start MongoDB
docker run -d \
  --name pangolin-mongo \
  -e MONGO_INITDB_ROOT_USERNAME=admin \
  -e MONGO_INITDB_ROOT_PASSWORD=your_password \
  -e MONGO_INITDB_DATABASE=pangolin \
  -p 27017:27017 \
  mongo:7

# Create Pangolin user
docker exec -it pangolin-mongo mongosh -u admin -p your_password
--authenticationDatabase admin

use pangolin
db.createUser({
  user: "pangolin",
  pwd: "pangolin_password",
  roles: [{ role: "readWrite", db: "pangolin" }]
})
```

## Production (MongoDB Atlas)

Create cluster at [cloud.mongodb.com](https://cloud.mongodb.com)

Create database user

Whitelist IP addresses

Get connection string

## Configuration

## Environment Variables

```
# Connection string format:
# mongodb://username:password@host:port/database

# Local development
DATABASE_URL=mongodb://pangolin:pangolin_password@localhost:27017/pangolin

# MongoDB Atlas
DATABASE_URL=mongodb+srv://pangolin:password@cluster0.xxxxx.mongodb.net/pangolin?retryWrites=true&w=majority

# With replica set
DATABASE_URL=mongodb://pangolin:password@host1:27017,host2:27017,host3:27017/pangolin?replicaSet=rs0

# Additional settings
MONGO_DB_NAME=pangolin
MONGO_MAX_POOL_SIZE=10
```

## Schema Initialization

### [!IMPORTANT]

Unlike PostgreSQL and SQLite backends, Pangolin **does not** automatically create indexes for MongoDB collections. You must create them manually or via a separate script for optimal performance.

Pangolin uses the following collections:

### Core Collections:

- `tenants` - Tenant records
- `warehouses` - Storage configurations
- `catalogs` - Catalog metadata
- `namespaces` - Namespace hierarchies
- `assets` - Table and view metadata
- `branches` - Branching information
- `tags` - Snapshot tags
- `commits` - Commit history
- `service\_users` - API identities
- `system\_settings` - Global configuration
- `merge\_operations` - Merge tracking

`merge\_conflicts` - Conflict resolution data

```
db.tenants.createIndex({ "name": 1 }, { unique: true })
```

```
// Warehouses
```

```
db.warehouses.createIndex({ "tenant_id": 1, "name": 1 }, { unique: true })
```

```
// Catalogs
```

```
db.catalogs.createIndex({ "tenant_id": 1, "name": 1 }, { unique: true })
```

```
// Namespaces
```

```
db.namespaces.createIndex({ "tenant_id": 1, "catalog_name": 1, "namespace_path": 1 }, { unique: true })
```

```
// Assets (Isolated by Branch)
```

```
db.assets.createIndex({
  "tenant_id": 1,
  "catalog_name": 1,
  "branch": 1,
  "namespace": 1,
  "name": 1
```

```
}, { unique: true })
db.assets.createIndex({ "namespace": 1 })
```

```
## Performance Tuning

### Recommended Settings

```javascript
// Enable profiling for slow queries
db.setProfilingLevel(1, { slowms: 100 })

// Check current profile level
db.getProfilingStatus()

// View slow queries
db.system.profile.find().sort({ ts: -1 }).limit(10)
```

## Monitoring

```
// Check database stats
db.stats()

// Check collection stats
db.catalogs.stats()

// Current operations
db.currentOp()

// Server status
db.serverStatus()
```

## Backup and Restore

### MongoDB Atlas

Atlas provides:

Continuous backups

Point-in-time recovery

Cross-region backups

### Manual Backups

```
# Backup
mongodump --uri="mongodb://pangolin:password@localhost:27017/pangolin"
--out=/backup/pangolin
```

```
# Restore
mongorestore --uri="mongodb://pangolin:password@localhost:27017/pangolin"
/backup/pangolin/pangolin

# Backup specific collection
mongodump --uri="mongodb://..." --collection=catalogs --out=/backup

# Restore specific collection
mongorestore --uri="mongodb://..." --collection=catalogs /backup/pangolin/catalogs.bson
```

## High Availability

### Replica Sets

```
# Initialize replica set
mongosh --eval "rs.initiate({
  _id: 'rs0',
  members: [
    { _id: 0, host: 'mongo1:27017' },
    { _id: 1, host: 'mongo2:27017' },
    { _id: 2, host: 'mongo3:27017' }
  ]
})"

# Check replica set status
mongosh --eval "rs.status()"
```

### Sharding

For very large deployments:

```
# Enable sharding on database
sh.enableSharding("pangolin")

# Shard collection by tenant_id
sh.shardCollection("pangolin.catalogs", { "tenant_id": 1 })
```

## Security Best Practices

**Authentication:** Always enable authentication

**TLS/SSL:** Use encrypted connections

**Network Security:** Bind to specific IPs, use firewalls

**Least Privilege:** Grant minimal required permissions

**Audit Logging:** Enable MongoDB audit logs

**Encryption at Rest:** Enable for Atlas or self-managed

## Troubleshooting

### Connection Issues

```
# Test connection
mongosh "mongodb://pangolin:password@localhost:27017/pangolin"

# Check if MongoDB is running
sudo systemctl status mongod

# Check logs
sudo tail -f /var/log/mongodb/mongod.log
```

### Performance Issues

```
// Explain query
db.catalogs.find({ tenant_id: "..."}).explain("executionStats")

// Check index usage
db.catalogs.aggregate([
  { $indexStats: {} }
])

// Compact collection
db.runCommand({ compact: 'catalogs' })
```

## Test Results

**Shared Regression Suite (store\_tests.rs)** passing:

`test\_mongo\_store\_compliance` (Covers CRUD, File IO, Metadata Logic)

Fully compliant with `CatalogStore` trait interface.

Verified with live integration (`test\_mongo.py`) including `Signer` (credential vending) and S3 file storage.

## Additional Resources

[MongoDB Official Documentation](#)

[MongoDB Atlas](#)

[MongoDB University](#)

## Next Steps

[PostgreSQL Backend](#)

[SQLite Backend](#)

[Backend Comparison](#)

[Warehouse Storage](#)

Source: postgresql.md

## PostgreSQL Backend Storage

PostgreSQL is a powerful, open-source relational database that provides excellent performance, reliability, and feature set for Pangolin's metadata storage.

### Overview

PostgreSQL stores all Pangolin catalog metadata in a structured SQL schema with:

Strong ACID guarantees

Foreign key constraints for data integrity

Efficient indexing for fast queries

Built-in replication and backup tools

Mature ecosystem and tooling

### Pros and Cons

#### Advantages

**Battle-tested:** Decades of production use

**ACID Compliance:** Strong consistency guarantees

**Rich Query Capabilities:** Complex joins, aggregations, window functions

**Foreign Keys:** Automatic referential integrity

**Managed Options:** AWS RDS, Azure Database, Google Cloud SQL

**Excellent Tooling:** pgAdmin, DBeaver, psql, and many others

**JSON Support:** JSONB for flexible property storage

-

**Full-text Search:** Built-in search capabilities

**Extensions:** PostGIS, pg\_partman, and hundreds more

## ⚠ Considerations

**Setup Complexity:** Requires database server installation

**Resource Usage:** Higher memory footprint than SQLite

**Vertical Scaling:** Primarily scales up (bigger server)

**Cost:** Managed services have ongoing costs

## Use Cases

### **Best For:**

Production deployments

Multi-user environments

High-concurrency workloads

Enterprise deployments

When you need strong consistency

### **Not Ideal For:**

Embedded applications

Edge/IoT devices

Serverless with cold starts

When you need horizontal scaling (consider MongoDB)

## Prerequisites

PostgreSQL 12+ installed

Database created for Pangolin

User with full permissions on the database

## Installation

### Local Development (Docker)

```
# Start PostgreSQL
docker run -d \
  --name pangolin-postgres \
```

```
-e POSTGRES_DB=pangolin \  
-e POSTGRES_USER=pangolin \  
-e POSTGRES_PASSWORD=your_password \  
-p 5432:5432 \  
postgres:16
```

```
# Verify connection
```

```
psql -h localhost -U pangolin -d pangolin
```

## Production (AWS RDS)

```
# Create RDS instance via AWS Console or CLI  
aws rds create-db-instance \  
  --db-instance-identifier pangolin-prod \  
  --db-instance-class db.t3.medium \  
  --engine postgres \  
  --engine-version 16.1 \  
  --master-username pangolin \  
  --master-user-password your_secure_password \  
  --allocated-storage 100 \  
  --storage-type gp3 \  
  --vpc-security-group-ids sg-xxxxx \  
  --db-subnet-group-name your-subnet-group \  
  --backup-retention-period 7 \  
  --preferred-backup-window "03:00-04:00" \  
  --preferred-maintenance-window "mon:04:00-mon:05:00"
```

## Configuration

### Environment Variables

```
# Connection string format:  
# postgresql://username:password@host:port/database  
# postgres://username:password@host:port/database  
  
# Local development  
DATABASE_URL=postgres://pangolin:your_password@localhost:5432/pangolin  
  
# Production (RDS)  
DATABASE_URL=postgresql://pangolin:password@pangolin-prod.xxxxx.us-east-1.rds.amazonaws.com:5432/pangolin  
  
# With SSL (recommended for production)  
DATABASE_URL=postgresql://pangolin:password@host:5432/pangolin?sslmode=require  
  
# Connection pool settings (optional)  
DATABASE_MAX_CONNECTIONS=10  
DATABASE_MIN_CONNECTIONS=2  
DATABASE_CONNECT_TIMEOUT=30
```

# Schema Initialization

Pangolin automatically initializes and migrates the PostgreSQL schema on startup using ``sqlx``. Manual schema application is generally not required unless you are performing a custom installation or troubleshooting.

The comprehensive schema includes:

## **Core Infrastructure:**

- ``tenants`` - Multi-tenant isolation records
- ``warehouses`` - Storage configurations (S3, Azure, GCP)
- ``catalogs`` - Iceberg catalog definitions
- ``namespaces`` - Namespace hierarchies and properties
- ``assets`` - Table and view metadata pointers (Isolated by ``branch_name``)
- ``branches`` - Git-like branch definitions
- ``tags`` - Immutable commit pointers (snapshots)
- ``commits`` - Detailed operation history
- ``metadata_locations`` - Physical Iceberg metadata file path tracking

## **Governance & Security:**

- ``users`` - Root and Tenant user accounts
- ``service_users`` - Machine-to-machine API identities
- ``roles`` - RBAC role definitions
- ``user_roles`` - Role assignments to users
- ``permissions`` - Direct (TBAC/RBAC) permission grants
- ``access_requests`` - Data discovery access workflows
- ``audit_logs`` - Comprehensive tamper-evident trail
- ``merge_operations`` - Branch merge tracking
- ``merge_conflicts`` - Conflict details for merges
- ``revoked_tokens`` - Blacklisted JWT identifiers
- ``active_tokens`` - (Internal) Session tracking for active tokens

## **System & Maintenance:**

- ``system_settings`` - Tenant-specific configuration overrides
- ``federated_sync_stats`` - Status tracking for cross-tenant federation

## **Indexes:**

Primary keys on all tables

Foreign key indexes for joining tenant/catalog/namespace context

Composite indexes for optimized listing (e.g., `idx\_assets\_tenant\_catalog`)

GIN indexes for efficient JSONB and Array searches (PostgreSQL specific)

## Manual Schema Application

[!WARNING]

Manual application of `postgres\_schema.sql` may bypass migration tracking. It is recommended to let Pangolin handle initialization automatically.

If you must apply the schema manually:

```
# Download the current consolidated schema
curl https://raw.githubusercontent.com/AlexMercedCoder/Pangolin/main/pangolin/pangolin_store/
sql/postgres_schema.sql -O

# Apply to your database
psql -h localhost -U pangolin -d pangolin -f postgres_schema.sql
```

## Performance Tuning

### Recommended PostgreSQL Settings

```
-- For production workloads
ALTER SYSTEM SET shared_buffers = '4GB';
ALTER SYSTEM SET effective_cache_size = '12GB';
ALTER SYSTEM SET maintenance_work_mem = '1GB';
ALTER SYSTEM SET checkpoint_completion_target = 0.9;
ALTER SYSTEM SET wal_buffers = '16MB';
ALTER SYSTEM SET default_statistics_target = 100;
ALTER SYSTEM SET random_page_cost = 1.1; -- For SSD storage
ALTER SYSTEM SET effective_io_concurrency = 200; -- For SSD storage
ALTER SYSTEM SET work_mem = '64MB';
ALTER SYSTEM SET min_wal_size = '1GB';
ALTER SYSTEM SET max_wal_size = '4GB';

-- Reload configuration
SELECT pg_reload_conf();
```

## Monitoring

```
-- Check active connections
SELECT count(*) FROM pg_stat_activity WHERE datname = 'pangolin';
```

```
-- Check table sizes
SELECT
    schemaname,
    tablename,
    pg_size_pretty(pg_total_relation_size(schemaname||'.'||tablename)) AS size
FROM pg_tables
WHERE schemaname = 'public'
ORDER BY pg_total_relation_size(schemaname||'.'||tablename) DESC;

-- Check index usage
SELECT
    schemaname,
    tablename,
    indexname,
    idx_scan,
    idx_tup_read,
    idx_tup_fetch
FROM pg_stat_user_indexes
ORDER BY idx_scan DESC;
```

## Backup and Restore

### Automated Backups (RDS)

AWS RDS provides automated backups:

Daily snapshots

Point-in-time recovery

Cross-region replication

### Manual Backups

```
# Backup
pg_dump -h localhost -U pangolin -d pangolin -F c -f pangolin_backup.dump

# Restore
pg_restore -h localhost -U pangolin -d pangolin -c pangolin_backup.dump

# Backup to SQL (human-readable)
pg_dump -h localhost -U pangolin -d pangolin > pangolin_backup.sql

# Restore from SQL
psql -h localhost -U pangolin -d pangolin < pangolin_backup.sql
```

## High Availability

# Replication

```
-- On primary server
ALTER SYSTEM SET wal_level = 'replica';
ALTER SYSTEM SET max_wal_senders = 3;
ALTER SYSTEM SET wal_keep_size = '1GB';

-- Create replication user
CREATE ROLE replicator WITH REPLICATION LOGIN PASSWORD 'repl_password';

-- On replica server
# In recovery.conf or postgresql.auto.conf
primary_conninfo = 'host=primary_host port=5432 user=replicator password=repl_password'
```

## Connection Pooling

Use PgBouncer for connection pooling:

```
# pgbouncer.ini
[databases]
pangolin = host=localhost port=5432 dbname=pangolin

[pgbouncer]
listen_port = 6432
listen_addr = *
auth_type = md5
auth_file = /etc/pgbouncer/userlist.txt
pool_mode = transaction
max_client_conn = 1000
default_pool_size = 25
```

Then connect to PgBouncer instead:

```
DATABASE_URL=postgresql://pangolin:password@localhost:6432/pangolin
```

## Troubleshooting

### Connection Issues

```
# Test connection
psql -h localhost -U pangolin -d pangolin -c "SELECT version();"

# Check if PostgreSQL is running
sudo systemctl status postgresql

# Check logs
sudo tail -f /var/log/postgresql/postgresql-16-main.log
```

# Performance Issues

```
-- Find slow queries
SELECT
  pid,
  now() - query_start AS duration,
  query,
  state
FROM pg_stat_activity
WHERE state != 'idle'
  AND now() - query_start > interval '5 seconds'
ORDER BY duration DESC;

-- Analyze table statistics
ANALYZE VERBOSE;

-- Rebuild indexes
REINDEX DATABASE pangolin;
```

## Security Best Practices

**Use SSL/TLS:** Always use `sslmode=require` in production

**Strong Passwords:** Use long, random passwords

**Least Privilege:** Grant only necessary permissions

**Network Security:** Use VPC/security groups to restrict access

**Regular Updates:** Keep PostgreSQL updated

**Audit Logging:** Enable PostgreSQL audit logging

**Encryption at Rest:** Use encrypted storage (EBS encryption for RDS)

## Migration from Other Backends

### From SQLite

```
# Export from SQLite
sqlite3 pangolin.db .dump > sqlite_dump.sql

# Convert to PostgreSQL format (manual editing required)
# Then import:
psql -h localhost -U pangolin -d pangolin -f converted_dump.sql
```

### From MongoDB

Custom migration script required. Contact support or see migration guide.

# Additional Resources

[PostgreSQL Official Documentation](#)

[AWS RDS PostgreSQL](#)

[Azure Database for PostgreSQL](#)

[Google Cloud SQL for PostgreSQL](#)

[PostgreSQL Performance Tuning](#)

## Next Steps

[MongoDB Backend](#)

[SQLite Backend](#)

[Backend Comparison](#)

[Warehouse Storage](#)

Source: [sqlite.md](#)

## SQLite Backend Storage

SQLite is a lightweight, embedded SQL database that provides zero-configuration storage for Pangolin's metadata.

### Overview

SQLite stores all Pangolin catalog metadata in a single file with:

Zero configuration required

ACID compliance

Full SQL support

Extremely low resource usage

Perfect for development and embedded deployments

## Pros and Cons

### Advantages

**Zero Configuration:** No separate database server needed

**Single File:** Entire database in one file

-

**Lightweight:** Minimal memory and CPU usage

**Fast:** Excellent performance for read-heavy workloads

**Portable:** Copy the file to move the database

**Embedded:** Perfect for edge devices and embedded systems

**ACID Compliant:** Full transaction support

**Foreign Keys:** Referential integrity support

**No Network:** No network latency

## ⚠ Considerations

**Single Writer:** Limited concurrent write performance

**No Replication:** Manual backup/restore required

**File-Based:** Not suitable for distributed systems

**Scalability:** Limited to single machine

**No User Management:** File-level permissions only

## Use Cases

### **Best For:**

Local development

Testing and CI/CD

Embedded applications

Edge/IoT deployments

Single-user applications

Prototyping

### **Not Ideal For:**

High-concurrency write workloads

Distributed systems

Multi-server deployments

When you need replication

Large-scale production (use PostgreSQL/MongoDB)

## Prerequisites

None! SQLite is embedded in Pangolin

## Installation

No installation required - SQLite is built into Pangolin.

## Configuration

### Environment Variables

```
# File-based database
DATABASE_URL=sqlite:///path/to/pangolin.db

# Relative path
DATABASE_URL=sqlite://./pangolin.db

# In-memory database (testing only, data lost on restart)
DATABASE_URL=sqlite::memory:

# Absolute path
DATABASE_URL=sqlite:///home/user/pangolin/data/pangolin.db
```

## Schema Initialization

Pangolin automatically initializes the SQLite schema on first startup using the bundled `sqlite\_schema.sql`.

The comprehensive schema includes:

#### **Core Infrastructure:**

- `tenants` - Multi-tenant isolation records
- `warehouses` - Storage configurations (S3, Azure, GCP)
- `catalogs` - Iceberg catalog definitions
- `namespaces` - Namespace hierarchies and properties
- `assets` - Table and view metadata pointers
- `branches` - Git-like branch definitions
- `tags` - Immutable commit pointers (snapshots)
- `commits` - Detailed operation history
- `metadata\_locations` - Physical Iceberg metadata file path tracking

#### **Governance & Security:**

- `users` - Root and Tenant user accounts

`roles` - RBAC role definitions

`user\_roles` - Role assignments to users

`permissions` - Direct (TBAC/RBAC) permission grants

`access\_requests` - Data discovery access workflows

`audit\_logs` - Comprehensive tamper-evident trail

`active\_tokens` - Session tracking for active tokens

### **System & Maintenance:**

`business\_metadata` - Rich discovery metadata for assets

`system\_settings` - Tenant-specific configuration overrides

`federated\_sync\_stats` - Status tracking for cross-tenant federation

### **Indexes:**

Pragma-enforced Foreign Keys

Primary keys on all tables

Composite indexes for branch isolation (e.g., `UNIQUE(tenant\_id, catalog\_name, branch\_name, namespace\_path, name)`)

Composite indexes for common listing operations

## Manual Schema Application

```
# Download schema
curl https://raw.githubusercontent.com/AlexMercedCoder/Pangolin/main/pangolin/pangolin_store/sql/sqlite_schema.sql -O

# Apply schema
sqlite3 pangolin.db < sqlite_schema.sql
```

## Performance Tuning

### Recommended PRAGMA Settings

```
-- Enable foreign keys (required)
PRAGMA foreign_keys = ON;

-- Performance optimizations
PRAGMA journal_mode = WAL;           -- Write-Ahead Logging for better concurrency
PRAGMA synchronous = NORMAL;         -- Balance between safety and speed
PRAGMA cache_size = -64000;          -- 64MB cache
PRAGMA temp_store = MEMORY;          -- Use memory for temp tables
```

```
PRAGMA mmap_size = 30000000000;      -- 30GB memory-mapped I/O
PRAGMA page_size = 4096;              -- 4KB pages (default)
```

## Monitoring

```
-- Database size
SELECT page_count * page_size as size FROM pragma_page_count(), pragma_page_size();

-- Table sizes
SELECT
    name,
    SUM(pgsizes) as size
FROM dbstat
GROUP BY name
ORDER BY size DESC;

-- Index usage
EXPLAIN QUERY PLAN SELECT * FROM catalogs WHERE tenant_id = '...';
```

## Backup and Restore

### File-Based Backup

```
# Simple file copy (database must be idle)
cp pangolin.db pangolin_backup.db

# Using SQLite backup command (safe while database is in use)
sqlite3 pangolin.db ".backup pangolin_backup.db"

# Dump to SQL
sqlite3 pangolin.db .dump > pangolin_backup.sql

# Restore from SQL
sqlite3 pangolin_new.db < pangolin_backup.sql
```

## Automated Backups

```
#!/bin/bash
# backup_sqlite.sh

BACKUP_DIR="/backups/pangolin"
TIMESTAMP=$(date +%Y%m%d_%H%M%S)
DB_FILE="/path/to/pangolin.db"

mkdir -p $BACKUP_DIR
sqlite3 $DB_FILE ".backup $BACKUP_DIR/pangolin_${TIMESTAMP}.db"
```

```
# Keep only last 7 days
find $BACKUP_DIR -name "pangolin_*.db" -mtime +7 -delete
```

## High Availability

SQLite does not support built-in replication. For high availability:

**Use PostgreSQL or MongoDB** for production

**File-level replication:** Use tools like Litestream

**Regular backups:** Automated backup scripts

## Litestream (Continuous Replication)

```
# Install Litestream
curl -s https://api.github.com/repos/benbjohnson/litestream/releases/latest \
  | grep "browser_download_url.*linux-amd64.tar.gz" \
  | cut -d : -f 2,3 \
  | tr -d \" \
  | wget -qi -

# Configure litestream.yml
replicas:
  - url: s3://my-bucket/pangolin.db

# Run Litestream
litestream replicate pangolin.db
```

## Security Best Practices

**File Permissions:** Restrict access to database file

```
chmod 600 pangolin.db
chown pangolin:pangolin pangolin.db
```

**Encryption at Rest:** Use filesystem encryption (LUKS, dm-crypt)

**Backup Encryption:** Encrypt backup files

```
sqlite3 pangolin.db .dump | gpg -c > pangolin_backup.sql.gpg
```

**Network Security:** SQLite has no network access (inherently secure)

## Migration

### To PostgreSQL

```
# Export from SQLite
sqlite3 pangolin.db .dump > sqlite_dump.sql

# Convert to PostgreSQL (requires manual editing)
# Then import to PostgreSQL
psql -U pangolin -d pangolin -f converted_dump.sql
```

## To MongoDB

Custom migration script required.

## Troubleshooting

### Database Locked

```
# Check for locks
lsof | grep pangolin.db

# Force unlock (use with caution)
fuser -k pangolin.db
```

## Corruption

```
# Check integrity
sqlite3 pangolin.db "PRAGMA integrity_check;"

# Recover from corruption
sqlite3 pangolin.db ".recover" | sqlite3 pangolin_recovered.db
```

## Performance Issues

```
-- Analyze database
ANALYZE;

-- Vacuum to reclaim space
VACUUM;

-- Rebuild indexes
REINDEX;
```

## Test Results

**All 6 SQLite tests passing:**

```
`test_sqlite_tenant_crud`  
`test_sqlite_warehouse_crud`  
`test_sqlite_catalog_crud`  
`test_sqlite_namespace_operations`  
`test_sqlite_asset_operations`  
`test_sqlite_multi_tenant_isolation`
```

## Development Workflow

```
# Start Pangolin with SQLite  
DATABASE_URL=sqlite:///./dev.db cargo run --bin pangolin_api  
  
# Inspect database  
sqlite3 dev.db  
sqlite> .tables  
sqlite> .schema catalogs  
sqlite> SELECT * FROM tenants;  
  
# Reset database  
rm dev.db  
# Restart Pangolin to recreate schema
```

## Additional Resources

[SQLite Official Documentation](#)

[SQLite Performance Tuning](#)

[Litestream \(Replication\)](#)

[SQLite Best Practices](#)

## Next Steps

[PostgreSQL Backend](#)

[MongoDB Backend](#)

[Backend Comparison](#)

[Warehouse Storage](#)

Source: README.md

## Best Practices

Comprehensive guides for deploying, operating, and optimizing Pangolin in production.

## Guides

### Operations

Deployment - Production deployment strategies, Docker, Kubernetes, HA setup

Scalability - Scaling API servers, databases, multi-tenancy, performance optimization

Security - Authentication, encryption, audit logging, compliance

### Data Management

Permissions Management - RBAC best practices, access control patterns

Branch Management - Git-like workflows, merge strategies, conflict resolution

Business Metadata - Metadata strategy, governance, data classification

### Technical

Apache Iceberg - Table design, partitioning, schema evolution, performance tuning

Generic Assets - Managing ML models, files, media, and other artifacts

## Quick Reference

### Deployment Checklist

- ☐ Infrastructure provisioned
- ☐ Secrets configured in secrets manager
- ☐ Database initialized
- ☐ Monitoring and alerting configured
- ☐ Backup procedures tested
- ☐ Load testing completed

### Security Checklist

- ☐ HTTPS/TLS enabled
- ☐ Strong JWT secrets
- ☐ Database encryption enabled
- ☐ Firewall rules configured

☐ Audit logging enabled

☐ Regular security patches

## Performance Checklist

☐ Appropriate partitioning strategy

☐ Connection pooling configured

☐ Caching enabled

☐ Regular compaction scheduled

☐ Metrics monitored

## Getting Help

**Documentation:** [Full Documentation](#)

**Features:** [Features Guide](#)

**CLI:** [CLI Documentation](#)

Source: branching.md

## Branch Management Best Practices

### Overview

This guide covers best practices for using Pangolin's Git-like branching features for safe data experimentation and development.

### Branching Strategy

#### Branch Types

##### **Main/Production Branch**

Default branch for production data

Protected from direct writes

All changes via merge from development branches

##### **Development Branches**

Short-lived feature branches

Experimentation and testing

Merged back to main after validation

## Long-Running Branches

Staging/QA environments

Regional variants

Historical snapshots

## Naming Conventions

```
# Feature branches
feature/customer-segmentation
feature/new-metrics-2025

# Environment branches
staging
qa
production

# User branches
dev/alice/experiment
dev/bob/analysis

# Release branches
release/2025-q4
release/v2.0
```

## Branch Lifecycle

### 1. Create Branch

```
# Create from main
pangolin-user create-branch \
  --catalog analytics \
  --name dev/alice/customer-analysis \
  --from main

# Create from specific snapshot
pangolin-user create-branch \
  --catalog analytics \
  --name experiment/pricing-model \
  --from main \
  --snapshot-id abc123
```

## Best Practices

Always branch from known good state

Use descriptive names

Document branch purpose

## 2. Development

```
# Work on branch
from pyiceberg.catalog import load_catalog

catalog = load_catalog("pangolin", ...)

# Specify branch in table reference
table = catalog.load_table("analytics.sales@dev/alice/customer-analysis")

# Make changes
table.append(new_data)
table.update(...)
```

### Isolation Benefits

Experiment without affecting production

Test schema changes safely

Validate data transformations

## 3. Testing & Validation

```
# Compare branch to main
main_table = catalog.load_table("analytics.sales@main")
branch_table = catalog.load_table("analytics.sales@dev/alice/customer-analysis")

# Validate row counts
assert branch_table.scan().to_arrow().num_rows > main_table.scan().to_arrow().num_rows

# Validate schema compatibility
assert branch_table.schema() == main_table.schema()

# Run data quality checks
run_data_quality_tests(branch_table)
```

## 4. Merge

```
# Merge back to main
pangolin-user merge-branch \
  --catalog analytics \
  --source dev/alice/customer-analysis \
  --target main \
  --strategy fast-forward
```

```
# Or create merge commit
pangolin-user merge-branch \
  --catalog analytics \
  --source dev/alice/customer-analysis \
  --target main \
  --strategy merge-commit \
  --message "Add customer segmentation analysis"
```

## 5. Cleanup

```
# Delete merged branch
pangolin-admin delete-branch \
  --catalog analytics \
  --name dev/alice/customer-analysis

# Or keep for historical reference
pangolin-user create-tag \
  --catalog analytics \
  --name customer-analysis-v1 \
  --branch dev/alice/customer-analysis
```

## Merge Strategies

### Fast-Forward Merge

#### When to Use

Linear history desired

No conflicts expected

Branch is ahead of target

```
# Fast-forward merge
pangolin-user merge-branch \
  --source feature/new-metrics \
  --target main \
  --strategy fast-forward
```

#### Result

Before:

```
main:    A---B
          \
feature:  C---D
```

After:

```
main:    A---B---C---D
```

# 3-Way Merge

## When to Use

Parallel development

Both branches have changes

Need to preserve history

```
# 3-way merge
pangolin-user merge-branch \
  --source feature/schema-update \
  --target main \
  --strategy merge-commit
```

## Result

```
Before:
main:    A---B---E
          \
feature:  C---D

After:
main:    A---B---E---M
          \   /
feature:  C---D
```

# Conflict Resolution

## Detecting Conflicts

```
# Attempt merge
pangolin-user merge-branch \
  --source feature/update \
  --target main

# If conflicts detected
Error: Merge conflicts detected
- Table: sales.orders
- Conflict: Schema mismatch
- Base: schema v1
- Source: schema v2
- Target: schema v1.1
```

# Resolution Strategies

## 1. Manual Resolution

```
# List conflicts
pangolin-admin list-conflicts --merge-id <uuid>

# Resolve each conflict
pangolin-admin resolve-conflict \
  --merge-id <uuid> \
  --conflict-id <uuid> \
  --resolution use-source # or use-target, or manual

# Complete merge
pangolin-admin complete-merge --merge-id <uuid>
```

## 2. Abort and Retry

```
# Abort merge
pangolin-admin abort-merge --merge-id <uuid>

# Rebase branch on target
# Resolve conflicts in branch
# Retry merge
```

# Common Patterns

## Feature Development

```
# 1. Create feature branch
pangolin-user create-branch --name feature/add-customer-ltv --from main

# 2. Develop and test
# ... make changes ...

# 3. Validate
run_tests()

# 4. Merge to main
pangolin-user merge-branch --source feature/add-customer-ltv --target main

# 5. Clean up
pangolin-admin delete-branch --name feature/add-customer-ltv
```

## Environment Promotion

```
# Dev → Staging → Production

# 1. Develop in dev branch
pangolin-user create-branch --name dev --from main

# 2. Promote to staging
```

```
pangolin-user merge-branch --source dev --target staging

# 3. Test in staging
run_integration_tests()

# 4. Promote to production
pangolin-user merge-branch --source staging --target main
```

## A/B Testing

```
# Create variant branches
pangolin-user create-branch --name variant-a --from main
pangolin-user create-branch --name variant-b --from main

# Apply different transformations
apply_transformation_a(variant-a)
apply_transformation_b(variant-b)

# Run experiments
results_a = run_experiment(variant-a)
results_b = run_experiment(variant-b)

# Merge winning variant
if results_a > results_b:
    pangolin-user merge-branch --source variant-a --target main
else:
    pangolin-user merge-branch --source variant-b --target main
```

## Time Travel & Rollback

```
# Create branch from historical snapshot
pangolin-user create-branch \
    --name rollback/before-incident \
    --from main \
    --snapshot-id <snapshot-before-issue>

# Validate data is correct
validate_data(rollback/before-incident)

# Merge to restore
pangolin-user merge-branch \
    --source rollback/before-incident \
    --target main \
    --message "Rollback to state before data quality issue"
```

## Branch Protection

# Protected Branches

## Main/Production

```
# Branch protection rules
branches:
  main:
    protected: true
    require_merge: true
    allowed_mergers:
      - tenant-admin
      - data-engineer
    require_tests: true
    require_approval: true
```

## Enforcement

```
# Prevent direct writes to main
table = catalog.load_table("analytics.sales@main")
table.append(data) # Error: Direct writes to protected branch not allowed

# Require merge workflow
Create branch → Make changes → Merge to main
```

# Performance Considerations

## Branch Overhead

### Metadata Only

Branches are lightweight (metadata pointers)

No data duplication until writes

Fast branch creation

### Copy-on-Write

```
main:    [snapshot-1] → [snapshot-2]
        ↓
branch:  [snapshot-1] → [snapshot-3] (only new data)
```

# Cleanup Strategy

## Delete Stale Branches

```
# Find branches older than 30 days with no activity
pangolin-admin list-branches --catalog analytics --inactive-days 30

# Delete after review
```

```
pangolin-admin delete-branch --name old-feature-branch
```

## Archive Instead of Delete

```
# Tag before deleting
pangolin-user create-tag --name archive/old-feature --branch old-feature-branch
pangolin-admin delete-branch --name old-feature-branch
```

# Monitoring

## Branch Metrics

```
-- Track branch count
SELECT catalog, COUNT(*) as branch_count
FROM branches
GROUP BY catalog;

-- Find long-running branches
SELECT name, created_at, DATEDIFF(NOW(), created_at) as age_days
FROM branches
WHERE age_days > 30
ORDER BY age_days DESC;
```

## Merge Activity

```
-- Merge frequency
SELECT
    DATE_TRUNC('week', timestamp) as week,
    COUNT(*) as merge_count
FROM audit_logs
WHERE action = 'MERGE_BRANCH'
GROUP BY week
ORDER BY week DESC;
```

# Troubleshooting

## Merge Conflicts

### Common Causes

Schema evolution in both branches

Concurrent data modifications

Incompatible transformations

## **Resolution**

Understand both changes

Choose appropriate resolution strategy

Test merged result

Document decision

## **Branch Divergence**

### **Symptoms**

Large number of conflicts

Merge takes long time

Unexpected results

### **Prevention**

Merge main into branch regularly

Keep branches short-lived

Communicate schema changes

## **Checklist**

### **Before Creating Branch**

- ☐ Clear purpose defined
- ☐ Appropriate base branch selected
- ☐ Naming convention followed
- ☐ Team notified if needed

### **During Development**

- ☐ Regular commits/snapshots
- ☐ Tests passing
- ☐ Documentation updated
- ☐ Sync with main periodically

### **Before Merging**

- ☐ All tests passing

- ☐ Data quality validated
- ☐ Schema compatibility verified
- ☐ Conflicts resolved
- ☐ Approval obtained (if required)

## After Merging

- ☐ Branch deleted or archived
- ☐ Tag created if needed
- ☐ Team notified
- ☐ Documentation updated

## Additional Resources

[Branch Management Guide](#)

[Merge Operations](#)

[Conflict Resolution](#)

[Git Operations \(PyPangolin\)](#)

Source: deployment.md

## Deployment Best Practices

### Overview

This guide covers best practices for deploying Pangolin in production environments, from initial setup to ongoing operations.

### Pre-Deployment Planning

### Infrastructure Assessment

#### Compute Requirements

**API Server:** 2-4 CPU cores, 4-8GB RAM minimum for production

**Metadata Store:** Size based on catalog scale (see Backend Storage section)

**Network:** Low-latency connection to object storage (S3/Azure/GCS)

#### Storage Requirements

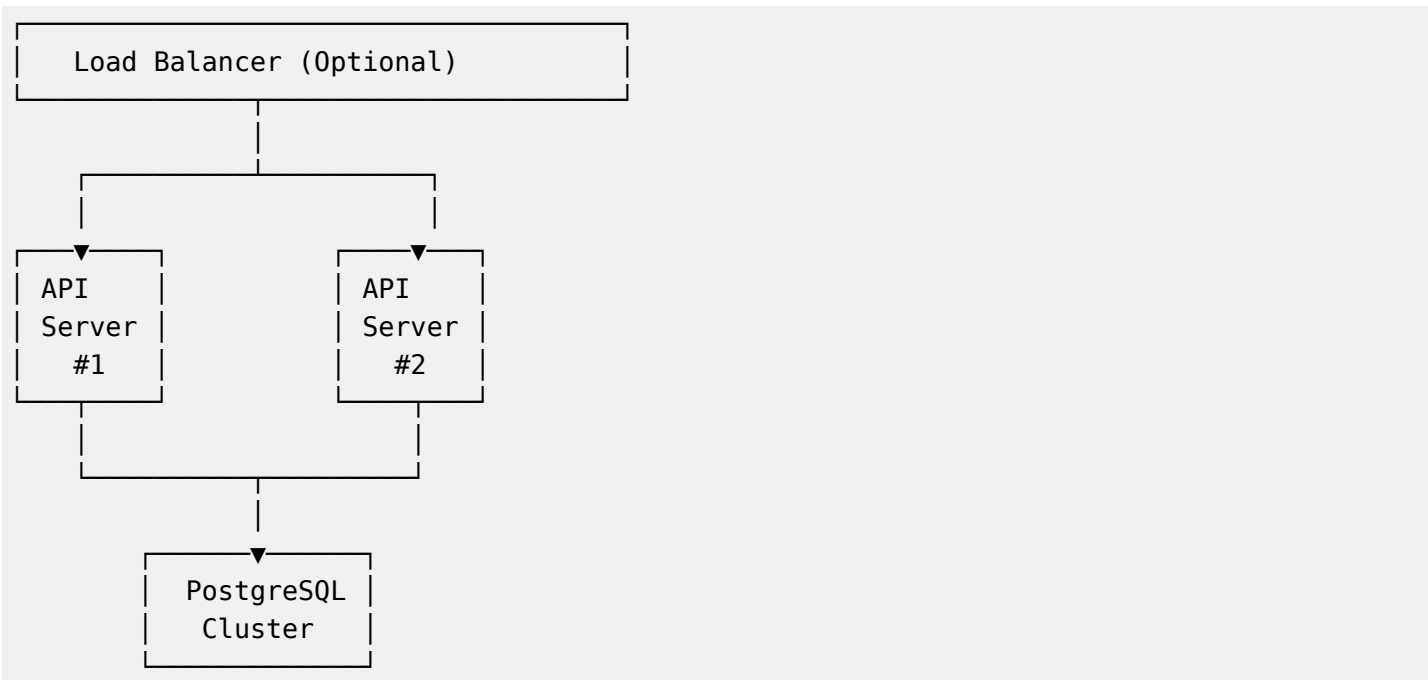
**Metadata Backend:** PostgreSQL (recommended) or MongoDB for production

**Object Storage:** S3, Azure Blob, or GCS for table data

**Backup Storage:** Separate location for metadata backups

## Deployment Architecture

### Single-Region Deployment



### Multi-Region Deployment

Deploy API servers in each region

Use read replicas for metadata store

Configure region-specific object storage

Implement cross-region replication for critical catalogs

## Environment Configuration

### Production Environment Variables

```
# Server Configuration
PANGOLIN_STORE_TYPE=postgresql
PANGOLIN_AUTH_MODE=auth
PANGOLIN_JWT_SECRET=<strong-random-secret> # Use secrets manager
PANGOLIN_JWT_EXPIRATION_HOURS=24

# Database Connection
DATABASE_URL=postgresql://user:pass@host:5432/pangolin
DATABASE_MAX_CONNECTIONS=20
```

```
DATABASE_IDLE_TIMEOUT=300
```

```
# Storage Configuration
```

```
AWS_REGION=us-east-1
```

```
AZURE_STORAGE_ACCOUNT=<account>
```

```
GCS_PROJECT_ID=<project>
```

```
# Logging
```

```
RUST_LOG=info,pangolin_api=debug
```

```
LOG_FORMAT=json # For structured logging
```

```
# Performance
```

```
WORKER_THREADS=4
```

```
MAX_REQUEST_SIZE=10485760 # 10MB
```

## Secrets Management

**DO NOT** store secrets in environment files or version control.

**Recommended Approaches:**

**AWS Secrets Manager** - For AWS deployments

**Azure Key Vault** - For Azure deployments

**HashiCorp Vault** - Platform-agnostic

**Kubernetes Secrets** - For K8s deployments

```
# Example: Fetch from AWS Secrets Manager
```

```
export PANGOLIN_JWT_SECRET=$(aws secretsmanager get-secret-value \
```

```
--secret-id pangolin/jwt-secret \
```

```
--query SecretString --output text)
```

## Container Deployment

### Docker Best Practices

#### Production Dockerfile

```
FROM rust:1.92 as builder
```

```
WORKDIR /app
```

```
COPY . .
```

```
RUN cargo build --release --bin pangolin_api
```

```
FROM debian:bookworm-slim
```

```
RUN apt-get update && apt-get install -y \
```

```
ca-certificates \
```

```
libssl3 \
```

```
&& rm -rf /var/lib/apt/lists/*
```

```
COPY --from=builder /app/target/release/pangolin_api /usr/local/bin/  
EXPOSE 8080  
CMD ["pangolin_api"]
```

## Docker Compose for Production

```
version: '3.8'  
  
services:  
  api:  
    image: your-registry/pangolin-api:latest  
    restart: unless-stopped  
    ports:  
      - "8080:8080"  
    environment:  
      - PANGOLIN_STORE_TYPE=postgresql  
      - DATABASE_URL=postgresql://postgres:5432/pangolin  
    depends_on:  
      - postgres  
    healthcheck:  
      test: ["CMD", "curl", "-f", "http://localhost:8080/health"]  
      interval: 30s  
      timeout: 10s  
      retries: 3  
  
  postgres:  
    image: postgres:16-alpine  
    restart: unless-stopped  
    volumes:  
      - pgdata:/var/lib/postgresql/data  
    environment:  
      - POSTGRES_PASSWORD=${DB_PASSWORD}  
      - POSTGRES_DB=pangolin  
  
volumes:  
  pgdata:
```

## Kubernetes Deployment

### Deployment Manifest

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: pangolin-api  
spec:  
  replicas: 3  
  selector:  
    matchLabels:  
      app: pangolin-api
```

```

template:
  metadata:
    labels:
      app: pangolin-api
  spec:
    containers:
      - name: api
        image: your-registry/pangolin-api:v0.2.0
        ports:
          - containerPort: 8080
        env:
          - name: PANGOLIN_JWT_SECRET
            valueFrom:
              secretKeyRef:
                name: pangolin-secrets
                key: jwt-secret
        resources:
          requests:
            memory: "2Gi"
            cpu: "1000m"
          limits:
            memory: "4Gi"
            cpu: "2000m"
        livenessProbe:
          httpGet:
            path: /health
            port: 8080
            initialDelaySeconds: 30
            periodSeconds: 10
        readinessProbe:
          httpGet:
            path: /health
            port: 8080
            initialDelaySeconds: 5
            periodSeconds: 5

```

## High Availability

### API Server HA

#### Load Balancing

Use multiple API server instances (minimum 2)

Configure health checks on `/health` endpoint

Implement session affinity if needed (stateless preferred)

#### Graceful Shutdown

```

# Handle SIGTERM for graceful shutdown
# Drain existing connections before terminating

```

# Database HA

## PostgreSQL

Use managed service (RDS, Cloud SQL, Azure Database)

Configure automatic failover

Enable point-in-time recovery

Set up read replicas for read-heavy workloads

## MongoDB

Deploy as replica set (minimum 3 nodes)

Use managed service (Atlas, DocumentDB)

Configure automatic failover

# Monitoring & Observability

## Health Checks

```
# API Health
curl http://localhost:8080/health

# Database Connection
curl http://localhost:8080/health/db
```

## Metrics to Monitor

### Application Metrics

Request rate and latency (p50, p95, p99)

Error rate (4xx, 5xx responses)

Active connections

JWT token generation/validation rate

### Database Metrics

Connection pool utilization

Query latency

Deadlocks and slow queries

Replication lag (if using replicas)

### System Metrics

CPU and memory utilization

Disk I/O

Network throughput

## Logging

### Structured Logging

```
{  
  "timestamp": "2025-12-24T20:00:00Z",  
  "level": "INFO",  
  "message": "Table created",  
  "tenant_id": "uuid",  
  "user_id": "uuid",  
  "catalog": "production",  
  "table": "sales.orders"  
}
```

### Log Aggregation

Use centralized logging (ELK, Splunk, CloudWatch)

Retain logs for compliance (30-90 days minimum)

Set up alerts for error patterns

## Backup & Disaster Recovery

### Metadata Backup

#### PostgreSQL

```
# Daily automated backups  
pg_dump -h localhost -U postgres pangolin > backup_$(date +%Y%m%d).sql  
  
# Point-in-time recovery enabled  
# Retain backups for 30 days minimum
```

#### MongoDB

```
# Snapshot backups  
mongodump --uri="mongodb://localhost:27017/pangolin" --out=/backup/$(date +%Y%m%d)
```

## Recovery Testing

Test restore procedures quarterly

Document recovery time objective (RTO) and recovery point objective (RPO)

Maintain runbooks for common failure scenarios

## Security Hardening

### Network Security

Deploy API behind firewall/security groups

Use TLS/HTTPS for all connections

Restrict database access to API servers only

Implement rate limiting

### Authentication

Use strong JWT secrets (256-bit minimum)

Rotate secrets regularly (quarterly)

Implement token expiration (24 hours recommended)

Enable multi-factor authentication for admin users

## Performance Optimization

### Database Optimization

#### Indexing

```
-- Ensure proper indexes exist
CREATE INDEX idx_assets_catalog ON assets(catalog_id);
CREATE INDEX idx_assets_namespace ON assets(namespace);
CREATE INDEX idx_audit_timestamp ON audit_logs(timestamp);
```

#### Connection Pooling

Configure appropriate pool size (10-20 connections per API instance)

Set idle timeout (5 minutes)

Monitor pool exhaustion

### Caching

Implement caching for frequently accessed metadata

Use Redis for distributed caching

Set appropriate TTLs based on data volatility

## Upgrade Strategy

### Rolling Updates

**Test in Staging** - Always test upgrades in non-production first

**Backup** - Take full metadata backup before upgrade

**Rolling Deployment** - Update one instance at a time

**Verify** - Check health and metrics after each instance

**Rollback Plan** - Keep previous version ready for quick rollback

### Database Migrations

```
# Run migrations before deploying new API version
./pangolin-migrate up

# Verify migration success
./pangolin-migrate status
```

## Compliance & Audit

### Audit Logging

Enable comprehensive audit logging

Store audit logs separately from application logs

Retain for compliance period (typically 1-7 years)

Implement log integrity verification

### Access Control

Follow principle of least privilege

Regular access reviews (quarterly)

Automated user deprovisioning

Monitor privileged operations

## Cost Optimization

# Resource Right-Sizing

Monitor actual resource usage

Scale down during off-peak hours

Use spot/preemptible instances for non-critical workloads

# Storage Optimization

Implement lifecycle policies for object storage

Archive old metadata to cheaper storage tiers

Clean up orphaned files regularly

# Troubleshooting

## Common Issues

### High Latency

Check database query performance

Review connection pool settings

Verify network latency to object storage

### Connection Errors

Verify database connectivity

Check connection pool exhaustion

Review firewall/security group rules

### Authentication Failures

Verify JWT secret configuration

Check token expiration settings

Review user permissions

# Checklist

## Pre-Deployment

[ ] Infrastructure provisioned and tested

[ ] Secrets configured in secrets manager

[ ] Database initialized and migrated

☐ Monitoring and alerting configured

☐ Backup procedures tested

☐ Load testing completed

## Post-Deployment

☐ Health checks passing

☐ Metrics being collected

☐ Logs aggregating correctly

☐ Backup jobs running

☐ Documentation updated

☐ Team trained on operations

## Additional Resources

[Scalability Best Practices](#)

[Security Best Practices](#)

[Monitoring Guide](#)

Source: generic-assets.md

## Generic Assets Best Practices

### Overview

This guide covers best practices for managing non-table assets in Pangolin, including ML models, files, videos, and other data artifacts.

### Asset Types

### Supported Generic Assets

#### **Data Files**

Parquet, CSV, JSON

Lance (vector database format)

Vortex (columnar format)

#### **ML Artifacts**

Trained models (PyTorch, TensorFlow, scikit-learn)

Model metadata and metrics

Training datasets

Feature stores

## **Media Files**

Images (JPEG, PNG, TIFF)

Videos (MP4, AVI, MOV)

Audio files

## **Other**

Directories/datasets

Database connection strings

Custom file types

# Registration

## File Assets

### **CSV Files**

```
from pyangolin import PangolinClient
from pyangolin.assets import CsvAsset

client = PangolinClient("http://localhost:8080")
client.login("admin", "password")

# Register and write CSV
csv_asset = CsvAsset.write(
    client,
    catalog="data_files",
    namespace="exports",
    name="customer_export_2025",
    data=df,
    location="s3://bucket/exports/customers.csv"
)

# Read back
df = CsvAsset.read(client, "data_files", "exports", "customer_export_2025")
```

### **JSON Files**

```
from pyangolin.assets import JsonAsset

# Register JSON
```

```

json_asset = JsonAsset.write(
    client,
    catalog="data_files",
    namespace="configs",
    name="app_config",
    data=config_dict,
    location="s3://bucket/configs/app.json"
)

```

## ML Models

### Model Registration

```

from pyangolin.assets import MLModelAsset

# Register trained model
model_asset = MLModelAsset.register(
    client,
    catalog="ml_models",
    namespace="production",
    name="customer_churn_v2",
    location="s3://bucket/models/churn/v2/model.pkl",
    framework="scikit-learn",
    version="2.0.1",
    metrics={
        "accuracy": 0.92,
        "precision": 0.89,
        "recall": 0.94,
        "f1_score": 0.91
    },
    training_data="ml_features.customer_features@2025-12-01",
    description="Customer churn prediction model trained on Q4 2025 data"
)

```

### Model Metadata

```

# Comprehensive model metadata
metadata = {
    "framework": "pytorch",
    "framework_version": "2.1.0",
    "model_type": "transformer",
    "architecture": "bert-base",
    "parameters": 110000000,
    "training": {
        "dataset": "ml_features.text_corpus",
        "epochs": 10,
        "batch_size": 32,
        "learning_rate": 0.001,
        "optimizer": "adam"
    },
    "performance": {

```

```

        "train_loss": 0.15,
        "val_loss": 0.18,
        "test_accuracy": 0.94
    },
    "deployment": {
        "environment": "production",
        "endpoint": "https://api.example.com/predict",
        "sla_ms": 100
    }
}

```

## Vector Data (Lance)

### Lance Dataset

```

from pyangolin.assets import LanceAsset

# Write vector embeddings
lance_asset = LanceAsset.write(
    client,
    catalog="vectors",
    namespace="embeddings",
    name="product_embeddings",
    data=embeddings_df,
    location="/data/lance/products"
)

# Read for similarity search
embeddings = LanceAsset.read(client, "vectors", "embeddings", "product_embeddings")

```

## Media Files

### Image Assets

```

from pyangolin.assets import ImageAsset

# Register image dataset
image_asset = ImageAsset.register(
    client,
    catalog="media",
    namespace="product_images",
    name="catalog_photos_2025",
    location="s3://bucket/images/products/",
    format="jpeg",
    resolution="1920x1080",
    count=10000,
    total_size_mb=5000
)

```

## Video Assets

```
from pyangolin.assets import VideoAsset

# Register video collection
video_asset = VideoAsset.register(
    client,
    catalog="media",
    namespace="training_videos",
    name="onboarding_series",
    location="s3://bucket/videos/onboarding/",
    format="mp4",
    codec="h264",
    duration_seconds=3600,
    resolution="1920x1080"
)
```

# Organization

## Namespace Structure

### By Asset Type

```
data_files/
├─ csv/
├─ json/
├─ parquet/
└─ lance/

ml_models/
├─ production/
├─ staging/
└─ experiments/

media/
├─ images/
├─ videos/
└─ audio/
```

### By Project/Domain

```
customer_analytics/
├─ models/
├─ datasets/
├─ reports/
└─ visualizations/

marketing/
├─ campaigns/
├─ assets/
└─ analytics/
```

# Versioning

## Model Versioning

### Semantic Versioning

```
# Version progression
models = [
    "customer_churn_v1.0.0", # Initial release
    "customer_churn_v1.1.0", # Feature addition
    "customer_churn_v1.1.1", # Bug fix
    "customer_churn_v2.0.0", # Major update
]
```

### Git-Style Tagging

```
# Tag production model
pangolin-user create-tag \
  --catalog ml_models \
  --name production-2025-12-24 \
  --asset customer_churn_v2
```

## Dataset Versioning

### Snapshot-Based

```
# Reference specific snapshot
dataset = "ml_features.customer_features@snapshot-abc123"

# Reference by date
dataset = "ml_features.customer_features@2025-12-01"
```

# Metadata Management

## Rich Metadata

### File Assets

```
{
  "asset_type": "csv",
  "format": "csv",
  "compression": "gzip",
  "row_count": 1000000,
  "size_bytes": 50000000,
  "schema": {
    "columns": ["id", "name", "email", "created_at"],
    "types": ["int64", "string", "string", "timestamp"]
  },
}
```

```
"created_by": "etl-pipeline",
"source_system": "crm",
"refresh_schedule": "daily"
}
```

## ML Models

```
{
  "asset_type": "ml_model",
  "framework": "tensorflow",
  "task": "classification",
  "input_schema": {
    "features": ["age", "income", "tenure"],
    "types": ["float", "float", "int"]
  },
  "output_schema": {
    "prediction": "binary",
    "confidence": "float"
  },
  "performance_metrics": {
    "accuracy": 0.92,
    "auc_roc": 0.95
  },
  "deployment_status": "production",
  "monitoring_dashboard": "https://monitoring.example.com/models/churn"
}
```

# Access Patterns

## Discovery

### Searchable Metadata

```
# Search for ML models
results = client.search(
    catalog="ml_models",
    query="classification",
    filters={
        "asset_type": "ml_model",
        "framework": "pytorch",
        "deployment_status": "production"
    }
)
```

## Tagging

```
# Tag assets for discovery
pangolin-admin set-metadata \
  --catalog ml_models \
  --namespace production \
```

```
--asset customer_churn \  
--tags "ml,classification,production,customer-analytics"
```

## Lineage Tracking

### Model Lineage

```
{  
  "model": "customer_churn_v2",  
  "lineage": {  
    "training_data": [  
      "ml_features.customer_features@2025-12-01",  
      "ml_features.transaction_history@2025-12-01"  
    ],  
    "parent_model": "customer_churn_v1",  
    "derived_models": [  
      "customer_churn_v2_quantized",  
      "customer_churn_v2_onnx"  
    ]  
  }  
}
```

## Storage Optimization

### File Organization

#### Hierarchical Structure

```
s3://bucket/  
├── models/  
│   ├── customer_churn/  
│   │   ├── v1/  
│   │   │   ├── model.pkl  
│   │   │   └── metadata.json  
│   │   └── v2/  
│   │       ├── model.pkl  
│   │       └── metadata.json  
│   └── product_recommendation/  
└── datasets/  
    ├── training/  
    └── validation/
```

## Compression

### Appropriate Compression

```
# CSV with gzip  
CsvAsset.write(..., compression="gzip")
```

```
# Parquet with snappy
ParquetAsset.write(..., compression="snappy")

# Models with pickle protocol 5
import pickle
with open("model.pkl", "wb") as f:
    pickle.dump(model, f, protocol=5)
```

## Lifecycle Policies

### S3 Lifecycle

```
{
  "Rules": [{
    "Id": "ArchiveOldModels",
    "Filter": {"Prefix": "models/experiments/"},
    "Status": "Enabled",
    "Transitions": [{
      "Days": 90,
      "StorageClass": "GLACIER"
    }]
  }]
}
```

## Governance

## Access Control

### Model Access

```
# Grant read access to production models
pangolin-admin grant-permission \
  --user data-scientist@company.com \
  --catalog ml_models \
  --namespace production \
  --role READ

# Grant write access to experiments
pangolin-admin grant-permission \
  --user data-scientist@company.com \
  --catalog ml_models \
  --namespace experiments \
  --role READ_WRITE
```

## Compliance

## PII in Training Data

```
{  
  "model": "customer_churn",  
  "compliance": {  
    "contains_pii": true,  
    "pii_fields": ["email", "phone"],  
    "anonymization": "applied",  
    "retention_policy": "delete_after_90_days",  
    "gdpr_compliant": true  
  }  
}
```

## Best Practices Checklist

### Registration

- ☐ Descriptive names following conventions
- ☐ Comprehensive metadata
- ☐ Appropriate namespace organization
- ☐ Version information included

### Metadata

- ☐ Asset type clearly specified
- ☐ Format and schema documented
- ☐ Performance metrics (for models)
- ☐ Lineage information
- ☐ Tags for discovery

### Storage

- ☐ Appropriate compression used
- ☐ Hierarchical organization
- ☐ Lifecycle policies configured
- ☐ Backup strategy defined

### Governance

- ☐ Access controls configured

[ ] Compliance requirements met

[ ] Audit logging enabled

[ ] Retention policies defined

## Operations

[ ] Monitoring configured

[ ] Versioning strategy defined

[ ] Deprecation process documented

[ ] Cleanup procedures automated

## Additional Resources

[Generic Assets Guide](#)

[PyPangolin File Assets](#)

[PyPangolin ML Models](#)

[Lance Format](#)

Source: iceberg.md

# Apache Iceberg Best Practices

## Overview

This guide covers best practices for using Apache Iceberg tables with Pangolin, optimizing for performance, reliability, and maintainability.

## Table Design

## Partitioning Strategy

### Time-Based Partitioning

```
from pyiceberg.schema import Schema
from pyiceberg.types import NestedField, StringType, TimestampType, LongType
from pyiceberg.partitioning import PartitionSpec, PartitionField
from pyiceberg.transforms import DayTransform
```

```
# Schema
```

```
schema = Schema(
    NestedField(1, "event_id", LongType(), required=True),
    NestedField(2, "event_time", TimestampType(), required=True),
```

```

    NestedField(3, "user_id", StringType(), required=True),
    NestedField(4, "event_type", StringType(), required=True)
)

# Partition by day
partition_spec = PartitionSpec(
    PartitionField(source_id=2,    field_id=1000,    transform=DayTransform(),
name="event_day")
)

```

## Best Practices

Partition by query patterns (e.g., date for time-series data)

Avoid over-partitioning (target 100MB-1GB per partition)

Use transforms (day, month, hour) instead of raw values

Consider cardinality (avoid high-cardinality partitions)

## Multi-Dimensional Partitioning

```

# Partition by date and region
partition_spec = PartitionSpec(
    PartitionField(source_id=2,    field_id=1000,    transform=DayTransform(),
name="event_day"),
    PartitionField(source_id=5,    field_id=1001,    transform=IdentityTransform(),
name="region")
)

```

# Schema Evolution

## Adding Columns

```

# Add new column (safe operation)
table.update_schema().add_column("new_field", StringType()).commit()

# Add with default value
table.update_schema().add_column(
    "status",
    StringType(),
    doc="Order status"
).commit()

```

## Renaming Columns

```

# Rename column (metadata-only operation)
table.update_schema().rename_column("old_name", "new_name").commit()

```

## Type Promotion

```
# Safe type promotions
table.update_schema().update_column("age", IntegerType()).commit() # int -> long
table.update_schema().update_column("price", DoubleType()).commit() # float -> double
```

## Best Practices

Use schema evolution instead of recreating tables

Test schema changes in development first

Document schema changes in metadata

Avoid dropping columns (mark as deprecated instead)

## Sorting

### Sort Order

```
from pyiceberg.table.sorting import SortOrder, SortField
from pyiceberg.transforms import IdentityTransform

# Sort by frequently filtered columns
sort_order = SortOrder(
    SortField(source_id=3, transform=IdentityTransform()), # user_id
    SortField(source_id=2, transform=IdentityTransform()) # event_time
)

table.update_properties().set("write.distribution-mode", "hash").commit()
```

## Benefits

Faster queries on sorted columns

Better compression

Improved data skipping

## Write Operations

### Batch Writes

#### Optimize Batch Size

```
# Target 100MB-1GB per file
table.update_properties().set("write.target-file-size-bytes", "536870912").commit() # 512MB

# Append data in batches
batch_size = 100000
for i in range(0, len(data), batch_size):
    batch = data[i:i+batch_size]
```

```
table.append(batch)
```

## Upserts/Merges

### Merge-on-Read

```
# Efficient for frequent updates
table.update_properties().set("write.merge.mode", "merge-on-read").commit()

# Update rows
table.update().where("user_id = 'user123']").set("status", "active").commit()
```

### Copy-on-Write

```
# Better for read-heavy workloads
table.update_properties().set("write.merge.mode", "copy-on-write").commit()
```

## Compaction

### Auto-Compaction

```
# Enable automatic compaction
table.update_properties().set("write.metadata.compression-codec", "gzip").commit()
table.update_properties().set("write.metadata.metrics.default", "truncate(16)").commit()
```

### Manual Compaction

```
# Compact small files
pangolin-admin optimize-catalog \
  --catalog production \
  --namespace events \
  --table user_events \
  --compact-files
```

## Read Operations

## Partition Pruning

### Leverage Partitions

```
# Query with partition filter (fast)
df = table.scan(
    row_filter="event_day >= '2025-12-01' AND event_day < '2025-12-31'"
).to_pandas()

# Without partition filter (slow - full scan)
```

```
df = table.scan(  
    row_filter="event_type = 'click'"  
) .to_pandas()
```

## Column Projection

### Select Only Needed Columns

```
# Good - only read necessary columns  
df = table.scan(  
    selected_fields=("event_id", "event_time", "user_id")  
) .to_pandas()  
  
# Bad - reads all columns  
df = table.scan().to_pandas()
```

## Predicate Pushdown

### Filter Early

```
# Pushdown to file level  
df = table.scan(  
    row_filter="user_id = 'user123' AND event_time > '2025-12-01'"  
) .to_pandas()
```

## Snapshot Management

## Retention Policy

### Configure Retention

```
# Keep snapshots for 7 days  
table.update_properties().set("history.expire.max-snapshot-age-ms",  
    "604800000").commit()  
  
# Keep minimum 10 snapshots  
table.update_properties().set("history.expire.min-snapshots-to-keep", "10").commit()
```

### Manual Cleanup

```
# Remove old snapshots  
pangolin-admin optimize-catalog \  
    --catalog production \  
    --remove-old-snapshots \  
    --retention-days 30
```

# Time Travel

## Query Historical Data

```
# Query as of timestamp
df = table.scan(
    snapshot_id=table.history()[0].snapshot_id
).to_pandas()

# Query as of specific time
import datetime
timestamp = datetime.datetime(2025, 12, 1).timestamp() * 1000
df = table.scan(as_of_timestamp=timestamp).to_pandas()
```

# Performance Optimization

## File Size

### Target File Size

```
# Optimal: 100MB-1GB per file
table.update_properties().set("write.target-file-size-bytes", "536870912").commit() # 512MB
```

### Avoid Small Files

Combine small writes into batches

Run regular compaction

Use appropriate batch sizes

## Metadata Caching

### Enable Caching

```
# Cache metadata for faster queries
catalog_properties = {
    "cache-enabled": "true",
    "cache.expiration-interval-ms": "300000" # 5 minutes
}
```

# Statistics

### Collect Statistics

```
# Enable column statistics
table.update_properties().set("write.metadata.metrics.default", "full").commit()
```

```
# Specific columns
table.update_properties().set("write.metadata.metrics.column.user_id",
"counts").commit()
```

## Data Quality

## Schema Validation

### Enforce Schema

```
# Require schema compatibility
table.update_properties().set("commit.manifest.min-count-to-merge", "5").commit()
table.update_properties().set("commit.manifest-merge.enabled", "true").commit()
```

## Constraints

### Not Null Constraints

```
# Define required fields in schema
schema = Schema(
    NestedField(1, "id", LongType(), required=True), # NOT NULL
    NestedField(2, "name", StringType(), required=False) # NULLABLE
)
```

## Maintenance

## Regular Tasks

### Daily

Monitor table growth

Check query performance

Review error logs

### Weekly

Compact small files

Review partition distribution

Check snapshot count

### Monthly

Expire old snapshots

Remove orphaned files

Optimize table layout

## Monitoring

### Key Metrics

```
# Table size
metadata = table.metadata
total_size = sum(f.file_size_in_bytes for f in table.scan().plan_files())

# File count
file_count = len(list(table.scan().plan_files()))

# Snapshot count
snapshot_count = len(table.history())

# Partition count
partition_count = len(table.scan().plan_files())
```

## Migration

### From Hive/Parquet

#### In-Place Migration

```
# Register existing Parquet as Iceberg
from pyiceberg.catalog import load_catalog

catalog = load_catalog("pangolin")
catalog.register_table(
    identifier="analytics.legacy_table",
    metadata_location="s3://bucket/path/to/metadata/v1.metadata.json"
)
```

#### Copy Migration

```
# Copy data to new Iceberg table
source_df = spark.read.parquet("s3://bucket/legacy/data")
source_df.writeTo("analytics.new_iceberg_table").create()
```

## Best Practices Checklist

### Table Design

[ ] Appropriate partitioning strategy

- [ ] Optimal file size (100MB-1GB)
- [ ] Sort order defined for common queries
- [ ] Schema documented

## Operations

- [ ] Batch writes configured
- [ ] Compaction scheduled
- [ ] Snapshot retention policy set
- [ ] Statistics collection enabled

## Performance

- [ ] Partition pruning utilized
- [ ] Column projection used
- [ ] Metadata caching enabled
- [ ] Regular maintenance scheduled

## Governance

- [ ] Schema evolution documented
- [ ] Data quality rules defined
- [ ] Retention policies configured
- [ ] Access patterns monitored

## Additional Resources

[PyIceberg Integration](#)

[Table Formats Guide](#)

[Maintenance Operations](#)

[PyPangolin Iceberg Guide](#)

Source: metadata.md

# Business Metadata Management Best Practices

## Overview

This guide covers best practices for managing business metadata in Pangolin to enhance data discovery and governance.

# Metadata Strategy

## Types of Metadata

### **Technical Metadata** (Auto-Generated)

Schema information

File formats

Partition specs

Statistics

### **Business Metadata** (User-Defined)

Business definitions

Data ownership

Data quality rules

Usage guidelines

### **Operational Metadata**

Access patterns

Query performance

Data lineage

Refresh schedules

## Business Metadata Schema

### Standard Fields

```
{
  "owner": "data-team@company.com",
  "steward": "alice@company.com",
  "domain": "sales",
  "classification": "internal",
  "pii": false,
  "retention_days": 2555, // 7 years
  "refresh_schedule": "daily",
  "description": "Customer transaction data for sales analytics",
  "tags": ["sales", "customer", "transactions"],
  "quality_score": 95,
  "last_validated": "2025-12-24"
}
```

# Custom Fields by Domain

## Sales Domain

```
{
  "sales_region": "north_america",
  "product_category": "software",
  "revenue_impact": "high"
}
```

## Finance Domain

```
{
  "gl_account": "4000-revenue",
  "cost_center": "engineering",
  "fiscal_year": "2025"
}
```

# Metadata Management

## Setting Metadata

```
# Via CLI
pangolin-admin set-metadata \
  --catalog analytics \
  --namespace sales \
  --asset customers \
  --key owner \
  --value "sales-team@company.com"

# Bulk metadata
pangolin-admin set-metadata \
  --catalog analytics \
  --namespace sales \
  --asset customers \
  --metadata '{
    "owner": "sales-team@company.com",
    "steward": "alice@company.com",
    "domain": "sales",
    "pii": true,
    "description": "Customer master data"
  }'
```

```
# Via Python
from pyangolin import PangolinClient

client = PangolinClient("http://localhost:8080")
client.login("admin", "password")
```

```
# Set metadata
client.set_metadata(
    catalog="analytics",
    namespace="sales",
    asset="customers",
    metadata={
        "owner": "sales-team@company.com",
        "domain": "sales",
        "pii": True
    }
)
```

## Metadata Templates

### Create Reusable Templates

```
# templates/sales_table.yaml
metadata:
  domain: sales
  owner: sales-team@company.com
  classification: internal
  retention_days: 2555
  tags:
    - sales
    - customer_data
```

### Apply Templates

```
# Apply template to new tables
for table in customers orders products; do
  pangolin-admin set-metadata \
    --catalog analytics \
    --namespace sales \
    --asset $table \
    --template templates/sales_table.yaml
done
```

## Data Governance

### Data Ownership

#### Clear Ownership Model

```
Data Owner (Business)
├─ Accountable for data quality
├─ Approves access requests
└─ Defines retention policies
```

```
Data Steward (Technical)
```

- └─ Implements data quality rules
- └─ Manages technical metadata
- └─ Monitors data health

#### Data Consumer (User)

- └─ Uses data responsibly
- └─ Reports quality issues
- └─ Follows usage guidelines

## Document Ownership

```
{
  "table": "sales.customers",
  "owner": {
    "name": "Sales VP",
    "email": "vp-sales@company.com",
    "role": "business_owner"
  },
  "steward": {
    "name": "Alice Johnson",
    "email": "alice@company.com",
    "role": "data_steward"
  },
  "contact": "sales-data-team@company.com"
}
```

# Data Classification

## Classification Levels

```
classifications:
  public:
    description: "Publicly available data"
    access: "all_users"
    encryption: "optional"

  internal:
    description: "Internal business data"
    access: "employees_only"
    encryption: "required"

  confidential:
    description: "Sensitive business data"
    access: "need_to_know"
    encryption: "required"
    audit_logging: "mandatory"

  restricted:
    description: "Highly sensitive data (PII, financial)"
    access: "explicit_approval"
    encryption: "required"
```

```
audit_logging: "mandatory"
data_masking: "required"
```

## Apply Classifications

```
# Tag PII data
pangolin-admin set-metadata \
  --catalog production \
  --namespace customers \
  --asset personal_info \
  --metadata '{
    "classification": "restricted",
    "pii": true,
    "pii_fields": ["email", "phone", "ssn"],
    "data_masking_required": true
  }'
```

## Data Quality

### Quality Metadata

```
{
  "quality_rules": [
    {
      "rule": "not_null",
      "columns": ["customer_id", "email"],
      "severity": "critical"
    },
    {
      "rule": "unique",
      "columns": ["customer_id"],
      "severity": "critical"
    },
    {
      "rule": "valid_email",
      "columns": ["email"],
      "severity": "high"
    }
  ],
  "quality_score": 98.5,
  "last_quality_check": "2025-12-24T10:00:00Z",
  "quality_issues": 0
}
```

## Discovery & Search

### Searchable Metadata

## Optimize for Discovery

```
{
  "title": "Customer Master Data",
  "description": "Comprehensive customer information including demographics, contact details, and account status. Updated daily from CRM system.",
  "keywords": ["customer", "crm", "demographics", "contact"],
  "use_cases": [
    "Customer segmentation",
    "Marketing campaigns",
    "Sales analytics"
  ],
  "sample_queries": [
    "SELECT * FROM customers WHERE region = 'NA' LIMIT 10",
    "SELECT COUNT(*) FROM customers GROUP BY customer_segment"
  ]
}
```

## Rich Descriptions

# Customer Transactions Table

## Overview

Daily transaction data from point-of-sale systems across all retail locations.

## Schema

- `transaction\_id`: Unique identifier for each transaction
- `customer\_id`: Links to customers table
- `amount`: Transaction amount in USD
- `timestamp`: Transaction timestamp (UTC)

## Usage Guidelines

- Use for sales analysis and customer behavior studies
- Join with customers table for demographic analysis
- Partition by date for performance

## Known Issues

- Transactions before 2024-01-01 may have missing customer\_id
- Amount field includes tax (use amount\_pretax for revenue calculations)

## Contact

For questions, contact: sales-analytics@company.com

## Tagging Strategy

### Hierarchical Tags

```
tags:
  domain:
    - sales
    - marketing
    - finance
```

```
data_type:
  - transactional
  - dimensional
  - aggregated

sensitivity:
  - public
  - internal
  - confidential

geography:
  - north_america
  - europe
  - asia_pacific
```

## Apply Tags

```
# Tag by domain
pangolin-admin set-metadata \
  --catalog analytics \
  --namespace sales \
  --asset transactions \
  --tags "domain:sales,data_type:transactional,geography:north_america"
```

# Metadata Lifecycle

## New Asset Onboarding

```
#!/bin/bash
# Metadata onboarding checklist

CATALOG=$1
NAMESPACE=$2
ASSET=$3

# 1. Set basic metadata
pangolin-admin set-metadata \
  --catalog "$CATALOG" \
  --namespace "$NAMESPACE" \
  --asset "$ASSET" \
  --metadata '{
    "created_date": "'$(date -I)'",
    "status": "draft"
  }'

# 2. Assign owner
echo "Who is the data owner? (email)"
read OWNER
pangolin-admin set-metadata --key owner --value "$OWNER"
```

```
# 3. Set classification
echo "Classification level? (public/internal/confidential/restricted)"
read CLASSIFICATION
pangolin-admin set-metadata --key classification --value "$CLASSIFICATION"

# 4. Add description
echo "Please provide a description:"
read DESCRIPTION
pangolin-admin set-metadata --key description --value "$DESCRIPTION"

# 5. Mark as active
pangolin-admin set-metadata --key status --value "active"
```

## Metadata Updates

### Version Control

```
{
  "metadata_version": "2.0",
  "last_updated": "2025-12-24T15:00:00Z",
  "updated_by": "alice@company.com",
  "change_log": [
    {
      "date": "2025-12-24",
      "field": "owner",
      "old_value": "bob@company.com",
      "new_value": "alice@company.com",
      "reason": "Team reorganization"
    }
  ]
}
```

## Metadata Retirement

```
# Mark as deprecated
pangolin-admin set-metadata \
  --catalog analytics \
  --namespace legacy \
  --asset old_table \
  --metadata '{
    "status": "deprecated",
    "deprecated_date": "2025-12-24",
    "replacement": "analytics.new_namespace.new_table",
    "deletion_date": "2026-03-24"
  }'
```

## Automation

# Metadata Extraction

## Auto-Generate from Schema

```
def generate_metadata_from_schema(table):
    """Extract metadata from table schema"""
    return {
        "column_count": len(table.schema().fields),
        "columns": [
            {
                "name": field.name,
                "type": str(field.field_type),
                "nullable": field.optional
            }
            for field in table.schema().fields
        ],
        "partition_spec": str(table.spec()),
        "sort_order": str(table.sort_order())
    }
```

## Infer from Usage

```
def infer_metadata_from_usage(table_name):
    """Infer metadata from query logs"""
    queries = get_query_logs(table_name)

    return {
        "popular_columns": get_most_queried_columns(queries),
        "common_filters": get_common_where_clauses(queries),
        "typical_joins": get_join_patterns(queries),
        "query_frequency": len(queries),
        "unique_users": count_unique_users(queries)
    }
```

# Metadata Validation

## Validation Rules

```
def validate_metadata(metadata):
    """Validate required metadata fields"""
    required_fields = ["owner", "description", "classification"]

    for field in required_fields:
        if field not in metadata:
            raise ValueError(f"Missing required field: {field}")

    if not is_valid_email(metadata["owner"]):
        raise ValueError("Invalid owner email")

    if len(metadata["description"]) < 50:
        raise ValueError("Description too short (minimum 50 characters)")
```

```

valid_classifications = ["public", "internal", "confidential", "restricted"]
if metadata["classification"] not in valid_classifications:
    raise ValueError(f"Invalid classification: {metadata['classification']}")

```

## Reporting

### Metadata Coverage

```

-- Check metadata completeness
SELECT
    catalog,
    namespace,
    COUNT(*) as total_assets,
    SUM(CASE WHEN owner IS NOT NULL THEN 1 ELSE 0 END) as with_owner,
    SUM(CASE WHEN description IS NOT NULL THEN 1 ELSE 0 END) as with_description,
    SUM(CASE WHEN classification IS NOT NULL THEN 1 ELSE 0 END) as with_classification,
    ROUND(100.0 * SUM(CASE WHEN owner IS NOT NULL THEN 1 ELSE 0 END) / COUNT(*), 2) as
owner_coverage_pct
FROM assets
GROUP BY catalog, namespace
ORDER BY owner_coverage_pct ASC;

```

### Metadata Quality Dashboard

```

def generate_metadata_dashboard():
    """Generate metadata quality metrics"""
    return {
        "total_assets": count_all_assets(),
        "metadata_coverage": {
            "owner": calculate_coverage("owner"),
            "description": calculate_coverage("description"),
            "classification": calculate_coverage("classification"),
            "tags": calculate_coverage("tags")
        },
        "data_quality": {
            "avg_quality_score": get_avg_quality_score(),
            "assets_with_quality_rules": count_assets_with_quality_rules()
        },
        "governance": {
            "pii_assets": count_pii_assets(),
            "classified_assets": count_classified_assets(),
            "orphaned_assets": count_orphaned_assets() # No owner
        }
    }

```

## Best Practices Checklist

## For Data Owners

- [ ] Assign clear ownership to all datasets
- [ ] Provide comprehensive descriptions
- [ ] Classify data appropriately
- [ ] Define data quality rules
- [ ] Document retention policies
- [ ] Review metadata quarterly

## For Data Stewards

- [ ] Validate metadata completeness
- [ ] Enforce metadata standards
- [ ] Monitor data quality
- [ ] Update technical metadata
- [ ] Maintain data lineage
- [ ] Automate where possible

## For Data Consumers

- [ ] Search metadata before requesting access
- [ ] Report metadata inaccuracies
- [ ] Follow usage guidelines
- [ ] Provide feedback on data quality
- [ ] Document new use cases

## Additional Resources

[Business Catalog Guide](#)

[Tag Management](#)

[Data Discovery](#)

Source: permissions.md

## Permissions Management Best Practices

# Overview

This guide covers best practices for managing permissions in Pangolin using Role-Based Access Control (RBAC).

## Permission Model

### Roles Hierarchy

Root (System Administrator)

- └─ Full system access
- └─ Tenant management
- └─ Cross-tenant operations

Tenant Admin

- └─ Full tenant access
- └─ User management
- └─ Warehouse/catalog creation
- └─ Permission grants within tenant

Tenant User

- └─ Catalog access (as granted)
- └─ Table operations (as granted)
- └─ Self-service discovery

Service User

- └─ API-only access
- └─ Programmatic operations
- └─ Limited to granted permissions

## Permission Scopes

### Catalog-Level

```
# Grant access to entire catalog
pangolin-admin grant-permission \
  --user analyst@company.com \
  --catalog production \
  --role READ
```

### Namespace-Level

```
# Grant access to specific namespace
pangolin-admin grant-permission \
  --user analyst@company.com \
  --catalog production \
  --namespace sales \
  --role READ_WRITE
```

## Asset-Level (via tags)

```
# Grant access to tagged assets
pangolin-admin grant-permission \
  --user analyst@company.com \
  --catalog production \
  --tag "department:sales" \
  --role READ
```

# Best Practices

## 1. Principle of Least Privilege

### Start Minimal

```
# Don't grant broad access initially
pangolin-admin grant-permission --user new-user --catalog "*" --role ADMIN

# Grant specific access as needed
pangolin-admin grant-permission --user new-user --catalog analytics --namespace reports
--role READ
```

### Expand Gradually

Start with READ access

Upgrade to READ\_WRITE when needed

ADMIN role only for catalog owners

## 2. Role-Based Assignment

### Define Organizational Roles

```
# Example role definitions
roles:
  data_analyst:
    catalogs:
      - analytics: READ
      - reports: READ_WRITE
    namespaces:
      - sales.*: READ

  data_engineer:
    catalogs:
      - raw_data: READ_WRITE
      - processed: READ_WRITE
    namespaces:
      - etl.*: ADMIN

  data_scientist:
```

```
catalogs:
  - ml_features: READ_WRITE
  - experiments: ADMIN
```

## Implement via Scripts

```
#!/bin/bash
# Grant permissions for data analyst role

USERS=("alice@company.com" "bob@company.com")
for user in "${USERS[@]}; do
  pangolin-admin grant-permission --user "$user" --catalog analytics --role READ
  pangolin-admin grant-permission --user "$user" --catalog reports --role READ_WRITE
done
```

## 3. Tag-Based Access Control

### Organize by Tags

```
# Tag assets by department
{
  "table": "sales.customers",
  "tags": {
    "department": "sales",
    "sensitivity": "high",
    "region": "us-east"
  }
}
```

### Grant by Tags

```
# Sales team gets access to sales data
pangolin-admin grant-permission \
  --user sales-team@company.com \
  --catalog production \
  --tag "department:sales" \
  --role READ_WRITE

# Regional access
pangolin-admin grant-permission \
  --user eu-analyst@company.com \
  --catalog production \
  --tag "region:eu" \
  --role READ
```

## 4. Service User Permissions

### Dedicated Service Users

```
# ETL pipeline - write to staging, read from raw
pangolin-admin create-service-user --name etl-pipeline
pangolin-admin grant-permission --service-user etl-pipeline --catalog raw_data --role
READ
pangolin-admin grant-permission --service-user etl-pipeline --catalog staging --role
READ_WRITE

# BI tool - read-only access
pangolin-admin create-service-user --name tableau-connector
pangolin-admin grant-permission --service-user tableau-connector --catalog analytics
--role READ
```

## Document Service Users

```
# Service User Registry

| Name | Purpose | Permissions | Owner | Last Rotated |
|-----|-----|-----|-----|-----|
| etl-pipeline | Nightly ETL | raw_data:READ, staging:RW | Data Eng | 2025-12-01 |
| tableau-connector | BI Dashboards | analytics:READ | Analytics | 2025-11-15 |
```

## 5. Temporary Access

### Time-Limited Grants

```
# Grant temporary access for incident investigation
pangolin-admin grant-permission \
  --user incident-responder@company.com \
  --catalog production \
  --role READ \
  --expires-in 24h

# Revoke after investigation
pangolin-admin revoke-permission \
  --user incident-responder@company.com \
  --catalog production
```

## 6. Access Request Workflow

### User Self-Service

```
# User requests access
pangolin-user request-access \
  --catalog analytics \
  --namespace sales \
  --role READ \
  --reason "Q4 sales analysis project"

# Admin reviews and approves
pangolin-admin list-access-requests
```

```
pangolin-admin update-access-request --id <uuid> --status approved
```

```
# System automatically grants permission
```

## Permission Patterns

### Development → Staging → Production

#### Environment Isolation

```
# Developers: Full access to dev, read-only to staging
pangolin-admin grant-permission --user dev-team --catalog dev --role ADMIN
pangolin-admin grant-permission --user dev-team --catalog staging --role READ

# QA: Read-write to staging, read-only to production
pangolin-admin grant-permission --user qa-team --catalog staging --role READ_WRITE
pangolin-admin grant-permission --user qa-team --catalog production --role READ

# Production: Limited access, strict controls
pangolin-admin grant-permission --user prod-operators --catalog production --role READ_WRITE
```

## Data Sensitivity Levels

#### Classification-Based Access

```
# Public data - broad access
pangolin-admin grant-permission --tag "sensitivity:public" --role READ

# Internal data - employees only
pangolin-admin grant-permission --tag "sensitivity:internal" --group employees --role READ

# Confidential - specific teams
pangolin-admin grant-permission --tag "sensitivity:confidential" --group finance --role READ

# Restricted - need-to-know basis
pangolin-admin grant-permission --tag "sensitivity:restricted" --user authorized-user --role READ
```

## Monitoring & Auditing

### Regular Access Reviews

#### Quarterly Review Process

```
# 1. Export current permissions
pangolin-admin list-permissions --format csv > permissions_Q4_2025.csv

# 2. Review with managers
# 3. Revoke unnecessary access
pangolin-admin revoke-permission --user inactive-user --catalog analytics

# 4. Document review
echo "Q4 2025 access review completed" >> access_review_log.txt
```

## Permission Change Auditing

### Track All Changes

```
-- Query audit logs for permission changes
SELECT
    timestamp,
    actor,
    action,
    resource,
    details
FROM audit_logs
WHERE action IN ('GRANT_PERMISSION', 'REVOKE_PERMISSION')
ORDER BY timestamp DESC
LIMIT 100;
```

### Alert on Sensitive Changes

```
alerts:
- name: Admin Permission Granted
  condition: action = 'GRANT_PERMISSION' AND role = 'ADMIN'
  notify: security-team@company.com

- name: Production Access Granted
  condition: action = 'GRANT_PERMISSION' AND catalog = 'production'
  notify: data-governance@company.com
```

## Unused Permission Detection

### Identify Inactive Access

```
-- Find users with permissions but no recent activity
SELECT
    p.user_id,
    p.catalog,
    p.role,
    MAX(a.timestamp) as last_access
FROM permissions p
LEFT JOIN audit_logs a ON p.user_id = a.user_id AND p.catalog = a.catalog
GROUP BY p.user_id, p.catalog, p.role
```

```
HAVING MAX(a.timestamp) < NOW() - INTERVAL '90 days'  
OR MAX(a.timestamp) IS NULL;
```

## Common Scenarios

### Onboarding New User

```
#!/bin/bash  
# Onboarding script for data analyst  
  
USER_EMAIL=$1  
MANAGER_EMAIL=$2  
  
# 1. Create user  
pangolin-admin create-user \  
  --username "$USER_EMAIL" \  
  --email "$USER_EMAIL" \  
  --role tenant-user  
  
# 2. Grant standard analyst permissions  
pangolin-admin grant-permission --user "$USER_EMAIL" --catalog analytics --role READ  
pangolin-admin grant-permission --user "$USER_EMAIL" --catalog sandbox --role READ_WRITE  
  
# 3. Notify manager  
echo "User $USER_EMAIL onboarded with standard analyst permissions" | \  
  mail -s "New User Onboarded" "$MANAGER_EMAIL"
```

### Offboarding User

```
#!/bin/bash  
# Offboarding script  
  
USER_EMAIL=$1  
  
# 1. Disable user  
pangolin-admin update-user --email "$USER_EMAIL" --active false  
  
# 2. Revoke all tokens  
pangolin-admin list-user-tokens --user "$USER_EMAIL" | \  
  xargs -I {} pangolin-admin delete-token --id {}  
  
# 3. Document for audit  
echo "$(date): User $USER_EMAIL offboarded" >> offboarding_log.txt
```

## Project-Based Access

```
# Grant access for project duration
```

```
PROJECT_TEAM=("alice@company.com" "bob@company.com" "charlie@company.com")
PROJECT_CATALOG="customer_churn_analysis"

for user in "${PROJECT_TEAM[@]}; do
    pangolin-admin grant-permission \
        --user "$user" \
        --catalog "$PROJECT_CATALOG" \
        --role READ_WRITE \
        --expires-in 90d
done

# Set reminder to review after 90 days
```

## Troubleshooting

### Permission Denied Issues

#### Debug Process

```
# 1. Check user's current permissions
pangolin-admin list-permissions --user user@company.com

# 2. Check catalog/namespace exists
pangolin-admin list-catalogs

# 3. Verify user is active
pangolin-admin list-users --filter "email=user@company.com"

# 4. Check audit logs for recent denials
pangolin-admin list-audit-events --user user@company.com --action ACCESS_DENIED
```

### Over-Permissioned Users

#### Identify and Remediate

```
# Find users with ADMIN role
pangolin-admin list-permissions --role ADMIN

# Review necessity
# Downgrade if not needed
pangolin-admin revoke-permission --user user@company.com --catalog analytics --role ADMIN
pangolin-admin grant-permission --user user@company.com --catalog analytics --role READ_WRITE
```

## Compliance

# GDPR/Privacy Considerations

## Data Access Logging

Log all access to PII-tagged data

Retain logs for compliance period

Provide access reports on request

## Right to Access

```
# Generate user's data access report
pangolin-admin list-audit-events \
  --user user@company.com \
  --action READ \
  --tag "pii:true" \
  --start-date 2025-01-01
```

# SOC 2 / ISO 27001

## Access Control Requirements

- ☐ Documented permission model
- ☐ Regular access reviews (quarterly)
- ☐ Segregation of duties
- ☐ Audit trail of all changes
- ☐ Automated provisioning/deprovisioning

## Checklist

### Initial Setup

- ☐ Define organizational roles
- ☐ Document permission model
- ☐ Create service users for applications
- ☐ Set up access request workflow
- ☐ Configure audit logging

### Ongoing Operations

- ☐ Quarterly access reviews
- ☐ Monitor permission changes

- [ ] Review service user keys
- [ ] Audit unused permissions
- [ ] Update documentation

## Security

- [ ] Principle of least privilege enforced
- [ ] No wildcard permissions in production
- [ ] Service users have minimal permissions
- [ ] Sensitive data access logged
- [ ] Compliance requirements met

## Additional Resources

[Security Best Practices](#)

[RBAC Documentation](#)

[Audit Logging](#)

Source: scalability.md

## Scalability Best Practices

### Overview

This guide covers strategies for scaling Pangolin to handle growing data volumes, user bases, and query loads.

### Scaling Dimensions

#### Vertical Scaling (Scale Up)

##### **When to Scale Up**

Single-tenant deployments with predictable growth

Cost-effective for small to medium workloads

Simpler operational model

##### **Limits**

Hardware constraints (CPU, RAM, disk)

Single point of failure

Downtime required for upgrades

## Horizontal Scaling (Scale Out)

### When to Scale Out

Multi-tenant deployments

Unpredictable or bursty workloads

High availability requirements

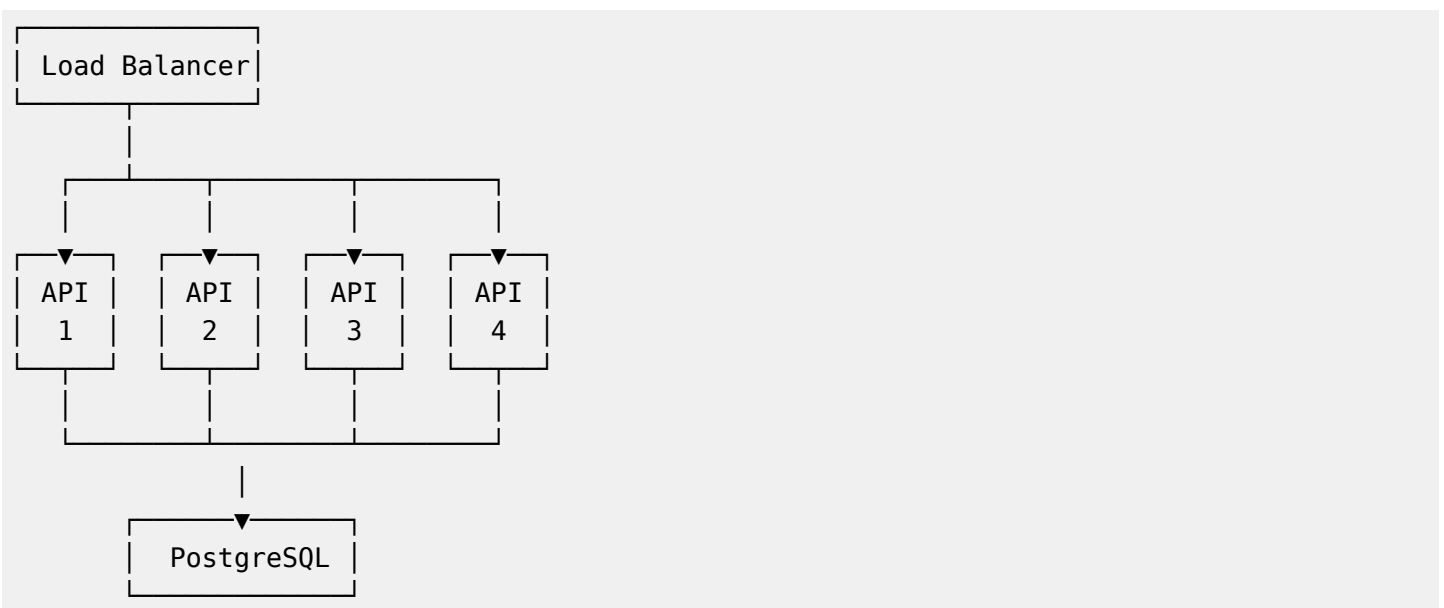
Large-scale deployments (100+ users, 1000+ tables)

## API Server Scaling

### Stateless Design

Pangolin API servers are stateless, enabling easy horizontal scaling.

### Load Balancing Strategy



### Scaling Triggers

CPU utilization > 70%

Request latency p95 > 500ms

Request queue depth > 100

### Auto-Scaling Configuration (Kubernetes)

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
```

```

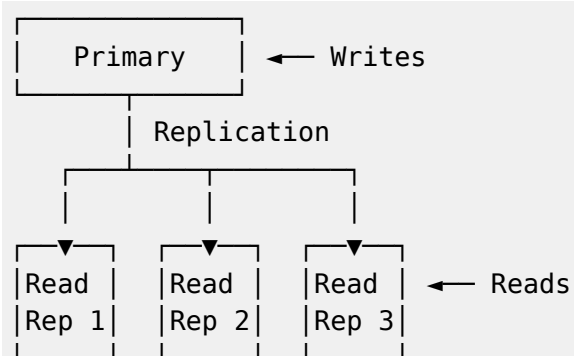
metadata:
  name: pangolin-api-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: pangolin-api
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 80

```

## Database Scaling

### PostgreSQL Scaling

#### Read Replicas



#### Connection Pooling

```

# PgBouncer configuration
[databases]
pangolin = host=localhost port=5432 dbname=pangolin

[pgbouncer]
pool_mode = transaction
max_client_conn = 1000
default_pool_size = 20

```

```
reserve_pool_size = 5
```

## Partitioning Strategy

```
-- Partition audit logs by month
CREATE TABLE audit_logs (
    id UUID PRIMARY KEY,
    timestamp TIMESTAMP NOT NULL,
    action VARCHAR(50),
    ...
) PARTITION BY RANGE (timestamp);

CREATE TABLE audit_logs_2025_01 PARTITION OF audit_logs
    FOR VALUES FROM ('2025-01-01') TO ('2025-02-01');
```

## MongoDB Scaling

### Sharding

```
// Shard by tenant_id for multi-tenancy
sh.shardCollection("pangolin.assets", { tenant_id: 1, catalog: 1 })

// Shard by catalog for single-tenant
sh.shardCollection("pangolin.assets", { catalog: 1, namespace: 1 })
```

## Multi-Tenancy Scaling

### Tenant Isolation Strategies

#### Shared Infrastructure (Cost-Effective)

All tenants share same API servers and database

Logical isolation via tenant\_id

Suitable for: 100s-1000s of small tenants

#### Dedicated Infrastructure (High Isolation)

Each tenant gets dedicated resources

Physical isolation

Suitable for: Enterprise customers, compliance requirements

#### Hybrid Approach (Recommended)

Shared API servers

Tenant-specific database schemas or databases

Balance of cost and isolation

## Tenant-Specific Scaling

```
-- Monitor per-tenant resource usage
SELECT
    tenant_id,
    COUNT(*) as table_count,
    SUM(size_bytes) as total_size
FROM assets
GROUP BY tenant_id
ORDER BY total_size DESC;
```

## Catalog Scaling

### Namespace Organization

#### Flat Structure (Small Scale)

```
catalog
├─ namespace1
├─ namespace2
└─ namespace3
```

#### Hierarchical Structure (Large Scale)

```
catalog
├─ department
│   ├── team1
│   │   ├── project_a
│   │   └─ project_b
│   └─ team2
└─ shared
```

#### Best Practices

Limit namespace depth to 3-4 levels

Use consistent naming conventions

Implement namespace quotas

## Table Distribution

#### Anti-Pattern: Single Mega-Catalog

```
production (10,000 tables)
```

## Best Practice: Multiple Catalogs

```
sales_analytics (500 tables)
marketing_data (300 tables)
finance_reports (200 tables)
operations_metrics (400 tables)
```

# Metadata Scaling

## Metadata Size Management

### Audit Log Archival

```
-- Archive logs older than 90 days
INSERT INTO audit_logs_archive
SELECT * FROM audit_logs
WHERE timestamp < NOW() - INTERVAL '90 days';

DELETE FROM audit_logs
WHERE timestamp < NOW() - INTERVAL '90 days';
```

### Snapshot Cleanup

```
# Remove snapshots older than retention policy
# Implement via maintenance operations
pangolin-admin optimize-catalog --catalog production \
  --remove-old-snapshots --retention-days 30
```

# Indexing Strategy

### Essential Indexes

```
-- Catalog operations
CREATE INDEX idx_assets_catalog_namespace ON assets(catalog_id, namespace);
CREATE INDEX idx_assets_tenant ON assets(tenant_id);

-- Search and discovery
CREATE INDEX idx_assets_name ON assets(name);
CREATE INDEX idx_assets_kind ON assets(kind);

-- Audit queries
CREATE INDEX idx_audit_tenant_time ON audit_logs(tenant_id, timestamp DESC);
CREATE INDEX idx_audit_action ON audit_logs(action);
```

# Caching Strategies

# Metadata Caching

## Redis Configuration

```
# Cache frequently accessed metadata
cache:
  type: redis
  host: redis-cluster
  ttl: 300 # 5 minutes

# Cache keys
patterns:
  - "catalog*:metadata"
  - "namespace*:list"
  - "permissions:user:*
```

## Cache Invalidation

Invalidate on write operations

Use pub/sub for multi-instance coordination

Implement cache warming for critical data

## Query Result Caching

```
# Cache expensive queries
@cache(ttl=600) # 10 minutes
def get_catalog_summary(catalog_id):
    return {
        'table_count': count_tables(catalog_id),
        'total_size': calculate_size(catalog_id),
        'namespaces': list_namespaces(catalog_id)
    }
```

# Object Storage Scaling

## S3 Performance Optimization

### Prefix Distribution

```
# Anti-pattern: Sequential prefixes
s3://bucket/table-00001/
s3://bucket/table-00002/
s3://bucket/table-00003/

# Best practice: Hash-based prefixes
s3://bucket/a7f3/table-00001/
s3://bucket/b2e9/table-00002/
s3://bucket/c4d1/table-00003/
```

## Request Rate Limits

S3: 3,500 PUT/COPY/POST/DELETE, 5,500 GET/HEAD per prefix per second

Use multiple prefixes for high-throughput workloads

## Multi-Region Storage

```
Primary Region: us-east-1
├─ Catalog metadata: RDS us-east-1
└─ Table data: S3 us-east-1
```

```
Secondary Region: eu-west-1
├─ Read replica: RDS eu-west-1
└─ Replicated data: S3 eu-west-1
```

## Performance Benchmarks

### Target Metrics

#### API Response Times

List catalogs: < 100ms (p95)

Get table metadata: < 50ms (p95)

Create table: < 200ms (p95)

Search: < 500ms (p95)

#### Throughput

1,000 requests/second per API instance

10,000 concurrent users per deployment

#### Scale Limits

100,000 tables per catalog

1,000 catalogs per deployment

10,000 users per tenant

## Monitoring for Scale

## Key Performance Indicators

```
-- Track growth trends
SELECT
    DATE_TRUNC('day', created_at) as date,
```

```
COUNT(*) as new_tables,  
SUM(COUNT(*)) OVER (ORDER BY DATE_TRUNC('day', created_at)) as cumulative  
FROM assets  
GROUP BY DATE_TRUNC('day', created_at)  
ORDER BY date DESC  
LIMIT 30;
```

## Capacity Planning

### Forecast Resource Needs

Monitor growth rate (tables, users, queries)

Project 6-12 months ahead

Plan scaling actions proactively

### Alerting Thresholds

Database connections > 80% of pool

Disk usage > 70%

API latency p95 > 500ms

Error rate > 1%

## Cost Optimization at Scale

### Resource Efficiency

#### Right-Sizing

Start small, scale based on actual usage

Use auto-scaling to handle peaks

Scale down during off-hours

#### Storage Tiering

```
# Move old data to cheaper storage  
aws s3 cp s3://hot-bucket/old-data/ s3://archive-bucket/old-data/ \  
--storage-class GLACIER_IR
```

## Multi-Tenancy Economics

### Shared Resources

API servers: Shared across all tenants

Database: Shared with logical isolation

Monitoring: Centralized

### **Dedicated Resources (Premium Tier)**

Dedicated API instances

Isolated database

Priority support

## Scaling Checklist

### Before Scaling

- ☐ Identify bottleneck (CPU, memory, database, network)
- ☐ Review current resource utilization
- ☐ Analyze query patterns and hot spots
- ☐ Test scaling strategy in staging

### During Scaling

- ☐ Monitor metrics continuously
- ☐ Validate health checks
- ☐ Test failover scenarios
- ☐ Verify data consistency

### After Scaling

- ☐ Measure performance improvement
- ☐ Update capacity planning
- ☐ Document changes
- ☐ Review cost impact

## Additional Resources

[Deployment Best Practices](#)

[Performance Tuning Guide](#)

[Database Optimization](#)

Source: security.md

# Security Best Practices

## Overview

This guide covers security best practices for Pangolin deployments, from authentication to data protection.

## Authentication & Authorization

### JWT Token Security

#### Strong Secrets

```
# Generate cryptographically secure secret (256-bit)
openssl rand -base64 32

# Store in secrets manager, never in code
export PANGOLIN_JWT_SECRET=$(aws secretsmanager get-secret-value ...)
```

#### Token Expiration

```
# Set appropriate expiration (24 hours recommended)
PANGOLIN_JWT_EXPIRATION_HOURS=24

# For service users, use longer expiration with rotation
SERVICE_TOKEN_EXPIRATION_HOURS=720 # 30 days
```

#### Token Rotation

Rotate JWT secrets quarterly

Implement graceful rotation (support old + new for overlap period)

Force re-authentication after secret rotation

## Multi-Factor Authentication

#### For Admin Users

Require MFA for root and tenant-admin roles

Use time-based one-time passwords (TOTP)

Implement backup codes for account recovery

#### Implementation via OAuth

```
# Configure OAuth with MFA-enabled provider
PANGOLIN_OAUTH_PROVIDER=okta
PANGOLIN_OAUTH_REQUIRE_MFA=true
```

# Role-Based Access Control (RBAC)

## Principle of Least Privilege

### User Roles Hierarchy

```
Root (System Admin)
  ↓
Tenant Admin
  ↓
Tenant User
  ↓
Service User (API-only)
```

### Permission Assignment

```
# Grant minimum necessary permissions
pangolin-admin grant-permission \
  --user data-analyst \
  --catalog analytics \
  --namespace sales \
  --role READ

# Avoid wildcard grants in production
--namespace "*" # Too broad
--namespace "sales" # Specific
```

## Service Users

### API Key Management

```
# Create service user with specific permissions
pangolin-admin create-service-user \
  --name etl-pipeline \
  --description "Nightly ETL job"

# Rotate keys regularly (90 days)
pangolin-admin rotate-service-user-key --id <uuid>

# Revoke compromised keys immediately
pangolin-admin delete-service-user --id <uuid>
```

### Best Practices

One service user per application/pipeline

Document service user purpose

Monitor service user activity

Implement key rotation automation

## Network Security

### TLS/HTTPS

#### Enforce HTTPS

```
# Nginx configuration
server {
    listen 80;
    server_name pangolin.example.com;
    return 301 https://$server_name$request_uri;
}

server {
    listen 443 ssl http2;
    server_name pangolin.example.com;

    ssl_certificate /path/to/cert.pem;
    ssl_certificate_key /path/to/key.pem;
    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_ciphers HIGH:!aNULL:!MD5;

    location / {
        proxy_pass http://pangolin-api:8080;
    }
}
```

### Firewall Rules

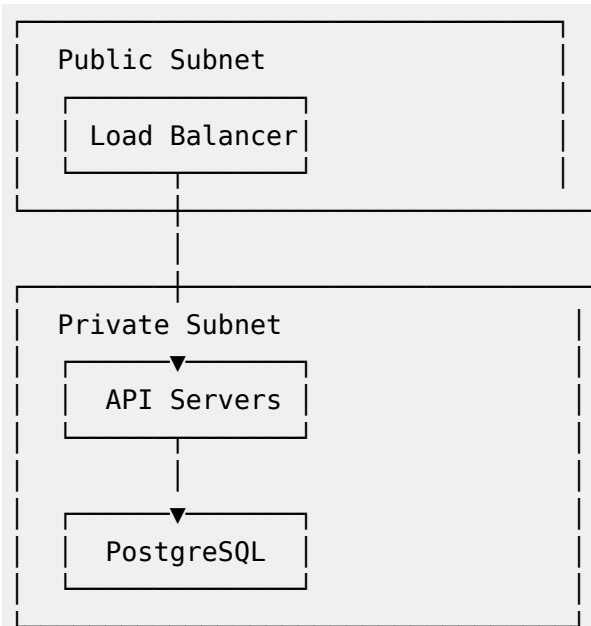
#### API Server

```
# Allow only necessary ports
- 443 (HTTPS) from load balancer
- 8080 (API) from internal network only
- Block all other inbound traffic
```

#### Database

```
# Restrict to API servers only
- 5432 (PostgreSQL) from API server IPs only
- 27017 (MongoDB) from API server IPs only
- Block public internet access
```

# VPC/Network Isolation



## Data Protection

### Encryption at Rest

#### Database Encryption

```
# PostgreSQL
# Enable transparent data encryption (TDE)
# Use encrypted EBS volumes for RDS

# MongoDB
# Enable encryption at rest
mongod --enableEncryption --encryptionKeyFile /path/to/keyfile
```

#### Object Storage Encryption

```
# S3 Server-Side Encryption
aws s3api put-bucket-encryption \
  --bucket my-pangolin-bucket \
  --server-side-encryption-configuration '{
    "Rules": [{
      "ApplyServerSideEncryptionByDefault": {
        "SSEAlgorithm": "AES256"
      }
    }]
  }'

# Or use KMS for key management
"SSEAlgorithm": "aws:kms",
"KMSMasterKeyID": "arn:aws:kms:..."
```

# Encryption in Transit

## Database Connections

```
# PostgreSQL with SSL
DATABASE_URL=postgresql://user:pass@host:5432/pangolin?sslmode=require

# MongoDB with TLS
mongodb://user:pass@host:27017/pangolin?tls=true
```

## Object Storage

Always use HTTPS endpoints

Verify SSL certificates

Use signed URLs for temporary access

# Credential Management

## Secrets Storage

### DO NOT

- ~Store secrets in environment files
- ~Commit secrets to version control
- ~Log secrets in application logs
- ~Pass secrets via command-line arguments

### DO

- ~Use secrets manager (AWS Secrets Manager, Vault, etc.)
- ~Rotate secrets regularly
- ~Audit secret access
- ~Use IAM roles when possible

# Credential Vending

## S3 Temporary Credentials

```
# Vend short-lived STS credentials
credentials = sts.assume_role(
    RoleArn='arn:aws:iam::account:role/pangolin-data-access',
    DurationSeconds=3600 # 1 hour
)

# Return to client for direct S3 access
```

```
return {  
  'access_key_id': credentials['AccessKeyId'],  
  'secret_access_key': credentials['SecretAccessKey'],  
  'session_token': credentials['SessionToken'],  
  'expiration': credentials['Expiration']  
}
```

## Best Practices

Minimum credential lifetime (1-12 hours)

Scope credentials to specific resources

Monitor credential usage

Revoke on suspicious activity

## Audit Logging

### Comprehensive Logging

#### Log All Security Events

```
{  
  "event": "authentication_failure",  
  "timestamp": "2025-12-24T20:00:00Z",  
  "user": "john.doe",  
  "ip_address": "192.168.1.100",  
  "reason": "invalid_password",  
  "attempts": 3  
}
```

#### Critical Events to Log

Authentication (success/failure)

Authorization decisions

Permission changes

User creation/deletion

Service user key rotation

Data access (read/write)

Configuration changes

## Log Retention

### Compliance Requirements

Security logs: 1-7 years (depending on industry)

Audit logs: 90 days minimum

Access logs: 30 days minimum

## Storage

Use immutable storage (S3 Object Lock, WORM)

Encrypt logs at rest

Implement log integrity verification

## Monitoring & Alerting

### Security Alerts

alerts:

- name: Multiple Failed Logins  
condition: failed\_login\_count > 5 in 5 minutes  
action: notify\_security\_team
- name: Privilege Escalation  
condition: role\_change to tenant-admin or root  
action: notify\_security\_team
- name: Unusual Access Pattern  
condition: access\_from\_new\_country  
action: require\_mfa\_verification
- name: Service User Key Rotation Overdue  
condition: key\_age > 90 days  
action: notify\_admin

## Vulnerability Management

### Dependency Scanning

```
# Scan Rust dependencies
cargo audit

# Scan Docker images
docker scan your-registry/pangolin-api:latest

# Scan infrastructure as code
checkov -d ./terraform
```

### Patch Management

## Update Schedule

Critical security patches: Within 24 hours

High severity: Within 7 days

Medium severity: Within 30 days

Low severity: Next maintenance window

## Testing Process

Test in development environment

Deploy to staging

Run security tests

Deploy to production with rollback plan

# Compliance

## Data Residency

### Geographic Restrictions

```
# Deploy in specific regions
AWS_REGION=eu-west-1 # EU data stays in EU
AZURE_LOCATION=westeurope

# Configure cross-region replication carefully
# Ensure compliance with GDPR, CCPA, etc.
```

## Data Classification

### Sensitivity Levels

```
# Tag assets with sensitivity
{
  "table": "customers.personal_data",
  "classification": "PII",
  "retention_policy": "7_years",
  "encryption_required": true,
  "access_logging": "mandatory"
}
```

## Access Reviews

### Quarterly Reviews

```
# Generate access report
```

```
pangolin-admin list-permissions --format csv > access_review_Q4_2025.csv

# Review and revoke unnecessary permissions
pangolin-admin revoke-permission --user inactive-user --catalog analytics
```

# Incident Response

## Security Incident Playbook

### 1. Detection

Monitor alerts

Review audit logs

User reports

### 2. Containment

```
# Immediately revoke compromised credentials
pangolin-admin revoke-token-by-id --id <token-id>

# Disable compromised user
pangolin-admin update-user --id <user-id> --active false

# Rotate affected secrets
pangolin-admin rotate-service-user-key --id <service-user-id>
```

### 3. Investigation

```
# Review audit logs
pangolin-admin list-audit-events \
  --user <compromised-user> \
  --start-date 2025-12-01 \
  --end-date 2025-12-24

# Check for unauthorized access
pangolin-admin list-audit-events \
  --action "READ" \
  --resource-type "TABLE" \
  --user <compromised-user>
```

### 4. Recovery

Restore from clean backup if needed

Reset all affected credentials

Apply security patches

### 5. Post-Incident

Document incident

Update security procedures

Conduct team training

## Security Checklist

### Deployment

☐ HTTPS/TLS enabled

☐ Strong JWT secrets configured

☐ Secrets stored in secrets manager

☐ Database encryption enabled

☐ Firewall rules configured

☐ VPC/network isolation implemented

### Operations

☐ Audit logging enabled

☐ Security monitoring configured

☐ Alerts set up

☐ Backup encryption enabled

☐ Incident response plan documented

### Maintenance

☐ Quarterly access reviews

☐ Regular security patches

☐ Dependency scanning

☐ Penetration testing (annual)

☐ Secret rotation (quarterly)

## Additional Resources

[Permissions Management Best Practices](#)

[Audit Logging Guide](#)

[Deployment Security](#)

Source: README.md

# Pangolin CLI Documentation

## Overview

The Pangolin CLI provides command-line tools for managing your Pangolin data catalog infrastructure.

## CLI Tools

**`pangolin-admin`** - Administrative tool for managing tenants, warehouses, catalogs, users, and permissions

**`pangolin-user`** - User-focused tool for data discovery and catalog operations

## Quick Start

## Installation

```
# Build from source
cd pangolin
cargo build --release --bin pangolin-admin

# The binary will be at: target/release/pangolin-admin
```

## First Steps

```
# 1. Login as root
pangolin-admin login --username admin --password password

# 2. Create a tenant
pangolin-admin create-tenant --name my-company \
  --admin-username tenant-admin \
  --admin-password secure-password

# 3. Login as tenant admin
pangolin-admin login --username tenant-admin --password secure-password

# 4. Create a warehouse
pangolin-admin create-warehouse my-warehouse --type s3

# 5. Create a catalog
pangolin-admin create-catalog my-catalog --warehouse my-warehouse
```

## Documentation Index

# Admin Tool (`pangolin-admin`)

[Admin Overview](#) - Complete admin tool reference

[Tenant Management](#) - Creating and managing tenants

[User Management](#) - User creation and administration

[Warehouse Management](#) - Storage backend configuration

[Catalog Management](#) - Local catalog operations

[Federated Catalogs](#) - External catalog integration

[Permission Management](#) - RBAC and access control

[Service Users](#) - API keys and service accounts

[Token Management](#) - User token operations

[Audit Logging](#) - Viewing and analyzing audit events

[Merge Operations](#) - Managing branch merges

[Update Operations](#) - Updating resources

[Metadata Management](#) - Business metadata operations

[Optimization Commands](#) - Performance and maintenance

# User Tool (`pangolin-user`)

[User Overview](#) - Complete user tool reference

[Branch Management](#) - Creating and managing branches

[Tag Management](#) - Versioning with tags

[Discovery](#) - Data discovery and search

[Access Requests](#) - Requesting permissions

[Token Management](#) - Personal token management

# Configuration & Setup

[Overview](#) - CLI tools overview

[Configuration](#) - CLI configuration files

[Docker Usage](#) - Running CLI in Docker

[Warehouse Management](#) - Multi-cloud warehouse setup

# Common Tasks

# Warehouse Operations

```
# Create S3 warehouse
pangolin-admin create-warehouse prod-s3 --type s3 \
  --bucket my-bucket \
  --access-key AKIA... \
  --secret-key ... \
  --region us-east-1

# Create Azure warehouse
pangolin-admin create-warehouse prod-azure --type azure

# Create GCS warehouse
pangolin-admin create-warehouse prod-gcs --type gcs

# List warehouses
pangolin-admin list-warehouses

# Update warehouse credentials
pangolin-admin update-warehouse --id <uuid>

# Delete warehouse
pangolin-admin delete-warehouse my-warehouse
```

# Catalog Operations

```
# Create catalog
pangolin-admin create-catalog my-catalog --warehouse my-warehouse

# List catalogs
pangolin-admin list-catalogs

# Delete catalog
pangolin-admin delete-catalog my-catalog
```

# User Management

```
# Create user
pangolin-admin create-user john.doe \
  --email john@example.com \
  --role tenant-user

# List users
pangolin-admin list-users

# Update user
pangolin-admin update-user --id <uuid> --active false
```

# Interactive Mode

The CLI supports an interactive REPL mode:

```
# Start interactive mode
pangolin-admin

# You'll see a prompt:
(admin:username@tenant)>

# Type commands without the 'pangolin-admin' prefix:
(admin:username@tenant)> list-warehouses
(admin:username@tenant)> create-catalog my-catalog --warehouse my-warehouse
(admin:username@tenant)> exit
```

## Environment Variables

```
# Set Pangolin API URL
export PANGOLIN_URL="http://localhost:8080"

# Set tenant context
export PANGOLIN_TENANT="my-tenant"

# Use in commands
pangolin-admin list-warehouses
```

## Configuration Profiles

The CLI saves authentication state in `~/.pangolin/config.json`:

```
{
  "base_url": "http://localhost:8080",
  "username": "admin",
  "token": "...",
  "tenant_id": "...",
  "tenant_name": "my-tenant"
}
```

Use multiple profiles:

```
# Use specific profile
pangolin-admin --profile production list-warehouses

# Switch profiles
pangolin-admin --profile staging login --username admin
```

## Getting Help

```
# General help
pangolin-admin help

# Command-specific help
pangolin-admin create-warehouse --help

# List all commands
pangolin-admin help | grep " "
```

## Troubleshooting

### Connection Issues

```
# Check API is running
curl http://localhost:8080/health

# Verify URL
echo $PANGOLIN_URL

# Test with explicit URL
pangolin-admin --url http://localhost:8080 list-tenants
```

### Authentication Issues

```
# Clear saved credentials
rm ~/.pangolin/config.json

# Login again
pangolin-admin login --username admin --password password
```

### Permission Issues

```
# Check current user context
pangolin-admin list-users # Shows current user

# Verify tenant context
# Root users must use 'use' command to switch tenants
pangolin-admin use my-tenant
```

## Examples

See individual documentation pages for detailed examples:

[Warehouse Management Examples](#)

[Multi-Cloud Setup](#)

[Federated Catalog Setup](#)

## Support

**Issues:** [GitHub Issues](#)

**Documentation:** [Full Documentation](#)

**Architecture:** [Architecture Docs](#)

## Version

Current CLI version: 0.1.0

For API compatibility, see [API Documentation](#).

## Source: admin-audit-logging.md

## Admin CLI - Audit Logging

This guide covers the audit logging commands available in the Pangolin Admin CLI. These commands allow administrators to track user actions, resource changes, and security events across the platform.

### List Audit Events

List audit events with support for comprehensive filtering and pagination.

### Syntax

```
pangolin-admin list-audit-events [OPTIONS]
```

### Parameters

``user-id <UUID>`` - Filter by user ID

``action <STRING>`` - Filter by action (e.g., ``create_table``, ``grant_permission``)

``resource-type <STRING>`` - Filter by resource type (e.g., ``table``, ``catalog``, ``user``)

``result <STRING>`` - Filter by result (``success`` or ``failure``)

``tenant-id <UUID>`` - Filter by tenant ID (Root users only - see cross-tenant queries)

``limit <NUMBER>`` - Maximum number of results to return (default: 50)

# Examples

## List the 50 most recent events:

```
pangolin-admin list-audit-events
```

## Filter by action and limit results:

```
pangolin-admin list-audit-events --action create_table --limit 10
```

## Filter by user and failed actions:

```
pangolin-admin list-audit-events --user-id "550e8400-e29b-41d4-a716-446655440000"
--result failure
```

## Filter by tenant (Root users only):

```
# View audit events for a specific tenant
pangolin-admin list-audit-events --tenant-id "tenant-uuid-here" --limit 20
```

# Output

ID	User	Action	Resource Type	Resource Name	Result
Timestamp					
ab12cd34	admin	create_table	table	sales_data	success
2025-12-18T10:30:00					
ef56gh78	user1	drop_table	table	old_data	failure
2025-12-18T10:35:00					
...					
Showing 2 events					

# Count Audit Events

Get the total count of audit events matching specific criteria.

# Syntax

```
pangolin-admin count-audit-events [OPTIONS]
```

# Parameters

``user-id <UUID>`` - Filter by user ID

``action <STRING>`` - Filter by action

``resource-type <STRING>`` - Filter by resource type

``result <STRING>`` - Filter by result

## Examples

### Get total event count:

```
pangolin-admin count-audit-events
```

### Count failure events:

```
pangolin-admin count-audit-events --result failure
```

## Output

```
Total audit events: 1542
```

## Get Audit Event

Retrieve full details for a specific audit event by its ID.

## Syntax

```
pangolin-admin get-audit-event --id <UUID>
```

## Parameters

``id <UUID>`` - The unique identifier of the audit event (required)

## Example

```
pangolin-admin get-audit-event --id "ab12cd34-e29b-41d4-a716-446655440000"
```

## Output

---

Audit Event Details

---

ID: ab12cd34-e29b-41d4-a716-446655440000

User: admin

User ID: 550e8400-e29b-41d4-a716-446655440000

Action: create\_table

```
Resource: sales_data (table)
Result: success
Timestamp: 2025-12-18T10:30:00.123456Z
IP Address: 192.168.1.100
User Agent: pangolin-cli/0.1.0
Metadata:
  schema: public
  format: parquet
```

---

## Use Cases

### Investigation

**Identify User Activity:** Filter by ``user_id`` to see all actions performed by a specific user.

**Trace Resource Changes:** Filter by ``resource_type`` and ``resource_name`` (via `grep`) to see history of a specific asset.

**Security Auditing:** key filter ``result=failure`` to identify potential unauthorized access attempts or system errors.

### Compliance

Export audit logs regularly using ``list-audit-events`` and piping to a file.

Verify actions match approved change requests.

### Notes

#### **Tenant Isolation:**

Tenant users only see audit logs for their own tenant

Root users can see all audit logs across all tenants

Use ``tenant-id`` filter to query specific tenant logs as Root

**Retention:** Audit logs are retained indefinitely by default (configurable per deployment).

**Performance:** Pagination (``limit`,`offset``) is recommended for large datasets.

Source: admin-catalogs.md

## Admin Catalog Management

Catalogs provide a logical namespace (Iceberg Catalog) on top of a physical

Warehouse.

# Commands

## List Catalogs

View all catalogs.

### Syntax:

```
pangolin-admin list-catalogs [--limit <N>] [--offset <N>]
```

### Options:

`^limit``: Maximum number of items to return (default: 100)

`^offset``: Number of items to skip (default: 0)

## Create Catalog

Create a new catalog backed by an existing warehouse.

[!IMPORTANT]

This command requires **Tenant Admin** privileges. The Root User cannot create catalogs.

### Syntax:

```
pangolin-admin create-catalog <name> --warehouse <warehouse_name>
```

### Example:

```
pangolin-admin create-catalog sales_catalog --warehouse main_lake
```

## Update Catalog

Modify a catalog's configuration.

### Syntax:

```
pangolin-admin update-catalog --id <uuid> [--name <new_name>]
```

### Example:

```
pangolin-admin update-catalog --id "catalog-uuid" --name "archived-sales"
```

## Delete Catalog

Delete a catalog.

**Syntax:**

```
pangolin-admin delete-catalog <name>
```

**Example:**

```
pangolin-admin delete-catalog temp_catalog
```

Source: admin-federated-catalogs.md

## Federated Catalogs (Admin CLI)

Manage federated catalogs that proxy requests to other Iceberg REST catalogs.

### Create Federated Catalog

Create a federated catalog that proxies to a remote Iceberg REST catalog:

```
pangolin-admin create-federated-catalog fed_catalog_b \  
-P uri=http://remote-server:8080/v1/catalog_b \  
--storage-location s3://my-bucket/fed_catalog_b \  
-P token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9... \  
--timeout 30
```

**Parameters:**

``name`` (required): Name of the federated catalog

``storage-location`` (required): S3/Azure/GCS path (required even for federated catalogs)

``-P` / `property`` (optional): Key-value properties for configuration (e.g., ``-P key=value``)

``timeout`` (optional): Timeout in seconds (default: 30)

### Common Properties:

``uri``: Base URL of the remote catalog (Required)

``token``: Bearer token for authentication

``credential``: Client credential (id:secret) for OAuth/Basic Auth

``warehouse``: Warehouse name to use

### Authentication Examples

## No Authentication

```
pangolin-admin create-federated-catalog public_catalog \  
-P uri=http://public-server:8080/v1/catalog \  
--storage-location s3://my-bucket/public_catalog
```

## Bearer Token

```
pangolin-admin create-federated-catalog secure_catalog \  
-P uri=http://secure-server:8080/v1/catalog \  
--storage-location s3://my-bucket/secure_catalog \  
-P token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

## Basic Auth / Client Credentials

```
pangolin-admin create-federated-catalog basic_catalog \  
-P uri=http://basic-server:8080/v1/catalog \  
--storage-location s3://my-bucket/basic_catalog \  
-P credential=admin:secret123
```

## API Key

```
pangolin-admin create-federated-catalog api_catalog \  
-P uri=http://api-server:8080/v1/catalog \  
--storage-location s3://my-bucket/api_catalog \  
-P x-api-key=abc123def456
```

## List Federated Catalogs

List all federated catalogs in the current tenant:

```
pangolin-admin list-federated-catalogs
```

### Example Output:

Name	URI	Properties
fed_catalog_b	http://remote:8080/v1/catalog_b	3 props
fed_catalog_c	http://other:8080/v1/catalog_c	2 props

## Test Federated Catalog

Test connectivity and authentication to a federated catalog:

```
pangolin-admin test-federated-catalog fed_catalog_b
```

### Example Output:

```
Testing connection to federated catalog 'fed_catalog_b'...
```

1. Checking catalog configuration...
2. Testing namespace listing...

---

```
Federated catalog 'fed_catalog_b' is working correctly!
```

---

```
Namespaces accessible: ['db_a', 'db_b']
```

## Delete Federated Catalog

Delete a federated catalog.

### Syntax:

```
pangolin-admin delete-federated-catalog <name>
```

### Example:

```
pangolin-admin delete-federated-catalog fed_catalog_b
```

## Operations

### Sync Federated Catalog

Trigger a manual synchronization of metadata from the remote catalog.

### Syntax:

```
pangolin-admin sync-federated-catalog <name>
```

### Example:

```
pangolin-admin sync-federated-catalog fed_catalog_b
```

### Get Federated Stats

View synchronization statistics and health metrics for a federated catalog.

### Syntax:

```
pangolin-admin get-federated-stats <name>
```

### Example:

```
pangolin-admin get-federated-stats fed_catalog_b
```

## Use Cases

### Cross-Tenant Data Sharing

Tenant A can access Tenant B's data through a federated catalog:

Tenant B creates a catalog and grants access to Tenant A

Tenant B generates a token for Tenant A

Tenant A creates a federated catalog pointing to Tenant B's catalog

Tenant A's users can now query Tenant B's data

### Multi-Region Data Access

Access data from different regions through federated catalogs:

```
# Create federated catalog for US region
pangolin-admin create-federated-catalog us_catalog \
  -P uri=http://us-server:8080/v1/catalog \
  --storage-location s3://my-bucket/us_catalog \
  -P token=$US_TOKEN

# Create federated catalog for EU region
pangolin-admin create-federated-catalog eu_catalog \
  -P uri=http://eu-server:8080/v1/catalog \
  --storage-location s3://my-bucket/eu_catalog \
  -P token=$EU_TOKEN
```

## Hybrid Cloud Setup

Federate catalogs across different cloud providers:

```
# AWS catalog
pangolin-admin create-federated-catalog aws_catalog \
  -P uri=http://aws-pangolin:8080/v1/catalog \
  --storage-location s3://aws-bucket/catalog \
  -P token=$AWS_TOKEN

# Azure catalog
pangolin-admin create-federated-catalog azure_catalog \
  -P uri=http://azure-pangolin:8080/v1/catalog \
```

```
--storage-location abfs://azure-container/catalog \  
-P token=$AZURE_TOKEN
```

## Troubleshooting

### Connection Timeout

If the test fails with a timeout:

```
```bash  
# Increase timeout  
pangolin-admin create-federated-catalog slow_catalog \  
-P uri=http://slow-server:8080/v1/catalog \  
--storage-location s3://my-bucket/slow_catalog \  
--timeout 60
```

### Authentication Errors

If you get 401/403 errors:

Verify the token is still valid

Check that the token has the correct permissions

Ensure the auth type matches the remote server's requirements

### Network Issues

If you get connection refused:

Verify the base URL is correct

Check network connectivity to the remote server

Ensure the remote server is running and accessible

Source: [admin-merge-operations.md](#)

## Admin CLI - Merge Operations

This guide covers the complete merge workflow for branch management in Pangolin.

### Overview

Merge operations allow you to merge changes from one branch into another. The workflow includes:

- List merge operations
- Check for conflicts
- Resolve conflicts
- Complete or abort the merge

# List Merge Operations

List all merge operations in the current catalog.

## Syntax

```
pangolin-admin list-merge-operations
```

## Parameters

None

## Example

```
pangolin-admin list-merge-operations
```

## Output

ID	Source Branch	Target Branch	Status
Created At			
merge-uuid-1	feature-branch	main	pending
2025-12-18T10:30:00			

## Notes

- Shows all merge operations for the current catalog
- Status can be: pending, in-progress, completed, aborted
- Use merge ID for subsequent operations

# Get Merge Operation

Get detailed information about a specific merge operation.

## Syntax

```
pangolin-admin get-merge-operation --id <MERGE_ID>
```

## Parameters

``id`` - UUID of the merge operation (required)

## Example

```
pangolin-admin get-merge-operation --id "merge-uuid"
```

## Output

Merge Operation Details
ID: merge-uuid
Source Branch: feature-branch
Target Branch: main
Status: pending
Created At: 2025-12-18T10:30:00

## Notes

- Provides full details of the merge operation
- Includes timestamps and status
- Shows completed timestamp if merge is finished

## List Conflicts

List all conflicts in a merge operation.

## Syntax

```
pangolin-admin list-conflicts --merge-id <MERGE_ID>
```

## Parameters

``merge-id`` - UUID of the merge operation (required)

## Example

```
pangolin-admin list-conflicts --merge-id "merge-uuid"
```

## Output

ID	Asset ID	Conflict Type	Resolved
conflict-uuid-1	table-1	schema	No
conflict-uuid-2	table-2	data	Yes

## Notes

- Shows all conflicts that need resolution
- Conflict types: schema, data, metadata
- Resolved status indicates if conflict has been addressed

## Resolve Conflict

Resolve a specific merge conflict.

## Syntax

```
pangolin-admin resolve-conflict \  
  --merge-id <MERGE_ID> \  
  --conflict-id <CONFLICT_ID> \  
  --resolution <RESOLUTION_STRATEGY>
```

## Parameters

- ``merge-id`` - UUID of the merge operation (required)
- ``conflict-id`` - UUID of the conflict (required)
- ``resolution`` - Resolution strategy (required)

## Resolution Strategies

-

`use-source` - Use changes from source branch

`use-target` - Keep target branch version

`manual` - Manual resolution (requires additional steps)

## Examples

```
# Use source branch changes
pangolin-admin resolve-conflict \
  --merge-id "merge-uuid" \
  --conflict-id "conflict-uuid" \
  --resolution "use-source"
```

```
# Keep target branch version
pangolin-admin resolve-conflict \
  --merge-id "merge-uuid" \
  --conflict-id "conflict-uuid" \
  --resolution "use-target"
```

## Output

```
Conflict resolved successfully!
```

## Notes

Must resolve all conflicts before completing merge

Resolution is permanent once applied

Use `list-conflicts` to verify all are resolved

## Complete Merge

Complete a merge operation after all conflicts are resolved.

## Syntax

```
pangolin-admin complete-merge --id <MERGE_ID>
```

## Parameters

`id` - UUID of the merge operation (required)

## Example

```
pangolin-admin complete-merge --id "merge-uuid"
```

## Output

```
Merge completed successfully!
```

## Notes

- All conflicts must be resolved first
- Finalizes the merge into target branch
- Cannot be undone after completion
- Target branch will contain merged changes

## Abort Merge

Abort a merge operation without applying changes.

## Syntax

```
pangolin-admin abort-merge --id <MERGE_ID>
```

## Parameters

`--id`` - UUID of the merge operation (required)

## Example

```
pangolin-admin abort-merge --id "merge-uuid"
```

## Output

```
Merge aborted successfully!
```

## Notes

- Cancels the merge operation
- No changes applied to target branch

-

Useful if conflicts are too complex

Can start a new merge later

## Complete Merge Workflow

### 1. Initiate Merge

```
# Create merge operation (via merge-branch command)
pangolin-admin merge-branch \
  --source "feature-branch" \
  --target "main"
```

### 2. Check Status

```
# List merge operations
pangolin-admin list-merge-operations

# Get specific merge details
pangolin-admin get-merge-operation --id "merge-uuid"
```

### 3. Handle Conflicts

```
# List conflicts
pangolin-admin list-conflicts --merge-id "merge-uuid"

# Resolve each conflict
pangolin-admin resolve-conflict \
  --merge-id "merge-uuid" \
  --conflict-id "conflict-1" \
  --resolution "use-source"

pangolin-admin resolve-conflict \
  --merge-id "merge-uuid" \
  --conflict-id "conflict-2" \
  --resolution "use-target"
```

### 4. Complete or Abort

```
# If all conflicts resolved, complete
pangolin-admin complete-merge --id "merge-uuid"

# Or abort if needed
pangolin-admin abort-merge --id "merge-uuid"
```

# Advanced Workflows

## Automated Merge Script

```
#!/bin/bash
MERGE_ID="$1"

# Check for conflicts
CONFLICTS=$(pangolin-admin list-conflicts --merge-id "$MERGE_ID" | grep "No" | wc -l)

if [ "$CONFLICTS" -eq 0 ]; then
    # No unresolved conflicts, complete merge
    pangolin-admin complete-merge --id "$MERGE_ID"
else
    echo "⚠ $CONFLICTS unresolved conflicts. Manual resolution required."
    pangolin-admin list-conflicts --merge-id "$MERGE_ID"
fi
```

## Conflict Resolution Strategy

```
#!/bin/bash
# Always use source for schema conflicts, target for data conflicts

MERGE_ID="$1"

# Get conflicts
pangolin-admin list-conflicts --merge-id "$MERGE_ID" > /tmp/conflicts.txt

# Parse and resolve
while read -r line; do
    CONFLICT_ID=$(echo "$line" | awk '{print $1}')
    CONFLICT_TYPE=$(echo "$line" | awk '{print $3}')

    if [ "$CONFLICT_TYPE" == "schema" ]; then
        pangolin-admin resolve-conflict \
            --merge-id "$MERGE_ID" \
            --conflict-id "$CONFLICT_ID" \
            --resolution "use-source"
    else
        pangolin-admin resolve-conflict \
            --merge-id "$MERGE_ID" \
            --conflict-id "$CONFLICT_ID" \
            --resolution "use-target"
    fi
done < /tmp/conflicts.txt
```

## Error Handling

# Common Errors

## Merge Not Found:

Error: Failed to get merge operation (404): Merge operation not found

Solution: Verify merge ID is correct

## Conflicts Not Resolved:

Error: Failed to complete merge (400): Unresolved conflicts remain

Solution: Resolve all conflicts first

## Invalid Resolution:

Error: Failed to resolve conflict (400): Invalid resolution strategy

Solution: Use valid strategy (use-source, use-target, manual)

# Best Practices

**Check Conflicts First:** Always list conflicts before resolving

**Resolve Systematically:** Resolve conflicts one by one

**Verify Resolution:** List conflicts again to confirm all resolved

**Test Before Complete:** Verify changes before completing merge

**Document Decisions:** Keep notes on resolution strategies

**Use Abort Wisely:** Don't hesitate to abort complex merges

# Related Commands

`^merge-branch`` - Initiate a merge operation

`^create-branch`` - Create a new branch

`^list-branches`` - List all branches

`^list-commits`` - View commit history

# Conflict Types

## Schema Conflicts

Column additions/removals

Data type changes

Constraint modifications

**Recommendation:** Usually use source (newer schema)

## Data Conflicts

Overlapping data changes

Concurrent updates

**Recommendation:** Case-by-case basis

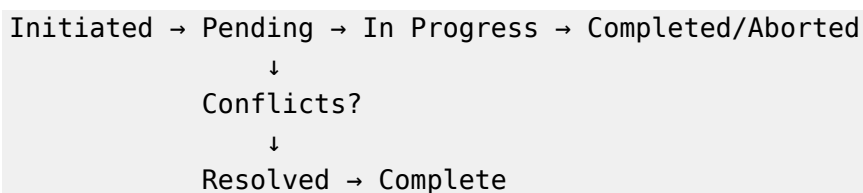
## Metadata Conflicts

Property changes

Tag modifications

**Recommendation:** Usually use source

## Merge States



## Tips

**Small Merges:** Merge frequently to avoid large conflicts

**Communication:** Coordinate with team before merging

**Backup:** Consider backing up before complex merges

**Testing:** Test merged changes in development first

**Documentation:** Document merge decisions for audit trail

Source: [admin-metadata.md](#)

## Admin Metadata Management

Business metadata allows you to attach technical and organizational context to any entity (Catalog, Warehouse, Table, View).

# Commands

## Get Metadata

Retrieve the metadata JSON attached to an entity.

### Syntax:

```
pangolin-admin get-metadata --entity-type <type> --entity-id <id>
```

**Entity Types:** `catalog`, `warehouse`, `table`, `view`.

### Example:

```
pangolin-admin get-metadata --entity-type table --entity-id sales.orders
```

## Set Metadata

Upsert a key-value pair into an entity's metadata.

### Syntax:

```
pangolin-admin set-metadata --entity-type <type> --entity-id <id> <key> <value>
```

### Example:

```
pangolin-admin set-metadata --entity-type table --entity-id sales.orders owner "Data Engineering"
```

## Delete Metadata

Remove all business metadata from an asset.

### Syntax:

```
pangolin-admin delete-metadata --asset-id <uuid>
```

### Example:

```
pangolin-admin delete-metadata --asset-id "table-uuid"
```

Source: admin-optimization-commands.md

## CLI Optimization Commands

This guide covers the 5 new optimization commands added to `pangolin-admin` for improved performance and usability.

# Overview

These commands provide efficient access to dashboard statistics, catalog summaries, asset search, bulk operations, and name validation without requiring multiple API calls.

# Commands

## 1. `stats` - Dashboard Statistics

Get comprehensive dashboard statistics in a single API call.

**Usage:**

```
pangolin-admin stats
```

**Output:**

```
Dashboard Statistics (Scope: system)
=====
Tenants:      3
Catalogs:     6
Warehouses:   2
Namespaces:   15
Tables:       42
Users:        5
Branches:     8
```

**Note:** `Tenants` count is only displayed for Root users with system-wide scope.

**Scope:** Results are scoped based on your role:

**Root:** System-wide statistics

**Tenant Admin:** Tenant-specific statistics

**Tenant User:** User-accessible resources only

## 2. `catalog-summary` - Catalog Summary

Get detailed summary for a specific catalog.

**Usage:**

```
pangolin-admin catalog-summary <catalog_name>
```

**Example:**

### Output:

```
Catalog: production
Namespaces: 12
Tables:    45
Branches:  3
Storage:    s3://my-bucket/warehouse/production
```

## 3. `search` - Asset Search

Search for assets across catalogs with optional filtering.

### Usage:

```
pangolin-admin search <query> [--catalog <name>] [--limit <n>]
```

### Examples:

```
# Basic search
pangolin-admin search "sales"

# Search within specific catalog
pangolin-admin search "customer" --catalog production

# Limit results
pangolin-admin search "analytics" --limit 20
```

### Output:

```
Found 5 results:
production.analytics.sales_data
production.analytics.customer_metrics
production.default.sales_summary
staging.analytics.sales_test
dev.analytics.sales_dev
```

### Features:

Full-text search across asset names

Catalog filtering

Pagination support

URL encoding for special characters

## 4. `bulk-delete` - Bulk Asset Deletion

Delete multiple assets in a single operation.

### Usage:

```
pangolin-admin bulk-delete --ids <uuid1,uuid2,...> [--confirm]
```

### Examples:

```
# Interactive confirmation
pangolin-admin bulk-delete --ids "123e4567-e89b-12d3-a456-426614174000,987fcdeb-51a2-43f7-8b9c-0123456789ab"

# Skip confirmation
pangolin-admin bulk-delete --ids "uuid1,uuid2" --confirm
```

### Output:

```
About to delete 2 assets:
- 123e4567-e89b-12d3-a456-426614174000
- 987fcdeb-51a2-43f7-8b9c-0123456789ab
Continue? (y/N): y

Bulk delete completed:
Succeeded: 2
Failed: 0
```

### Error Handling:

```
Bulk delete completed:
Succeeded: 3
Failed: 2
Errors:
- Asset not found: uuid1
- Permission denied: uuid2
```

## 5. `validate` - Name Validation

Validate resource name availability before creation.

### Usage:

```
pangolin-admin validate <resource_type> <name1> [name2...]
```

### Resource Types:

`catalog`

`warehouse`

### Examples:

```
# Validate single catalog name
pangolin-admin validate catalog new_production

# Validate multiple names
pangolin-admin validate catalog prod_v2 staging_v2 dev_v2

# Validate warehouse names
pangolin-admin validate warehouse s3_warehouse azure_warehouse
```

## Output:

```
Name validation results:
  new_production: ✓ Available
  existing_catalog: x Taken
    Reason: Name already exists
  invalid-name: x Taken
    Reason: Invalid name format
```

## Performance Benefits

Operation	Before	After	Improvement
Dashboard Load	5-10 API calls	1 API call	<b>5-10x faster</b>
Catalog Overview	3-4 API calls	1 API call	<b>3-4x faster</b>
Asset Search	Client-side filtering	Server-side filtering	<b>Reduced bandwidth</b>
Name Validation	Sequential checks	Batch validation	<b>N requests → 1 request</b>

## Configuration

### Environment Variables

```
# Set API URL (default: http://localhost:8081)
export PANGOLIN_URL=http://localhost:8080

# Set tenant context
export PANGOLIN_TENANT=my_tenant
```

## Using with Profiles

```
# Use specific profile
pangolin-admin --profile production stats

# Override URL
pangolin-admin --url http://api.example.com stats
```

# Interactive Mode

All commands work in both interactive REPL mode and non-interactive mode:

```
# Start interactive mode
pangolin-admin

# Then use commands without the prefix
(admin:user@tenant)> stats
(admin:user@tenant)> catalog-summary production
(admin:user@tenant)> search "sales" --limit 10
```

## Error Handling

All commands provide clear error messages:

```
# Catalog not found
$ pangolin-admin catalog-summary nonexistent
Error: Failed to get catalog summary: 404 Not Found

# Invalid resource type
$ pangolin-admin validate invalid_type some_name
Error: Invalid value 'invalid_type' for '<RESOURCE_TYPE>'
[possible values: catalog, warehouse]

# API connection error
$ pangolin-admin stats
Error: API Request Failed: Connection refused
```

## Best Practices

**Use `stats` for dashboards:** Replace multiple API calls with a single stats command

**Validate before creating:** Always validate names before attempting to create resources

**Use catalog filters:** Narrow search results by specifying catalog names

**Confirm bulk operations:** Review the list before confirming bulk deletes

**Set environment variables:** Configure `PANGOLIN\_URL` to avoid repetitive `url` flags

## See Also

[Admin CLI Overview](#)

[API Reference](#)

[Performance Optimizations](#)

Source: admin-permissions.md

## Admin Permissions Management

The permissions system in Pangolin allows for fine-grained access control over resources. Permissions are granted to **Roles**, and Roles are assigned to Users.

### Concepts

#### Roles

Pangolin uses a hybrid RBAC system:

**System Roles** (Fixed): ``Root``, ``TenantAdmin``, ``TenantUser``.

**Dynamic Roles** (Custom): User-defined collections of permissions (e.g., ``data_engineer``, ``analyst``).

#### Actions

``READ``: View metadata and read data.

``WRITE``: Modify metadata and write data.

``DELETE``: Delete resources.

``CREATE``: Create new resources.

``LIST``: List resources.

``MANAGE_ACCESS``: Grant/Revoke permissions.

``MANAGE_DISCOVERY``: Manage business metadata.

#### Resources

Resources are hierarchical strings.

**System**: ``system`` (Global control)

**Catalog**: ``catalog:{catalog_name}``

**Namespace**: ``namespace:{catalog_name}:{namespace_name}``

**Table**: ``table:{catalog_name}:{namespace_name}:{table_name}``

**Tag/Attribute**: ``tag:{name}`` or ``attribute:{key}``

#### Commands

# List Permissions

List all active permissions, optionally filtered.

## Syntax:

```
pangolin-admin list-permissions [--role <role>] [--user <user>]
```

## Examples:

```
# List all permissions in the system
pangolin-admin list-permissions

# See what the 'analyst' role can do
pangolin-admin list-permissions --role analyst
```

# Grant Permission

Grant a specific action on a resource to a **User**.

## Syntax:

```
pangolin-admin grant-permission <username> <action> <resource>
```

## Examples:

```
# Grant 'admin_user' read access to the 'sales' catalog
pangolin-admin grant-permission admin_user read catalog:sales

# Grant 'data_engineer' write access to a specific namespace
pangolin-admin grant-permission data_engineer write namespace:sales:region_us

# Allow 'audit_bot' to read tables tagged with 'compliance'
pangolin-admin grant-permission audit_bot read tag:compliance
```

# Revoke Permission

Remove a previously granted permission from a **Role**.

## Syntax:

```
pangolin-admin revoke-permission <role_name> <action> <resource>
```

## Examples:

```
# Revoke 'write' access to 'sales' catalog from 'analyst' role
pangolin-admin revoke-permission analyst write catalog:sales
```

# Assign Role

Assign a Role to a User (or Service User).

## Syntax:

```
pangolin-admin assign-role --user-id <user_id> --role-id <role_id>
```

## Example:

```
pangolin-admin assign-role --user-id 550e8400... --role-id 770e8400...
```

# Revoke User Role

Revoke a Role from a User.

## Syntax:

```
pangolin-admin revoke-user-role --user-id <user_id> --role-id <role_id>
```

## Example:

```
pangolin-admin revoke-user-role --user-id 550e8400... --role-id 770e8400...
```

Source: admin-service-users.md

# Service User CLI Commands

Complete reference for managing service users via the Pangolin Admin CLI.

## Overview

Service users provide API key-based authentication for programmatic access to Pangolin. They are ideal for:

CI/CD pipelines

Data integration tools

Monitoring systems

Microservices

## Commands

create-service-user

Create a new service user with an API key.

### Syntax:

```
pangolin-admin create-service-user \  
  --name <name> \  
  [--description <description>] \  
  [--role <role>] \  
  [--expires-in-days <days>]
```

### Arguments:

`name` (required): Name of the service user

`description` (optional): Description of purpose

`role` (optional): Role assignment (default: `tenant-user`)

Valid values: `tenant-user`, `tenant-admin`, `root`

`expires-in-days` (optional): Expiration in days from creation

### Example:

```
pangolin-admin create-service-user \  
  --name "ci-pipeline" \  
  --description "GitHub Actions CI/CD" \  
  --role "tenant-user" \  
  --expires-in-days 90
```

### Output:

Service user created successfully!

---

⚠ IMPORTANT: Save this API key - it will not be shown again!

---

Service User ID: 550e8400-e29b-41d4-a716-446655440000  
Name: ci-pipeline  
API Key: pgl\_AbCdEfGhIjKlMnOpQrStUvWxYz1234567890...  
Expires At: 2026-03-18T19:00:00Z

---

## list-service-users

List all service users for the current tenant.

### Syntax:

```
pangolin-admin list-service-users
```

Example Output:

ID		Name	Role	Active	Last Used
Expires At					
550e8400-e29b-41d4-a716-446655440000		ci-pipeline	tenant-user	✓	2 days ago
Never					
660e8400-e29b-41d4-a716-446655440001		data-sync	tenant-user	✓	Never
2026-06-01					

get-service-user

Get detailed information about a specific service user.

Syntax:

```
pangolin-admin get-service-user --id <service-user-id>
```

Example:

```
pangolin-admin get-service-user --id 550e8400-e29b-41d4-a716-446655440000
```

Output:

Service User Details	
ID: 550e8400-e29b-41d4-a716-446655440000	
Name: ci-pipeline	
Description: GitHub Actions CI/CD	
Role: tenant-user	
Active: Yes	
Created At: 2025-12-18T14:00:00Z	
Last Used: 2025-12-16T10:30:00Z	
Expires At: 2026-03-18T19:00:00Z	

update-service-user

Update service user properties.

Syntax:

```
pangolin-admin update-service-user \  
--id <service-user-id> \
```

```
[--name <new-name>] \  
[--description <new-description>] \  
[--active <true|false>]
```

### Arguments:

`id` (required): Service user ID

`name` (optional): New name

`description` (optional): New description

`active` (optional): Enable/disable service user

### Example - Update description:

```
pangolin-admin update-service-user \  
  --id 550e8400-e29b-41d4-a716-446655440000 \  
  --description "Updated: GitHub Actions + GitLab CI"
```

### Example - Deactivate service user:

```
pangolin-admin update-service-user \  
  --id 550e8400-e29b-41d4-a716-446655440000 \  
  --active false
```

## delete-service-user

Permanently delete a service user.

### Syntax:

```
pangolin-admin delete-service-user --id <service-user-id>
```

### Example:

```
pangolin-admin delete-service-user --id 550e8400-e29b-41d4-a716-446655440000
```

### Output:

```
Service user deleted successfully!
```

**Warning:** This action is permanent and immediately invalidates the API key.

## rotate-service-user-key

Rotate the API key for a service user.

## Syntax:

```
pangolin-admin rotate-service-user-key --id <service-user-id>
```

## Example:

```
pangolin-admin rotate-service-user-key --id 550e8400-e29b-41d4-a716-446655440000
```

## Output:

```
API key rotated successfully!
```

---

```
△ IMPORTANT: Save this new API key - it will not be shown again!
```

---

```
Service User ID: 550e8400-e29b-41d4-a716-446655440000
```

```
Name: ci-pipeline
```

```
New API Key: pgl_NewKeyXyZ123456789...
```

```
Expires At: 2026-03-18T19:00:00Z
```

---

**Important:** The old API key is immediately invalidated.

# Best Practices

## Security

### Save API Keys Securely

Store in secrets manager (AWS Secrets Manager, HashiCorp Vault, etc.)

Never commit to version control

Never log in plaintext

### Use Least Privilege

Assign minimum required role

Use `tenant-user` for most cases

Reserve `tenant-admin` for administrative tasks

### Set Expiration

Use `expires-in-days` for temporary access

Recommended: 90 days for production, 30 days for testing

### Regular Rotation

Rotate keys every 90 days

Rotate immediately if compromised

Update services with new key before old expires

## Monitoring

### Check Last Used

Review ``list-service-users`` output

Identify unused service users

Delete inactive service users

### Audit Activity

Monitor API access logs

Track service user operations

Alert on suspicious patterns

## Naming Conventions

Use descriptive names that indicate:

**Purpose:** ``ci-pipeline``, ``data-sync``, ``monitoring``

**Environment:** ``prod-ci``, ``staging-etl``

**Team:** ``analytics-team-bot``, ``ml-pipeline``

## Troubleshooting

### "Invalid or expired API key"

**Cause:** API key is incorrect, expired, or service user is inactive

#### Solutions:

Verify API key is correct (check for typos)

Check service user is active: ``get-service-user id <id>``

Check expiration date hasn't passed

Rotate key if needed: ``rotate-service-user-key id <id>``

### "Only admins can create service users"

**Cause:** Insufficient permissions

#### Solutions:

Ensure you're logged in as Tenant Admin or Root

Check your JWT token is valid

Re-login if token expired

## "Unknown variant TenantUser"

**Cause:** Role format is incorrect or case-sensitive.

**Solution:** Use the exact case expected by the API (e.g., ``tenant-user``, ``tenant-admin``, ``root``).

```
`role "tenant-user"`
```

```
`role "TenantUser"`
```

## Related Documentation

[Service Users API](#)

[Authentication](#)

[RBAC](#)

[CLI Overview](#)

Source: admin-system-settings.md

## System Settings Management

The ``pangolin-admin`` tool provides commands to view and update global system configuration. These settings are stored in the backend database and override default environment variables where applicable.

## Commands

### ``get-system-settings``

Retrieve the current global system configuration.

#### Usage:

```
pangolin-admin get-system-settings
```

#### Output:

```
{
  "allow_public_signup": false,
  "default_warehouse_bucket": "warehouse",
```

```
"default_retention_days": 30,  
"smtp_host": "smtp.example.com",  
"smtp_user": "admin"  
}
```

## `update-system-settings`

Update one or more global configuration parameters.

### Usage:

```
pangolin-admin update-system-settings \  
  --allow-public-signup true \  
  --default-retention-days 90
```

### Options:

- ``allow-public-signup <bool>``: Enable or disable public user registration.
- ``default-warehouse-bucket <string>``: Default bucket name for new warehouses.
- ``default-retention-days <int>``: Default data retention period.
- ``smtp-host <string>``: SMTP server hostname.
- ``smtp-port <int>``: SMTP server port.
- ``smtp-user <string>``: SMTP username.
- ``smtp-password <string>``: SMTP password (write-only).

### Example:

```
# Enable public signup and set default retention to 60 days  
pangolin-admin      update-system-settings      --allow-public-signup      true  
--default-retention-days 60
```

Source: admin-tenants.md

## Admin Tenant Management

Tenants are the highest level of isolation in Pangolin. Data, users, and configuration do not leak between tenants.

## Commands

### List Tenants

View all existing tenants. (Root Admin only)

**Syntax:**

```
pangolin-admin list-tenants
```

## Create Tenant

Initialize a new tenant environment. (Root Admin only)

**Syntax (Interactive):**

```
create-tenant --name <name> --admin-username <username> --admin-password <password>
```

**Syntax (Non-Interactive):**

```
pangolin-admin create-tenant --name <name> --admin-username <username> --admin-password <password>
```

**Example:**

```
pangolin-admin create-tenant --name acme_corp --admin-username admin --admin-password "p@ssword123"
```

## Update Tenant

Modify properties of an existing tenant. (Root Admin only)

**Syntax:**

```
pangolin-admin update-tenant --id <uuid> [--name <new_name>]
```

**Example:**

```
pangolin-admin update-tenant --id 123e4567-e89b-12d3 --name acme_holdings
```

## Delete Tenant

Permanently remove a tenant and all associated data. **Irreversible.** (Root Admin only)

**Syntax:**

```
pangolin-admin delete-tenant <id>
```

**Example:**

```
pangolin-admin delete-tenant 123e4567-e89b-12d3
```

Source: admin-token-management.md

## Admin CLI - Token Management

This guide covers token management commands for security and session control.

### Revoke Token (Self)

Revoke your own authentication token (logout).

#### Syntax

```
pangolin-admin revoke-token
```

#### Parameters

None

#### Example

```
# Logout by revoking your own token  
pangolin-admin revoke-token
```

#### Output

```
Token revoked successfully!  
You have been logged out. Please login again to continue.
```

#### Notes

Revokes the currently authenticated token

Effectively logs you out

You'll need to login again to use the CLI

Cannot be undone

#### Syntax

```
pangolin-admin revoke-token-by-id --id <TOKEN_ID>
```

## Parameters

`id` - UUID of the token to revoke (required)

## Example

```
# Revoke a specific token
pangolin-admin revoke-token-by-id --id "token-uuid"
```

## Output

```
Token token-uuid revoked successfully!
```

## List User Tokens (Admin)

List all active tokens for a specific user.

## Syntax

```
pangolin-admin list-user-tokens --user-id <USER_ID>
```

## Example

```
pangolin-admin list-user-tokens --user-id "user-uuid"
```

## Delete Token (Admin)

Forcefully delete a token record.

## Syntax

```
pangolin-admin delete-token --token-id <TOKEN_ID>
```

## Example

```
pangolin-admin delete-token --token-id "token-uuid"
```

## Use Cases

# Self-Logout

```
# When done working
pangolin-admin revoke-token
```

## Force User Logout (Security)

```
# If a user's credentials are compromised
pangolin-admin revoke-token-by-id --id "compromised-token-uuid"
```

## Session Management

```
# End all sessions for a user (requires listing tokens first)
# Note: Token listing endpoint may be needed
pangolin-admin revoke-token-by-id --id "token-1"
pangolin-admin revoke-token-by-id --id "token-2"
```

## Security Best Practices

### When to Revoke Tokens

**End of Work Session:** Always revoke when done

**Security Incident:** Immediately revoke compromised tokens

**User Offboarding:** Revoke all tokens when user leaves

**Suspicious Activity:** Revoke tokens showing unusual patterns

**Regular Rotation:** Periodically revoke and re-issue tokens

### Token Lifecycle

```
Login → Token Issued → Work Session → Revoke Token → Logout
```

## Multi-Device Sessions

Each login creates a new token:

Desktop: token-1

Laptop: token-2

CI/CD: token-3

Revoke individually or all at once.

## Workflow Examples

### Normal Logout

```
# 1. Finish work
pangolin-admin list-catalogs

# 2. Logout
pangolin-admin revoke-token

# 3. Login again later
pangolin-admin login --username admin --password password
```

### Security Incident Response

```
# 1. Login as admin
pangolin-admin login --username admin --password password

# 2. Identify compromised token
# (from security logs or monitoring)

# 3. Revoke the token
pangolin-admin revoke-token-by-id --id "compromised-token-uuid"

# 4. Notify the user
echo "Token revoked. Please contact security."
```

### Scheduled Token Rotation

```
#!/bin/bash
# Rotate tokens for all service users weekly

# Get all service user tokens
# (requires token listing endpoint)

# Revoke old tokens
for token_id in "${OLD_TOKENS[@]}"; do
    pangolin-admin revoke-token-by-id --id "$token_id"
done

# Users will need to re-authenticate
```

### Error Handling

# Common Errors

## 401 Unauthorized:

Error: API Request Failed: Failed to revoke token (401): Not authenticated

Solution: Login first

## 403 Forbidden:

Error: API Request Failed: Failed to revoke token (403): Insufficient permissions

Solution: Only admins can revoke other users' tokens

## 404 Not Found:

Error: API Request Failed: Failed to revoke token (404): Token not found

Solution: Token may already be revoked or invalid

# Comparison with Service User Keys

## Regular Tokens (JWT)

**Purpose:** User authentication

**Lifetime:** Session-based

**Revocation:** Via `revoke-token` commands

**Use:** Interactive CLI sessions

## Service User API Keys

**Purpose:** Service-to-service auth

**Lifetime:** Long-lived (30-90 days)

**Revocation:** Via `rotate-service-user-key`

**Use:** Automated systems, CI/CD

## Related Commands

`login` - Authenticate and get a token

`create-service-user` - Create service user with API key

`rotate-service-user-key` - Rotate service user API key

`list-service-users` - List service users

## Security Considerations

### Token Storage

Tokens stored in ` ~/.pangolin/config`

Protect this file (chmod 600)

Never commit to version control

### Token Exposure

If a token is exposed:

Immediately revoke it

Change your password

Review access logs

Notify security team

### Audit Trail

All token revocations are logged

Review logs regularly

Monitor for unusual patterns

## Best Practices

**Always Logout:** Revoke tokens when done

**Monitor Sessions:** Track active tokens

**Quick Response:** Revoke compromised tokens immediately

**Regular Rotation:** Implement token rotation policies

**Least Privilege:** Use service users for automation

**Audit Regularly:** Review token usage logs

Source: admin-update-operations.md

## Admin CLI - Update Operations

This guide covers the update commands for modifying existing resources in Pangolin.

## Update Tenant

Update properties of an existing tenant.

### Syntax

```
pangolin-admin update-tenant --id <TENANT_ID> --name <NEW_NAME>
```

### Parameters

``id`` - UUID of the tenant to update (required)

``name`` - New name for the tenant (optional)

### Example

```
# Update tenant name
pangolin-admin update-tenant \
  --id "8582f8f2-63af-4ecf-9aaf-82133bc42a16" \
  --name "production-tenant"
```

### Output

```
Tenant updated successfully!
```

### Notes

Requires root privileges

Tenant ID must be a valid UUID

Only specified fields will be updated

## Update User

Update properties of an existing user.

### Syntax

```
pangolin-admin update-user --id <USER_ID> [OPTIONS]
```

# Parameters

``id`` - UUID of the user to update (required)

``username`` - New username (optional)

``email`` - New email address (optional)

``active`` - Set active status (true/false) (optional)

## Examples

```
# Update username
pangolin-admin update-user \
  --id "user-uuid" \
  --username "new-username"

# Update email
pangolin-admin update-user \
  --id "user-uuid" \
  --email "newemail@example.com"

# Deactivate user
pangolin-admin update-user \
  --id "user-uuid" \
  --active false

# Update multiple fields
pangolin-admin update-user \
  --id "user-uuid" \
  --username "updated-user" \
  --email "updated@example.com" \
  --active true
```

## Output

```
User updated successfully!
```

## Notes

Requires tenant admin or root privileges

User ID must be a valid UUID

At least one optional parameter must be provided

## Update Warehouse

Update properties of an existing warehouse.

# Syntax

```
pangolin-admin update-warehouse --id <WAREHOUSE_ID> --name <NEW_NAME>
```

## Parameters

`id` - UUID of the warehouse to update (required)

`name` - New name for the warehouse (optional)

## Example

```
# Update warehouse name
pangolin-admin update-warehouse \
  --id "warehouse-uuid" \
  --name "production-warehouse"
```

## Output

```
Warehouse updated successfully!
```

## Notes

Requires tenant admin privileges

Warehouse ID must be a valid UUID

Name must be unique within the tenant

## Update Catalog

Update properties of an existing catalog.

## Syntax

```
pangolin-admin update-catalog --id <CATALOG_ID> --name <NEW_NAME>
```

## Parameters

`id` - UUID of the catalog to update (required)

`name` - New name for the catalog (optional)

## Example

```
# Update catalog name
pangolin-admin update-catalog \
  --id "catalog-uuid" \
  --name "production-catalog"
```

## Output

```
Catalog updated successfully!
```

## Notes

Requires tenant admin privileges

Catalog ID must be a valid UUID

Name must be unique within the tenant

## Common Patterns

### Getting Resource IDs

To update a resource, you first need its ID. Use list commands:

```
# Get tenant ID
pangolin-admin list-tenants

# Get user ID
pangolin-admin list-users

# Get warehouse ID
pangolin-admin list-warehouses

# Get catalog ID
pangolin-admin list-catalogs
```

## Batch Updates

For multiple updates, use a script:

```
#!/bin/bash
# Update multiple users
for user_id in "${USER_IDS[@]"; do
  pangolin-admin update-user --id "$user_id" --active true
done
```

# Error Handling

## Common Errors

### **404 Not Found:**

Error: API Request Failed: Failed to update tenant (404 Not Found): Tenant not found

Solution: Verify the ID is correct

### **403 Forbidden:**

Error: API Request Failed: Failed to update user (403 Forbidden): Insufficient permissions

Solution: Ensure you have appropriate privileges

### **400 Bad Request:**

Error: API Request Failed: Failed to update catalog (400 Bad Request): Name already exists

Solution: Choose a unique name

## Best Practices

**Verify Before Update:** List resources first to confirm IDs

**Use Descriptive Names:** Choose meaningful names for resources

**Test in Development:** Test updates in a dev environment first

**Document Changes:** Keep a log of resource updates

**Check Permissions:** Ensure you have the required privileges

## Related Commands

`~list-tenants`` - List all tenants

`~list-users`` - List all users

`~list-warehouses`` - List all warehouses

`~list-catalogs`` - List all catalogs

`~delete-tenant`` - Delete a tenant

`~delete-user`` - Delete a user

`~delete-warehouse`` - Delete a warehouse

``delete-catalog`` - Delete a catalog

Source: admin-users.md

## Admin User Management

Manage users who can authenticate to the Pangolin system for the current tenant.

### Commands

#### List Users

View all registered users and their primary roles.

**Syntax:**

```
pangolin-admin list-users
```

#### Create User

Create a new user account.

[!WARNING]

The **Root User** can ONLY create a ``tenant-admin`` for a new tenant. They cannot create regular ``tenant-user`` accounts. Regular user management is the responsibility of the Tenant Admin.

**Syntax:**

```
pangolin-admin create-user <username> --email <email> --role <role> [--password <password>] [--tenant-id <uuid>]
```

**Options:**

``role``: Assign a role (``root``, ``tenant-admin``, ``tenant-user``).

``password``: Provide password directly (omitting will trigger a secure prompt).

``tenant-id``: Target a specific tenant (Root only).

**Example:**

```
pangolin-admin create-user alice --email alice@acme.com --role tenant-user
```

#### Update User

Modify an existing user's profile or status.

**Syntax:**

```
pangolin-admin update-user --id <uuid> [--username <name>] [--email <email>] [--active <true|false>]
```

**Example:**

```
pangolin-admin update-user --id "user-uuid" --active false
```

## Delete User

Remove a user account.

**Syntax:**

```
pangolin-admin delete-user <username>
```

**Example:**

```
pangolin-admin delete-user bob
```

Source: admin-warehouses.md

## Admin Warehouse Management

Warehouses define the physical storage locations (S3, GCS, Azure, Local) where data is kept.

### Commands

#### List Warehouses

View configured warehouses and their storage types.

**Syntax:**

```
pangolin-admin list-warehouses
```

#### Create Warehouse

Configure a new storage location.

[!IMPORTANT]

This command requires **Tenant Admin** privileges. The Root User cannot create warehouses.

### Syntax (Full):

```
pangolin-admin create-warehouse --name <name> --type <type> --bucket <bucket>
--access-key <key> --secret-key <secret> --region <region> [--endpoint <url>]
```

**Supported Types:** `s3`, `gcs`, `azure`, `local`.

### Example:

```
pangolin-admin create-warehouse --name production --type s3 --bucket "my-data" --region
"us-east-1" ...
```

## Update Warehouse

Modify a warehouse configuration.

### Syntax:

```
pangolin-admin update-warehouse --id <uuid> [--name <new_name>]
```

### Example:

```
pangolin-admin update-warehouse --id "warehouse-uuid" --name "legacy-storage"
```

## Delete Warehouse

Remove a warehouse configuration. Does not delete the actual data in storage.

### Syntax:

```
pangolin-admin delete-warehouse <name>
```

### Example:

```
pangolin-admin delete-warehouse old_bucket
```

Source: admin.md

## Pangolin Admin CLI (`pangolin-admin`)

The `pangolin-admin` tool is designed for system administrators to manage Tenants, Users, Warehouses, Catalogs, and Governance policies.

## Usage Modes

# Interactive Mode (REPL)

Run the binary without a command to enter the interactive shell.

```
pangolin-admin  
# Optional: Specify server URL and Profile  
pangolin-admin --url http://localhost:8080 --profile prod
```

In this mode, the session is persistent. You can login once and run multiple commands.

## Non-Interactive Mode

Run a single command directly from the shell. Useful for scripts.

```
pangolin-admin login --username admin  
pangolin-admin list-tenants  
pangolin-admin --profile prod create-user --username newuser
```

# Authentication

**Login:** ``login username <user> [password <pass>] [tenant-id <uuid>]``

Omit ``tenant-id`` for Root user login

Include ``tenant-id`` for tenant-scoped login (resolves duplicate usernames across tenants)

**Logout:** Exit the REPL or simply don't use the session. Session tokens are stored in ``~/config/pangolin/cli/config.json``.

## Examples

### Root Login:

```
pangolin-admin login --username admin --password password
```

### Tenant-Scoped Login (for users with duplicate usernames):

```
# Get tenant ID first  
TENANT_ID=$(pangolin-admin list-tenants | grep "my-tenant" | awk '{print $1}')
```

```
# Login with tenant context  
pangolin-admin login --username user --password pass123 --tenant-id $TENANT_ID
```

# Core Management

## Tenants

``create-tenant name <name> admin-username <user> admin-password <pass>``: Create a new tenant.

``list-tenants``: List all platform tenants (Root only).

``update-tenant id <id> [name <name>]``: Update tenant properties.

``delete-tenant <id>``: Remove a tenant and its data.

## Users

``create-user <username> email <email> role <role>``: Create a new user.

``list-users``: List all registered users.

``update-user id <id> [username <name>] [email <email>] [active <bool>]``: Modify user profile.

``delete-user <username>``: Delete a user account.

## Warehouses

``create-warehouse name <name> type <s3|gcs|azure|local> ...``: Configure physical storage.

``list-warehouses``: List defined warehouses.

``update-warehouse id <id> [name <name>]``: Rename a warehouse.

``delete-warehouse <name>``: Detach a storage location.

## Catalogs

``create-catalog <name> warehouse <warehouse_name>``: provision an Iceberg catalog.

``list-catalogs``: View logical catalogs.

``update-catalog id <id> [name <name>]``: Rename a catalog.

``delete-catalog <name>``: Remove a catalog reference.

## Governance

### Permissions

``list-permissions role <role> user <user>``: View active access policies.

``grant-permission <username> <action> <resource>``: Grant user a specific permission.

``revoke-permission <role> <action> <resource>``: Revoke permission from a role.

# Metadata

See [Metadata Management Guide](#) for detailed attribution and explorer commands.

`^get-metadata entity-type <type> entity-id <id>``: Get entity properties.

`^set-metadata entity-type <type> entity-id <id> <key> <value>``: Set entity properties.

`^list-namespace-tree <catalog>``: Browse catalog structure.

## Update Operations

Update existing resources. See [Update Operations Guide](#) for details.

### Commands

`^update-tenant id <id> name <name>``: Update tenant properties

`^update-user id <id> [username <name>] [email <email>] [active <bool>]``: Update user properties

`^update-warehouse id <id> name <name>``: Update warehouse properties

`^update-catalog id <id> name <name>``: Update catalog properties

## Token Management

Manage authentication tokens for security. See [Token Management Guide](#) for details.

### Commands

`^revoke-token``: Revoke your own token (logout)

`^revoke-token-by-id id <token-id>``: Admin revoke any token

## Merge Operations

Complete merge workflow for branch management. See [Merge Operations Guide](#) for details.

### Commands

`^list-merge-operations``: List all merge operations

`^get-merge-operation id <id>``: Get merge details

`^list-conflicts merge-id <id>``: List merge conflicts

`^resolve-conflict merge-id <id> conflict-id <id> resolution <strategy>``: Resolve conflict

``complete-merge id <id>``: Complete a merge

``abort-merge id <id>``: Abort a merge

## Business Metadata & Governance

Manage business metadata and access requests. See [Business Metadata Guide](#) for details.

### Commands

``delete-metadata asset-id <id>``: Delete business metadata

``request-access asset-id <id> reason <reason>``: Request asset access

``list-access-requests``: List all access requests

``update-access-request id <id> status <status>``: Approve/deny access request

``get-asset-details id <id>``: Get asset details

## Service User Management

Service users provide API key authentication for machine-to-machine access.

### Commands

``create-service-user name <name> [description <desc>] [role <role>] [expires-in-days <days>]``: Create a new service user

Default role: ``tenant-user``

Returns API key (shown only once!)

``list-service-users``: List all service users with status

``get-service-user id <id>``: View detailed service user information

``update-service-user id <id> [name <name>] [description <desc>] [active <true|false>]``: Update service user properties

``delete-service-user id <id>``: Delete a service user

``rotate-service-user-key id <id>``: Rotate API key (invalidates old key immediately)

## Examples

### Create service user:

```
pangolin-admin create-service-user \  
  --name "ci-pipeline" \  
  --description "CI/CD automation" \  
  --role "tenant-user" \  
  --
```

```
--expires-in-days 90
```

### List service users:

```
pangolin-admin list-service-users
```

### Rotate API key:

```
pangolin-admin rotate-service-user-key --id <uuid>
```

## Important Notes

⚠ API keys are shown only once during creation/rotation - save them securely!

Valid roles: `tenant-user`, `tenant-admin`, `root`

Service users authenticate via `X-API-Key` header

Rotating a key immediately invalidates the old key

## Source: configuration.md

## CLI Configuration Guide

The Pangolin CLI tools (`pangolin-admin` and `pangolin-user`) share a common configuration file to persist session state and preferences.

## File Location

The configuration file is stored in `config.json` within your system's standard configuration directory (using the `com.pangolin.cli` namespace):

**Linux:** `~/.config/pangolin-cli/config.json`

**macOS:** `~/Library/Application Support/com.pangolin.cli/config.json`

**Windows:** `C:\Users\<User>\AppData\Roaming\pangolin\cli\config.json`

## Configuration Properties

The file is a standard JSON object with the following properties:

Property	Type	Description	Default
`base_url`	String	The URL of the Pangolin API server.	`http://localhost:8080`
`auth_token`	String	The JWT token for the currently logged-in session.	`null`
`username`	String	The username of the logged-in user.	`null`
`tenant_id`	String	The Unique Identifier of the active tenant.	`null`

| `tenant\_name` | String | The human-readable name of the active tenant. | `null` |

## Example `config.json`

```
{
  "base_url": "http://localhost:8080",
  "auth_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "username": "admin",
  "tenant_id": "123e4567-e89b-12d3-a456-426614174000",
  "tenant_name": "AcmeCorp"
}
```

## Manual Editing

## Multiple Profiles

You can manage multiple sessions (e.g., `dev`, `prod`, `dummy\_user`) using the `profile` flag.

**Default:** `~/config/pangolin/cli/config.json`

**Profile 'prod':** `~/config/pangolin/cli/config-prod.json`

To use a profile:

```
pangolin-admin --profile prod login ...
pangolin-user --profile dev list-catalogs
```

Each profile maintains its own `base_url`, `auth_token`, and `username`.

You can manually edit this file to change the target server or clear your session (by removing `auth_token` and `username`). However, it is recommended to use the CLI commands (`login`, `url`) to manage these settings.

## Source: docker-usage.md

## Using Pangolin CLI via Docker

This guide explains how to use the Pangolin CLI tools (`pangolin-admin` and `pangolin-user`) via the Docker container instead of installing binaries locally.

## Docker Image

The Pangolin CLI tools are available in the `alexmerced/pangolin-cli` Docker image:

```
docker pull alexmerced/pangolin-cli:0.1.0
```

# Available Commands

The Docker image includes both CLI tools:

`pangolin-admin` - Administrative operations (tenants, warehouses, catalogs, users, permissions)

`pangolin-user` - User operations (discovery, access requests, tokens, branches)

## Basic Usage

### Running Admin Commands

```
docker run --rm alexmerced/pangolin-cli:0.1.0 pangolin-admin --help
```

### Running User Commands

```
docker run --rm alexmerced/pangolin-cli:0.1.0 pangolin-user --help
```

## Configuration

### Environment Variables

Pass configuration via environment variables:

```
docker run --rm \  
  -e PANGOLIN_API_URL=http://your-api:8080 \  
  -e PANGOLIN_TOKEN=your_jwt_token \  
  alexmerced/pangolin-cli:0.1.0 \  
  pangolin-admin list-tenants
```

### Configuration File

Mount a configuration file into the container:

```
docker run --rm \  
  -v $(pwd)/config.json:/root/.pangolin/config.json \  
  alexmerced/pangolin-cli:0.1.0 \  
  pangolin-admin list-tenants
```

## Common Workflows

### Interactive Mode

For interactive sessions, use the `-it`` flags:

```
docker run -it \  
  -e PANGOLIN_API_URL=http://localhost:8080 \  
  alexmerced/pangolin-cli:0.1.0 \  
  pangolin-admin
```

## Scripting

Create a shell alias for easier usage:

```
# Add to your .bashrc or .zshrc  
alias pangolin-admin='docker run --rm -e PANGOLIN_API_URL=http://localhost:8080  
alexmerced/pangolin-cli:0.1.0 pangolin-admin'  
alias pangolin-user='docker run --rm -e PANGOLIN_API_URL=http://localhost:8080  
alexmerced/pangolin-cli:0.1.0 pangolin-user'  
  
# Then use like normal commands  
pangolin-admin list-tenants  
pangolin-user search "my dataset"
```

## Docker Compose Integration

Include CLI tools in your `docker-compose.yml``:

```
services:  
  cli:  
    image: alexmerced/pangolin-cli:0.1.0  
    environment:  
      - PANGOLIN_API_URL=http://pangolin-api:8080  
      - PANGOLIN_TOKEN=${PANGOLIN_TOKEN}  
    command: ["echo", "CLI tools ready. Use: docker compose run cli pangolin-admin  
--help"]  
    depends_on:  
      - pangolin-api
```

Run commands:

```
docker compose run --rm cli pangolin-admin list-tenants  
docker compose run --rm cli pangolin-user search "customers"
```

## Network Considerations

### Connecting to Local API

When the API is running on your host machine:

## Linux:

```
docker run --rm \
  --network host \
  -e PANGOLIN_API_URL=http://localhost:8080 \
  alexmerced/pangolin-cli:0.1.0 \
  pangolin-admin list-tenants
```

## Mac/Windows:

```
docker run --rm \
  -e PANGOLIN_API_URL=http://host.docker.internal:8080 \
  alexmerced/pangolin-cli:0.1.0 \
  pangolin-admin list-tenants
```

# Connecting to Dockerized API

When using Docker Compose or a custom network:

```
docker run --rm \
  --network your_network_name \
  -e PANGOLIN_API_URL=http://pangolin-api:8080 \
  alexmerced/pangolin-cli:0.1.0 \
  pangolin-admin list-tenants
```

# Examples

## Create a Tenant

```
docker run --rm \
  -e PANGOLIN_API_URL=http://localhost:8080 \
  -e PANGOLIN_TOKEN=your_root_token \
  alexmerced/pangolin-cli:0.1.0 \
  pangolin-admin create-tenant --name "acme"
```

## List Catalogs

```
docker run --rm \
  -e PANGOLIN_API_URL=http://localhost:8080 \
  -e PANGOLIN_TOKEN=your_token \
  alexmerced/pangolin-cli:0.1.0 \
  pangolin-admin list-catalogs
```

## Search for Datasets

```
docker run --rm \
  -e PANGOLIN_API_URL=http://localhost:8080 \
  -e PANGOLIN_TOKEN=your_token \
  alexmerced/pangolin-cli:0.1.0 \
  pangolin-user search "sales"
```

## Generate a Token

```
docker run --rm \
  -e PANGOLIN_API_URL=http://localhost:8080 \
  -e PANGOLIN_TOKEN=your_admin_token \
  alexmerced/pangolin-cli:0.1.0 \
  pangolin-admin generate-token --user-id user_123 --expiry 7d
```

## Advantages of Docker CLI

**No Installation:** No need to install Rust or compile binaries  
**Consistent Environment:** Same version across all platforms  
**Easy Updates:** `docker pull` to get the latest version  
**Isolation:** Doesn't affect your system  
**CI/CD Friendly:** Easy to integrate into pipelines

## Disadvantages

**Slower Startup:** Docker overhead on each command  
**Longer Commands:** More verbose than native binaries  
**Network Complexity:** Requires understanding Docker networking

## When to Use Docker vs Binaries

### Use Docker when:

You don't want to install binaries  
Running in CI/CD pipelines  
Need consistent environments across teams  
Already using Docker for other tools

### Use Binaries when:

Running many commands frequently  
Want faster execution  
Prefer simpler command syntax  
Working locally without Docker

# Troubleshooting

## Cannot Connect to API

Check network connectivity:

```
docker run --rm --network host alpine ping -c 3 localhost
```

## Permission Denied

Ensure your user is in the docker group:

```
sudo usermod -aG docker $USER  
# Log out and back in
```

## Token Issues

Verify your token is valid:

```
curl -H "Authorization: Bearer $PANGOLIN_TOKEN" http://localhost:8080/api/v1/tenants
```

## Related Documentation

[Admin CLI Reference](#)

[User CLI Reference](#)

[Configuration Guide](#)

[Binary Installation](#)

## Source: overview.md

## Pangolin CLI Tools Overview

Pangolin provides two command-line interface (CLI) tools designed for different operational personas. Both tools feature a synchronized **REPL (Interactive)** mode and a **Non-Interactive** mode for scripting.

### The Tools

#### 1. `pangolin-admin` (The Platform Tool)

Targeted at Root and Tenant Admins for infrastructure and identity management.

### Core Commands:

Command	Usage	Role
list-tenants	Manage platform organizations and isolation.	Root
create-user	Onboard Tenant Admins or standard Users.	Root / Admin
create-warehouse	Connect S3/Azure/GCS storage providers.	Admin
create-catalog	Provision Iceberg or Federated catalogs.	Admin
grant-permission	Assign RBAC/TBAC roles and access levels.	Admin
list-audit-events	Comprehensive forensic and security tracking.	Admin
create-service-user	Manage machine-to-machine API identities.	Admin
update-system-settings	Configure global platform defaults.	Root

## 2. `pangolin-user` (The Data Tool)

Targeted at Data Engineers and Analysts for discovery and versioning.

### Core Commands:

Command	Usage	Role
search	Discover tables/views via metadata engines.	All Users
create-branch	Fork catalogs for dev/testing isolation.	Engineer
merge-branch	Promote changes across catalog environments.	Engineer
create-tag	Mark immutable snapshot versions.	Analyst
generate-code	Get Spark/Pylceberg connection config.	All Users
get-token	Manage personal API keys and sessions.	All Users

## Architecture & Config

Both tools share the same configuration layer.

**Interactive Mode:** Simply run `pangolin-admin` or `pangolin-user` to enter the shell.

**Non-Interactive Mode:** Pass the command directly: `pangolin-user search query "sales"`.

### Config Paths:

**Linux:** `~/config/pangolin/config.json`

**macOS:** `~/Library/Application Support/com.pangolin.cli/config.json`

**Windows:** `C:\Users\<User>\AppData\Roaming\pangolin\cli\config.json`

## Pagination

Most list commands support standard pagination flags to handle large datasets effectively:

`limit <N>`: Restrict the number of results (default: 100).

`offset <N>`: Skip the first N results.

```
# Get the first 50 tables
pangolin-user search --query "sales" --limit 50
```

```
# Get the next 50
pangolin-user search --query "sales" --limit 50 --offset 50
```

## Quick Tip

Use the ``profile`` flag to manage multiple Pangolin environments (e.g., ``dev`` vs ``prod``).

```
pangolin-user login --profile prod --username "me@company.com"
```

## Source: user-access.md

### User Access Requests

If you need access to a resource you cannot currently view or modify, you can submit an access request through the CLI.

## Commands

### List Requests

View the status of your submitted access requests.

#### **Syntax:**

```
pangolin-user list-requests
```

### Request Access

Submit a new request for access to a resource.

#### **Syntax:**

```
pangolin-user request-access --resource <resource_identifier> --role <requested_role>
--reason <text>
```

#### **Example:**

```
pangolin-user request-access --resource catalog:marketing --role read --reason "Need
access for Q4 reporting"
```

The administrator will review your request and grant/deny access.

## Source: user-branches.md

# User Branch Management

Pangolin supports Git-like branching for your data. You can create branches to experiment, isolate changes, or manage data lifecycles.

## Commands

### List Branches

View all branches in a catalog.

#### Syntax:

```
pangolin-user list-branches --catalog <catalog_name>
```

#### Example:

```
pangolin-user list-branches --catalog sales
```

### Create Branch

Create a new branch from an existing one (defaulting to `main`).

#### Syntax:

```
pangolin-user create-branch --catalog <catalog> <name> [--from <source_branch>]  
[--branch-type <type>] [--assets <asset_list>]
```

#### Parameters:

`catalog` (required): Catalog name

`name` (required): Name of the new branch

`from` (optional): Source branch to branch from (default: `main`)

`branch-type` (optional): Branch type - `main`, `feature`, or `experiment`

`assets` (optional): Comma-separated list of assets for partial branching

#### Examples:

Basic branch creation:

```
# Create a feature branch from main  
pangolin-user create-branch --catalog sales feature/q3_adjustments  
  
# Create a branch from an existing branch  
pangolin-user create-branch --catalog sales experimental --from dev
```

## Enhanced Branching (with branch types):

```
# Create a feature branch with explicit type
pangolin-user create-branch --catalog sales feature-x \
  --branch-type feature

# Create an experiment branch
pangolin-user create-branch --catalog sales experiment-1 \
  --branch-type experiment
```

## Partial Branching (specific assets only):

```
# Create a branch with only specific tables
pangolin-user create-branch --catalog sales feature-subset \
  --branch-type feature \
  --assets "customers,orders,products"

# Create an experiment branch with a single table
pangolin-user create-branch --catalog sales test-pricing \
  --branch-type experiment \
  --assets "pricing_model"
```

## Use Cases:

**Feature Development:** Create feature branches to develop new data transformations

**Experimentation:** Create experiment branches to test data changes without affecting production

**Partial Branching:** Create branches with only the tables you need to work with, reducing overhead

**Data Isolation:** Isolate changes to specific datasets while keeping others unchanged

## Merge Branch

Merge changes from one branch into another (e.g., merging a feature branch back into main).

### Syntax:

```
pangolin-user merge-branch --catalog <catalog> --source <src> --target <tgt>
```

### Example:

```
pangolin-user merge-branch --catalog sales --source feature/q3_adjustments --target main
```

Source: user-discovery.md

# User Data Discovery

Explore the available datasets and integrate them into your tools.

## Commands

### List Catalogs

See which catalogs you have access to.

**Syntax:**

```
pangolin-user list-catalogs
```

### Search

Search for tables, views, or dashboards across all catalogs.

**Syntax:**

```
pangolin-user search <query>
```

**Example:**

```
pangolin-user search "customer revenue"
```

### Generate Code

Generate boilerplate code to connect to a specific table. Pangolin outputs code with syntax highlighting for immediate use.

**Syntax:**

```
pangolin-user generate-code --language <lang> --table <full_table_name>
```

**Supported Languages:** `pyiceberg`, `pyspark`, `dremio`, `sql`.

**Example:**

```
pangolin-user generate-code --language pyiceberg --table sales_catalog.public.orders
```

### Get Token

Generate a personal API key for use in external scripts or tools.

**Syntax:**

```
pangolin-user get-token [--description <text>] [--expires-in <days>]
```

**Example:**

```
pangolin-user get-token --description "CI/CD Pipeline" --expires-in 30
```

Source: user-tags.md

## User Tag Management

Tags are immutable references to specific commits in the catalog's history. They are useful for marking releases, snapshots, or important points in time.

## Commands

### List Tags

View all tags in a catalog.

**Syntax:**

```
pangolin-user list-tags --catalog <catalog>
```

**Example:**

```
pangolin-user list-tags --catalog sales
```

### Create Tag

Create a tag pointing to a specific commit.

**Syntax:**

```
pangolin-user create-tag --catalog <catalog> --name <tag_name> --commit-id <commit_sha>
```

**Example:**

```
pangolin-user create-tag --catalog sales --name v1.0.0 --commit-id 8a08734268e3...
```

Source: user-tokens.md

## Token Generation

Generate long-lived JWT tokens for automation, scripts, and external clients.

## User CLI

### Generate Token

Generate a JWT token for your account:

```
pangolin-user get-token --description "PyIceberg automation" --expires-in 90
```

#### Parameters:

``description`` (required): Description of what this token will be used for

``expires-in`` (optional): Token expiration in days (default: 90)

#### Example Output:

```
Token generated successfully!

Token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
Expires: 2025-03-18T12:00:00Z

△ Save this token securely. It will not be shown again.
Use this token with PyIceberg:
    catalog = load_catalog('my_catalog', uri='...', token='<TOKEN>')
```

## Using Tokens with Pylceberg

Once you have a token, use it to authenticate with Pylceberg:

```
from pyiceberg.catalog import load_catalog

catalog = load_catalog(
    "my_catalog",
    **{
        "uri": "http://localhost:8080/v1/my_catalog",
        "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
    }
)

# Now you can use the catalog
table = catalog.load_table("my_namespace.my_table")
df = table.scan().to_arrow()
```

## Security Best Practices

-

**Store tokens securely:** Never commit tokens to version control

**Use environment variables:** Store tokens in environment variables or secure vaults

**Set appropriate expiration:** Use shorter expiration times for sensitive environments

**Rotate tokens regularly:** Generate new tokens periodically

**Revoke unused tokens:** Use the token revocation API to revoke tokens you no longer need

## Token Revocation

Tokens can be revoked using the API:

```
# Self-revoke (revoke your own token)
curl -X POST http://localhost:8080/api/v1/tokens/revoke/self \
  -H "Authorization: Bearer <TOKEN>"

# Admin revoke (revoke another user's token)
curl -X POST http://localhost:8080/api/v1/tokens/revoke/<JTI> \
  -H "Authorization: Bearer <ADMIN_TOKEN>"
```

## Common Use Cases

### CI/CD Pipelines

Generate a token for your CI/CD pipeline to automate data operations:

```
# In your CI/CD setup script
TOKEN=$(pangolin-user get-token --description "GitHub Actions" --expires-in 365)
echo "PANGOLIN_TOKEN=$TOKEN" >> $GITHUB_ENV
```

### Data Science Notebooks

Generate a token for Jupyter notebooks:

```
pangolin-user get-token --description "Jupyter notebook" --expires-in 30
```

### Scheduled Jobs

Generate tokens for cron jobs or scheduled tasks:

```
pangolin-user get-token --description "Daily ETL job" --expires-in 180
```

## Source: user.md

# Pangolin User CLI (`pangolin-user`)

The `pangolin-user` tool is for data engineers and analysts to discover data, manage branches, and request access.

## Usage Modes

### Interactive Mode (REPL)

```
pangolin-user  
pangolin-user --profile dev
```

### Non-Interactive Mode

```
pangolin-user list-catalogs  
pangolin-user search "sales_data"
```

## Data Discovery

`list-catalogs`: View available catalogs.

`search <query>`: Search for tables and views.

`generate-code language <pyiceberg|pyspark|dremio|sql> table <table\_name>`:  
Generate connection snippets.

`get-token [description <text>] [expires-in <days>]`: Generate personal API key.

## Data Engineering

### Branching

`list-branches catalog <catalog>`: View branch history.

`create-branch catalog <catalog> <name> [from <branch>] [branch-type <type>]`:  
Create new fork.

`merge-branch catalog <catalog> source <src> target <tgt>`: Promote/Integrate changes.

### Tags

`list-tags catalog <catalog>`: View immutable release/snapshot tags.

`create-tag catalog <catalog> name <name> commit-id <id>`: Tag a specific version.

# Access Management

``request-access resource <res> role <role> reason <text>``: Request permission to an asset.

``list-requests``: View status of your pending/historical requests.

## Source: warehouse-management.md

# CLI Warehouse Management Guide

## Overview

The Pangolin Admin CLI provides comprehensive warehouse management capabilities for S3, Azure ADLS, and Google GCS storage backends.

## Creating Warehouses

### S3/MinIO Warehouses

Create an S3-compatible warehouse (works with AWS S3 and MinIO):

```
pangolin-admin create-warehouse my-s3-warehouse \  
  --type s3 \  
  --bucket my-bucket \  
  --access-key AKIAIOSFODNN7EXAMPLE \  
  --secret-key wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY \  
  --region us-east-1 \  
  --endpoint http://localhost:9000 # Optional, for MinIO
```

**Interactive Mode** (prompts for credentials):

```
pangolin-admin create-warehouse my-s3-warehouse --type s3
```

#### Properties Set:

``s3.bucket`` - S3 bucket name

``s3.access-key-id`` - AWS access key ID

``s3.secret-access-key`` - AWS secret access key

``s3.region`` - AWS region

``s3.endpoint`` - Custom endpoint (for MinIO)

### Azure ADLS Warehouses

Create an Azure Data Lake Storage warehouse:

```
pangolin-admin create-warehouse my-azure-warehouse --type azure
```

### Interactive Prompts:

```
Azure Storage Account Name: mystorageaccount
Azure Storage Account Key: *****
Azure Container Name: mycontainer
Use SAS token instead of account key? [y/N]: n
```

### Properties Set:

```
`adls.account-name` - Azure storage account name
`adls.account-key` - Azure storage account key
`azure.container` - Azure container name
`adls.sas-token` - SAS token (optional, if using SAS auth)
```

## Google GCS Warehouses

Create a Google Cloud Storage warehouse:

```
pangolin-admin create-warehouse my-gcs-warehouse --type gcs
```

### Interactive Prompts:

```
GCP Project ID: my-gcp-project
GCS Bucket Name: my-gcs-bucket
Service Account JSON File Path: /path/to/service-account.json
```

### Properties Set:

```
`gcs.project-id` - GCP project ID
`gcs.bucket` - GCS bucket name
`gcs.service-account-file` - Path to service account JSON file
```

## Listing Warehouses

View all warehouses in the current tenant:

```
pangolin-admin list-warehouses
```

### Output:

```
+-----+-----+
```

Name	Type
my-s3-warehouse	
my-azure-warehouse	

## Updating Warehouses

Update warehouse name and/or storage configuration:

```
pangolin-admin update-warehouse --id <warehouse-uuid>
```

### Interactive Flow:

```
=== Update Warehouse ===
Current name: my-s3-warehouse

What would you like to update?
1. Name only
2. Storage configuration
3. Both
Choice [1]: 2

=== Update Storage Configuration ===
Enter new storage configuration properties (key=value).
Press Enter with empty input to finish.
Example: s3.access-key-id=AKIA...

Property (or press Enter to finish): s3.access-key-id=AKIANEWKEY123
  ✓ Added: s3.access-key-id = AKIANEWKEY123
Property (or press Enter to finish): s3.secret-access-key=newsecret
  ✓ Added: s3.secret-access-key = newsecret
Property (or press Enter to finish):

Warehouse updated successfully!
```

## Update Name Only

```
pangolin-admin update-warehouse --id <uuid> --name new-warehouse-name
```

## Update Storage Configuration

Common use cases:

### Rotate S3 Credentials:

```
Property: s3.access-key-id=AKIANEWKEY
Property: s3.secret-access-key=newsecretkey
```

## Change Azure Account Key:

```
Property: adls.account-key=newaccountkey
```

## Update GCS Service Account:

```
Property: gcs.service-account-file=/new/path/to/sa.json
```

# Deleting Warehouses

Delete a warehouse by name:

```
pangolin-admin delete-warehouse my-warehouse-name
```

**Warning:** Deleting a warehouse does not delete the underlying storage bucket or data. It only removes the warehouse configuration from Pangolin.

## Property Reference

### S3/MinIO Properties

Property	Required	Description	Example
`s3.bucket`	Yes	S3 bucket name	`my-data-bucket`
`s3.access-key-id`	Yes	AWS access key ID	`AKIAIOSFODNN7EXAMPLE`
`s3.secret-access-key`	Yes	AWS secret access key	`wJalrXUtnFEMI/K7MDENG...`
`s3.region`	Yes	AWS region	`us-east-1`
`s3.endpoint`	No	Custom S3 endpoint	`http://localhost:9000`

### Azure ADLS Properties

Property	Required	Description	Example
`adls.account-name`	Yes	Azure storage account name	`mystorageaccount`
`adls.account-key`	Yes*	Azure storage account key	`base64encodedkey...`
`azure.container`	Yes	Azure container name	`mycontainer`
`adls.sas-token`	No	SAS token (alternative to account key)	`?sv=2020-08-04&ss=...`

\* Required unless using SAS token

### Google GCS Properties

Property	Required	Description	Example
`gcs.project-id`	Yes	GCP project ID	`my-gcp-project`
`gcs.bucket`	Yes	GCS bucket name	`my-gcs-bucket`

| `gcs.service-account-file` | Yes | Path to service account JSON | `/path/to/sa.json` |

## Best Practices

### Security

**Never commit credentials** to version control

**Use environment variables** for non-interactive scripts:

```
export S3_ACCESS_KEY="AKIA..."
export S3_SECRET_KEY="..."
pangolin-admin create-warehouse my-warehouse --type s3 \
  --bucket my-bucket \
  --access-key "$S3_ACCESS_KEY" \
  --secret-key "$S3_SECRET_KEY" \
  --region us-east-1
```

**Rotate credentials regularly** using the update command

**Use STS/temporary credentials** when possible (Azure SAS, AWS STS)

### Naming Conventions

Use descriptive warehouse names: `production-s3-us-east`, `dev-azure-westus`

Include environment and region in the name

Use lowercase with hyphens for consistency

### Multi-Cloud Strategy

When using multiple cloud providers:

**Create separate warehouses** for each cloud provider

**Use consistent naming:** `prod-s3-warehouse`, `prod-azure-warehouse`

**Document which catalogs use which warehouses**

**Test credential vending** with Pylceberg before production use

### Troubleshooting

#### "Root user cannot create warehouses"

**Problem:** Trying to create warehouse as root user

**Solution:** Login as a Tenant Admin:

```
pangolin-admin login --username tenant-admin --password <password>
```

## "Failed to create warehouse: 422 Unprocessable Entity"

**Problem:** Invalid property names or missing required fields

**Solution:** Ensure you're using correct property names (e.g., `s3.bucket` not `bucket`)

## "Warehouse created but Pylceberg can't connect"

**Problem:** Incorrect credentials or property names

**Solution:**

Verify credentials are correct

Check property names match Pylceberg conventions

Update warehouse with correct properties:

```
pangolin-admin update-warehouse --id <uuid>
```

## Examples

### Complete S3 Warehouse Setup

```
# 1. Login as tenant admin
pangolin-admin login --username admin --password mypassword

# 2. Create S3 warehouse
pangolin-admin create-warehouse production-s3 \
  --type s3 \
  --bucket prod-data-bucket \
  --access-key AKIAIOSFODNN7EXAMPLE \
  --secret-key wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY \
  --region us-east-1

# 3. Create catalog using the warehouse
pangolin-admin create-catalog prod-catalog --warehouse production-s3

# 4. Verify
pangolin-admin list-warehouses
pangolin-admin list-catalogs
```

### Rotate S3 Credentials

```
# Get warehouse ID
pangolin-admin list-warehouses

# Update credentials
pangolin-admin update-warehouse --id <warehouse-uuid>
# Choose option 2 (Storage configuration)
# Enter new credentials:
```

```
# s3.access-key-id=AKIANEWKEY
# s3.secret-access-key=newsecret
```

## Migrate from S3 to Azure

```
# 1. Create new Azure warehouse
pangolin-admin create-warehouse production-azure --type azure
# Follow prompts for Azure credentials

# 2. Create new catalog with Azure warehouse
pangolin-admin create-catalog prod-catalog-azure --warehouse production-azure

# 3. Migrate data (external process)
# 4. Update applications to use new catalog
# 5. Decommission old S3 warehouse when ready
```

## Related Documentation

[Storage & Connectivity Architecture](#) - Detailed multi-cloud architecture

[Pylceberg Integration Guide](#) - Testing warehouses with Pylceberg

[Authentication Guide](#) - CLI authentication methods

## See Also

`pangolin-admin create-catalog` - Create catalogs using warehouses

`pangolin-admin list-catalogs` - View catalogs and their warehouses

`pangolin-admin help` - View all available commands

## Source: environment-variables.md

## Environment Variables Reference

This document provides a comprehensive reference for all environment variables used by Pangolin API.

## Table of Contents

Storage Backend Configuration

Authentication & Security

Object Storage (S3/MinIO)

MongoDB Configuration

PostgreSQL Configuration

SQLite Configuration

Server Configuration

Logging

Quick Start Examples

## Storage Backend Configuration

### `DATABASE\_URL`

**Required:** No (defaults to in-memory storage)

**Type:** String (Connection URL)

**Description:** Primary configuration for selecting and connecting to the storage backend.

The format of this URL determines which backend is used:

Backend	URL Format	Example
PostgreSQL	<code>`postgresql://`</code> or <code>`postgres://`</code>	<code>`postgresql://user:pass@localhost:5432/pangolin`</code>
MongoDB	<code>`mongodb://`</code> or <code>`mongodb+srv://`</code>	<code>`mongodb://user:pass@localhost:27017`</code>
SQLite	<code>`sqlite://`</code> or ends with <code>`.db`</code>   <code>`sqlite://pangolin.db`</code> or <code>`/path/to/pangolin.db`</code>	
Memory	Not set or invalid format   <code>*(no value)*</code>	

### Examples:

```
# PostgreSQL
export DATABASE_URL="postgresql://pangolin:secret@localhost:5432/pangolin_db"

# MongoDB
export DATABASE_URL="mongodb://admin:password@localhost:27017"

# MongoDB Atlas
export DATABASE_URL="mongodb+srv://user:pass@cluster.mongodb.net"

# SQLite
export DATABASE_URL="sqlite://./pangolin.db"

# Memory (default - no persistence)
# Don't set DATABASE_URL or set to empty string
```

### `MONGO\_DB\_NAME`

**Required:** Only when using MongoDB

**Type:** String

**Default:** `pangolin`

**Description:** Name of the MongoDB database to use.

```
export DATABASE_URL="mongodb://localhost:27017"
export MONGO_DB_NAME="pangolin_production"
```

## `PANGOLIN\_STORAGE\_TYPE`

**Status:** ⚠ **DEPRECATED** - Not currently used

**Description:** This variable is not read by the API. Use `DATABASE\_URL` instead.

## Authentication & Security

### `PANGOLIN\_NO\_AUTH`

**Required:** No

**Type:** Boolean (`true` or `false`)

**Default:** `false`

**Description:** ⚠ **EVALUATION/DEVELOPMENT ONLY** - Disables authentication entirely.

When enabled:

Creates a default tenant with ID `00000000-0000-0000-0000-000000000000`

Auto-provisions a `tenant\_admin` user with password `password123`

All API requests bypass authentication

Tenant creation endpoint is disabled

**Example:**

```
export PANGOLIN_NO_AUTH=true
```

⚠ **WARNING:** Never use `PANGOLIN\_NO\_AUTH=true` in production environments!

### `JWT\_SECRET`

**Required:** Yes (for production with authentication)

**Type:** String

**Default:** Auto-generated (insecure for production)

**Description:** Secret key used for signing JWT tokens.

```
# Generate a secure secret
export JWT_SECRET=$(openssl rand -base64 32)
```

## Object Storage (S3/MinIO)

These variables configure access to S3-compatible object storage for Iceberg table metadata and data files.

## `AWS\_ACCESS\_KEY\_ID`

**Required:** Yes (for S3/MinIO storage)

**Type:** String

**Description:** AWS access key ID or MinIO access key.

```
export AWS_ACCESS_KEY_ID="AKIAIOSFODNN7EXAMPLE"
```

## `AWS\_SECRET\_ACCESS\_KEY`

**Required:** Yes (for S3/MinIO storage)

**Type:** String

**Description:** AWS secret access key or MinIO secret key.

```
export AWS_SECRET_ACCESS_KEY="wJalrXUtnFEMI/K7MDENG/bPxrFiCYEXAMPLEKEY"
```

## `AWS\_REGION`

**Required:** No

**Type:** String

**Default:** `us-east-1`

**Description:** AWS region for S3 buckets.

```
export AWS_REGION="us-west-2"
```

## `S3\_ENDPOINT`

**Required:** Only for MinIO or S3-compatible storage

**Type:** String (URL)

**Description:** Custom S3 endpoint URL (for MinIO, LocalStack, etc.).

```
# MinIO
```

```
export S3_ENDPOINT="http://localhost:9000"
```

```
# LocalStack
```

```
export S3_ENDPOINT="http://localhost:4566"
```

## `AWS\_ENDPOINT\_URL`

**Required:** No

**Type:** String (URL)

**Description:** Alternative to `S3\_ENDPOINT`. Both work the same way.

```
export AWS_ENDPOINT_URL="http://minio:9000"
```

# MongoDB Configuration

## `DATABASE\_URL` (MongoDB)

See Storage Backend Configuration above.

## `MONGO\_DB\_NAME`

See Storage Backend Configuration above.

## `MONGODB\_URI`

**Status:**  **NOT USED**

**Description:** This variable is not read. Use `DATABASE\_URL` instead.

## `MONGODB\_DATABASE`

**Status:**  **NOT USED**

**Description:** This variable is not read. Use `MONGO\_DB\_NAME` instead.

# PostgreSQL Configuration

## `DATABASE\_URL` (PostgreSQL)

See Storage Backend Configuration above.

### Connection String Format:

```
postgresql://[user[:password]@][host][:port][/dbname][?param1=value1&...]
```

### Example with all parameters:

```
export
DATABASE_URL="postgresql://pangolin_user:secure_password@db.example.com:5432/pangolin_db
?sslmode=require&connect_timeout=10"
```

# SQLite Configuration

## `DATABASE\_URL` (SQLite)

See Storage Backend Configuration above.

### Path Formats:

```
# Relative path
export DATABASE_URL="sqlite://pangolin.db"
```

```
# Absolute path
export DATABASE_URL="sqlite:///var/lib/pangolin/data.db"

# Alternative format (file path only)
export DATABASE_URL="/var/lib/pangolin/pangolin.db"
```

## Server Configuration

### `PANGOLIN\_HOST`

**Required:** No  
**Type:** String (IP address)  
**Default:** `0.0.0.0`  
**Description:** IP address to bind the server to.

```
# Listen on all interfaces
export PANGOLIN_HOST="0.0.0.0"

# Listen only on localhost
export PANGOLIN_HOST="127.0.0.1"
```

### `PANGOLIN\_PORT` or `PORT`

**Required:** No  
**Type:** Integer  
**Default:** `8080`  
**Description:** Port number for the API server.

```
export PANGOLIN_PORT=3000
# or
export PORT=3000
```

## Logging

### `RUST\_LOG`

**Required:** No  
**Type:** String (log level)  
**Default:** `info`  
**Description:** Controls logging verbosity.

**Valid levels:** `error`, `warn`, `info`, `debug`, `trace`

```
# Production
export RUST_LOG=info
```

```
# Development
export RUST_LOG=debug

# Troubleshooting
export RUST_LOG=trace

# Module-specific logging
export RUST_LOG=pangolin_api=debug,pangolin_store=info
```

## Quick Start Examples

### Development with Memory Storage

```
export PANGOLIN_NO_AUTH=true
export RUST_LOG=debug
export AWS_ACCESS_KEY_ID=minioadmin
export AWS_SECRET_ACCESS_KEY=minioadmin
export S3_ENDPOINT=http://localhost:9000
export AWS_REGION=us-east-1

./pangolin_api
```

### Production with MongoDB

```
export DATABASE_URL="mongodb://pangolin_user:secure_pass@mongo.example.com:27017"
export MONGO_DB_NAME="pangolin_production"
export JWT_SECRET="your-super-secret-jwt-key-here"
export AWS_ACCESS_KEY_ID="AKIAIOSFODNN7EXAMPLE"
export AWS_SECRET_ACCESS_KEY="wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
export AWS_REGION="us-east-1"
export RUST_LOG=info
export PANGOLIN_PORT=8080

./pangolin_api
```

### Production with PostgreSQL

```
export
DATABASE_URL="postgresql://pangolin:password@postgres.example.com:5432/pangolin?sslmode=
require"
export JWT_SECRET="your-super-secret-jwt-key-here"
export AWS_ACCESS_KEY_ID="AKIAIOSFODNN7EXAMPLE"
export AWS_SECRET_ACCESS_KEY="wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
export AWS_REGION="us-west-2"
export RUST_LOG=info

./pangolin_api
```

# Docker Compose with MinIO

```
version: '3.8'

services:
  pangolin-api:
    image: pangolin-api:latest
    ports:
      - "8080:8080"
    environment:
      - DATABASE_URL=mongodb://mongo:27017
      - MONGO_DB_NAME=pangolin
      - RUST_LOG=info
      - AWS_ACCESS_KEY_ID=minioadmin
      - AWS_SECRET_ACCESS_KEY=minioadmin
      - S3_ENDPOINT=http://minio:9000
      - AWS_REGION=us-east-1
    depends_on:
      - mongo
      - minio

  mongo:
    image: mongo:7.0
    ports:
      - "27017:27017"

  minio:
    image: minio/minio
    ports:
      - "9000:9000"
      - "9001:9001"
    environment:
      - MINIO_ROOT_USER=minioadmin
      - MINIO_ROOT_PASSWORD=minioadmin
    command: server /data --console-address ":9001"
```

## Common Pitfalls

### Using the wrong variable names

```
# WRONG - These are not used
export PANGOLIN_STORE_TYPE=mongo
export MONGODB_URI=mongodb://localhost:27017
export MONGODB_DATABASE=pangolin

# CORRECT
export DATABASE_URL=mongodb://localhost:27017
export MONGO_DB_NAME=pangolin
```

## Forgetting MongoDB database name

```
# WRONG - Will use default database name "pangolin"
export DATABASE_URL=mongodb://localhost:27017

# CORRECT - Explicitly set database name
export DATABASE_URL=mongodb://localhost:27017
export MONGO_DB_NAME=my_custom_db
```

## Missing S3 endpoint for MinIO

```
# WRONG - Will try to connect to AWS S3
export AWS_ACCESS_KEY_ID=minioadmin
export AWS_SECRET_ACCESS_KEY=minioadmin

# CORRECT - Specify MinIO endpoint
export AWS_ACCESS_KEY_ID=minioadmin
export AWS_SECRET_ACCESS_KEY=minioadmin
export S3_ENDPOINT=http://localhost:9000
```

## Using NO\_AUTH in production

```
# WRONG - Security risk!
export PANGOLIN_NO_AUTH=true # in production

# CORRECT - Use proper authentication
export JWT_SECRET=$(openssl rand -base64 32)
# Don't set PANGOLIN_NO_AUTH
```

## Environment Variable Priority

When multiple variables could configure the same thing:

**Storage Backend:** `DATABASE\_URL` (only this is used)

**MongoDB Database:** `MONGO\_DB\_NAME` (only this is used)

**S3 Endpoint:** `S3\_ENDPOINT` or `AWS\_ENDPOINT\_URL` (both work)

**Server Port:** `PANGOLIN\_PORT` or `PORT` (PANGOLIN\_PORT takes precedence)

## Troubleshooting

### API uses Memory storage instead of MongoDB

**Problem:** You set `MONGODB\_URI` but API still uses memory storage.

**Solution:** Use `DATABASE\_URL` instead:

```
export DATABASE_URL="mongodb://localhost:27017"  
export MONGO_DB_NAME="pangolin"
```

## Cannot connect to MinIO

**Problem:** S3 operations fail with connection errors.

**Solution:** Make sure to set the endpoint:

```
export S3_ENDPOINT="http://localhost:9000"  
# or  
export AWS_ENDPOINT_URL="http://localhost:9000"
```

## Authentication errors in development

**Problem:** Getting 401 Unauthorized errors.

**Solution:** For development/testing only:

```
export PANGOLIN_NO_AUTH=true
```

**Remember:** Never use this in production!

## See Also

[Deployment Guide](#)

[Docker Setup](#)

[Configuration Best Practices](#)

## Source: README.md

## Pangolin Features Guide

This directory contains detailed documentation for the core features of the Pangolin platform.

### Asset & Data Management

Basic data operations and lifecycle management.

Entities: Understanding the Pangolin core models.

Asset Management: Handling Tables, Views, and other assets.

Generic Assets: Cataloging ML Models, Videos, and Files.

Modern Table Formats: Support for Delta Lake, Hudi, and Paimon.

Warehouse Management: Configuring storage backends.

Time Travel: Querying historical data states.

Tag Management: Versioning with tags.

## Versioning & Branches

Git-like workflows for your data lake.

Branch Management: Creating and managing branches.

Merge Operations: Workflow for 3-way merges.

Merge Conflicts: Principles of data conflict resolution.

## Security & Access

Governance and authentication mechanisms.

Multi-Tenancy: Isolation principles and tenant management.

RBAC: Role-Based Access Control system.

Credential Vending: Cloud IAM and STS integration.

Security Concepts: Concept guide for vending and signing.

## Discovery & Audit

Finding data and tracking changes.

Business Catalog: Data discovery portal and business metadata.

Audit Logging: Comprehensive action tracking and compliance.

## Integration & Federation

Connecting to external systems.

Federated Catalogs: Connecting to remote Iceberg catalogs.

Catalog Management: Managing local and federated catalogs.

Service Users: API keys for programmatic access.

## Integration & Testing

Using Pangolin with other tools.

# Maintenance

Optimizing and maintaining your data lake.

Maintenance Operations: Snapshot management and orphan file cleanup.

# Feature Status Matrix

Feature	Category	Status
Multi-Tenancy	System	Implemented
Branching	Versioning	Implemented
3-Way Merge	Versioning	Implemented
RBAC	Security	Implemented
Audit Logging	Security	Implemented
Business Catalog	Discovery	Implemented
Credential Vending (Static)	Security	Implemented
Credential Vending (STS)	Security	Implemented
Federated Catalogs	Integration	Implemented
View Management	Data	Implemented

Source: `asset_management.md`

# Asset Management

Beyond standard Iceberg tables, Pangolin supports managing other asset types like Views, Materialized Views, Functions, and Procedures.

# Supported Assets

**Iceberg Table**: Standard Iceberg table.

**View**: A logical view definition.

**Materialized View**: A pre-computed view.

**Function**: A user-defined function.

**Procedure**: A stored procedure.

# API Endpoints

## Create View

``POST /v1/{prefix}/namespaces/{namespace}/views``

**Headers:**

`Authorization`: `Bearer <token>`

`X-Pangolin-Tenant`: `<Tenant-ID>`

**Body:**

```
{
  "name": "my_view",
  "location": "s3://warehouse/views/my_view",
  "properties": {
    "sql": "SELECT * FROM source_table"
  }
}
```

## Get View

`GET /v1/{prefix}/namespaces/{namespace}/views/{view}`

**Headers:**

`Authorization`: `Bearer <token>`

`X-Pangolin-Tenant`: `<Tenant-ID>`

**Response:**

```
{
  "name": "my_view",
  "location": "s3://warehouse/views/my_view",
  "properties": {
    "sql": "SELECT * FROM source_table"
  }
}
```

## Rename Asset

`POST /v1/{prefix}/tables/rename`

Works for both Tables and Views.

**Body:**

```
{
  "source": {
    "namespace": ["ns1"],
    "name": "old_name"
  },
  "destination": {
    "namespace": ["ns1"],
    "name": "new_name"
  }
}
```

## Update Asset

``POST /v1/{prefix}/namespaces/{namespace}/tables/{table}`` (Tables)  
``POST /v1/{prefix}/namespaces/{namespace}/views/{view}`` (Views)

## Delete Asset

``DELETE /v1/{prefix}/namespaces/{namespace}/tables/{asset_name}``

[!TIP]

While the Iceberg REST specification often separates Table and View endpoints for creation and retrieval, deletion is typically performed via the generic resource path. Pangolin's ``delete_table`` handler internally handles any asset type tracked by the catalog.

## Rename Asset

``POST /v1/{prefix}/tables/rename``

Works for both Tables and Views.

Source: `audit_logs.md`

## Audit Logging

Pangolin provides a comprehensive, multi-tenant audit logging system. Every action—from login attempts to table schema evolutions—is recorded with full context, providing an immutable trail for security, compliance, and debugging.

## Multi-Tenant Isolation

Audit logs are strictly partitioned by **Tenant ID**.

**Root Admin:** Can view logs for the platform itself (tenant creation, global settings).

**Tenant Admin:** Can view all logs belonging to their specific organization.

**Tenant User:** Only sees logs for their own individual actions.

[!NOTE]

Tenant isolation is enforced at the storage driver level. It is physically impossible for a User in Tenant A to query logs for Tenant B.

## Usage Guides

### 1. Via CLI (``pangolin-admin``)

The admin tool provides powerful filtering and counting capabilities for security reviews.

## List Recent Events:

```
# List last 50 events for a specific user
pangolin-admin list-audit-events --user-id <uuid> --limit 50

# Filter by failed actions
pangolin-admin list-audit-events --result failure
```

## Count Events:

```
# Get total count of table creations
pangolin-admin count-audit-events --action create_table
```

## 2. Via Management UI

Navigate to **Administration -> Audit Logs**.

Use the **Filter Sidebar** to narrow down by User, Action, or Time Range.

Click on any row to open the **Event Detail Modal**, which displays the raw JSON metadata.

## 3. Via REST API

Integrate Pangolin logs into your own SIEM (Splunk, Datadog) using the audit endpoints.

### Fetch Logs:

```
GET /api/v1/audit?action=commit_table&limit=100
Authorization: Bearer <token>
X-Pangolin-Tenant: <tenant-id>
```

## Event Data Model

Every log entry follows a standardized structure:

Field	Description
:---	:---
`id`	Unique UUID for the log entry.
`user_id`	UUID of the actor.
`username`	Readable name of the actor.
`action`	The operation (e.g., `create_table`, `merge_branch`).
`resource_name`	Path to the affected object (e.g., `catalog/ns/table`).
`result`	`success` or `failure`.
`timestamp`	UTC ISO8601 string.
`metadata`	Action-specific JSON (e.g., old/new schema ID).

## Logged Actions Detail

Category		Actions	
----		----	
<b>Catalog</b>		<code>`create_catalog`</code> , <code>`delete_catalog`</code> , <code>`update_catalog`</code>	
<b>Branching</b>		<code>`create_branch`</code> , <code>`delete_branch`</code> , <code>`merge_branch`</code>	
<b>Security</b>		<code>`login`</code> , <code>`grant_permission`</code> , <code>`rotate_api_key`</code>	
<b>Maintenance</b>		<code>`expire_snapshots`</code> , <code>`remove_orphan_files`</code>	
<b>Data (Iceberg)</b>		<code>`create_table`</code> , <code>`drop_table`</code> , <code>`commit_table`</code>	

## Best Practices

**Regular Audits:** Schedule a weekly review of ``failure`` results to identify potential misconfigurations or attack patterns.

**Webhook Integration:** Export high-severity actions (like ``drop_table`` or ``revoke_permission``) to your alerting system.

**Retention:** Configure ``PANGOLIN_AUDIT_RETENTION_DAYS`` (default 90) to match your legal requirements.

Source: `branch_management.md`

## Branch Management

Pangolin brings Git-like semantics to your data catalog, allowing you to isolate changes, experiment safely, and manage data lifecycles with high precision.

### Core Concepts

#### 1. Partial Branching (The "Asset Table")

Unlike Git, which tracks an entire repository, a Pangolin branch acts as a "Filtered View" or "Asset Table."

When you create a branch, you specify a set of assets (tables/views) to "fork."

Only the metadata pointers for those specific assets are copied.

This allows you to work on ``table_A`` in isolation without involving ``table_B`` through ``table_Z``.

#### 2. Fork-on-Write Behavior

Pangolin supports an **Auto-Add** mechanism during asset creation:

**New Assets:** If you create a new table using the ``@branch`` suffix (e.g., ``new_table@dev``), Pangolin will automatically create the ``dev`` branch if it does not exist.

**Existing Assets:** To work on an existing table in a new branch, the table must first be "added" to the branch metadata. This happens either during explicit branch creation

(specifying `assets`) or when the table is first written to on that branch.

[!WARNING]

If you attempt to **Update** a table on a branch where it hasn't been added yet (e.g., `update my\_table@new\_branch`), the operation will return `NOT\_FOUND`. You must first fork the table into the branch.

## Usage Guides

### 1. Via CLI

The `pangolin-user` tool provides the most direct way to manage branches.

#### Create a Branch:

```
# Create dev branch from main with specific tables
pangolin-user create-branch dev --from main --assets data_team.users,sales.orders
```

#### List Branches:

```
pangolin-user list-branches --catalog production
```

#### Merge a Branch:

```
# Integrates dev into main
pangolin-user merge-branch dev main --catalog production
```

### 2. Via API (Iceberg Clients)

Most Iceberg clients support branching through the `@` syntax in the table identifier.

#### Spark/Pylceberg Example:

```
# Create a new table on a branch (Auto-creates the branch)
df.writeTo("production.db.my_table@feature_x").create()

# Read from a branch
df = spark.read.table("production.db.my_table@feature_x")
```

### 3. Via Management UI

Navigate to the **Data Explorer**.

Select your **Catalog**.

Use the **Branch Selector** in the top-right to switch contexts.

To create a branch from the UI, click **Manage Branches** in the sidebar or the selector

dropdown.

# Branch Types & Permissions

Type	Permission Required	Use Case
----	----	----
<b>Main</b>	`Write` on Catalog	The source of truth.
<b>Experimental</b>	`ExperimentalBranching`	Short-lived feature testing.
<b>Ingest</b>	`IngestBranching`	Staging area for bulk data loads before production merge.

For detailed permission cascading, see the [Permissions System Guide](#).

## Best Practices

**Atomic Changes:** Use branches for schema evolutions to ensure `main` never sees a broken state.

**Audit Consistency:** All branch-specific operations are logged with the branch context, allowing you to trace `Who changed What on Which branch`.

**Merge Often:** To avoid complex 3-way merge conflicts, keep feature branches short-lived and rebase frequently.

Source: `business_catalog.md`

## Business Catalog & Discovery

Pangolin goes beyond technical metadata by providing a rich **Business Catalog** to help users find, understand, and access data.

### features

#### 1. Metadata Management

Annotate your data assets with meaningful context:

**Descriptions:** Rich text documentation for tables and columns.

**Tags:** Categorize assets (e.g., `PII`, `Financial`, `Gold`).

**Properties:** Custom key-value pairs for flexible metadata.

#### 2. Discovery Portal

The **Discovery Portal** (`/discovery`) is a search-centric interface designed for data consumers.

**Full Text Search:** Find tables by name, description, or tags.

**Filtering:** Narrow results by Catalog, Namespace, or Tags.

**Discoverability:** Assets can be marked as "Discoverable" to appear in global searches even if the user doesn't have explicit read access.

## 3. Unified Search API

Powering the Discovery Portal is the high-performance `/api/v1/search`` endpoint:

**Universal Scope:** Searches across all Catalogs, Namespaces, and Tables simultaneously.

**Optimized:** Uses backend-specific optimizations (In-Memory iteration, SQL ``LIKE`` indexes, MongoDB text search).

**Secure:** Automatically filters results based on user permissions and "Discoverable" flags.

## 4. Access Request Workflow

Streamline data access governance:

**Request Access:** Users can request access to discoverable tables directly from the search results.

**Approval Workflow:** Tenant Admins receive access requests in the Admin Console (`/admin/requests``).

**Audit Trail:** All requests and approvals are logged for compliance.

## RBAC Integration

**MANAGE\_DISCOVERY:** A specific permission action required to edit business metadata or manage discoverability.

**Search Filtering:** Search results are automatically filtered based on the user's permissions, ensuring they only see what they are allowed to see (or what is explicitly discoverable).

Source: `catalog_management.md`

## Catalog Management

Pangolin supports managing multiple Iceberg catalogs within a single tenant. This document outlines the behavior and features of catalog management.

## Overview

Catalogs in Pangolin serve as the top-level container for namespaces and tables. Each catalog is associated with a Warehouse which defines the underlying object storage

configuration.

## Catalog Types

**Local:** Managed directly by Pangolin. Metadata is stored in Pangolin's backend.

**Federated:** Proxies requests to an external catalog (e.g., Snowflake, Glue, another REST catalog).

## Naming and Collisions

Catalog names must be unique within a Tenant.

## Behavior by Storage Backend

### **Postgres / SQLite (Production):**

**Constraint:** The database enforces a ``UNIQUE(tenant_id, name)`` constraint.

**Collision Behavior:** Attempting to create a catalog with a name that already exists in the same tenant will result in an error. The API will currently return a ``500 Internal Server Error`` (to be improved to ``409 Conflict``).

**Cross-Tenant:** Different tenants can have catalogs with the same name without conflict.

### **Memory Store (Development/Demo):**

**Constraint:** Based on a generic Key-Value map.

**Collision Behavior: Silent Overwrite.** Creating a catalog with an existing name will replace the old catalog configuration. No error is returned.

**Note:** This behavior is specific to the in-memory development store and does not reflect production safety.

## Creation

Catalogs can be created via:

**UI:** \*Catalogs -> Create Catalog\*

**API:** ``POST /api/v1/catalogs``

## Source: entities.md

## Entity & Model Overview

Pangolin's data model is designed to support multi-tenancy and Git-like branching for lakehouse assets.

# Core Entities

## Tenant

Represents an isolated customer or organization.

**ID:** UUID

**Name:** String

**Properties:** Key-Value map

## Warehouse

Represents a physical storage location for data.

**ID:** UUID

**Name:** String

**Storage Config:** Configuration for S3, Azure, GCS, etc.

**Tenant ID:** UUID of the owning tenant

## Catalog

A collection of namespaces.

**Name:** String (e.g., `warehouse`, `lake`)

**Properties:** Key-Value map

## Namespace

A logical grouping of assets, similar to a database or schema.

**Name:** Multi-part identifier (e.g., `["marketing", "campaigns"]`)

**Properties:** Key-Value map

## Asset

A tracked object in the lakehouse (Table, View, Model, etc.).

**ID:** UUID

**Name:** String

**Kind:** `IcebergTable`, `DeltaTable`, `HudiTable`, `ParquetTable`, `View`, `MIModel`, etc.

**Location:** Storage URI

**Properties:** Key-Value map

## Branch

A named pointer to a specific commit history.

**Name:** String (e.g., `main`, `dev`)

**Type:** `Ingest` (production-ready) or `Experimental` (isolated testing)

**Head Commit:** Pointer to the latest commit on this branch.

**Assets:** List of asset names tracked by this branch (for partial branching).

## Commit

An immutable record of a state change.

**ID:** UUID

**Parent ID:** UUID of the previous commit (optional)

**Timestamp:** Unix timestamp

**Author:** User who made the change

**Message:** Description of the change

**Operations:** List of changes (`Put Asset` or `Delete Asset`)

## Merge Operation

Tracked process for integrating changes between branches.

**ID:** UUID

**Status:** `Pending`, `Conflicted`, `Resolving`, `Ready`, `Completed`, or `Aborted`.

**Conflicts:** List of detected conflicts requiring resolution.

**Resolution Strategy:** `AutoMerge`, `TakeSource`, `TakeTarget`, `Manual`, `ThreeWayMerge`.

## Source: federated\_catalogs.md

## Federated Catalogs

Federated Catalogs allow Pangolin to act as a secure proxy for external Iceberg REST catalogs. This enables centralized governance, unified authentication, and cross-tenant data sharing.

## Usage Guides

## 1. Via CLI (`pangolin-admin`)

Manage external connections directly from your terminal.

### Create Federated Catalog:

```
pangolin-admin create-federated-catalog partner_prod \  
-P uri="https://api.partner.io/v1/catalog" \  
--storage-location "s3://partner-bucket/data" \  
-P token="secret_partner_token"
```

### Test Connectivity:

```
pangolin-admin test-federated-catalog partner_prod
```

## 2. Via Management UI

Navigate to **Infrastructure -> Catalogs**.

Click **Create Catalog** and select **Federated** as the type.

Enter the target REST API URL and select your authentication method (Basic, Bearer, or API Key).

Click **Test Connection** before saving to ensure Pangolin can reach the target.

## Cross-Tenant Federation Pattern

A common use case is sharing data between two different Pangolin tenants (e.g., \*Engineering\* sharing with \*Data Science\*).

**Source Tenant:** Create a **Service User** and copy its API key.

**Target Tenant:** Create a **Federated Catalog** using the Source's URL and the API key.

**Result:** Users in the Target Tenant can query the Source's data using their own Pangolin credentials.

## Security & Constraints

**Single Sign-On:** Users authenticate to their local Pangolin; Pangolin handles the backend handshake with the external catalog.

**Audit Logging:** All requests passing through the proxy are logged in the \*local\* tenant's audit trail.

**Read-Through:** Federated catalogs are strictly for Iceberg REST operations. Pangolin-specific features (like branching or access requests) are not supported on federated assets.

## Best Practices

**Timeout Management:** Set a reasonable timeout (default 30s) to prevent slow external catalogs from blocking local resources.

**Service Users:** Always use a dedicated Service User on the external side rather than a personal account.

**HTTPS Only:** In production, never use unencrypted HTTP for federated catalog URLs.

## Source: generic\_assets.md

# Generic Assets (Universal Catalog)

Pangolin is not just for Iceberg tables. It serves as a **Universal Data Catalog**, allowing you to govern, tag, and discover any file-based asset in your data lake—from ML models and video datasets to raw CSVs and PDF documentation.

## Overview

Generic Assets are managed entities in the catalog that point to a location (S3/Azure/GCP) but don't strictly follow the Iceberg table format. They share the same governance features as tables:

**RBAC:** Control who can view or modify the asset.

**Discovery:** Full-text search and tagging.

**Lineage:** Track relationships (e.g., "This ML Model was trained on Table X").

**Branching:** Include generic assets in your data branches.

## Supported Asset Types

Pangolin natively supports the following types:

**Data Files:** ``ParquetTable``, ``CsvTable``, ``JsonTable``

**AI/ML:** ``MIModel``

**Modern Formats:** ``DeltaTable``, ``HudiTable``, ``ApachePaimon``, ``Lance``, ``Vortex``, ``Nimble``

**Media:** ``VideoFile``, ``ImageFile``

**System:** ``View``, ``DbConnString``, ``Directory``, ``Other``

## Usage Guide (API)

Currently, Generic Assets are managed primarily via the REST API.

### 1. Registering an Asset

**Endpoint:** `POST /api/v1/catalogs/{catalog}/namespaces/{namespace}/assets`

## Example 1: Registering an ML Model

Catalog a trained model artifact stored in S3.

```
curl -X POST http://localhost:8080/api/v1/catalogs/analytics/namespaces/ml_models/assets \
-H "Authorization: Bearer <token>" \
-H "X-Pangolin-Tenant: <tenant-id>" \
-H "Content-Type: application/json" \
-d '{
  "name": "fraud_detection_v2",
  "kind": "MLModel",
  "location": "s3://my-datalake/models/fraud_v2.tar.gz",
  "properties": {
    "framework": "pytorch",
    "version": "2.1.0",
    "accuracy": "0.985",
    "training_date": "2023-11-15",
    "trained_by": "alice@example.com"
  }
}'
```

## Example 2: Registering a Video Dataset

Catalog a folder of raw video footage for ingestion.

```
curl -X POST http://localhost:8080/api/v1/catalogs/raw_data/namespaces/videos/assets \
-d '{
  "name": "cctv_logs_2023",
  "kind": "VideoFile",
  "location": "s3://raw-zone/cctv/2023/",
  "properties": {
    "format": "mp4",
    "codec": "h264",
    "retention": "30_days"
  }
}'
```

## Example 3: Registering an External Delta Table

Catalog an existing Delta Lake table for discovery (without managing it via Iceberg).

```
curl -X POST http://localhost:8080/api/v1/catalogs/analytics/namespaces/silver/assets \
-d '{
  "name": "marketing_leads",
  "kind": "DeltaTable",
  "location": "abfss://data@account.dfs.core.windows.net/delta/leads",
  "properties": {
```

```
    "managed_by": "databricks"
  }
}'
```

## 2. Discovering Assets

Generic assets appear alongside Iceberg tables in search results.

### Search API:

```
GET /api/v1/assets/search?q=fraud
```

### Response:

```
[
  {
    "id": "uuid...",
    "name": "fraud_detection_v2",
    "kind": "MLModel",
    "catalog": "analytics",
    "namespace": "ml_models"
  },
  {
    "id": "uuid...",
    "name": "fraud_transactions",
    "kind": "IcebergTable",
    "catalog": "analytics",
    "namespace": "finance"
  }
]
```

## Usage Guide (UI)

You can interactively register and manage generic assets directly from the **Data Explorer**.

**Navigate:** Go to **Data Explorer** and drill down into a **Catalog** and **Namespace**.

**Register:** Click the "**Register Asset**" button in the top-right corner.

### Configure:

**Name:** Enter a unique name for the asset.

**Type:** Select from the dropdown (e.g., `ML Model`, `Video File`, `Delta Table`).

**Location:** Provide the full S3/Azure/GCS URI (e.g., `s3://my-bucket/path/to/asset`).

**Properties:** Add key-value pairs for metadata (e.g., `accuracy: 0.99`, `owner: data-team`).

**Submit:** Click **Register**. The asset will now be discoverable in search.

## Related Documentation

Modern Table Formats: Specific guide for cataloging **Delta Lake**, **Hudi**, and **Paimon**.

Asset Management: General governance guide.

## Security

Generic Assets leverage the same permission model as tables.

**READ**: Required to view metadata or search for the asset.

**WRITE**: Required to update properties or location.

### Example Policy:

Grant Data Scientists access to register models.

```
{
  "effect": "Allow",
  "action": "RegisterAsset",
  "resource": "catalog/analytics/namespace/ml_models/*"
}
```

Source: iam\_roles.md

## Using Cloud IAM Roles with Pangolin

### Overview

Pangolin supports role-based authentication for cloud storage, allowing you to avoid storing static credentials. Instead of distributing long-lived keys, Pangolin can assume IAM roles (AWS) or use Managed Identities (Azure/GCP) to vend temporary credentials to clients like Spark, Dremio, and Pylceberg.

### Vending Strategies

When configuring a **Warehouse**, you select a ``vending_strategy`` that defines how credentials are provisioned.

#### 1. AwsSts (AWS Security Token Service)

Pangolin assumes a specified IAM Role and vends a temporary ``AccessKey``, ``SecretKey``, and ``SessionToken``.

**Best for:** Production environments on AWS.

**Config:** Requires ``role_arn`` and optional ``external_id``.

## 2. AwsStatic

Pangolin vends the static credentials configured in the warehouse.

**Best for:** Centralizing credentials so they don't have to be shared with every developer or tool individually.

**Config:** Uses the ``access_key`` and ``secret_key`` from the warehouse config.

## 3. AzureSas

Pangolin generates a Shared Access Signature (SAS) token for the specific storage container.

**Best for:** Azure Blob Storage users.

## 4. GcpDownscoped

Pangolin vends a downscoped OAuth2 token with access restricted to specific buckets/prefixes.

**Best for:** Google Cloud Storage users.

## Configuration Example

Creating a warehouse with an AWS STS strategy using the Admin CLI:

```
pangolin-admin create-warehouse --name "prod-s3" --type "s3"
# Then update with strategy (or use the interactive wizard)
```

### Using the API directly:

```
{
  "name": "s3-iam-warehouse",
  "storage_config": {
    "s3.bucket": "acme-data",
    "s3.region": "us-east-1",
    "s3.role-arn": "arn:aws:iam::123456789012:role/PangolinDataAccess"
  },
  "vending_strategy": {
    "AwsSts": {
      "role_arn": "arn:aws:iam::123456789012:role/PangolinDataAccess"
    }
  }
}
```

## Current Status & Support

Strategy	Status	Notes
<b>AwsStatic</b>	Implemented	Full support for S3/MinIO
<b>AwsSts</b>	Implemented	Full support for temporary credential vending
<b>AzureSas</b>	Planned	Awaiting Azure SDK integration
<b>GcpDownscoped</b>	Planned	Awaiting GCP SDK integration

[!IMPORTANT]

For providers/strategies marked as **Planned**, clients must provide their own storage credentials (e.g., via environment variables or Spark properties) even if the warehouse is configured in Pangolin.

## Best Practices

**Principle of Least Privilege:** Ensure the IAM role assumed by Pangolin only has access to the specific buckets/prefixes used by the warehouse.

**Short Session Durations:** Prefer shorter TTLs for vended credentials (e.g., 1 hour) to minimize the impact of token leakage.

**Use External IDs:** When configuring cross-account ``AwsSts`` roles, always use an ``external_id`` to prevent the "confused deputy" problem.

## Related Documentation

[Warehouse Management](#)

[Security & Credential Vending](#)

[Architecture: Signer Trait](#)

Source: maintenance.md

## Table Maintenance

Maintaining your Iceberg tables is critical for both storage efficiency and query performance. Pangolin provides built-in utilities to manage metadata growth and clean up unreferenced data files.

### Available Operations

#### 1. Expire Snapshots

Over time, Iceberg tables accumulate snapshots. Expiring old snapshots reduces the size of the metadata files and allows for the deletion of data files that are no longer part of any valid state.

**Endpoint:** ``POST /v1/{prefix}/namespaces/{ns}/tables/{table}/maintenance``

**Payload:**

```
{
  "action": "expire_snapshots",
  "older_than_timestamp": 1735689600000,
  "retain_last": 10
}
```

### Logic:

Identifies snapshots older than the timestamp OR outside the `retain\_last` count.

Removes these snapshots from the metadata.

Triggers the underlying storage provider (S3/Azure/GCS) to delete unreferenced manifests and data files.

## 2. Remove Orphan Files

Failed write jobs or uncommitted transactions can leave "orphan" files in your storage bucket that aren't tracked by any metadata.

**Endpoint:** `POST /v1/{prefix}/namespaces/{ns}/tables/{table}/maintenance`

### Payload:

```
{
  "action": "remove_orphan_files",
  "older_than_timestamp": 1735689600000
}
```

### Logic:

Scans the table's storage location.

Compares files on disk with those referenced in *\*all\** valid snapshots.

Deletes files not mentioned in metadata (subject to the `older\_than` safety buffer).

## Permissions Required

To run maintenance operations, the user must have the following permissions:

Operation	Action	Scope
----	----	----
<b>All Maintenance</b>	`write`	Asset or Namespace

[!IMPORTANT]

Because maintenance operations can physically delete data from your cloud storage, it is highly recommended to only grant these permissions to **Service Users** or **Data Administrators**.

## Tooling Status

Interface	Status
**REST API**	Fully Supported
**Python SDK**	Supported via `table.expire_snapshots()`
**Pangolin CLI**	Coming Soon
**Management UI**	Supported in Asset Details view

## Best Practices

**Retention Policy:** Set a standard retention (e.g., 7 days or 100 snapshots) to avoid metadata bloat.

**Safety Buffers:** When removing orphan files, always use an ``older_than`` buffer of at least 24 hours to avoid deleting files from currently running ingest jobs.

**Audit Trails:** Monitor maintenance actions in the **Audit Logs** to ensure they are running as scheduled.

Source: `merge_conflicts.md`

## Merge Principles & Conflict Management

In a Data Lakehouse, merging branches is more complex than merging code because the underlying data is massive and mutable. Pangolin implements specific strategies to handle these updates safely.

[!NOTE]

For the step-by-step API guide and technical details on the 3-Way Merge algorithm, see the [Merge Operations](#) guide.

## Types of Changes

Pangolin tracks three types of changes to assets (tables/views) when merging a branch:

**Append (New Data):** Adding new files/rows to a table.

**Schema Update:** Changing columns (add, drop, rename, type change).

**Rewrite (Optimize/Update/Delete):** Rewriting existing data files (e.g., compaction or row-level updates).

## When Do Conflicts Occur?

A **Merge Conflict** occurs when the *\*same asset\** has been modified on both the **Source Branch** (the one you are merging) and the **Target Branch** (usually ``main``) since they diverged, in a way that cannot be automatically reconciled.

Change Type A (Target)	Change Type B (Source)	Conflict?	Resolution
Append	Append	No	Pangolin automatically combines both sets of new files.
Schema Change	Append	Maybe	If schema change breaks new data (e.g.,

deleted a column present in new data), it conflicts. |

| **Rewrite** | **Append** | **No** | Generally safe; new data is added to the rewritten table state. |

| **Rewrite** | **Rewrite** | **YES** | **Concurrent Update Conflict**. Both branches modified the same partition/files. |

| **Schema Change** | **Schema Change** | **YES** | Divergent schemas can rarely be auto-merged. |

## The "Concurrent Update" Problem

If Branch A compacts ``partition_1`` and Branch B deletes rows from ``partition_1``, merging them is impossible without losing data. One operation must be discarded or re-played.

## Best Practices to Avoid Conflicts

### 1. Short-Lived Branches

The longer a branch exists, the more likely ``main`` has diverged. Merge frequently.

### 2. Isolate Scope (Partial Branching)

Use Pangolin's **Partial Branching** feature to only include the specific tables you are working on.

\*Bad\*: Branching the massive ``sales_fact`` table just to update the ``dim_users`` table.

\*Good\*: Create a branch containing only ``dim_users``.

### 3. Coordinate Compaction

Schedule maintenance jobs (Compaction/Vacuum) on the ``main`` branch during windows when other write branches are not active, or coordinate with your team.

### 4. Rebase/Sync Often

If your branch is long-lived, periodically **Rebase** (merge ``main`` into your feature branch) to resolve conflicts early in your isolated environment rather than at the final merge.

Source: `merge_operations.md`

## Merging & Conflict Resolution

Pangolin supports advanced branching and merging capabilities for data catalogs, similar to Git but specialized for data lake table formats like Iceberg. This document explains the merge logic, conflict detection, and resolution strategies.

# Overview

Pangolin's Enhanced Merge Conflict Resolution system provides intelligent conflict detection and resolution when merging branches. Unlike simple merge operations that fail on any conflict, this system:

- Detects** conflicts automatically during merge operations
- Categorizes** conflicts by type (schema, deletion, metadata, data overlap)
- Tracks** merge operations with full lifecycle management
- Enables** manual resolution with multiple strategies
- Supports** automatic resolution for non-critical conflicts

## Merge Logic

Pangolin uses a **3-Way Merge** algorithm when a common ancestor (base commit) can be detected. If no common ancestor is found (e.g., disjoint histories), it falls back to a 2-way merge (Source overwrites Target).

### 1. Base Commit Detection

- When you initiate a merge from `Feature Branch` to `Main`, Pangolin:
- Traverses the commit history of both branches.
- Finds the most recent common ancestor (Base Commit).
- Calculates changes:

- Source Changes:** Base -> Source Head
- Target Changes:** Base -> Target Head

### 2. Conflict Detection

Pangolin detects conflicts by comparing modifications to assets in Source and Target relative to the Base.

Scenario	Source	Target	Result
:---	:---	:---	:---
<b>No Change</b>	Same as Base	Same as Base	No Change
<b>Fast-Forward</b>	Modified	Same as Base	Update to Source
<b>Incoming</b>	Same as Base	Modified	Keep Target
<b>Conflict</b>	Modified (A)	Modified (B)	<b>CONFLICT</b>
<b>Deletion</b>	Deleted	Modified	<b>CONFLICT</b> (Deletion Conflict)
<b>Resurrection</b>	Created	Created	<b>CONFLICT</b> (if different content)

## Conflict Types

# 1. Schema Change Conflicts

Occur when the same table has different schemas in source and target branches.

**Example:** Both branches added columns independently or changed column types.

**Resolution:** Requires manual review to merge schema changes properly.

# 2. Deletion Conflicts

Occur when an asset is deleted in one branch but modified in another.

**Example:** Source deleted `temp\_table` while Target added data to it.

**Resolution:** Choose to keep (with modifications) or delete the asset.

# 3. Metadata Conflicts

Occur when asset properties (tags, description, properties) differ between branches.

**Resolution:** Can often be auto-resolved for non-critical properties.

# 4. Data Overlap Conflicts

Occur when the same partitions of a table are modified in both branches.

**Resolution:** Requires manual review to determine which version to keep.

## Merge Workflow

### 1. Initiate Merge

```
POST /api/v1/branches/merge
{
  "catalog": "production",
  "source_branch": "feature-branch",
  "target_branch": "main"
}
```

#### Response (Conflicts Detected):

```
{
  "status": "conflicted",
  "operation_id": "550e8400-e29b...",
  "conflicts": 3,
  "message": "Merge has 3 conflicts that need resolution"
}
```

## 2. List Conflicts

```
GET /api/v1/merge-operations/{operation_id}/conflicts
```

## 3. Resolve Conflicts

```
POST /api/v1/conflicts/{conflict_id}/resolve
{
  "strategy": "TakeSource",
  "resolved_value": null
}
```

### Resolution Strategies:

`TakeSource`: Use the source branch version.

`TakeTarget`: Use the target branch version.

`Manual`: Provide custom resolution (e.g., a merged schema) in `resolved\_value`.

`AutoMerge`: Automatically merge non-conflicting changes.

## 4. Complete Merge

After all conflicts are resolved:

```
POST /api/v1/merge-operations/{operation_id}/complete
```

## 5. Abort Merge

If you decide not to proceed:

```
POST /api/v1/merge-operations/{operation_id}/abort
```

## Merge Operation Lifecycle

```
Pending → Conflicted → Resolving → Ready → Completed
    ↓                                     ↑
    └──────────────────────────────────┘
    Aborted
```

## Managing Merge Operations

### List All Merge Operations

```
GET /api/v1/catalogs/{catalog_name}/merge-operations
```

## Get Merge Operation Details

```
GET /api/v1/merge-operations/{operation_id}
```

## Best Practices

**Merge Frequently:** Minimize divergence between branches.

**Review Thoroughly:** Schema and deletion conflicts can be destructive.

**Test After Merge:** Always validate data and schema after a complex merge.

Source: multi\_tenancy.md

## Multi-Tenancy

Pangolin is designed from the ground up as a multi-tenant platform, providing strong isolation between different organizations (tenants) while sharing the same underlying infrastructure.

## Core Hierarchy

The hierarchy in Pangolin follows a strict tenant-first model:

**Tenant:** The root isolation unit (e.g., "Acme Corp").

**Warehouse:** Storage configurations (S3, Azure, GCS) belong to a tenant.

**Catalog:** Data catalogs (Iceberg) belong to a tenant and reference a warehouse.

**User:** Users are scoped to a tenant (except for the global Root user).

## Isolation Levels

### 1. Data Isolation

Each tenant typically has its own **Warehouse**. This ensures that data files for different tenants are stored in separate S3 buckets or prefixes, preventing accidental cross-tenant data access.

### 2. Identity Isolation

Users created within a tenant cannot see or access resources in another tenant unless explicitly granted cross-tenant permissions. Each tenant can manage its own set of

users and service accounts.

## 3. Namespace Isolation

Identifiers (Catalogs, Namespaces, Assets) are scoped to the tenant. Two different tenants can have a catalog named `production` without any conflict.

## Tenant Management

Multi-tenancy is managed by the **Root** user (Platform Administrator).

**Root User (Platform Admin):** Global administrator with full access to all tenants. Responsible for tenant creation, listing, and deletion, as well as global system settings.

**Tenant Admin:** A highly privileged user within a specific tenant. Capable of managing all resources within their tenant, including users, service users, warehouses, catalogs, and permissions.

**Tenant User:** A standard user within a tenant, with access dictated by assigned RBAC policies.

## Configuration

When using the API or CLI, the tenant context is derived from:

**JWT Claims:** For standard user authentication.

**API Key Context:** For service users.

**X-Pangolin-Tenant Header:** Required for certain administrative operations or when using Pylceberg with cross-tenant access.

For more details on managing tenants, see the [CLI: Admin Tenants](#) guide.

Source: `pyiceberg_testing.md`

## Pylceberg Testing Guide

This guide provides instructions for testing Pangolin with Pylceberg. For detailed configuration options (Auth, Vending, Azure, GCP), see the [Pylceberg Integration Guides](#).

## Prerequisites

**Install Pylceberg and dependencies:**

```
pip install "pyiceberg[s3fs,pyarrow]" pyjwt
```

**Start Pangolin server in NO\_AUTH mode (for testing):**

```
PANGOLIN_NO_AUTH=1 cargo run --release --bin pangolin_api
```

### Or start with authentication (production):

```
cargo run --release --bin pangolin_api
```

## Quick Start (NO\_AUTH Mode)

The easiest way to test PyIceberg is using NO\_AUTH mode:

```
from pyiceberg.catalog import load_catalog

# Connect without authentication
catalog = load_catalog(
    "pangolin",
    **{
        "uri": "http://localhost:8080",
        "prefix": "analytics", # Your catalog name
    }
)

# Test connection
print(f"Connected: {catalog}")

# List namespaces
namespaces = catalog.list_namespaces()
print(f"Namespaces: {namespaces}")
```

## Complete Test Scenarios

### Scenario 1: Basic Connection

```
from pyiceberg.catalog import load_catalog

catalog = load_catalog(
    "pangolin",
    **{
        "uri": "http://localhost:8080",
        "prefix": "analytics",
    }
)

print(f"✓ Connected to catalog: {catalog}")
```

### Scenario 2: Namespace Operations

```
# Create namespace
catalog.create_namespace("test_namespace", properties={"owner": "test"})
print("✓ Created namespace")

# List namespaces
namespaces = catalog.list_namespaces()
print(f"✓ Namespaces: {namespaces}")

# Get namespace properties
props = catalog.load_namespace_properties("test_namespace")
print(f"✓ Properties: {props}")
```

## Scenario 3: Table Creation

```
from pyiceberg.schema import Schema
from pyiceberg.types import NestedField, IntegerType, StringType

# Define schema
schema = Schema(
    NestedField(1, "id", IntegerType(), required=True),
    NestedField(2, "name", StringType(), required=False),
    NestedField(3, "email", StringType(), required=False),
)

# Create table
table = catalog.create_table(
    identifier="test_namespace.users",
    schema=schema,
    location="memory://test-bucket/analytics/test_namespace/users"
)

print(f"✓ Created table: {table}")
```

## Scenario 4: Data Write

```
import pyarrow as pa

# Create data
data = pa.Table.from_pydict({
    "id": [1, 2, 3],
    "name": ["Alice", "Bob", "Charlie"],
    "email": ["alice@example.com", "bob@example.com", "charlie@example.com"]
})

# Write data
table.append(data)
print(f"✓ Wrote {len(data)} rows")
```

## Scenario 5: Data Read

```
# Scan table
scan = table.scan()
result = scan.to_arrow()

print(f"✓ Read {len(result)} rows")
print("\nData:")
print(result.to_pandas())
```

## Scenario 6: List Tables

```
# List tables in namespace
tables = catalog.list_tables("test_namespace")
print(f"✓ Tables: {tables}")
```

# Authentication

Pangolin supports both **OAuth 2.0** and **Basic Authentication**.

**Note:** For detailed configuration of OAuth providers (Google, GitHub, etc.), please see [Authentication](#).

## Option 1: Standard OAuth2 (Recommended for Production)

The most secure way to authenticate scripts and services is using the standard OAuth2 client credentials flow with a Service User.

```
from pyiceberg.catalog import load_catalog

catalog = load_catalog(
    "pangolin",
    **{
        "uri": "http://localhost:8080/v1/default/",
        "type": "rest",
        "credential": "service_user_uuid:service_user_api_key",
        "oauth2-server-uri": "http://localhost:8080/v1/default/v1/oauth/tokens",
        "scope": "catalog"
    }
)
```

[!NOTE]

**URI Suffix:** Ensure your `uri` ends with a trailing slash `/`.

**Auth Endpoint:** `oauth2-server-uri` must be the full path: `{base\_url}/v1/{catalog\_name}/v1/oauth/tokens`. Pangolin also supports the `/api/v1/iceberg/{catalog\_name}/v1/oauth/tokens` prefix for compatibility.

**Credential Format:** Must be `client\_id:client\_secret` (UUID:API\_Key).

## Option 2: Using an Existing Token (Temporary)

If you have a JWT token from the UI or another login flow, you can use it directly.

```
from pyiceberg.catalog import load_catalog

catalog = load_catalog(
    "pangolin",
    **{
        "uri": "http://localhost:8080/api/v1/iceberg/default",
        "type": "rest",
        "token": "YOUR_JWT_TOKEN",          # Token from UI or Login
    }
)
```

## Option 2: Basic Authentication (Root User Only)

**Only the Root User** can use the `credential` parameter directly. Standard Tenant Users must use a Token (Option 1).

```
catalog = load_catalog(
    "pangolin",
    **{
        "uri": "http://localhost:8080/iceberg/default",
        "type": "rest",
        "credential": "root_user:root_password",
    }
)
```

## Obtaining a Token (for Scripts/Service Accounts)

For automated scripts acting as a Tenant User ("Service Account"), you must programmatically exchange your credentials for a token before initializing the catalog.

```
import requests
from pyiceberg.catalog import load_catalog

# 1. Login to get Token
response = requests.post("http://localhost:8080/api/v1/users/login", json={
    "username": "my-service-account",
    "password": "my-secure-password",
    "tenant_id": "00000000-0000-0000-0000-000000000001" # Optional if user unique
})
token = response.json()["token"]

# 2. Use Token
catalog = load_catalog(
    "pangolin",
```

```

    **{
        "uri": "http://localhost:8080/iceberg/default",
        "type": "rest",
        "token": token
    }
)

```

## Complete Test Script

Save as `test\_pyiceberg.py`:

```

#!/usr/bin/env python3
"""
PyIceberg Integration Test for Pangolin
"""

import sys
from pyiceberg.catalog import load_catalog
from pyiceberg.schema import Schema
from pyiceberg.types import NestedField, IntegerType, StringType
import pyarrow as pa

def test_connection():
    """Test 1: Basic connection"""
    print("\n=== Test 1: Connection ===")
    catalog = load_catalog(
        "pangolin",
        **{
            "uri": "http://localhost:8080",
            "prefix": "analytics",
        }
    )
    print(f"✓ Connected to catalog: {catalog}")
    return catalog

def test_namespaces(catalog):
    """Test 2: Namespace operations"""
    print("\n=== Test 2: Namespaces ===")

    # List existing
    namespaces = catalog.list_namespaces()
    print(f"Existing namespaces: {namespaces}")

    # Create namespace
    catalog.create_namespace("test_namespace")
    print(f"✓ Created namespace: test_namespace")

    # Verify
    namespaces = catalog.list_namespaces()
    print(f"✓ Updated namespaces: {namespaces}")

def test_table_creation(catalog):

```

```

"""Test 3: Table creation"""
print("\n=== Test 3: Table Creation ===")

schema = Schema(
    NestedField(1, "id", IntegerType(), required=True),
    NestedField(2, "name", StringType(), required=False),
    NestedField(3, "email", StringType(), required=False),
)

table = catalog.create_table(
    identifier="test_namespace.users",
    schema=schema,
    location="memory://test-bucket/analytics/test_namespace/users"
)

print(f"✓ Created table: {table}")
return table

def test_data_write(table):
    """Test 4: Write data"""
    print("\n=== Test 4: Data Write ===")

    data = pa.Table.from_pydict({
        "id": [1, 2, 3],
        "name": ["Alice", "Bob", "Charlie"],
        "email": ["alice@example.com", "bob@example.com", "charlie@example.com"]
    })

    table.append(data)
    print(f"✓ Wrote {len(data)} rows")

def test_data_read(table):
    """Test 5: Read data"""
    print("\n=== Test 5: Data Read ===")

    scan = table.scan()
    result = scan.to_arrow()

    print(f"✓ Read {len(result)} rows")
    print("\nData:")
    print(result.to_pandas())

def test_list_tables(catalog):
    """Test 6: List tables"""
    print("\n=== Test 6: List Tables ===")

    tables = catalog.list_tables("test_namespace")
    print(f"✓ Tables in test_namespace: {tables}")

def main():
    print("=" * 60)
    print("Pangolin PyIceberg Integration Test")
    print("=" * 60)

```

```

try:
    catalog = test_connection()
    test_namespaces(catalog)
    table = test_table_creation(catalog)
    test_data_write(table)
    test_data_read(table)
    test_list_tables(catalog)

    print("\n" + "=" * 60)
    print("✓ ALL TESTS PASSED!")
    print("=" * 60)
    return 0

except Exception as e:
    print("\n" + "=" * 60)
    print(f"✗ TESTS FAILED: {e}")
    print("=" * 60)
    import traceback
    traceback.print_exc()
    return 1

if __name__ == "__main__":
    sys.exit(main())

```

Run with:

```
python3 test_pyiceberg.py
```

## Troubleshooting

### 401 Unauthorized

#### **In NO\_AUTH mode:**

Ensure server is started with `PANGOLIN\_NO\_AUTH=1`

Restart server if you changed the environment variable

#### **With authentication:**

Check token is valid and not expired

Verify token contains correct tenant\_id

Decode token to inspect: `jwt.decode(token, options={"verify\_signature": False})`

### 404 Not Found

Verify catalog name in `prefix` parameter matches existing catalog

-

Check catalog was created: ``curl http://localhost:8080/api/v1/catalogs``

## Connection Refused

Ensure Pangolin server is running

Check server is listening on port 8080

Verify no firewall blocking connection

## Table Creation Fails

Check namespace exists first

Verify storage location is accessible

Review server logs for detailed error

## Verification Checklist

- ☐ Server starts successfully
- ☐ Connection to catalog works
- ☐ Can list namespaces
- ☐ Can create namespace
- ☐ Can create table
- ☐ Can write data
- ☐ Can read data back
- ☐ Can list tables

## Next Steps

After successful testing:

### **Production Setup:**

Remove ``PANGOLIN_NO_AUTH`` environment variable

Set up proper JWT secret: ``PANGOLIN_JWT_SECRET``

Configure token generation endpoint

Implement token refresh logic

### **Multi-Tenant Testing:**

Create multiple tenants

Generate tokens for each tenant

Verify tenant isolation

### **Performance Testing:**

Large dataset writes

Concurrent operations

Query performance

## Additional Resources

[Client Configuration](#) - Detailed client setup

[Getting Started](#) - Quick start guide

[Warehouse Management](#) - Warehouse and catalog setup

Source: rbac.md

# Role-Based Access Control (RBAC)

## Overview

Pangolin implements a hierarchical role-based access control system to manage permissions across tenants, catalogs, and resources.

### **Key Features:**

3-tier role hierarchy (Root, TenantAdmin, TenantUser)

Granular permission scopes (Catalog, Namespace, Asset, Tag)

Service user support for programmatic access

Tenant isolation

## User Roles

### Root

**Scope:** Global (all tenants)

**Permissions:** Full system access

#### **Can:**

Create/delete tenants

Manage all users across all tenants

-

Access all catalogs and resources

Configure system-wide settings

**Use Case:** System administrators

## TenantAdmin

**Scope:** Single tenant

**Permissions:** Full access within their tenant

**Can:**

Create/delete users in their tenant

Create/delete catalogs, warehouses

Manage permissions for their tenant

Create service users

Create federated catalogs

**Cannot:**

Access other tenants

Create new tenants

Modify system settings

**Use Case:** Team leads, data platform administrators

## TenantUser

**Scope:** Single tenant with granular permissions

**Permissions:** Based on assigned permissions

**Can:**

Read/write to authorized catalogs

Create tables in authorized namespaces

Query authorized tables

**Cannot:**

Create users or service users

Modify tenant settings

Access unauthorized resources

**Use Case:** Data engineers, data scientists, analysts

# Permission Scopes

## Catalog-Level

Permissions apply to an entire catalog and all its namespaces/tables.

**Example:** Grant user read access to `analytics` catalog

```
{
  "user_id": "uuid",
  "scope": "Catalog",
  "resource": "analytics",
  "action": "Read"
}
```

## Namespace-Level

Permissions apply to a specific namespace and its tables.

**Example:** Grant user write access to `sales.transactions` namespace

```
{
  "user_id": "uuid",
  "scope": "Namespace",
  "resource": "analytics/sales",
  "action": "Write"
}
```

## Asset-Level

Permissions apply to a specific table or view.

**Example:** Grant user read access to specific table

```
{
  "user_id": "uuid",
  "scope": "Asset",
  "resource": "analytics/sales/transactions",
  "action": "Read"
}
```

## Permission Actions

**Read:** View metadata, query data

**Write:** Create tables, insert/update data

**Delete:** Drop tables, delete data

**Admin:** Full control (create, read, update, delete)

## Managing Users

### Create User

```
POST /api/v1/users
Authorization: Bearer <admin-token>
Content-Type: application/json

{
  "username": "data_engineer",
  "email": "engineer@example.com",
  "password": "secure-password",
  "role": "tenant-user"
}
```

### Assign Role to User

```
POST /api/v1/users/{user_id}/roles
Authorization: Bearer <admin-token>
Content-Type: application/json

{
  "role-id": "role-uuid"
}
```

### Grant Permission (to User or Role)

```
POST /api/v1/permissions
Authorization: Bearer <admin-token>
Content-Type: application/json

{
  "user-id": "optional-user-uuid",
  "role-id": "optional-role-uuid",
  "scope": "Catalog",
  "resource": "analytics",
  "action": "Read"
}
```

### Revoke Permission

```
DELETE /api/v1/permissions/{permission_id}
Authorization: Bearer <admin-token>
```

# Service Users

Service users are programmatic identities with API key authentication. See [Service Users](#) for details.

## Key Points:

Service users inherit the same RBAC system

Can be assigned TenantAdmin or TenantUser roles

Authenticate via `X-API-Key` header

Support expiration and rotation

## Best Practices

### 1. Principle of Least Privilege

Grant minimum necessary permissions:

Use TenantUser role by default

Grant catalog-level access only when needed

Prefer namespace or asset-level permissions

### 2. Separate Service Users by Purpose

Create dedicated service users for each use case:

```
- ci_pipeline_user (Write to staging catalog)
- analytics_reader (Read from analytics catalog)
- etl_processor (Write to raw, read from staging)
```

### 3. Regular Permission Audits

Review user permissions periodically:

```
GET /api/v1/users/{user_id}/permissions
```

### 4. Use Groups (Future)

When available, use groups to manage permissions for teams.

### 5. Monitor Access

Review audit logs for unauthorized access attempts:

```
GET /api/v1/audit?user_id={user_id}
```

## Permission Inheritance

Permissions follow a hierarchical model:

```
Root
├─ TenantAdmin (Tenant A)
│   └─ TenantUser (Catalog: analytics)
│       └─ Namespace: sales
│           └─ Asset: transactions
```

### Rules:

Root has access to everything

TenantAdmin has access to all resources in their tenant

TenantUser permissions are explicitly granted

More specific permissions override general ones

## Common Scenarios

### Scenario 1: Data Engineer with Full Catalog Access

```
# Create user
POST /api/v1/users
{
  "username": "data_engineer",
  "role": "tenant-user"
}

# Grant catalog-level write access
POST /api/v1/permissions
{
  "user-id": "uuid",
  "scope": "Catalog",
  "resource": "analytics",
  "action": "Write"
}
```

### Scenario 2: Analyst with Read-Only Access

```
# Create user
POST /api/v1/users
{
  "username": "analyst",
```

```
"role": "tenant-user"
}

# Grant namespace-level read access
POST /api/v1/permissions
{
  "user-id": "uuid",
  "scope": "Namespace",
  "resource": "analytics/sales",
  "action": "Read"
}
```

## Scenario 3: CI/CD Pipeline

```
# Create service user
POST /api/v1/service-users
{
  "name": "ci_pipeline",
  "role": "tenant-user",
  "expires_in_days": 90
}

# Grant write access to staging catalog
POST /api/v1/permissions
{
  "user-id": "service_user_uuid",
  "scope": "Catalog",
  "resource": "staging",
  "action": "Write"
}
```

## Troubleshooting

### "Forbidden" errors

**Cause:** User lacks required permissions.

**Solution:**

Check user's role: `GET /api/v1/users/{user\_id}`

List user's permissions: `GET /api/v1/users/{user\_id}/permissions`

Grant missing permission

### Permission not taking effect

**Cause:** Permission cache or session issue.

## Solution:

Re-authenticate to get new JWT token

Verify permission was created: `GET /api/v1/permissions`

## Related Documentation

[Service Users](#) - API key authentication

[Authentication](#) - User authentication

[Audit Logs](#) - Access monitoring

[Permissions System](#) - Permission details

Source: security\_vending.md

## Security & Credential Vending (Multi-Cloud)

Pangolin provides mechanisms to securely vend temporary credentials to clients for S3, Azure ADLS Gen2, and Google Cloud Storage, enabling direct data access while maintaining security.

## Overview

Instead of sharing long-term cloud credentials with clients (e.g., Spark jobs, Dremio, Trino), Pangolin acts as a trusted intermediary. It authenticates the client and then issues temporary, scoped credentials for specific storage resources.

### Benefits:

- ~No long-term credentials in client configurations
- ~Automatic credential rotation (STS for S3, OAuth2 for Azure/GCP)
- ~Scoped access to specific table locations
- ~Centralized audit trail of data access
- ~Support for cross-account/cross-cloud access
- ~**Multi-cloud support:** S3, Azure ADLS Gen2, Google Cloud Storage
- ⚠ **Local Filesystem:** Supported for dev/test (no credential vending involved)

## Configuration

## Prerequisites

**AWS Credentials:** Pangolin needs AWS credentials with permissions to:

Call `sts:AssumeRole`` (for STS vending)

Access S3 buckets (for static credential vending)

Generate presigned URLs

**IAM Role** (for STS vending): Create an IAM role that Pangolin can assume with S3 access permissions.

## Environment Variables

```
# AWS Configuration
AWS_REGION=us-east-1
AWS_ACCESS_KEY_ID=AKIA...
AWS_SECRET_ACCESS_KEY=...

# For STS Credential Vending
PANGOLIN_STS_ROLE_ARN=arn:aws:iam::123456789012:role/PangolinDataAccess
PANGOLIN_STS_SESSION_DURATION=3600 # 1 hour (default)

# For MinIO or S3-compatible storage
AWS_ENDPOINT_URL=http://minio:9000
AWS_ALLOW_HTTP=true
```

## Warehouse Configuration

To enable credential vending, configure the ``vending_strategy`` in your warehouse definition.

### **AWS S3 (STS Mode):**

```
curl -X POST http://localhost:8080/api/v1/warehouses \
-H "X-Pangolin-Tenant: <tenant-id>" \
-H "Content-Type: application/json" \
-d '{
  "name": "production_warehouse",
  "storage_config": {
    "type": "s3",
    "bucket": "my-data-bucket",
    "region": "us-east-1",
    "s3.role-arn": "arn:aws:iam::123456789012:role/PangolinDataAccess"
  },
  "vending_strategy": {
    "AwsSts": {
      "role_arn": "arn:aws:iam::123456789012:role/PangolinDataAccess"
    }
  }
}'
```

### **AWS S3 (Static Mode):**

```
curl -X POST http://localhost:8080/api/v1/warehouses \
-H "X-Pangolin-Tenant: <tenant-id>" \
-H "Content-Type: application/json" \
-d '{
  "name": "dev_warehouse",
  "storage_config": {
    "type": "s3",
    "bucket": "my-dev-bucket",
    "region": "us-east-1",
    "s3.access-key-id": "AKIA...",
    "s3.secret-access-key": "...",
  },
  "vending_strategy": "AwsStatic"
}'
```

## Azure ADLS Gen2 Configuration

### OAuth2 Mode (Recommended):

```
curl -X POST http://localhost:8080/api/v1/warehouses \
-H "X-Pangolin-Tenant: <tenant-id>" \
-H "Content-Type: application/json" \
-d '{
  "name": "azure_warehouse",
  "storage_config": {
    "type": "azure",
    "azure.account-name": "mystorageaccount",
    "azure.container": "data",
    "azure.client-id": "azure-client-id",
    "azure.client-secret": "azure-client-secret",
    "azure.tenant-id": "azure-tenant-id"
  },
  "vending_strategy": {
    "AzureOAuth": {
      "client_id": "azure-client-id",
      "client_secret": "azure-client-secret",
      "tenant_id": "azure-tenant-id"
    }
  }
}'
```

### Account Key Mode:

```
curl -X POST http://localhost:8080/api/v1/warehouses \
-H "X-Pangolin-Tenant: <tenant-id>" \
-H "Content-Type: application/json" \
-d '{
  "name": "azure_warehouse",
  "storage_config": {
    "type": "azure",
    "azure.account-name": "mystorageaccount",
```

```

    "azure.container": "data",
    "azure.account-key": "your-account-key"
  },
  "vending_strategy": "AzureSas"
}'

```

## Google Cloud Storage Configuration

```

curl -X POST http://localhost:8080/api/v1/warehouses \
-H "X-Pangolin-Tenant: <tenant-id>" \
-H "Content-Type: application/json" \
-d '{
  "name": "gcp_warehouse",
  "storage_config": {
    "type": "gcs",
    "gcp.project-id": "my-gcp-project",
    "gcp.bucket": "my-data-bucket",
    "gcp.service-account-key": "{...json key...}"
  },
  "vending_strategy": "GcpDownscoped"
}'

```

## Features

### Presigned URLs

Get a presigned URL to download a specific file (e.g., a metadata file or data file) without needing AWS credentials.

**Endpoint:** `GET /v1/{prefix}/namespaces/{namespace}/tables/{table}/presign?location=s3://bucket/key``

**Request:**

```

curl
"http://localhost:8080/v1/analytics/namespaces/sales/tables/transactions/presign?location=s3://my-bucket/data/file.parquet" \
-H "Authorization: Bearer <token>" \
-H "X-Pangolin-Tenant: <tenant-id>"

```

**Response:**

```

{
  "url":
  "https://bucket.s3.amazonaws.com/key?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=.
  .."
}

```

**Use Case:** Web applications, data preview tools, or clients that can't handle AWS credentials.

## PyIceberg Integration

PyIceberg automatically uses Pangolin's credential vending when configured correctly for all supported cloud providers.

### Automatic Credential Vending (S3)

```
from pyiceberg.catalog import load_catalog

catalog = load_catalog(
    "pangolin",
    **{
        "uri": "http://localhost:8080",
        "prefix": "analytics",
        "token": "your-jwt-token",
        # No S3 credentials needed - Pangolin vends them automatically!
    }
)

# PyIceberg will request credentials from Pangolin for each table access
table = catalog.load_table("sales.transactions")
df = table.scan().to_pandas() # Pangolin vends S3 credentials automatically
```

### Automatic Credential Vending (Azure)

```
catalog = load_catalog(
    "pangolin_azure",
    **{
        "uri": "http://localhost:8080",
        "prefix": "azure_catalog",
        "token": "your-jwt-token",
        # No Azure credentials needed - Pangolin vends them automatically!
        # Pangolin provides: adls.token, adls.account-name, adls.container
    }
)

table = catalog.load_table("sales.transactions")
df = table.scan().to_pandas() # Pangolin vends Azure credentials automatically
```

### Automatic Credential Vending (GCP)

```
catalog = load_catalog(
    "pangolin_gcp",
    **{
        "uri": "http://localhost:8080",
        "prefix": "gcp_catalog",
```

```

        "token": "your-jwt-token",
        # No GCP credentials needed - Pangolin vends them automatically!
        # Pangolin provides: gcp-oauth-token, gcp-project-id
    }
)

table = catalog.load_table("sales.transactions")
df = table.scan().to_pandas() # Pangolin vends GCP credentials automatically

```

## How it works:

Pylceberg requests table metadata from Pangolin

Pangolin includes temporary cloud credentials in the response (based on warehouse type)

Pylceberg uses these credentials to read data files from cloud storage

Credentials expire after the configured duration (default: 1 hour)

## Client-Provided Credentials

If you prefer to manage credentials yourself:

### S3:

```

catalog = load_catalog(
    "pangolin",
    **{
        "uri": "http://localhost:8080",
        "prefix": "analytics",
        "token": "your-jwt-token",
        "s3.access-key-id": "AKIA...",
        "s3.secret-access-key": "...",
    }
)

```

### Azure:

```

catalog = load_catalog(
    "pangolin_azure",
    **{
        "uri": "http://localhost:8080",
        "prefix": "azure_catalog",
        "token": "your-jwt-token",
        "adls.account-name": "mystorageaccount",
        "adls.account-key": "...",
    }
)

```

### GCP:

```

catalog = load_catalog(
    "pangolin_gcp",
    **{
        "uri": "http://localhost:8080",
        "prefix": "gcp_catalog",
        "token": "your-jwt-token",
        "gcp-project-id": "my-project",
        "gcs.service-account-key": "/path/to/key.json",
    }
)

```

## IAM Policy Examples

### Pangolin Service Role Policy

This is the policy for the IAM role that Pangolin assumes to vend credentials:

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
        "s3:PutObject",
        "s3:DeleteObject",
        "s3:ListBucket"
      ],
      "Resource": [
        "arn:aws:s3:::my-data-bucket/*",
        "arn:aws:s3:::my-data-bucket"
      ]
    }
  ]
}

```

### Trust Policy for Pangolin

Allow Pangolin's AWS account/role to assume the data access role:

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::YOUR-PANGOLIN-ACCOUNT:role/PangolinService"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}

```

```
}  
]  
}
```

## Security Best Practices

### 1. Use STS Credential Vending

Always use `AwsSts` strategy in production to vend temporary credentials instead of sharing static credentials.

### 2. Scope Credentials to Table Locations

Pangolin vends credentials scoped to specific table locations, limiting blast radius if credentials are compromised.

### 3. Set Short Expiration Times

Configure `PANGOLIN\_STS\_SESSION\_DURATION` to the minimum time needed (default: 3600 seconds / 1 hour).

### 4. Use IAM Conditions

Add IAM policy conditions to restrict access by IP, time, or other factors:

```
{  
  "Condition": {  
    "IpAddress": {  
      "aws:SourceIp": "10.0.0.0/8"  
    }  
  }  
}
```

### 5. Monitor Credential Usage

Review CloudTrail logs for STS AssumeRole calls to detect unusual patterns.

### 6. Rotate Static Credentials

If using static credentials (`AwsStatic`), rotate them regularly.

## Troubleshooting

"Access Denied" when reading data

**Cause:** Vended credentials don't have permissions for the S3 location.

**Solution:**

Verify the IAM role has S3 permissions for the table location

Check the role ARN in warehouse configuration

Verify Pangolin can assume the role: ``aws sts assume-role role-arn <arn> role-session-name test``

## "Credentials expired" errors

**Cause:** STS credentials expired during a long-running query.

**Solution:**

Increase ``PANGOLIN_STS_SESSION_DURATION`` (max: 43200 seconds / 12 hours)

Configure your client to refresh credentials automatically

For very long queries, consider using static credentials

## Pylceberg not using vended credentials

**Cause:** Pylceberg may be using client-provided credentials instead.

**Solution:**

Remove ``s3.access-key-id`` and ``s3.secret-access-key`` from Pylceberg config

Verify warehouse has ``vending_strategy`` configured correctly

Check Pangolin logs for credential vending requests

## "Invalid security token" errors

**Cause:** Clock skew between Pangolin server and AWS.

**Solution:**

Sync server time with NTP: ``sudo ntpdate -s time.nist.gov``

Verify server timezone is set correctly

Check CloudTrail for timestamp-related errors

## Cross-account access not working

**Cause:** Trust policy or permissions issue.

**Solution:**

Verify trust policy allows Pangolin's role to assume the target role

Check both the trust policy and the permissions policy

Test with AWS CLI: `aws sts assume-role role-arn <target-role> role-session-name test``

## Advanced Configuration

### Custom Session Duration

```
# Set custom session duration (in seconds)
PANGOLIN_STS_SESSION_DURATION=7200 # 2 hours
```

### External ID for Cross-Account Access

For enhanced security in cross-account scenarios:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::PANGOLIN-ACCOUNT:role/PangolinService"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "StringEquals": {
          "sts:ExternalId": "unique-external-id-12345"
        }
      }
    }
  ]
}
```

Set in Pangolin:

```
PANGOLIN_STS_EXTERNAL_ID=unique-external-id-12345
```

### Regional Endpoints

For better performance, use regional STS endpoints:

```
AWS_STS_REGIONAL_ENDPOINTS=regional
```

## Related Documentation

[Warehouse Management](#) - Creating and configuring warehouses

Authentication - User authentication and tokens

Client Configuration - Pylceberg, Spark, Trino setup

AWS S3 Storage - S3 storage backend configuration

## Source: service\_users.md

# Service Users

Service Users are dedicated programmatic identities designed for secure machine-to-machine access. They bypass the standard JWT session flow in favor of long-lived, high-entropy API keys.

## Usage Guides

### 1. Via CLI (`pangolin-admin`)

Administrators can provision and rotate keys directly from the terminal.

#### Create Service User:

```
pangolin-admin create-service-user \  
  --name "etl-pipeline" \  
  --role "TenantUser"
```

[!NOTE]

Ensure you record the **Service User ID (UUID)** from the output. If the CLI does not display the ID, use the Management UI or the API directly to retrieve it after creation.

#### Rotate API Key:

```
pangolin-admin rotate-service-user-key --id <service-user-uuid>
```

### 2. Via Management UI

Navigate to **Identity -> Service Users**.

Click **Create Service User**, enter a name and expiration date.

**Important:** The API key will be displayed in a modal. **Copy it immediately**, as it will never be shown again.

**Key Rotation:** To rotate a key, click the **Rotate Key** icon. A new key will be generated and displayed once; the old key is immediately invalidated.

To deactivate a user, use the delete action or manage their active status (roadmap feature).

# Authentication

Service users authenticate by sending the `X-API-Key` header with every request.

```
curl -H "X-API-Key: pgl_YOUR_SECRET_KEY" \
https://your-pangolin-api.com/api/v1/catalogs
```

## Integration: PyIceberg (Standard OAuth2)

PyIceberg supports standard OAuth2, which is the recommended way to use Service Users.

```
from pyiceberg.catalog import load_catalog

catalog = load_catalog(
    "pangolin",
    **{
        "uri": "http://localhost:8080/v1/default/",
        "credential": "<service_user_uuid>:<api_key>",
        "oauth2-server-uri": "http://localhost:8080/v1/default/v1/oauth/tokens",
        "scope": "catalog",
        "type": "rest",
    }
)
```

## Integration: Custom Clients (X-API-Key)

For simple HTTP clients that don't support OAuth2, you can use the header:

```
import requests
headers = {"X-API-Key": "pgl_YOUR_SECRET_KEY"}
response = requests.get("https://api.pangolin.io/api/v1/catalogs", headers=headers)
```

## Best Practices

**Never Share Keys:** Treat API keys exactly like root passwords.

**Rotation:** Rotate keys at least every 90 days. If a key is potentially exposed, rotate it immediately.

**Least Privilege:** Only grant `TenantAdmin` to service users that specifically need to manage other users or infrastructure. Most pipelines only need `TenantUser`.

**Monitor Usage:** Check the `last\_used` timestamp in the UI to identify and prune stale identities.

Source: `table_formats.md`

# Modern Table Formats (Delta, Hudi, Paimon)

Pangolin supports cataloging external table formats like **Delta Lake**, **Apache Hudi**, and **Apache Paimon** as Generic Assets. This allows you to maintain a unified discovery layer across your entire data estate, even if some tables are managed by different engines (e.g., Databricks, EMR, Flink).

## Overview

Unlike Iceberg tables, which Pangolin manages natively (handling commits, snapshots, and manifest files), other modern table formats are treated as **governed pointers**.

### What Pangolin Does:

- **Discovery:** Tables appear in search results with their specific type (e.g., ``DeltaTable``).

- **Governance:** Apply RBAC policies and tags to these tables.

- **Lineage:** Track them as sources or sinks in your data pipelines.

- **Metadata:** Store custom properties (e.g., ``managed_by: databricks``, ``last_compacted: 2023-10-01``).

### What Pangolin Does NOT Do:

- **Transaction Management:** Pangolin does not process Delta/Hudi commits. The query engine (Spark/Flink/Trino) handles the ACID guarantees.

- **Schema Evolution:** You cannot change the schema of a Delta table via Pangolin's API.

## Supported Formats

### 1. Delta Lake (``DeltaTable``)

Developed by Databricks, widely used in Spark ecosystems.

**Root Path:** The directory containing the ``_delta_log`` folder.

**Best Practice:** Point Pangolin to the table root, not the ``_delta_log`` folder itself.

```
`s3://bucket/warehouse/db/my_table`
```

```
`s3://bucket/warehouse/db/my_table/_delta_log`
```

### 2. Apache Hudi (``HudiTable``)

Popular for streaming and upsert-heavy workloads.

**Root Path:** The directory containing the ``.hoodie`` folder.

**Best Practice:** Use properties to note the table type (Copy-on-Write vs

Merge-on-Read).

### 3. Apache Paimon (`ApachePaimon`)

A streaming data lake format (formerly Flink Table Store).

**Root Path:** The root directory of the Paimon table.

## How to Catalog These Tables

### Uses Case: The "Unmanaged" Catalog

You have existing pipelines writing Delta tables using Databricks. You want these tables to be discoverable by analysts in Pangolin alongside your native Iceberg tables.

### Option 1: Via Management UI (Data Explorer)

Navigate to the **Data Explorer**.

Drill down into the **Catalog** and **Namespace** where you want to register the table.

Click the "**Register Asset**" button (top-right).

Fill in the details:

**Name:** `customer\_churn\_prediction`

**Type:** Select `Delta Lake Table` (or Hudi/Paimon).

**Location:** `s3://finance-data/delta/churn\_preds/`

(Optional) Add Properties:

`owner`: `data-science-team`

`update\_frequency`: `daily`

Click **Register**.

### Option 2: Via REST API

```
# Register a Delta Table
curl -X POST http://localhost:8080/api/v1/catalogs/analytics/namespaces/gold/assets \
  -H "Authorization: Bearer <token>" \
  -H "X-Pangolin-Tenant: <tenant-id>" \
  -H "Content-Type: application/json" \
  -d '{
    "name": "customer_churn_prediction",
    "kind": "DeltaTable",
    "location": "s3://finance-data/delta/churn_preds/",
    "properties": {
      "provider": "delta",
```

```
"managed_by": "databricks-job-123"  
}  
'
```

## Interoperability Patterns

### XTable (formerly OneTable)

If you use tools like **Apache XTable** to translate metadata between formats (e.g., Delta -> Iceberg), you can register the *\*same\** data location twice with different semantics:

**Primary:** Register as ``DeltaTable`` (the source of truth).

**Read-Replica:** Register as ``IcebergTable`` (using the XTable-generated metadata) if you want Pangolin to serve it to Iceberg clients.

### UniForm (Delta Lake 3.0)

If your Delta tables use **UniForm** (Universal Format) to generate Iceberg metadata:

You should register the table as an ``IcebergTable`` in Pangolin pointing to the ``metadata`` directory generated by UniForm if you want full Iceberg client compatibility.

Registering it as a ``DeltaTable`` is still useful for discovery but avoids Pangolin trying to serve the Iceberg metadata directly.

## FAQ

**Q: Can I run SQL queries on these tables via Pangolin?**

**A:** No. Pangolin is a catalog, not a query engine. You use Dremio, Trino, Spark, or StarRocks to query the tables. Pangolin ensures you can *\*find\** them and know *\*where\** they are.

**Q: Do these tables show up in Iceberg clients?**

**A:** No. Generic assets are filtered out of the standard Iceberg REST API responses (``loadTable``) because they don't have valid Iceberg metadata. They only appear in Pangolin's asset search and discovery APIs.

Source: tag\_management.md

## Tag Management

For details on configured Tag-Based Access Control policies, see [Permissions System](#).

Tags allow for categorical organization and security governance of assets. ory, such as releases or stable versions.

## Data Model

A `Tag` consists of:

`name`: Unique identifier for the tag within a catalog.

`commit\_id`: The ID of the commit the tag points to.

## API Endpoints

### Create Tag

`POST /api/v1/tags`

#### Headers:

`Authorization`: `Bearer <token>`

`X-Pangolin-Tenant`: `<Tenant-ID>`

#### Request Body:

```
{
  "name": "v1.0",
  "commit_id": "uuid-of-commit"
}
```

#### Response:

```
{
  "name": "v1.0",
  "commit_id": "uuid-of-commit"
}
```

### List Tags

`GET /api/v1/tags`

#### Response:

```
[
  {
    "name": "v1.0",
    "commit_id": "uuid-of-commit"
  }
]
```

### Delete Tag

`DELETE /api/v1/tags/:name`

**Response:** `204 No Content`

Source: time\_travel.md

## Time Travel Queries

### Overview

Pangolin supports Iceberg's time-travel capabilities, allowing users to query tables as they existed at a specific point in time or snapshot. This is essential for:

Auditing historical data states

Reproducing past analysis results

Recovering from accidental data modifications

Comparing data across time periods

### Usage

Time-travel queries are performed using Iceberg's standard query parameters.

#### By Snapshot ID

Query the table as it was at a specific snapshot:

```
GET /v1/analytics/namespaces/sales/tables/transactions?snapshot-id=123456789
Authorization: Bearer <token>
X-Pangolin-Tenant: <tenant-id>
```

#### By Timestamp

Query the table as it was at a specific timestamp (milliseconds since epoch):

```
GET /v1/analytics/namespaces/sales/tables/transactions?as-of-timestamp=1678886400000
Authorization: Bearer <token>
X-Pangolin-Tenant: <tenant-id>
```

## Client Examples

### Pylceberg

```
from pyiceberg.catalog import load_catalog
from datetime import datetime
```

```

catalog = load_catalog(
    "pangolin",
    **{
        "uri": "http://localhost:8080",
        "prefix": "analytics",
        "token": "your-jwt-token",
    }
)

table = catalog.load_table("sales.transactions")

# Query by snapshot ID
snapshot_id = 123456789
df = table.scan(snapshot_id=snapshot_id).to_pandas()

# Query by timestamp
timestamp = datetime(2024, 1, 15, 10, 30, 0)
df = table.scan(as_of_timestamp=timestamp).to_pandas()

# Query as of yesterday
from datetime import timedelta
yesterday = datetime.now() - timedelta(days=1)
df = table.scan(as_of_timestamp=yesterday).to_pandas()

```

## Spark SQL

```

-- Query by snapshot ID
SELECT * FROM analytics.sales.transactions VERSION AS OF 123456789;

-- Query by timestamp
SELECT * FROM analytics.sales.transactions TIMESTAMP AS OF '2024-01-15 10:30:00';

-- Query as of yesterday
SELECT * FROM analytics.sales.transactions TIMESTAMP AS OF current_timestamp() -
INTERVAL 1 DAY;

```

## Trino

```

-- Query by snapshot ID
SELECT * FROM iceberg.analytics.sales.transactions FOR VERSION AS OF 123456789;

-- Query by timestamp
SELECT * FROM iceberg.analytics.sales.transactions FOR TIMESTAMP AS OF TIMESTAMP
'2024-01-15 10:30:00';

```

## Snapshot Management

# List Snapshots

```
from pyiceberg.catalog import load_catalog

catalog = load_catalog("pangolin", **config)
table = catalog.load_table("sales.transactions")

# Get table metadata
metadata = table.metadata

# List all snapshots
for snapshot in metadata.snapshots:
    print(f"Snapshot ID: {snapshot.snapshot_id}")
    print(f"Timestamp: {snapshot.timestamp_ms}")
    print(f"Operation: {snapshot.summary.get('operation')}")
    print("---")
```

## Get Snapshot Details

```
# Get specific snapshot
snapshot_id = 123456789
snapshot = table.snapshot_by_id(snapshot_id)

print(f"Snapshot ID: {snapshot.snapshot_id}")
print(f"Parent ID: {snapshot.parent_snapshot_id}")
print(f"Timestamp: {snapshot.timestamp_ms}")
print(f"Manifest List: {snapshot.manifest_list}")
```

## Implementation Details

When a time-travel parameter is provided:

**Pangolin retrieves** the current metadata for the table

**Verifies** if the requested snapshot exists in the metadata history

**Returns** the metadata pointing to the requested snapshot

**Client reads** data files from the snapshot's manifest list

If the snapshot is not found (e.g., expired and cleaned up), a `404 Not Found` is returned.

## Snapshot Retention

### Default Retention

Iceberg tables retain snapshots based on configuration:

**Default:** Last 5 snapshots

**Time-based:** Snapshots from last 7 days

## Configure Retention

```
from pyiceberg.catalog import load_catalog

catalog = load_catalog("pangolin", **config)
table = catalog.load_table("sales.transactions")

# Update table properties
table.update_properties(
    **{
        "history.expire.max-snapshot-age-ms": "604800000", # 7 days
        "history.expire.min-snapshots-to-keep": "10"
    }
)
```

## Expire Snapshots

```
# Expire snapshots older than 7 days
table.expire_snapshots(
    older_than=datetime.now() - timedelta(days=7),
    retain_last=5
)
```

## Use Cases

### Audit Historical Data

```
# Compare data from last month to current
last_month = datetime.now() - timedelta(days=30)
df_last_month = table.scan(as_of_timestamp=last_month).to_pandas()
df_current = table.scan().to_pandas()

# Find differences
diff = df_current.merge(df_last_month, how='outer', indicator=True)
new_records = diff[diff['_merge'] == 'left_only']
deleted_records = diff[diff['_merge'] == 'right_only']
```

## Recover from Accidental Deletion

```
# Find snapshot before deletion
snapshots = table.metadata.snapshots
for snapshot in reversed(snapshots):
```

```
if snapshot.summary.get('operation') == 'delete':  
    # Use snapshot before this one  
    previous_snapshot = snapshot.parent_snapshot_id  
    df = table.scan(snapshot_id=previous_snapshot).to_pandas()  
    break
```

## Reproduce Past Analysis

```
# Reproduce analysis from specific date  
analysis_date = datetime(2024, 1, 15)  
df = table.scan(as_of_timestamp=analysis_date).to_pandas()  
  
# Run same analysis  
result = df.groupby('category').agg({'amount': 'sum'})
```

## Best Practices

### 1. Configure Appropriate Retention

Balance between:

**Storage costs:** More snapshots = more metadata storage

**Audit requirements:** Compliance may require longer retention

**Recovery needs:** Keep enough history for rollback scenarios

### 2. Use Timestamp for Business Logic

Prefer timestamp-based queries for business logic:

```
# Good: Business-friendly  
df = table.scan(as_of_timestamp=datetime(2024, 1, 1)).to_pandas()  
  
# Less ideal: Requires knowing snapshot IDs  
df = table.scan(snapshot_id=123456789).to_pandas()
```

### 3. Document Snapshot IDs for Critical States

For important milestones, document snapshot IDs:

```
# Production release snapshot  
PROD_RELEASE_V1 = 123456789  
df = table.scan(snapshot_id=PROD_RELEASE_V1).to_pandas()
```

## 4. Regular Snapshot Cleanup

Automate snapshot expiration to manage storage:

```
# Weekly cleanup job
table.expire_snapshots(
    older_than=datetime.now() - timedelta(days=30),
    retain_last=10
)
```

## 5. Test Time Travel Queries

Verify time travel works before relying on it:

```
# Test query
snapshots = table.metadata.snapshots
if len(snapshots) > 1:
    df = table.scan(snapshot_id=snapshots[0].snapshot_id).to_pandas()
    assert len(df) >= 0 # Verify query works
```

## Troubleshooting

### "Snapshot not found"

**Cause:** Snapshot expired or never existed.

**Solution:**

List available snapshots: ``table.metadata.snapshots``

Check retention policy

Use a more recent snapshot or timestamp

### "Timestamp too old"

**Cause:** No snapshots exist for the requested timestamp.

**Solution:**

Check oldest available snapshot: ``min(s.timestamp_ms for s in table.metadata.snapshots)``

Use a more recent timestamp

Increase snapshot retention if needed

## Performance issues with old snapshots

**Cause:** Old snapshots may reference many small files.

**Solution:**

Consider table compaction

Use more recent snapshots when possible

Optimize query filters to reduce data scanned

## Related Documentation

[Branch Management](#) - Git-like branching for catalogs

[Merge Conflicts](#) - Merging branches

[Pylceberg Testing](#) - Pylceberg integration

Source: `warehouse_management.md`

## Warehouse Management

### Overview

Warehouses in Pangolin define storage configurations and credential management strategies for your Iceberg tables. Each warehouse represents a storage backend (S3, Azure Blob, GCS) and controls how clients access data.

**Key Concepts:**

**Warehouse:** Storage configuration and credential vending settings

**Catalog:** References a warehouse and defines a storage location

**Credential Vending:** Automatic provisioning of temporary credentials to clients

## Creating a Warehouse

### Basic Warehouse (Static Credentials)

```
curl -X POST http://localhost:8080/api/v1/warehouses \
  -H "X-Pangolin-Tenant: <tenant-id>" \
  -H "Authorization: Bearer <token>" \
  -H "Content-Type: application/json" \
  -d '{
    "name": "dev_warehouse",
    "use_sts": false,
    "storage_config": {
      "type": "s3",
      "bucket": "my-dev-bucket",
      "region": "us-east-1"
    }
  }'
```

```
}  
}'
```

### Configuration:

`use\_sts: false` - Clients use static credentials from their environment

Suitable for development and testing

## Production Warehouse (STS Credential Vending)

```
curl -X POST http://localhost:8080/api/v1/warehouses \  
-H "X-Pangolin-Tenant: <tenant-id>" \  
-H "Authorization: Bearer <token>" \  
-H "Content-Type: application/json" \  
-d '{  
  "name": "production_warehouse",  
  "vending_strategy": {  
    "type": "AwsSts",  
    "role_arn": "arn:aws:iam::123456789012:role/PangolinDataAccess"  
  },  
  "storage_config": {  
    "s3.bucket": "my-prod-bucket",  
    "s3.region": "us-east-1"  
  }  
}'
```

### Configuration:

`vending\_strategy`: Defines how Pangolin provisions temporary credentials (STS, SAS, OAuth).

Required for production environments where direct IAM access is prohibited.

## Storage Backend Configuration

### AWS S3

```
{  
  "type": "s3",  
  "bucket": "my-bucket",  
  "region": "us-east-1",  
  "role_arn": "arn:aws:iam::123456789012:role/DataAccess"  
}
```

### Azure Blob Storage

```
{
```

```
"type": "azure",  
"account_name": "mystorageaccount",  
"container": "data"  
}
```

## Google Cloud Storage

```
{  
  "type": "gcs",  
  "bucket": "my-gcs-bucket",  
  "project_id": "my-project"  
}
```

## MinIO (S3-Compatible)

```
"type": "s3",  
"bucket": "minio-bucket",  
"endpoint": "http://minio:9000",  
"allow_http": true,  
"s3.path-style-access": "true"  
}
```

## API Endpoints

### List Warehouses

`GET /api/v1/warehouses`

#### Headers:

`Authorization`: `Bearer <token>`

`X-Pangolin-Tenant`: `<Tenant-ID>`

### Create Warehouse

`POST /api/v1/warehouses`

#### Body (with STS):

```
{  
  "name": "main_warehouse",  
  "use_sts": true,  
  "storage_config": {  
    "type": "s3",  
    "bucket": "my-bucket",  
    "region": "us-east-1",  
  }  
}
```

```
"role_arn": "arn:aws:iam::123456789012:role/PangolinRole"
}
```

## Get Warehouse

`GET /api/v1/warehouses/{name}`

## Catalog Association

After creating a warehouse, create catalogs that reference it:

```
curl -X POST http://localhost:8080/api/v1/catalogs \
-H "X-Pangolin-Tenant: <tenant-id>" \
-H "Content-Type: application/json" \
-d '{
  "name": "analytics",
  "warehouse_name": "main_warehouse",
  "storage_location": "s3://my-bucket/analytics"
}'
```

## Best Practices

**Use STS in Production:** Set `use\_sts: true` for production warehouses

**Static Credentials for Development:** Use `use\_sts: false` for local development

**Separate Warehouses by Environment:** Create different warehouses for dev, staging, production

**Scope Storage Locations:** Use catalog's `storage\_location` to organize data

## Related Documentation

[Security & Credential Vending](#) - Detailed credential vending guide

[AWS S3 Storage](#) - S3 backend configuration

[Client Configuration](#) - Pylceberg, Spark, Trino setup

## Source: README.md

## Getting Started with Pangolin

Welcome to Pangolin! This directory contains everything you need to set up, configure, and evaluate the platform.

# Onboarding

Get up and running in minutes.

Quick Start Guide: A step-by-step walkthrough of your first tenant, catalog, and table.

Evaluating Pangolin: Using `NO\_AUTH` mode for rapid local testing.

## ⚙️ Configuration

Fine-tune Pangolin for your environment.

Environment Variables: Comprehensive list of all configuration options.

Configuration Overview: Principles of runtime and storage setup.

Nested Namespaces: Guide to creating and managing hierarchical namespaces.

Storage Backend Logic: How backends determine which credentials to use (vending vs client-side).

Dependencies: System requirements and library overview.

## Client Integration

Connect your favorite tools to the Pangolin REST Catalog.

Client Configuration: Setup guides for Pylceberg, PySpark, and Trino.

## Authentication Modes

Choose your security model.

Auth Mode: Standard operational mode with user authentication.

No Auth Mode: For local development and testing.

## Deployment

Move from local testing to production.

Deployment Guide: Instructions for local, Docker, and production environments.

Docker Deployment: Detailed Docker setup and configuration.

## Quick Launch Reference

If you have Docker installed, the fastest way to see Pangolin in action is:

```
# Clone and start with Docker Compose
git clone https://github.com/your-org/pangolin.git
cd pangolin
docker-compose up -d
```

Visit the [Quick Start Guide](#) to perform your first data operations!

# Source: auth-mode.md

## Authentication Modes & Guide

Pangolin supports two distinct operational modes: **Auth Mode** (Production) and **No-Auth Mode** (Development). This guide details how to configure, access, and manage identity in both scenarios.

### 1. Auth Mode vs. No-Auth Mode

Feature	Auth Mode (Default)	No-Auth Mode
Use Case	Production, Multi-Tenancy, Public Internet	Local Dev, CI/CD, Rapid Prototyping
Security	Enforced via JWT Tokens	Disabled (Open Access)
Identity	Users must log in; Identity is verified	Requests default to `admin`; Password checks bypassed
Setup	Requires `PANGOLIN_JWT_SECRET`	Requires `PANGOLIN_NO_AUTH=true`
Tenant Context	Strict isolation; Users belong to 1 Tenant	Default Tenant used automatically if unspecified

### 2. Configuration (Environment Variables)

#### Root User Credentials

The **Root User** is the system superadmin. These credentials are used to bootstrap the system (create the first tenant).

##### Auth Mode:

```
# Secure credentials for production
export PANGOLIN_ROOT_USER="super_admin"
export PANGOLIN_ROOT_PASSWORD="Start123!ComplexPassword"
export PANGOLIN_JWT_SECRET="<random-32-char-string>"
```

##### No-Auth Mode:

```
# Credentials are ignored, but variables can still be set
export PANGOLIN_NO_AUTH="true"
```

### 3. UI Login Nuances

The Management UI adapts its login form based on the detected mode.

#### Auth Mode UI

**Standard Login (Root):** By default, the "**Tenant-specific login**" checkbox is **unchecked**.

Entering just a Username/Password sends a `null` Tenant ID.

This triggers the **Root Authentication** flow (checking `PANGOLIN\_ROOT\_USER`).

**Tenant User Login:** Accessing a specific tenant requires:

**Checking** the "Tenant-specific login" box.

Entering the **Tenant ID** (UUID).

Entering Username/Password.

**Behavior:** Invalid credentials return a 401 error. Session expires when the JWT expires.

## No-Auth Mode UI

**Auto-Detection:** The UI detects `NO\_AUTH` state on load.

**Root Access (Bypass):** To view the Root Dashboard (System Admin view), use these specific credentials:

**Username:** `root`

**Password:** `root`

\*Note: This is a client-side convenience bypass that gives you full UI access without needing a server token.\*

**Tenant Admin Access:** The UI may auto-login as the default admin, or you can use the credentials below.

## 4. Root User Workflows

The Root User's primary job is **System bootstrapping**. They generally do **not** query data.

### Workflow A: Bootstrapping in No-Auth Mode

When Pangolin starts in No-Auth Mode (`PANGOLIN\_NO\_AUTH=true`), it automatically:

Creates a default Tenant (`00000000-0000-0000-0000-000000000000`).

Creates a default **Tenant Admin** with:

**Username:** `tenant\_admin`

**Password:** `password123`

Allows **Anonymous API Access:** Requests with *\*no\** Authorization header are automatically treated as this `tenant\_admin`.

## Workflow B: Logging In as Root (Auth Mode)

### Via API (cURL):

```
# Auth Mode: Must provide correct password & null tenant-id
curl -X POST http://localhost:8080/api/v1/users/login \
  -d '{"username":"super_admin", "password":"...", "tenant-id":null}'
```

## Workflow B: Creating a Tenant & Admin (Bootstrapping)

Only the Root user can create tenants.

### 1. Create Tenant:

```
# Requires Root Token
POST /api/v1/tenants
{
  "name": "acme-corp"
}
# Response: Returns { "id": "uuid-for-acme", ... }
```

### 2. Create Tenant Admin:

```
# Requires Root Token
# Note: You are creating a user *inside* the new tenant
POST /api/v1/users
{
  "username": "acme_admin",
  "password": "temporary_password",
  "tenant_id": "uuid-for-acme", # Critical: Links user to tenant
  "role": "TenantAdmin"
}
```

## 5. Token Generation Guide

Authentication tokens (JWTs) act as your API keys for all operations.

### Generating a Token (Generic)

**Endpoint:** `POST /api/v1/tokens` (Requires Admin privileges) or `POST /api/v1/users/login` (Self-service).

### Payload:

```
{
  "username": "data_engineer",
  "password": "secure_password",
  "tenant_id": "uuid-for-acme"
}
```

Response:

```
{
  "token": "eyJhbGciOiJIUzI1Ni...",
  "expires_in": 86400,
  "tenant_id": "uuid-for-acme"
}
```

Using the Token

Pass it in the HTTP Header:

```
Authorization: Bearer eyJhbGciOiJIUzI1Ni...
```

6. Permissions Matrix: Who can touch what?

Pangolin employs a strict hierarchy. Here is what each user type can access.

Resource	Root User	Tenant Admin	Tenant User
Manage Tenants	Create/Delete	No Access	No Access
Manage Users	(Global)	(Own Tenant Only)	No Access
Create Warehouses	(Out of Scope)	Full Control	No Access
Create Catalogs	(Out of Scope)	Full Control	No Access
Manage Roles	(Out of Scope)	Full Control	No Access
Query Data	No Access***	Full Access	By Permission Only
View Audit Logs	(System Wide)	(Own Tenant Only)	No Access

\*\*\*Note on Root Data Access: By design, the Root user cannot query data (tables/views) directly. They must authenticate as a Tenant Admin or specific User to modify actual data. This enforcement ensures "System Administrators" cannot casually browse sensitive data "Business Data".

Tenant User Permissions (RBAC)

A standard Tenant User starts with ZERO permissions. They cannot see Catalogs or query Tables until granted a Role:

Reader Role: Can `SELECT` from tables.

Writer Role: Can `INSERT`/`UPDATE`/`DELETE`.

Discovery Role: Can search for assets but not query contents.

See [Permissions Management](#) for how to assign these granular rights.

Source: client\_configuration.md

Client Configuration

This guide shows how to configure Apache Iceberg clients (Pylceberg, PySpark, Trino, etc.) to connect to Pangolin.

## Authentication Modes

Pangolin supports two authentication modes:

### 1. NO\_AUTH Mode (Testing/Evaluation)

Set `PANGOLIN\_NO\_AUTH=1` environment variable on the server

No authentication required from clients

Uses default tenant automatically

**Not recommended for production**

### 2. Bearer Token Authentication (Production)

JWT tokens with tenant information

Iceberg REST specification compliant

Required for multi-tenant deployments

## Pylceberg

### Installation

```
pip install "pyiceberg[s3fs,pyarrow]" pyjwt
```

### NO\_AUTH Mode (Testing)

When the Pangolin server is running with `PANGOLIN\_NO\_AUTH=1`:

```
from pyiceberg.catalog import load_catalog

catalog = load_catalog(
    "pangolin",
    **{
        "uri": "http://localhost:8080",
        "prefix": "analytics", # Catalog name
        "s3.endpoint": "http://localhost:9000",
        "s3.access-key-id": "minioadmin",
        "s3.secret-access-key": "minioadmin",
        "s3.region": "us-east-1",
    }
)
```

```
# List namespaces
namespaces = catalog.list_namespaces()
print(namespaces)
```

## Bearer Token Authentication (Production)

### Step 1: Generate Token

```
curl -X POST http://localhost:8080/api/v1/tokens \
  -H "Content-Type: application/json" \
  -d '{
    "tenant_id": "000000000-0000-0000-0000-000000000001",
    "username": "user@example.com",
    "expires_in_hours": 24
  }'
```

Response:

```
{
  "token": "eyJ0eXAiOiJKV1QiLCJhbGc...",
  "expires_at": "2025-12-14T02:37:49+00:00",
  "tenant_id": "000000000-0000-0000-0000-000000000001"
}
```

### Step 2: Use Token with Pylceberg

```
from pyiceberg.catalog import load_catalog

# Token from /api/v1/tokens endpoint
token = "eyJ0eXAiOiJKV1QiLCJhbGc..."

catalog = load_catalog(
    "pangolin",
    **{
        "uri": "http://localhost:8080",
        "prefix": "analytics", # Catalog name
        "token": token,        # Bearer token authentication
        # S3 configuration is Optional here!
        # If the warehouse has Vending enabled, Pangolin will provide credentials.
    }
)
```

[!TIP]

**Credential Vending:** If your Pangolin warehouse is configured with a `vending\_strategy` (like `AwsStatic` or `AwsSts`), you **do not** need to provide `s3.access-key-id` or `s3.secret-access-key` in your client configuration. Pangolin will automatically vend temporary credentials to the client.

### Step 3: Generate Token Programmatically

```
import jwt
import datetime
from pyiceberg.catalog import load_catalog

def generate_token(tenant_id: str, secret: str = "secret") -> str:
    """Generate JWT token with tenant_id"""
    payload = {
        "sub": "api-user",
        "tenant_id": tenant_id,
        "roles": ["User"],
        "exp": datetime.datetime.utcnow() + datetime.timedelta(hours=24)
    }
    return jwt.encode(payload, secret, algorithm="HS256")

# Generate token
token = generate_token("00000000-0000-0000-0000-000000000001")

# Use with PyIceberg
catalog = load_catalog(
    "pangolin",
    **{
        "uri": "http://localhost:8080",
        "prefix": "analytics",
        "token": token,
    }
)
```

## Configuration File (`~/pyiceberg.yaml`)

## NO\_AUTH Mode

```
catalog:
  pangolin:
    uri: http://localhost:8080
    prefix: analytics
    s3.endpoint: http://localhost:9000
    s3.access-key-id: minioadmin
    s3.secret-access-key: minioadmin
    s3.region: us-east-1
```

## With Bearer Token

```
catalog:
  pangolin:
    uri: http://localhost:8080
    prefix: analytics
    token: eyJ0eXAiOiJKV1QiLCJhbGc... # From /api/v1/tokens
```

```
s3.endpoint: http://localhost:9000
s3.access-key-id: minioadmin
s3.secret-access-key: minioadmin
s3.region: us-east-1
```

Then in Python:

```
from pyiceberg.catalog import load_catalog
catalog = load_catalog("pangolin")
```

## PySpark

### Installation

```
pip install pyspark
```

### Configuration

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("Pangolin Example") \
        .config("spark.jars.packages",
"org.apache.iceberg:iceberg-spark-runtime-3.5_2.12:1.5.0,org.apache.iceberg:iceberg-aws-
bundle:1.5.0") \
        .config("spark.sql.extensions",
"org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions") \
        .config("spark.sql.catalog.pangolin", "org.apache.iceberg.spark.SparkCatalog") \
        .config("spark.sql.catalog.pangolin.catalog-impl",
"org.apache.iceberg.rest.RESTCatalog") \
        .config("spark.sql.catalog.pangolin.uri", "http://localhost:8080/v1/analytics") \
        .config("spark.sql.catalog.pangolin.io-impl", "org.apache.iceberg.aws.s3.S3FileIO")
\
    .config("spark.sql.catalog.pangolin.s3.endpoint", "http://localhost:9000") \
    .config("spark.sql.catalog.pangolin.s3.access-key-id", "minioadmin") \
    .config("spark.sql.catalog.pangolin.s3.secret-access-key", "minioadmin") \
    .config("spark.sql.catalog.pangolin.s3.path-style-access", "true") \
    .config("spark.sql.catalog.pangolin.header.Authorization", f"Bearer {token}") \
    #
    Bearer token
    .getOrCreate()
```

## Trino

### Configuration (`catalog/pangolin.properties`)

```
connector.name=iceberg
iceberg.catalog.type=rest
iceberg.rest.uri=http://localhost:8080/v1/analytics
iceberg.rest.http-headers=Authorization:Bearer eyJ0eXAiOiJKV1QiLCJhbGc...
fs.s3a.endpoint=http://localhost:9000
fs.s3a.access-key=minioadmin
fs.s3a.secret-key=minioadmin
fs.s3a.path.style.access=true
```

# Apache Flink

## SQL Client Configuration

You can create an Iceberg catalog in Flink SQL using the `rest` catalog type.

```
CREATE CATALOG pangolin WITH (
  'type'='iceberg',
  'catalog-type'='rest',
  'uri'='http://localhost:8080/v1/analytics',
  'token'='<YOUR_JWT_TOKEN>',
  'warehouse'='s3://warehouse-bucket/analytics'
);

USE CATALOG pangolin;
```

### [!NOTE]

Ensure you have the `flink-table-api-java-bridge` and `iceberg-flink-runtime` jars available in your Flink environment.

## Dremio

Dremio supports connecting to Iceberg REST catalogs as a source.

## Configuration Steps

**Add Source:** Click "Add Data Source" and select "**Iceberg REST Catalog**".

### General Settings:

**Name:** `pangolin\_analytics`

**Endpoint URL:** `http://pangolin:8080/v1/analytics/iceberg` (Ensure reachable from Dremio)

### Authentication:

If using **No-Auth Mode**, no further auth is needed.

If using **Token Auth**, you may need to pass the token via the `header.Authorization`

property in the "Advanced Options" -> "Connection Properties" if Dremio's UI doesn't explicitly ask for a Bearer token yet.

\*Property\*: `header.Authorization`

\*Value\*: `Bearer <YOUR\_JWT\_TOKEN>`

### Credential Vending:

Enable "**Use vended credentials**" (supported by Pangolin) to let Dremio use the credentials provided by the catalog endpoint for S3/Azure/GCS access.

## Environment Variables

For production deployments, use environment variables:

### Pylceberg

```
import os
from pyiceberg.catalog import load_catalog

catalog = load_catalog(
    "pangolin",
    **{
        "uri": os.getenv("PANGOLIN_URI", "http://localhost:8080"),
        "prefix": os.getenv("PANGOLIN_CATALOG", "analytics"),
        "token": os.getenv("PANGOLIN_TOKEN"), # JWT token
        "s3.endpoint": os.getenv("S3_ENDPOINT"),
        "s3.access-key-id": os.getenv("AWS_ACCESS_KEY_ID"),
        "s3.secret-access-key": os.getenv("AWS_SECRET_ACCESS_KEY"),
        "s3.region": os.getenv("AWS_REGION", "us-east-1"),
    }
)
```

## Troubleshooting

### Connection Issues

#### 401 Unauthorized:

In NO\_AUTH mode: Ensure server has `PANGOLIN\_NO\_AUTH=1` set

In production: Check that Bearer token is valid and not expired

**404 Not Found:** Ensure the catalog name in the prefix/URI matches an existing catalog

**S3 Access Denied:** Verify S3 credentials and endpoint configuration

### Debugging

Enable debug logging:

### **Pylceberg:**

```
import logging
logging.basicConfig(level=logging.DEBUG)
```

### **Check Token:**

```
import jwt
decoded = jwt.decode(token, options={"verify_signature": False})
print(f"Tenant ID: {decoded.get('tenant_id')}")
print(f"Expires: {decoded.get('exp')}")
```

## Migration from Custom Headers

If you were using ``header.X-Pangolin-Tenant``:

### **Old (Not compatible with Pylceberg):**

```
"header.X-Pangolin-Tenant": "00000000-0000-0000-0000-000000000001"
```

### **New (Iceberg REST spec compliant):**

```
"token": generate_token("00000000-0000-0000-0000-000000000001")
```

The custom header approach still works for direct API calls but is not supported by Pylceberg due to its authentication architecture.

## Source: configuration.md

## Configuration Overview

Pangolin follow a "Configuration-over-Code" philosophy, allowing for flexible deployments across different infrastructures.

## Runtime Configuration

Most settings are managed via **Environment Variables** at startup. This includes everything from the port number to the metadata persistence backend.

For a complete list of variables, see [Environment Variables](#).

For deployment-specific patterns, see [Deployment Guide](#).

## Client Configuration Discovery

Pangolin implements the Iceberg REST specification for **Config Discovery**. Clients (like Spark or Pylceberg) can query the ``/v1/config`` endpoint to receive default properties and overrides.

This is particularly useful for:

**S3 Endpoint Overrides:** Telling clients to use a local MinIO instance.

**IO Implementation:** Specifying the storage driver (e.g., ``S3FileIO``).

**Warehouse Context:** Providing default warehouse identifiers.

## Example Discovery Call

```
GET http://localhost:8080/v1/config?warehouse=main
```

### Response:

```
{
  "defaults": {
    "s3.endpoint": "http://minio:9000",
    "s3.path-style-access": "true"
  },
  "overrides": {
    "io-impl": "org.apache.iceberg.aws.s3.S3FileIO"
  }
}
```

## Storage Management

While core system configuration uses environment variables, **Storage Connectors** (Warehouses) are configured dynamically via the Admin API/CLI. This allows you to add or modify storage locations without restarting the Pangolin service.

To learn how to configure storage, see [Warehouse Management](#).

Source: dependencies.md

## Dependencies Overview

Pangolin is built with the following key Rust crates:

### Core Frameworks

Axum: Ergonomic and modular web application framework.

Tokio: Asynchronous runtime for Rust.

Tower: Modular and reusable components for building robust networking clients and

servers.

## Data & Serialization

Serde: Framework for serializing and deserializing Rust data structures.

Serde JSON: JSON support for Serde.

Bytes: Utilities for working with bytes (used in federated proxy).

## Storage

Object Store: Unified interface for object storage (S3, GCS, Azure, Local).

DashMap: Concurrent associative array for high-performance in-memory storage.

SQLx: Async SQL toolkit for PostgreSQL, MongoDB, and SQLite.

Features: ``postgres``, ``mongodb``, ``sqlite``, ``uuid``, ``chrono``, ``json``

MongoDB: Official MongoDB driver for Rust.

## Security & Authentication

JSON Web Token: JWT creation and validation.

bcrypt: Password hashing for users and API key hashing for service users.

AWS SDK for S3: Official AWS SDK for S3 operations and STS credential vending.

AWS Config: AWS configuration loading.

## HTTP Client (Federated Catalogs)

reqwest: HTTP client for forwarding requests to external catalogs.

## Utilities

Uuid: UUID generation for entities.

Chrono: Date and time handling.

Tracing: Application-level tracing for async Rust.

Anyhow: Flexible error handling.

## Development Dependencies

fokio-test: Testing utilities for async code.

mockall: Mocking framework for tests.

# Feature-Specific Dependencies

## Service Users

``bcrypt`` - API key hashing

``uuid`` - Service user ID generation

``chrono`` - Expiration tracking

## Merge Conflict Resolution

``uuid`` - Merge operation and conflict ID generation

``chrono`` - Timestamp tracking

``serde`` - Conflict serialization

## Federated Catalogs

``request`` - HTTP client for proxying requests

``bytes`` - Request/response body handling

``axum::http`` - HTTP types for forwarding

## Cloud Provider SDKs (Optional)

For credential vending with AWS STS, Azure OAuth2, or GCP service accounts:

**AWS SDK:** ``aws-config``, ``aws-sdk-sts``, ``aws-credential-types``, ``aws-smithy-types``

**Azure Identity:** ``azure_identity``, ``azure_core``

**GCP Auth:** ``gcp_auth``

**Build with cloud features:**

```
# All cloud providers
cargo build --features cloud-credentials

# Individual providers
cargo build --features aws-sts
cargo build --features azure-oauth
cargo build --features gcp-oauth
```

**Default build** (without features) uses placeholder implementations for development.

## System Requirements

## Minimum

Rust 1.75+

2GB RAM (for development)

Linux, macOS, or Windows

## Recommended (Production)

Rust 1.82+

4GB+ RAM

Linux (Ubuntu 20.04+ or similar)

**Backend Storage** (choose one):

PostgreSQL 12+ (recommended for production)

MongoDB 5+ (recommended for cloud-native)

SQLite 3.35+ (recommended for development/embedded)

**Warehouse Storage:** S3, Azure Blob, or GCS

## Building from Source

```
# Clone repository
git clone https://github.com/your-org/pangolin.git
cd pangolin

# Build all crates
cargo build --release

# Run tests
cargo test

# Run API server
cargo run --bin pangolin_api --release
```

## Docker Dependencies

If using Docker deployment:

Docker 20.10+

Docker Compose 1.29+ (optional)

## Related Documentation

-

[Getting Started](#) - Quick start guide

[Configuration](#) - Configuration options

[Deployment](#) - Production deployment

# Source: deployment.md

## Deployment Guide

This guide covers how to deploy Pangolin in various environments, from local development to production-ready Docker containers.

### Local Development

To run Pangolin locally for development or testing, ensure you have the [Rust toolchain](#) installed.

```
# Clone the repository
git clone https://github.com/your-org/pangolin.git
cd pangolin

# Run the API server
cargo run --bin pangolin_api
```

The server listens on `0.0.0.0:8080` by default. You can change this using the `PORT` environment variable.

### Docker Deployment

Pangolin is designed to be cloud-native and easily containerized.

#### 1. Using Docker Compose (Recommended for Evaluation)

Pangolin provides a `docker-compose.yml` that sets up the API server along with common dependencies like MinIO (S3), PostgreSQL, and MongoDB.

```
# Start all services
docker-compose up -d

# Check logs
docker-compose logs -f pangolin-api
```

#### Services Exposed:

**Pangolin API:** `http://localhost:8080`

**MinIO Console:** `http://localhost:9001` (Credentials: `minioadmin`/`minioadmin`)

-

**PostgreSQL:** `localhost:5432`

## 2. Building the Image Manually

```
# From the root of the repository
docker build -t pangolin:latest .
```

## 3. Running the Container

```
docker run -p 8080:8080 \
  -e PANGOLIN_STORAGE_TYPE=memory \
  -e PANGOLIN_NO_AUTH=true \
  alexmerced/pangolin-api:latest
```

[!NOTE]

For production deployments, we recommend pinning to a specific version, e.g., `alexmerced/pangolin-api:0.1.0`.

## Production Best Practices

When moving to production, consider the following:

### 1. Persistent Metadata Storage

Do not use `memory` storage in production. Configure a robust backend like **PostgreSQL** or **MongoDB**.

```
export DATABASE_URL="postgresql://user:pass@db-host:5432/pangolin"
```

### 2. Enable Authentication

Ensure `PANGOLIN\_NO\_AUTH` is NOT set to `true`. Set a strong `PANGOLIN\_JWT\_SECRET`.

### 3. Use Cloud IAM Roles

For storage access, prefer `AwsSts` or `AzureSas` vending strategies over static keys. See [Credential Vending](#).

### 4. Monitoring & Logging

Set `RUST\_LOG=info` or `debug` and integrate the container logs with your logging provider.

## Related Documentation

[Environment Variables](#)

[Client Configuration](#)

[Multi-Tenancy](#)

Source: `docker_deployment.md`

## Docker Deployment

Docker deployment documentation has been consolidated into the main [Deployment Guide](#).

Please refer to that guide for instructions on using Docker, Docker Compose, and building images manually.

Source: `env_vars.md`

## Environment Variables

Pangolin is configured via environment variables. This guide lists all available options.

### Core API Configuration

Variable	Description	Default
<code>`PORT`</code>	The port the API server will listen on.	<code>`8080`</code>
<code>`RUST_LOG`</code>	Log level ( <code>`error`</code> , <code>`warn`</code> , <code>`info`</code> , <code>`debug`</code> , <code>`trace`</code> ).	<code>`info`</code>

### Metadata Persistence

Pangolin stores its own metadata (tenants, users, catalogs) in a backend database.

Variable	Description	Default
<code>`DATABASE_URL`</code>	Connection string for Postgres, MongoDB, or SQLite.	(None)
<code>`PANGOLIN_STORAGE_TYPE`</code>	Storage driver if <code>`DATABASE_URL`</code> is missing. ( <code>`memory`</code> , <code>`postgres`</code> , <code>`mongo`</code> , <code>`sqlite`</code> ).	<code>`memory`</code>
<code>`MONGO_DB_NAME`</code>	Database name (when using MongoDB).	<code>`pangolin`</code>

[!NOTE]

If ``DATABASE_URL`` is provided, the driver is automatically inferred from the URI scheme (e.g., ``postgresql://``, ``mongodb://``, ``sqlite://``).

### Authentication & Security

Variable	Description	Default
<code>`PANGOLIN_NO_AUTH`</code>	<b>Evaluation Mode.</b> Auto-provisions a default tenant and	

admin user. Set to `"true"` to enable. | `false`` |  
`PANGOLIN_JWT_SECRET``	Secret key for JWT signing. **MUST** be changed in production.	`default_secret``
`PANGOLIN_SEED_ADMIN``	Auto-provision a Tenant Admin even if `NO_AUTH`` is false. Set to `"true"`.	`false``
`PANGOLIN_ADMIN_USER``	Username for seed admin.	`tenant_admin``
`PANGOLIN_ADMIN_PASSWORD``	Password for seed admin.	`password123``
`PANGOLIN_ROOT_USER``	Initial Root user for multi-tenant bootstrapping.	`admin``
`PANGOLIN_ROOT_PASSWORD``	Password for Root user.	`password``

## OAuth 2.0 (External Providers)

Required for enabling "Login with Google/GitHub/Microsoft" in the UI.

Variable	Provider
<code>OAUTH_GOOGLE_CLIENT_ID`</code> / <code>_SECRET`</code>	Google
<code>OAUTH_MICROSOFT_CLIENT_ID`</code> / <code>_SECRET`</code>	Microsoft
<code>OAUTH_GITHUB_CLIENT_ID`</code> / <code>_SECRET`</code>	GitHub

## Cloud Provider Features

When building with cloud features (`features aws-sts``, etc.), these standard variables are used by the underlying SDKs for the **Signer** logic.

Variable	Description
<code>AWS_ACCESS_KEY_ID`</code>	AWS Credentials for STS / Signer.
<code>AWS_SECRET_ACCESS_KEY`</code>	AWS Credentials for STS / Signer.
<code>AWS_REGION`</code>	Default AWS region.
<code>AWS_ENDPOINT_URL`</code>	Override for MinIO or custom S3 backends.

## Security Checklist

**Disable NO\_AUTH:** Ensure `PANGOLIN_NO_AUTH`` is unset or `false`` in production.

**Rotate JWT Secret:** Use a 32+ character random string.

**Strong Root Password:** Change the default `admin/password`` immediately.

**Use DATABASE\_URL:** Avoid the `memory`` store for any non-trivial use case.

Source: [evaluating-pangolin.md](#)

## Evaluating Pangolin

The easiest way to evaluate Pangolin and test its features (like Pylceberg integration, branching, and multi-tenancy) is to use the **NO\_AUTH`** mode locally.

This mode disables strict authentication requirements and enables an **Auto-Provisioning** system that sets up a ready-to-use environment for you.

# 1. Quick Start

Run the Pangolin server with the `PANGOLIN\_NO\_AUTH` environment variable set to `true`.

```
# In the pangolin/ directory
PANGOLIN_NO_AUTH=true cargo run --bin pangolin_api
```

## What Happens?

When the server starts in this mode, it will:

**Create a Default Tenant:** A system tenant (ID: `00000000-0000-0000-0000-000000000000`) is created automatically.

**Auto-Provision a Admin:** A Tenant Administrator user (`tenant\_admin`) is created for you.

**Generate Credentials:** A valid, long-lived JWT token is generated.

**Print Configuration:** The server's startup logs will display a handy banner with everything you need.

## 2. Connecting with PyIceberg

Look at your terminal output for a block that looks like this:

```
=====
WARNING: NO_AUTH MODE ENABLED - FOR EVALUATION ONLY
=====
Initial Tenant Admin Auto-Provisioned:
Username: tenant_admin
Password: password123
Tenant ID: 00000000-0000-0000-0000-000000000000
-----
PyIceberg Configuration Snippet:
-----
catalog = load_catalog(
    "local",
    **{
        "type": "rest",
        "uri": "http://127.0.0.1:8080/api/v1/catalogs/my_catalog/iceberg",
        "token": "eyJ0eXAiOiJKV1QiLCJ... ", # <-- Your Auto-Generated Token
    }
)
=====
```

Simply **copy and paste** that Python snippet into your script or notebook. Pangolin uses the JWT token to carry your Tenant context, which is the standard way to interact with the Iceberg REST API.

## 3. Using the CLI

You can also use the `pangolin-admin` CLI easily in this mode.

### Setup

Configure your CLI to point to your local instance. Since `NO\_AUTH` mode auto-provisions a user, you can just login with the default credentials:

```
pangolin-admin login --username tenant_admin --password password123
```

### Try These Commands

Once logged in, verify your access and start exploring:

#### 1. Create a Warehouse

Define where your data lives (e.g., an S3 bucket or local memory for testing).

```
pangolin-admin create-warehouse my_warehouse --type memory
```

#### 2. Create a Catalog

Link a catalog name to your warehouse.

```
pangolin-admin create-catalog my_catalog --warehouse my_warehouse
```

\*Note: This matches the URI in the Python snippet above (`.../catalogs/my\_catalog/iceberg`).\*

#### 3. Manage Users

Create other users to test permissions.

```
pangolin-admin create-user data_scientist --role tenant-user --email ds@example.com
```

## Nuances of NO\_AUTH Mode

While `NO\_AUTH` mode makes evaluation simple, keep these important details in mind:

**It is NOT Secure:** Do not use this mode in production. Anyone with network access to the server can act as the root user.

**Root Access:** The server internally treats requests without specific auth headers as "Root" requests, but the auto-provisioned user is a **Tenant Admin**. This allows you to test the exact permissions strictness that a real user would face.

**Tenant Isolation:** Even in `NO\_AUTH` mode, Pangolin enforces tenant isolation. You must provide the correct `X-Pangolin-Tenant` header (or have it in your valid JWT token) to access resources. The auto-provisioned token handles this for you.

Source: getting\_started.md

# Getting Started with Pangolin

This guide will walk you through setting up Pangolin and performing an end-to-end workflow.

## Authentication Modes

Pangolin supports two primary operational modes: **No-Auth Mode** and **Auth Mode**.

### 1. No-Auth Mode (`PANGOLIN\_NO\_AUTH=true`)

Designed for rapid local evaluation and testing.

**Single Tenant:** All operations are automatically tied to a fixed "default" tenant.

**No Headers Required:** You don't need `Authorization` or `X-Pangolin-Tenant` headers.

**Anonymous Access:** Direct access to all catalog features.

**Limitations:** Only one tenant allowed; no RBAC enforcement.

### 2. Auth Mode (`PANGOLIN\_NO\_AUTH=false`)

The standard mode for multi-tenant production environments.

**Tenant Isolation:** Every request MUST include a valid `Authorization: Bearer <JWT>` token.

**RBAC Enforcement:** Permissions are checked at the Catalog, Namespace, and Asset levels.

**Tenant Context:** Users only see resources belonging to their specific tenant.

**Multiple Tenants:** Supports infinite logical isolation across different organizations.

## User Scopes

Pangolin uses a three-tier identity model to balance platform oversight with organizational autonomy.

Scope	Typical ID	Key Responsibilities	Focus
Root User	`admin`	Create Tenants, Configure Global Settings, System Metrics.	Platform
Tenant Admin		Organization Lead   Manage Users, Warehouses, Catalogs, Audit Logs.	Governance
Tenant User		Analyst / Engineer   Create Namespaces, Manage Tables, Branching, Tagging.	Data

## Role Details:

**Root:** Has "god mode" over the platform but cannot see into tenant table data directly unless granted a role.

**Tenant Admin:** Controls everything \*within\* their tenant. They set up the warehouses (S3/Azure) and the catalogs.

**Tenant User:** Restricted to the data assets. They cannot manage other users or infrastructure.

## Installation & Setup

### Clone the Repository

```
git clone https://github.com/your-repo/pangolin.git
cd pangolin
```

### Run the Server

#### For Testing (NO\_AUTH mode):

```
PANGOLIN_NO_AUTH=true cargo run --bin pangolin_api
```

#### For Production:

```
cargo run --bin pangolin_api
```

The server will start at `http://localhost:8080`.

## End-to-End Walkthrough

We will simulate a real-world scenario: A "Data Team" setting up their environment, creating tables, experimenting on a branch, and merging changes back.

### 1. Create a Tenant

Pangolin is multi-tenant by default. First, create a tenant for your organization.

```
curl -X POST http://localhost:8080/api/v1/tenants \
  -u admin:password \
  -H "Content-Type: application/json" \
  -d '{
    "name": "Acme Corp",
    "id": "00000000-0000-0000-0000-000000000001"
  }'
```

\*Note: We are using a fixed UUID for simplicity. The `-u admin:password` corresponds to the default Root User credentials in `example.env`.\*

## 2. Create a Warehouse

A warehouse stores storage credentials and configuration for **Iceberg table data**.

[!TIP]

**Storage Clarification:**

**Catalog Metadata Storage** (controlled by `PANGOLIN\_STORAGE\_TYPE` env var): Where Pangolin stores its catalog metadata (tenants, catalogs, branches, etc.)

**Table Data Storage** (controlled by warehouse config): Where Iceberg table data files are stored

These are separate! For example, you can store catalog metadata in Postgres while table data lives in S3.

```
curl -X POST http://localhost:8080/api/v1/warehouses \
-H "X-Pangolin-Tenant: 00000000-0000-0000-0000-000000000001" \
-H "Content-Type: application/json" \
-d '{
  "name": "main_warehouse",
  "vending_strategy": {"type": "AwsStatic"},
  "storage_config": {
    "type": "s3",
    "bucket": "demo-bucket",
    "region": "us-east-1",
    "access_key_id": "minioadmin",
    "secret_access_key": "minioadmin",
    "endpoint": "http://localhost:9000"
  }
}'
```

\*Note: We use `vending\_strategy: AwsStatic` for this quick start. For production with IAM Role assumption, see [Credential Vending](#).\*

## 3. Create a Catalog

A catalog references a warehouse and specifies a storage location. The catalog name is what clients use in their connection URIs.

```
curl -X POST http://localhost:8080/api/v1/catalogs \
-H "X-Pangolin-Tenant: 00000000-0000-0000-0000-000000000001" \
-H "Content-Type: application/json" \
-d '{
  "name": "analytics",
  "warehouse_name": "main_warehouse",
  "storage_location": "s3://demo-bucket/analytics"
}'
```

## 4. Create a Namespace

Namespaces organize your tables (like schemas in a database).

```
curl -X POST http://localhost:8080/v1/analytics/namespaces \
-H "X-Pangolin-Tenant: 00000000-0000-0000-0000-000000000001" \
-H "Content-Type: application/json" \
-d '{"namespace": ["data_team"]}'
```

\*Note: The URL uses the catalog name `analytics` as the prefix.\*

## 5. Create a Table on `main`

Create an Iceberg table in the `data\_team` namespace. By default, operations target the `main` branch.

```
curl -X POST http://localhost:8080/v1/analytics/namespaces/data_team/tables \
-H "X-Pangolin-Tenant: 00000000-0000-0000-0000-000000000001" \
-H "Content-Type: application/json" \
-d '{
  "name": "users",
  "location": "s3://demo-bucket/analytics/data_team/users",
  "schema": {
    "type": "struct",
    "fields": [
      {"id": 1, "name": "id", "type": "int", "required": true},
      {"id": 2, "name": "name", "type": "string", "required": false}
    ]
  }
}'
```

## 6. Create a Feature Branch (`dev`)

Create a new branch named `dev` to experiment safely. We will use **Partial Branching** to only track the `users` table on this branch.

```
curl -X POST http://localhost:8080/api/v1/branches \
-H "X-Pangolin-Tenant: 00000000-0000-0000-0000-000000000001" \
-H "Content-Type: application/json" \
-d '{
  "name": "dev",
  "catalog": "analytics",
  "from_branch": "main",
  "branch_type": "experimental",
  "assets": ["data_team.users"]
}'
```

## 7. Update Table on `dev`

Modify the table schema on the `dev` branch. Note the `?branch=dev` query parameter.

```
curl -X POST http://localhost:8080/v1/analytics/namespaces/data_team/tables/users?branch=dev \
-H "X-Pangolin-Tenant: 00000000-0000-0000-0000-000000000001" \
-H "Content-Type: application/json" \
-d '{
  "updates": [
    {"action": "add-column", "name": "email", "type": "string"}
  ]
}'
```

## 8. Verify Isolation

Check that the `main` branch is unaffected.

```
# Check main (should NOT have email column)
curl http://localhost:8080/v1/analytics/namespaces/data_team/tables/users?branch=main \
-H "X-Pangolin-Tenant: 00000000-0000-0000-0000-000000000001"

# Check dev (SHOULD have email column)
curl http://localhost:8080/v1/analytics/namespaces/data_team/tables/users?branch=dev \
-H "X-Pangolin-Tenant: 00000000-0000-0000-0000-000000000001"
```

## 9. Merge `dev` into `main`

Once satisfied, merge the changes back to `main`.

```
curl -X POST http://localhost:8080/api/v1/branches/merge \
-H "X-Pangolin-Tenant: 00000000-0000-0000-0000-000000000001" \
-H "Content-Type: application/json" \
-d '{
  "catalog": "analytics",
  "source_branch": "dev",
  "target_branch": "main"
}'
```

Now, `main` will contain the schema updates made in `dev`.

## Next Steps

Explore [Branch Management](#) for advanced strategies.

Learn about [Credential Vending](#).

Set up [Client Configuration](#) for Pylceberg, PySpark, Trino, or Dremio.

## Production Setup

For production deployments, disable `NO_AUTH` mode and use Bearer token

authentication:

## 1. Start Server (No NO\_AUTH variable)

```
# Set JWT secret for production
export PANGOLIN_JWT_SECRET="your-secret-key-here"
cargo run --release --bin pangolin_api
```

## 2. Generate Tokens

```
curl -X POST http://localhost:8080/api/v1/tokens \
-H "Content-Type: application/json" \
-d '{
    "tenant_id": "00000000-0000-0000-0000-000000000001",
    "username": "user@example.com",
    "expires_in_hours": 24
}'
```

Response:

```
{  
  "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpcyI6ImFkbWUiLCJyb2tkeSI6ImNpdG8iLCJhdWQiOiJkaXIiLCJmcm9udGVudCkiOiJkYXRlZDQwLjE0MDE1OTg0MTAifQ==",  
  "expires_at": "2025-12-14T02:37:49+00:00",  
  "tenant_id": "000000000-0000-0000-0000-000000000001"  
}
```

### 3. Use Token with Pylceberg

```
from pyiceberg.catalog import load_catalog

catalog = load_catalog(
    "pangolin",
    **{
        "uri": "http://localhost:8080",
        "prefix": "analytics",
        "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1cm9jaCI6ImFub3R5bWVudCIsImV4cCI6MTYxMjQyODk5fQ.eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1cm9jaCI6ImFub3R5bWVudCIsImV4cCI6MTYxMjQyODk5fQ", # From /api/v1/tokens
    }
)
```

## 4. Tenant Creation Restriction

In NO\_AUTH mode, attempting to create additional tenants will fail:

```
curl -X POST http://localhost:8080/api/v1/tenants \
-H "Content-Type: application/json" \
-d '{"name": "test-tenant"}'
```

Response (403 Forbidden):

```
{
  "error": "Cannot create additional tenants in NO_AUTH mode",
  "message": "NO_AUTH mode is only meant for evaluation and testing with a single
default tenant. Please enable authentication and use Bearer tokens if you want to create
multiple tenants.",
  "hint": "Remove PANGOLIN_NO_AUTH environment variable and use /api/v1/tokens endpoint
to generate tokens"
}
```

## Next Steps

[Pylceberg Testing Guide](#) - Comprehensive Pylceberg testing

[Client Configuration](#) - Configure various Iceberg clients

[Warehouse Management](#) - Manage warehouses and catalogs

[API Reference](#) - Complete API documentation

## Source: logging-in.md

### Logging In to Pangolin

This guide covers how to authenticate with Pangolin across all interfaces: **UI**, **CLI**, **API**, and **Python SDK**. It specifically explains when you need a **Tenant UUID** and how to find it.

### Core Concept: Root vs. Tenant Users

Pangolin has two types of users, which determines how you log in:

User Type	Scope	Requires Tenant UUID?	Login Context
<b>Root User</b>	Superadmin for the entire system	<b>NO</b>	Authenticates against the System Root. Can manage all tenants.
<b>Tenant User</b>	Scoped to a specific Organization (Tenant)	<b>YES</b>	Authenticates against a specific Tenant. Username is only unique *within* that tenant.

### The Tenant UUID

#### What is it?

A **Tenant UUID** (Universally Unique Identifier) is the unique ID for an Organization/Tenant (e.g., `a1b2c3d4-...`).

#### When do I need it?

**Root User (`admin`): NEVER** use a Tenant UUID. You must log in globally (send `null` or omit the ID).

\*Note:\* Even in **No-Auth Mode**, the `admin` account is Global. If you try to log in with the Default Tenant UUID (`0000...`), it may fail because the `admin` user does not exist *inside* that tenant.

**Tenant User (e.g., `alice`): ALWAYS** need it.

**Default Tenant Admin (`tenant\_admin`): ALWAYS** need it (Use `00000000-0000-0000-0000-000000000000`).

\*Note:\* In **No-Auth Mode**, this is often the user you interact with implicitly, so it's easy to confuse this with Root.

## ⚠ Common Confusion: No-Auth Mode

You might feel like you need a UUID in No-Auth mode. This is usually because:

You are interacting with the **Default Tenant** (`0000...`), which requires that UUID.

Or you are using the `root`/`root` client-side bypass (which is a UI implementation detail, not a real backend user).

### Rule of Thumb:

If you are `admin` (System Owner) -> **No UUID**.

If you are anyone else (including Default Admin) -> **Use UUID**.

## How to find your Tenant UUID

### 1. Via UI (If you are already logged in as Root/Admin)

Go to the **Tenants** page.

The UUID is listed under the **ID** column for each tenant.

### 2. Via CLI (If you have Root access)

```
pangolin tenants list
```

Output:

ID	Name	Description
-----	-----	-----
00000000-0000-0000-0000-000000000000	Default	Default Tenant
a1b2-c3d4-e5f6-g7h8	Acme Corp	Primary Tenant
...		

### 3. Via API

```
curl -H "Authorization: Bearer <root-token>" http://localhost:8080/api/v1/tenants
```

### 4. Convenience URL (If provided by Administrator)

If your administrator has provided a link like ``http://<pangolin-ui>/login/<your-tenant-uuid>``, navigating to it will automatically:

Pre-fill your **Tenant UUID**.

Expand the **Tenant-specific login** section.

## Logging In via UI

The Login screen adapts to your needs:

### As Root User (`admin`)

Enter Username: `admin` (or your configured root user).

Enter Password.

**Leave "Tenant-specific login" unchecked.**

Click **Sign In**.

### As Tenant User

Enter Username.

Enter Password.

Check **"Tenant-specific login"**.

Enter your **Tenant UUID**.

Click **Sign In**.

## Logging In via CLI

### As Root User

```
# No tenant-id flag needed
pangolin login --username admin --password password123
```

### As Tenant User

```
# Must provide tenant-id
pangolin login --username alice --password secret --tenant-id <UUID>
```

## Logging In via API

The API endpoint: `POST /api/v1/users/login`

### As Root User

Send `tenant-id` as `null` (or omit it entirely if supported by your client).

```
curl -X POST http://localhost:8080/api/v1/users/login \
-H "Content-Type: application/json" \
-d '{
  "username": "admin",
  "password": "password123",
  "tenant-id": null
}'
```

### As Tenant User

Send `tenant-id` with the UUID.

```
curl -X POST http://localhost:8080/api/v1/users/login \
-H "Content-Type: application/json" \
-d '{
  "username": "alice",
  "password": "secret",
  "tenant-id": "alb2c3d4-..."
}'
```

## Logging In via Python SDK (`pypangolin`)

### As Root User

```
from pangolin import PangolinClient

client = PangolinClient(base_url="http://localhost:8080")
client.authenticate("admin", "password123") # No tenant_id arg
```

### As Tenant User

```
from pangolin import PangolinClient

client = PangolinClient(base_url="http://localhost:8080")
```

```
client.authenticate("alice", "secret", tenant_id="alb2c3d4-...")
```

Source: `nested_namespaces.md`

## Working with Nested Namespaces

Pangolin supports hierarchical (nested) namespaces, allowing you to organize tables into logical structures like ``finance.payroll.2023`` or ``raw.iot.sensors``.

### Overview

In the Iceberg spec, a namespace is a list of strings (e.g., ``["finance", "payroll"]``).

**Storage:** Maps to nested directories in your warehouse (e.g., ``s3://bucket/wh/finance/payroll/``).

**API:** Represented as JSON arrays in bodies, and usually dot-separated or URL-encoded strings in path parameters.

### Creating Nested Namespaces

Currently, the most reliable way to create nested namespaces is via the **REST API**, as the UI currently supports single-level creation.

### API Endpoint

``POST /v1/{catalog}/namespaces``

### Request Body

To create ``marketing.campaigns.q1``:

```
curl -X POST http://localhost:8080/v1/analytics/namespaces \
-H "Authorization: Bearer <token>" \
-H "X-Pangolin-Tenant: <tenant-id>" \
-H "Content-Type: application/json" \
-d '{
  "namespace": ["marketing", "campaigns", "q1"],
  "properties": {
    "owner": "marketing-team",
    "retention": "365d"
  }
}'
```

**Note:** You do NOT need to create the parent ``marketing`` or ``marketing.campaigns`` explicitly before creating the child. However, creating them explicitly is good practice for setting properties at each level.

# Specifying Namespaces in API Calls

When referencing a nested namespace in URL paths (e.g., to list tables), the handling depends on the client and the specific endpoint.

## 1. Dot Notation (Common)

For most HTTP clients and simple setups, you can often use dot-notation if your namespace parts don't contain dots themselves.

```
`GET /v1/analytics/namespaces/marketing.campaigns.q1/tables`
```

## 2. URL Encoding (Robust)

The default separator for Iceberg REST is the Unit Separator character ``\u001F`` (ASCII 31). However, many implementations (including typical usage patterns in Pangolin) accept encoded dots or specific delimiters.

If accessing programmatically:

**Pylceberg:** Handles this automatically.

**REST Clients:** If ``marketing.campaigns.q1`` fails, ensure you are encoding the separator used by your namespace creation.

## Client Examples

### Pylceberg (Python)

Pylceberg creates generic ``["part1", "part2"]`` namespaces natively.

```
from pyiceberg.catalog import load_catalog

catalog = load_catalog("pangolin", **config)

# Create nested namespace
catalog.create_namespace(["marketing", "campaigns", "q1"])

# List tables
tables = catalog.list_tables("marketing.campaigns.q1")
```

### Spark (SQL)

```
-- Creating nested namespaces (if supported by the catalog property set)
CREATE NAMESPACE pangolin.marketing.campaigns.q1;

-- Creating a table in a nested namespace
CREATE TABLE pangolin.marketing.campaigns.q1.leads (
  id bigint,
  email string
```

```
) USING iceberg;
```

## Storage Layout

Pangolin maps namespaces to physical folders in your object storage.

For a warehouse bucket `my-datalake`:

**Namespace:** `[ "marketing", "campaigns" ]`

**Table:** `leads`

**Path:** `s3://my-datalake/analytics/marketing/campaigns/leads/`

This hierarchical structure is preserved, making it easy to browse data directly in S3/Azure/GCP consoles if needed.

## Source: no-auth-mode.md

### No Auth Mode

Pangolin supports a specialized "No Auth" mode designed for local development, testing, and evaluation where setting up a full authentication provider (like OAuth/OIDC) is not feasible or desired.

### Triggering No Auth Mode

To enable No Auth mode, set the following environment variable before starting the `pangolin\_api` server:

```
export PANGOLIN_NO_AUTH=true
```

## Behavior & Features

### 1. Automatic Tenant Admin Access

When accessing the API without any credentials, the system automatically treats the request as coming from the default **Tenant Admin** of the default tenant.

**Username:** `tenant\_admin`

**Role:** `TenantAdmin`

**Tenant ID:** `00000000-0000-0000-0000-000000000000`

This allows you to perform administrative tasks (create users, manage catalogs, etc.) immediately without logging in.

## 2. "Easy Auth" (Selective Authentication)

While No Auth mode defaults to Admin for unauthenticated requests, it **respects the `Authorization` header** if provided. This is crucial for verifying permissions:

**If you provide a token:** The system processes the request as that specific user.

**If you provide NO token:** The system processes the request as the default Tenant Admin.

## 3. Password Bypass

To facilitate logging in as specific users (e.g., `TenantUser` for testing access restrictions) without setting up true hashes:

The `.login()` endpoint bypasses password verification in this mode.

You can log in as *any* existing user with *any* password.

## Usage Guide

### Using the CLI

Since the CLI handles authentication tokens automatically, you can mix modes:

**Admin Mode (Default):** Just run commands.

```
pangolin-admin catalog list # Works immediately
```

**User Mode:** Login as a specific user to switch context.

```
pangolin-user login --username data_analyst --password any
# Now subsequent commands run as data_analyst
```

### Using the UI

The UI will automatically detect No Auth mode and may auto-login as `tenant\_admin`.

To test different roles:

**Logout** via the profile menu.

**Login** with the username you created (e.g., `data\_analyst`) and any password.

You are now exploring the UI restricted by that user's permissions.

### Using the API (cURL / Python)

**Admin Request:**

```
curl http://localhost:8080/api/v1/catalogs
```

**User Request** (after getting token via ``/login``):

```
curl -H "Authorization: Bearer <token>" http://localhost:8080/api/v1/catalogs
```

## Testing with Pylceberg

The server prints a convenient configuration snippet on startup when in No Auth mode. This snippet works immediately because the token vended is for the default ``tenant_admin``.

## Configuration

Use this snippet in your Python scripts to connect Pylceberg to your local Pangolin instance:

```
from pyiceberg.catalog import load_catalog

catalog = load_catalog(
    "local",
    **{
        "type": "rest",
        "uri": "http://127.0.0.1:8080/api/v1/catalogs/sales/iceberg",
        # Default Admin Token (valid for 24h/until restart)
        "token": "YOUR_TOKEN_FROM_SERVER_LOGS",
        # Critical for No Auth mode routing
        "header.X-Pangolin-Tenant": "00000000-0000-0000-0000-000000000000"
    }
)

# Verify connection
print(catalog.list_namespaces())
```

## Key Notes for Pylceberg

**Tenant Header:** You **MUST** include ``"header.X-Pangolin-Tenant"`` in the config properties. Without it, the server won't know which tenant context to use for the Iceberg REST endpoints, even with a valid token.

**Token:** The token printed in the logs is a valid JWT for the ``tenant_admin`` role. You can use it as-is.

**URI:** Point to the specific catalog's Iceberg endpoint (e.g., ``.../catalogs/sales/iceberg``). Ensure the catalog exists first (e.g., via the seeding script).

Source: storage-backend-logic.md

# Storage Backend Logic

This document explains how Pangolin's storage backends (MemoryStore, SQLiteStore, PostgresStore, MongoStore) determine which storage credentials to use when reading and writing Iceberg metadata and data files.

## Overview

Pangolin supports two credential modes:

**Credential Vending:** Catalog server provides temporary credentials to clients

**Client-Side Credentials:** Clients provide their own storage credentials

The storage backend automatically handles both scenarios using a **warehouse lookup** → **fallback** pattern.

## How It Works

### Step 1: Warehouse Lookup

When a file operation occurs (read or write), the backend:

Extracts the storage location from the file path (e.g., `s3://test-warehouse/catalog/...`)

Calls `get\_warehouse\_for\_location(location)` to find a matching warehouse

Searches all warehouses for one where the bucket/container name appears in the location path

#### Example:

```
// Location: "s3://test-warehouse/catalog/namespace/table/metadata.json"
// Warehouse config: { "s3.bucket": "test-warehouse", "s3.endpoint": "...", ... }
// Match: location.contains("test-warehouse") → true
```

### Step 2: Use Warehouse Config (Credential Vending)

If a warehouse is found:

Uses the warehouse's `storage\_config` to create an object store client

The `storage\_config` contains:

`s3.endpoint`, `s3.bucket`, `s3.region`

`s3.access-key-id`, `s3.secret-access-key` (if using static credentials)

OR `vending\_strategy` for temporary credential vending

**Object store cache** reuses this client for subsequent operations

### Code Path:

```
if let Some(warehouse) = self.get_warehouse_for_location(path).await? {
    let cache_key = self.get_object_store_cache_key(&warehouse.storage_config, path);
    let store = self.object_store_cache.get_or_insert(cache_key, || {
        Arc::new(create_object_store(&warehouse.storage_config, path).unwrap())
    });
    // Use store for read/write
}
```

## Step 3: Fallback to Environment Variables (Client-Side)

If NO warehouse is found (or warehouse lookup fails):

Falls back to environment-based S3 client

Reads credentials from environment variables:

``S3\_ENDPOINT`` or AWS default

``AWS\_ACCESS\_KEY\_ID``

``AWS\_SECRET\_ACCESS\_KEY``

``AWS\_REGION``

**Does NOT use AWS credential chain** (no EC2 metadata service, no IAM roles)

### Code Path:

```
// Fallback for S3 paths when no warehouse found
let mut builder = AmazonS3Builder::new() // NOT from_env()!
    .with_bucket_name(bucket)
    .with_allow_http(true);

if let Ok(endpoint) = std::env::var("S3_ENDPOINT") {
    builder = builder.with_endpoint(endpoint);
}

if let Ok(key_id) = std::env::var("AWS_ACCESS_KEY_ID") {
    builder = builder.with_access_key_id(key_id);
}

// ... etc
```

## Credential Vending vs Client-Side

Aspect	Credential Vending	Client-Side
<b>Warehouse Lookup</b>	Succeeds (warehouse config found)	Fails (no warehouse or catalog without warehouse)
<b>Credentials Source</b>	Warehouse ``storage_config``	Environment variables or Pylceberg config
<b>Security</b>	Temporary credentials, server-controlled	⚠ Long-lived credentials,

client-controlled |  
| **Use Case** | Multi-tenant, managed access | Development, testing, simple setups |  
| **Pylceberg Config** | Minimal (just token + tenant header) | Full S3 credentials in catalog config |

## Troubleshooting

### Issue: "Failed to write to S3" with AWS credential chain errors

#### Symptom:

```
error sending request for url (http://169.254.169.254/latest/api/token)
```

**Cause:** Using ``AmazonS3Builder::from_env()`` which tries AWS credential discovery.

**Fix:** Ensure code uses ``AmazonS3Builder::new()`` and explicitly sets credentials.

### Issue: Warehouse not found, falling back to env credentials

**Symptom:** Files written but warehouse config not used.

#### Causes:

**Bucket name mismatch:** Catalog's ``storage_location`` uses different bucket than warehouse's ``s3.bucket``

**Catalog without warehouse:** Catalog created without ``warehouse_name`` field

#### Fix:

```
# Ensure catalog has matching storage_location
catalog_response = requests.post(
    f"{API_URL}/api/v1/catalogs",
    json={
        "name": "my_catalog",
        "warehouse_name": "my_warehouse", # ← Links to warehouse
        "catalog_type": "Local",
        "storage_location": "s3://my-bucket/catalog", # ← Must contain warehouse's
bucket
        "properties": {}
    }
)
```

### Issue: Pylceberg writes failing

**Symptom:** Table creation fails with "Invalid JSON" or connection errors.

#### Common Causes:

**Port mismatch:** MinIO on non-standard ports (use 9000:9000, not 9010:9000)

**Missing path-style access:** Add `"s3.path-style-access": "true"` for MinIO

**Wrong catalog endpoint:** Use `"uri": "http://localhost:8080"` with `"prefix": "catalog\_name"`

### Working Pylceberg Config:

```
catalog = load_catalog(  
    "my_catalog",  
    **{  
        "type": "rest",  
        "type": "rest",  
        "uri": "http://localhost:8080/api/v1/catalogs/my_catalog",  
        "token": "your_jwt_token",  
        "header.X-Pangolin-Tenant": "tenant_id",  
        # Client-side credentials (if not using vending)  
        "s3.endpoint": "http://localhost:9000",  
        "s3.access-key-id": "minioadmin",  
        "s3.secret-access-key": "minioadmin",  
        "s3.region": "us-east-1",  
        "s3.path-style-access": "true",  
    }  
)
```

## Implementation Details

### MemoryStore

**Warehouse Lookup:** `get\_warehouse\_for\_location()` - iterates `DashMap`, uses `location.contains(bucket)`

**Object Store Cache:** `DashMap<String, Arc<dyn ObjectStore>>`

**Metadata Cache:** `moka::future::Cache<String, Vec<u8>>`

### SQLiteStore

**Warehouse Lookup:** `get\_warehouse\_for\_location()` - queries `SQLite`, uses `location.contains(bucket)`

**Object Store Cache:** Same as `MemoryStore`

**Metadata Cache:** Same as `MemoryStore`

**Key Fix:** Changed from exact bucket matching to `contains()` for compatibility

### PostgresStore & MongoStore

Follow same pattern as `SQLiteStore`

Warehouse lookup queries respective database

Same caching infrastructure

## Best Practices

**Always set `storage\_location`** when creating catalogs to ensure warehouse lookup succeeds

**Use credential vending** for production multi-tenant deployments

**Use client-side credentials** for development and testing

**Match bucket names** between warehouse config and catalog storage\_location

**Use standard ports** for MinIO (9000) to avoid Pylceberg issues

**Enable path-style access** for MinIO and S3-compatible services

Source: README.md

## Known Issues

This section documents verified issues, limitations, and architectural quirks present in the current release.

### v0.4.0

#### SQL Backend Token Listing (SQLite/Postgres)

**Description:** Active token lists may be empty for Root users or ephemeral accounts when using SQL backends due to a strict `JOIN` dependency.

**Status:** Identified. Fix proposed for v0.5.x.

Source: token\_listing\_sqlite\_join.md

## Known Issue: SQL Backend Token Listing for Root/Ephemeral Users

**Affected Versions:** v0.4.0 and prior

**Affected Backends:** SQLite, PostgreSQL

**Unaffected Backends:** Memory, MongoDB

## Summary

When listing active tokens via `/api/v1/users/me/tokens` or `/api/v1/users/{id}/tokens`, SQL-based backends (SQLite, PostgreSQL) fail to return results for users who do not exist in the `users` database table (e.g., Ephemeral Root Users) or if there is a data

mismatch in the `JOIN` operation. This results in empty token lists even when valid tokens exist.

## Technical Detail

The `list\_active\_tokens` implementation in SQL backends relies on a `JOIN` between the `active\_tokens` table and the `users` table to filter tokens by `tenant\_id`:

```
SELECT t.token_id, t.user_id, t.token, t.expires_at
FROM active_tokens t
JOIN users u ON t.user_id = u.id
WHERE u.tenant_id = ? ...
```

## The Problem

**Ephemeral Root Users:** The default "System Root" (`admin`/`password` or env vars) is often ephemeral and does not have a persisted row in the `users` table.

**Result:** The `JOIN` fails (Inner Join finds no matching `user\_id`), so the query returns 0 rows.

**Schema Sensitivity:** Any mismatch between the token's stored `user\_id` and the `users` table record causes specific tokens to disappear from lists.

## Comparison with Other Backends

**Memory Store & MongoDB:** These backends store the `tenant\_id` directly on the token record/document (`TokenInfo`). The list operation performs a direct filter on the tokens collection without needing to look up the user. This makes them robust for all user types.

## Workaround

**Use Memory Store** for robust local development involving Root user token management.

**Persist the User:** If using SQLite/Postgres, ensure the user (even Root) is explicitly created in the `users` table with the correct Tenant ID.

\*Note\*: Root users are typically tenant-less, which makes the `WHERE u.tenant\_id = ?` clause problematic even if the user exists, unless the query logic handles `NULL` tenants (which it currently doesn't for tenant-scoped inputs).

## Future Fix

The intended fix for v0.5.x is to denormalize the `tenant\_id` column into the `active\_tokens` table for SQL backends, mirroring the NoSQL/Memory architecture.

**Proposed Schema Change:**

```
ALTER TABLE active_tokens ADD COLUMN tenant_id TEXT;
```

```
CREATE INDEX idx_tokens_tenant ON active_tokens(tenant_id);
-- Migration logic to populate from users or defaults
```

### Proposed Query Change:

```
SELECT * FROM active_tokens WHERE tenant_id = ? AND expires_at > ?
-- No JOIN required
```

Source: permissions.md

## Permissions System

Pangolin implements a hierarchical, additive Role-Based Access Control (RBAC) system. It ensures that data is highly secure by default while providing flexibility through cascading grants and Tag-Based Access Control (TBAC).

### Permission Hierarchy & Cascading

Permissions in Pangolin are inherited downwards. A grant at a higher level automatically applies to all resources contained within that scope.

Scope	Includes	Use Case
---	---	---
<b>Catalog</b>	All Namespaces & Assets	Full access for a specific Data Team.
<b>Namespace</b>	All child Assets (Tables/Views)	Access to a specific "Schema" or project area.
<b>Asset</b>	Specific Table/View only	Granular access to a single sensitive dataset.

### Cascading Logic

If you grant `Read` on a **Namespace**, the user can read **every table** existing now OR created in the future within that namespace.

Grants are **Additive**: If a user is denied `Write` at the Catalog level but granted `Write` on a specific Table, they **can** write to that table.

## Usage Guides

### 1. Via CLI (`pangolin-admin`)

The admin tool is the primary way to manage grants.

#### Grant Permission:

```
# Grant Read on a specific namespace to a user
pangolin-admin grant-permission <user-id> \
  --action read \
  --scope namespace \
```

```
--catalog production \  
--namespace sales
```

### Revoke Permission:

```
pangolin-admin revoke-permission <permission-id>
```

### List Permissions:

```
pangolin-admin list-permissions --user <user-id>
```

## 2. Via Management UI

Log in as a **Tenant Admin**.

Navigate to **Identity -> Users** or **Identity -> Roles**.

Select a User/Role and click the **Permissions** tab.

Click **Add Permission** and select the Scope (Catalog, Namespace, or Asset) and the allowed Actions.

## Tag-Based Access Control (TBAC)

Manage access at scale using classification tags rather than individual table grants.

**Tag an Asset:** (e.g., Tag `orders` as `FINANCIAL`).

**Grant Tag Access:** Grant a role `Read` on the `Tag` scope for `FINANCIAL`.

**Result:** Any user with that role can access all assets tagged `FINANCIAL` across all catalogs.

## Advanced Activity Tracking

Pangolin tracks "Who granted What to Whom" in the **Audit Logs**. Every permission change is a first-class event, allowing security teams to reconstruct the history of access for any dataset.

## Best Practices

**Favor Roles over Users:** Create a role like `Data\_Analyst` and grant it permissions, then assign users to that role.

**Start Specific:** Start with `Asset` or `Namespace` level grants before jumping to `Catalog` wide access.

**Audit Regularly:** Use the `Audit Logs` to verify that permissions are being used as intended.

Source: README.md

## Pylceberg Integration Guides

Pangolin acts as a standard Apache Iceberg REST Catalog, making it compatible with the Python-native **Pylceberg** library.

### Getting Started

Choose the guide that matches your environment:

#### Authenticated (Production)

For multi-tenant environments where security is enforced via JWT tokens.

Credential Vending (Recommended): Pangolin manages storage keys.

Client-Provided Credentials: You provide storage keys to the client.

#### Service Users (Machine Identity)

For automation and CI/CD where persistent access is needed.

API Key Authentication: Using Service User API keys.

#### No-Auth (Evaluation)

For rapid local testing and development.

Credential Vending: Test the vending flow without RBAC.

Client-Provided Credentials: Basic local storage connectivity.

### Multi-Cloud Support

Detailed configuration for different storage backends.

Azure & GCP Integration: Connecting to ADLS Gen2 and Google Cloud Storage.

### Key Concepts

#### X-Iceberg-Access-Delegation

To enable **Credential Vending**, Pylceberg must send the ``X-Iceberg-Access-Delegation`` header with the value ``vended-credentials``. This tells Pangolin that the client expects temporary storage keys for its operations.

#### Tenant Context

While Pangolin extracts tenant context from the authentication token, we **strongly recommend** explicitly setting the ``header.X-Pangolin-Tenant`` property in your Pylceberg configuration. This ensures reliable routing for all operations, especially during initial connection.

Source: `auth_api_key.md`

## Pylceberg: Authenticated with Service User API Keys

Service Users provide an alternative to JWT tokens for machine-to-machine communication. Instead of short-lived tokens, you can use a persistent API key.

### Configuration

Unlike standard OIDC/OAuth2 flows, API keys are passed via the ``X-API-Key`` header. In Pylceberg, you configure this using the ``header.`` prefix.

### With Credential Vending (Recommended)

```
from pyiceberg.catalog import load_catalog

catalog = load_catalog(
    "pangolin",
    **{
        "type": "rest",
        "uri": "http://localhost:8080/v1/my_catalog",

        # API Key Authentication
        "header.X-API-Key": "pgl_key_your_api_key_here",

        # Tenant Routing (Recommended for Service Users)
        "header.X-Pangolin-Tenant": "550e8400-e29b-41d4-a716-446655440000",

        # Enable Credential Vending
        "header.X-Iceberg-Access-Delegation": "vended-credentials",
    }
)
```

### With Client-Provided Credentials

```
from pyiceberg.catalog import load_catalog

catalog = load_catalog(
    "pangolin",
    **{
        "type": "rest",
        "uri": "http://localhost:8080/v1/my_catalog",
        "header.X-API-Key": "pgl_key_your_api_key_here",
    }
)
```

```

    "header.X-Pangolin-Tenant": "550e8400-e29b-41d4-a716-446655440000",

    # Local MinIO or specific S3 credentials
    "s3.access-key-id": "minioadmin",
    "s3.secret-access-key": "minioadmin",
    "s3.region": "us-east-1",
    "s3.endpoint": "http://localhost:9000",
    "s3.path-style-access": "true",
  }
)

```

## Security Considerations

**Persistent Access:** API keys do not expire unless configured with an expiration date or revoked.

**Tenant Scoping:** API keys are locked to a specific tenant.

**Secret Management:** Handle API keys as secrets. Do not hardcode them in version control.

## Key Differences from JWT

Feature	JWT (OAuth2)	API Key (Service User)
Duration	Short-lived (Minutes/Hours)	Long-lived (Months/Years)
Identity	Individual User	Machine/Account
Header	`Authorization: Bearer <token>`	`X-API-Key: <key>`
Tenant	Extracted from Token	Requires `X-Pangolin-Tenant` (Recommended)

Source: `auth_client_creds.md`

**PyIceberg: Authenticated with Client Credentials**

In this scenario, you provide a valid Bearer Token for detailed RBAC and Tenant tracking, but you still manage storage credentials manually on the client side.

## Configuration

```

from pyiceberg.catalog import load_catalog

catalog = load_catalog(
    "pangolin",
    **{
        "type": "rest",
        "uri": "http://localhost:8080/v1/my_catalog",
        "token": "YOUR_JWT_TOKEN", # Your Bearer Token
    }
)

```

```

    # Client-Provided Storage Credentials
    "s3.access-key-id": "YOUR_ACCESS_KEY",
    "s3.secret-access-key": "YOUR_SECRET_KEY",
    "s3.region": "us-east-1",
  }
)

```

## When to use

Multi-tenant environments where clients already have their own direct S3/Azure/GCP access established.

Situations where your storage provider does not yet support the vending strategies implemented in Pangolin.

## Comparison to Vending

Aspect	Client Credentials	Credential Vending
<b>Security</b>	Long-lived keys on client.	Short-lived keys on client.
<b>Simplicity</b>	Client must manage secrets.	Client only needs a JWT token.
<b>Auditability</b>	Access logged at Storage level.	Access logged at both levels.

Source: `auth_vended_creds.md`

## PyIceberg: Authenticated with Credential Vending

This is the **Gold Standard** for secure Lakehouse implementations. Clients authenticate via Token, and the Catalog vends short-lived, scoped credentials for specific table operations.

## Configuration

```

from pyiceberg.catalog import load_catalog

catalog = load_catalog(
    "pangolin",
    **{
        "type": "rest",
        "uri": "http://localhost:8080/v1/my_catalog",
        "token": "YOUR_JWT_TOKEN",

        # Enable Credential Vending
        "header.X-Iceberg-Access-Delegation": "vended-credentials",
    }
)

```

# How it works

**Authentication:** Pylceberg sends the `token` in the Authorization header.

**Access Delegation:** The `X-Iceberg-Access-Delegation` header tells Pangolin to return temporary storage credentials (STS, SAS, or Downscoped tokens) in the `load-table` response.

**Data Access:** Pylceberg uses these temporary credentials to read/write data directly to S3, ADLS, or GCS without needing long-lived keys on the client machine.

## When to use

Production Multi-Tenant SaaS.

Zero-Trust environments.

Scenarios requiring strict audit logging of data access.

Source: [multi\\_cloud.md](#)

## Pylceberg Multi-Cloud Integration

Pangolin's credential vending and REST support extend beyond AWS to Azure Blob Storage (ADLS Gen2) and Google Cloud Storage (GCS).

### Azure ADLS Gen2

## Dependencies

Install Pylceberg with the `adlfs` and `pyarrow` extras:

```
pip install "pyiceberg[adlfs,pyarrow]"
```

## Configuration (Authenticated)

When using Azure, the catalog URI remains the same, but the storage properties change.

```
from pyiceberg.catalog import load_catalog

catalog = load_catalog(
    "azure_catalog",
    **{
        "type": "rest",
        "uri": "http://localhost:8080/v1/azure_catalog",
        "token": "YOUR_JWT_TOKEN",

        # Enable Vending
```

```

    "header.X-Iceberg-Access-Delegation": "vended-credentials",

    # Optional: Direct Key Access
    # "adls.account-name": "mystorageaccount",
    # "adls.account-key": "YOUR_ACCOUNT_KEY"
}
)

```

## Supported Properties (Pylceberg Compatible)

Pylceberg Property	Description	Vended by Pangolin
`adls.token`	OAuth2 access token for Azure AD authentication.	Yes (OAuth2 mode)
`adls.account-name`	Azure storage account name.	Yes (all modes)
`adls.account-key`	Azure storage account key.	Yes (account key mode)
`adls.container`	Container name within the storage account.	Yes (all modes)

**Note:** When using Pangolin's credential vending, you don't need to provide these properties manually. Pangolin vends them automatically based on your warehouse configuration.

## Google Cloud Storage (GCS)

### Dependencies

Install Pylceberg with the `gcsfs` and `pyarrow` extras:

```
pip install "pyiceberg[gcsfs,pyarrow]"
```

### Configuration (Authenticated)

```

from pyiceberg.catalog import load_catalog

catalog = load_catalog(
    "gcp_catalog",
    **{
        "type": "rest",
        "uri": "http://localhost:8080/v1/gcp_catalog",
        "token": "YOUR_JWT_TOKEN",

        # Enable Vending
        "header.X-Iceberg-Access-Delegation": "vended-credentials",

        # Optional: Direct Service Account Access
        # "gcs.project-id": "my-project",
        # "gcs.service-account-key": "/path/to/key.json"
    }
)

```

# Supported Properties (Pylceberg Compatible)

Pylceberg Property	Description	Vended by Pangolin
----	----	----
<code>`gcp-oauth-token`</code>	OAuth2 access token for GCP authentication.	Yes (OAuth2 mode)
<code>`gcp-project-id`</code>	GCP project ID.	Yes (all modes)
<code>`gcs.service-account-key`</code>	Path to service account JSON key (client-provided).	No (client manages)

**Note:** When using Pangolin's credential vending, you only need to provide ``token`` for authentication. Pangolin vends ``gcp-oauth-token`` and ``gcp-project-id`` automatically.

## On-Premise & S3-Compatible (MinIO)

For testing or private cloud deployments using MinIO or other S3-compatible storage.

## Configuration

```
from pyiceberg.catalog import load_catalog

catalog = load_catalog(
    "minio",
    **{
        "type": "rest",
        "uri": "http://localhost:8080/v1/minio",
        "token": "YOUR_TOKEN",

        # Mandatory for MinIO
        "s3.endpoint": "http://localhost:9000",
        "s3.path-style-access": "true",
        "s3.region": "us-east-1",
    }
)
```

Property	Description
----	----
<code>`s3.endpoint`</code>	The full HTTP(S) URL of your MinIO server.
<code>`s3.path-style-access`</code>	Must be <code>`true`</code> for MinIO to use <code>`endpoint/bucket/key`</code> instead of <code>`bucket.endpoint/key`</code> .

## Hybrid Cloud Scenarios

Pangolin allows you to manage catalogs across different clouds from a single control plane. One tenant can have an S3 warehouse for analytics and a GCS warehouse for ML workloads.

## Key Considerations

**Signer Implementation:** Ensure your Pangolin server is built with the appropriate

cloud SDK features (``features azure-oauth`` or ``features gcp-oauth``).

**Library Versions:** Earlier versions of Pylceberg had limited support for non-S3 backends. We recommend using **Pylceberg 0.7.0+** for the best multi-cloud experience.

**Region Consistency:** Ensure the ``region`` or ``location`` properties in your Pangolin Warehouse configuration match the physical bucket locations to minimize latency and costs.

Source: `no_auth_client_creds.md`

## Pylceberg: No Auth with Client Credentials

In this scenario, the Pangolin server is running in ``NO_AUTH`` mode, and the client provides storage credentials directly.

### Configuration

```
from pyiceberg.catalog import load_catalog

catalog = load_catalog(
    "pangolin",
    **{
        "type": "rest",
        "uri": "http://localhost:8080/v1/my_catalog",

        # Client-Provided Storage Credentials
        "s3.access-key-id": "minioadmin",
        "s3.secret-access-key": "minioadmin",
        "s3.region": "us-east-1",
        "s3.endpoint": "http://localhost:9000", # Required for local MinIO
        "s3.path-style-access": "true",        # Required for local MinIO
    }
)
```

### When to use

Initial development and local testing against a MinIO container.

Simple integration tests where complex auth/vending flows are out of scope.

Source: `no_auth_vended_creds.md`

## Pylceberg: No Auth with Credential Vending

In this scenario, Pangolin's ``NO_AUTH`` mode is enabled (for evaluation), but it still securely vends scoped credentials to Pylceberg.

# Configuration

```
from pyiceberg.catalog import load_catalog

catalog = load_catalog(
    "pangolin",
    **{
        "type": "rest",
        "uri": "http://localhost:8080/v1/my_catalog",

        # Enable Credential Vending
        "header.X-Iceberg-Access-Delegation": "vended-credentials",

        # No storage keys required!
    }
)
```

## When to use

Internal scenarios where you want to evaluate the vending workflow without setting up full JWT authentication.

Demonstration and local development where you want to mimic production credentials behavior.

[!IMPORTANT]

Even in `NO\_AUTH` mode, Pangolin requires a tenant context. If no token is provided, it defaults to the system tenant (`00000000-0000-0000-0000-000000000000`).

## Source: README.md

## Pangolin Reference Documentation

This section contains detailed reference guides for all major Pangolin concepts and operations, covering API, CLI, Python SDK, and UI usage.

## Core Concepts

**Tenants**: Managing isolation units and context switching.

**Users**: Identity management for humans.

**Access Control**: RBAC (Roles) and ABAC/TBAC (Tags).

**Service Users**: Programmatic access keys for automation.

**Tokens**: Session management and authentication.

**Auditing**: Governance logs and monitoring.

# Storage & Catalogs

**Warehouses**: Configuring object storage (S3, Azure, GCS).

**Catalogs**: Managing local and federated Iceberg catalogs.

## Data Management

**Assets**: Managing tables, views, and generic files.

**Business Metadata**: Tagging and custom properties.

**Version Control**: Branching, merging, and conflict resolution.

Source: `access_control.md`

## Access Control Reference (RBAC & TBAC/ABAC)

Pangolin uses a multi-layered security model:

**Tenant Isolation**: Strict separation of resources between tenants.

**Role-Based Access Control (RBAC)**: Permissions assigned to users via roles.

**Tag-Based Access Control (TBAC/ABAC)**: Permissions granted on assets based on their tags (business metadata).

### 1. Concepts

**Tenant Isolation**: Users are confined to their tenant. Cross-tenant access is impossible except for the Root user.

#### Roles:

``root``: Global superuser.

``tenant-admin``: Admin for a specific tenant.

``tenant-user``: Regular user, requires explicit permissions.

#### Permissions:

**Scopes**: ``Catalog``, ``Namespace``, ``Asset``, ``Tag``.

**Actions**: ``Read``, ``Write``, ``Delete``, ``Admin``.

### 2. API

**Base Endpoint**: ``/api/v1/permissions``

#### Grant Permission (Standard RBAC)

**Method:** `POST`

**Body:**

```
{
  "user_id": "uuid-of-user",
  "scope": "Catalog",
  "resource": "analytics",
  "action": "Read"
}
```

## Grant Permission (Tag-Based / ABAC)

Grant access to *any* asset that checks the "PII" tag.

**Body:**

```
{
  "user_id": "uuid-of-compliance-officer",
  "scope": "Tag",
  "resource": "PII",
  "action": "Read"
}
```

*\*Note: For Tag scope, the `resource` field is the Tag Name.\**

## Revoke Permission

**Method:** `DELETE`

**Path:** `/api/v1/permissions/{permission\_id}`

## List Permissions

**Method:** `GET`

**Path:** `/api/v1/permissions`

**Params:** `user={uuid}`, `role={role\_name}`

## 3. CLI

### Grant Permission

```
# Grant Read on 'analytics' catalog (RBAC)
pangolin-admin grant-permission \
  --username alice \
  --action Read \
  --resource analytics
```

```
# Grant Read on all assets tagged 'Public' (TBAC)
pangolin-admin grant-permission \
  --username alice \
  --action Read \
  --scope Tag \
  --resource Public
```

## Revoke Permission

```
pangolin-admin revoke-permission --id <permission-uuid>
```

## List Permissions

```
# List all permissions
pangolin-admin list-permissions

# Filter by user
pangolin-admin list-permissions --user alice
```

# 4. Python SDK (`pypangolin`)

## Grant

```
# RBAC
client.permissions.grant(
    user_id="uuid...",
    scope="Catalog",
    resource="analytics",
    action="Read"
)

# TBAC (Tag-Based)
client.permissions.grant(
    user_id="uuid...",
    scope="Tag",
    resource="Confidential",
    action="Read"
)
```

## List

```
perms = client.permissions.list(user_id="uuid...")
for p in perms:
    print(f"{p.scope}: {p.resource} -> {p.action}")
```

# Revoke

```
client.permissions.revoke(permission_id="uuid...")
```

## 5. UI

**Log in** as a **Tenant Admin**.

Navigate to **Permissions**.

**Grant**: Click "**Grant Permission**".

Select **User** or **Role**.

Select **Scope**: Choose **Tag** for ABAC/TBAC strategies.

Enter **Resource**: The name of the tag (e.g., `PII`, `Financial`).

Select **Action**.

Click **Grant**.

Source: assets.md

## Assets Reference

Assets represent data entities (Tables, Views) or generic files managed in catalogs.

### 1. API

**Base Endpoint**: `/api/v1/assets` or via Catalog structure.

#### Get Asset by ID

**Method**: `GET`

**Path**: `/api/v1/assets/{asset\_id}`

#### Search Assets

**Method**: `GET`

**Path**: `/api/v1/search/assets`

**Params**: `q={query}`

#### Register Generic Asset

**Method**: `POST`

**Path:** ``/api/v1/catalogs/{catalog}/namespaces/{ns}/assets``

**Body:**

```
{
  "name": "raw-csv",
  "kind": "file",
  "location": "s3://bucket/path/file.csv",
  "properties": {}
}
```

## 2. CLI

### Search

```
pangolin-admin search "sales"
```

### Get Details

```
pangolin-admin get-asset-details --id <uuid>
```

### View Catalog Tree

```
pangolin-admin explorer tree --catalog analytics
```

## 3. Python SDK (`pypangolin`)

### Search

```
results = client.search.search(query="sales", limit=5)
for r in results:
    print(r.name, r.catalog, r.namespace)
```

### Get Asset

```
# By ID
asset = client.search.get_asset("uuid")

# By Path
asset =
client.catalogs.get("analytics").namespaces("sales").tables("transactions").get()
```

# Register Generic Asset

```
client.catalogs.namespaces("analytics").register_asset(  
    namespace="raw",  
    name="data-file",  
    kind="csv",  
    location="s3://bucket/data.csv"  
)
```

## 4. UI

### Log in.

Navigate to **Explorer** (or **Catalogs > [Select Catalog]**).

**Browse:** Use the sidebar tree to navigate namespaces.

**Search:** Use the global search bar at the top.

**Details:** Click an asset to view Schema, History, and Metadata.

## Source: auditing.md

## Auditing Reference

Pangolin provides comprehensive audit logs for governance.

### 1. API

**Base Endpoint:** `/api/v1/audit``

### List Audit Events

**Method:** `GET``

**Path:** `/api/v1/audit``

**Parameters:**

``user_id``: Filter by actor.

``action``: Filter by action (e.g., ``create_table``).

``resource_type``: Filter by type (e.g., ``Catalog``).

``result``: ``Success`` or ``Failure``.

``limit``, ``offset``: Pagination.

```
curl "http://localhost:8080/api/v1/audit?action=create_table&limit=10" \  
-H "Authorization: Bearer <token>"
```

## Get Event Details

**Method:** `GET`

**Path:** `/api/v1/audit/{event\_id}`

## 2. CLI

### List Events

```
# List recent
pangolin-admin list-audit-events --limit 20

# Filter by user
pangolin-admin list-audit-events --user-id <uuid>

# Filter by failure
pangolin-admin list-audit-events --result Failure
```

### Get Event

```
pangolin-admin get-audit-event --id <uuid>
```

## 3. Python SDK (`pypangolin`)

### List Events

```
events = client.audit.list(
    limit=50,
    action="delete_table",
    result="Success"
)
for e in events:
    print(e.timestamp, e.username, e.action, e.resource_name)
```

### Get Event

```
event = client.audit.get("event-uuid")
print(event.metadata)
```

## 4. UI

**Log in as Tenant Admin.**

Navigate to **Audit Logs** (usually in the **Governance** or **Admin** section).

**Browse:** View the chronological list of events.

**Filter:** Use the filter bar to search by Actor, Action, or Status.

**Details:** Click on an event row to see full JSON metadata (IP address, User Agent, request details).

## Source: catalogs.md

# Catalogs Reference

Catalogs are Iceberg REST catalogs. Pangolin supports two types:

**Local:** Managed purely by Pangolin, stored in a Warehouse.

**Federated:** Proxies to another Iceberg catalog (e.g., Dremio, Snowflake, Unity, Glue).

## 1. API

### Local Catalogs

**Endpoint:** `/api/v1/catalogs`

**Method:** `POST`

**Body:**

```
{
  "name": "analytics",
  "warehouse_name": "s3-prod",
  "catalog_type": "Local"
}
```

### Federated Catalogs

**Endpoint:** `/api/v1/federated-catalogs`

**Method:** `POST`

**Body:**

```
{
  "name": "snowflake-mirror",
  "config": {
    "uri": "https://...",
    "warehouse": "optional-warehouse-ref",
    "credential": "optional-credential",
    "properties": {}
  }
}
```

```
}
```

## 2. CLI

### Create Local Catalog

```
pangolin-admin create-catalog analytics --warehouse s3-prod
```

### Create Federated Catalog

```
pangolin-admin create-federated-catalog snowflake-mirror \  
  --storage-location "s3://bucket/prefix" \  
  -P uri="https://snowflake..." \  
  -P token="sess:..."
```

### Sync Federated Catalog

Triggers a metadata discovery from the remote catalog.

```
pangolin-admin sync-federated-catalog snowflake-mirror
```

## 3. Python SDK (`pypangolin`)

### Create Local Catalog

```
cat = client.catalogs.create(  
    name="analytics",  
    warehouse="s3-prod",  
    type="Local"  
)
```

### Create Federated Catalog

```
fed_cat = client.federated_catalogs.create(  
    name="snowflake-mirror",  
    uri="https://...",  
    properties={"warehouse": "my-wh"}  
)
```

### Sync

```
client.federated_catalogs.sync("snowflake-mirror")
```

## 4. UI

**Log in** as a **Tenant Admin**.

Navigate to **Catalogs** in the sidebar.

**Create:** Click "**Create Catalog**".

**Name:** Identifier.

**Type:** Choose **Local** (Standard) or **Federated**.

**Details:**

Local: Select a **Warehouse**.

Federated: Enter URI and credentials.

Click **Save**.

**Browse:** Click on a catalog name to open the **Data Explorer**.

Source: metadata.md

## Business Metadata Reference

Metadata allows attaching custom key-value pairs to assets for governance and discovery.

### 1. API

**Base Endpoint:** `/api/v1/metadata`

#### Set Metadata

**Method:** `PUT`

**Path:** `/api/v1/metadata`

**Body:**

```
{
  "entity_type": "Asset",
  "entity_id": "uuid...",
  "key": "owner",
  "value": "data-team"
}
```

## Get Metadata

**Method:** `GET`

**Path:** `/api/v1/metadata/{entity\_type}/{entity\_id}`

## Delete Metadata

**Method:** `DELETE`

**Path:** `/api/v1/metadata/{entity\_type}/{entity\_id}/{key}`

## 2. CLI

### Set Metadata

```
pangolin-admin set-metadata \  
  --entity-type Asset \  
  --entity-id <uuid> \  
  --key owner \  
  --value data-team
```

### Get Metadata

```
pangolin-admin get-metadata \  
  --entity-type Asset \  
  --entity-id <uuid>
```

## 3. Python SDK (`pypangolin`)

### Set

```
client.metadata.set(  
    entity_type="Asset",  
    entity_id="uuid...",  
    properties={"owner": "data-team", "sla": "24h"}  
)
```

### Get

```
meta = client.metadata.get("Asset", "uuid...")  
print(meta)
```

## 4. UI

Navigate to an **Asset** in the **Explorer**.

Select the **Metadata** tab.

**Add:** Click "Add Property", enter key/value, Save.

**Edit/Delete:** Use the icons next to existing properties.

### Source: service\_users.md

## Service Users Reference

Service Users are headless accounts for machine-to-machine interaction (CI/CD, ETL, etc.). They authenticate via **API Keys**.

### 1. API

**Base Endpoint:** `/api/v1/service-users`

#### Create Service User

**Method:** `POST`

**Body:**

```
{
  "name": "ci-bot",
  "role": "tenant-user",
  "expires_in_days": 90
}
```

**Response:** Returns the **API Key** (only once).

#### Rotate Key

**Method:** `POST`

**Path:** `/api/v1/service-users/{id}/rotate`

### 2. CLI

#### Create

```
pangolin-admin create-service-user \
  --name "ci-bot" \
  --role "tenant-user" \
```

```
--expires-in-days 365
```

### Output:

```
API Key: pgl_AbCd...  
(Save this key!)
```

## List

```
pangolin-admin list-service-users
```

## Rotate Key

```
pangolin-admin rotate-service-user-key --id <uuid>
```

## 3. Python SDK (`pypangolin`)

### Create

```
su = client.service_users.create(  
    name="airflow-bot",  
    role="tenant-user",  
    expires_in_days=30  
)  
print(f"ID: {su.id}")  
print(f"Key: {su.api_key}") # Only available here
```

### Rotate

```
new_creds = client.service_users.rotate_key("uuid...")  
print(f"New Key: {new_creds.api_key}")
```

## 4. UI

### Log in as a Tenant Admin.

Navigate to **Service Users** (often a tab within **Users** or a separate sidebar item).

**Create:** Click "**Create Service User**".

Enter Name and Role.

Set Expiration.

**COPY THE KEY:** A modal will show the key. Copy it immediately.

**Rotate:** Select a service user -> **Rotate Key**. A new key is generated and shown.

Source: tenants.md

## Tenants Reference

Tenants are the top-level isolation unit in Pangolin. Each tenant has its own set of users, catalogs, and resources.

### 1. API

**Base Endpoint:** `/api/v1/tenants`

#### List Tenants

**Method:** `GET`

**Path:** `/api/v1/tenants`

**Auth:** Root Access Token

**Parameters:**

`limit` (query, optional): Max results

`offset` (query, optional): Pagination offset

```
curl -X GET http://localhost:8080/api/v1/tenants \
-H "Authorization: Bearer <root-token>"
```

#### Create Tenant

**Method:** `POST`

**Path:** `/api/v1/tenants`

**Auth:** Root Access Token

**Body:**

```
{
  "name": "acme-corp"
}
```

```
curl -X POST http://localhost:8080/api/v1/tenants \
-H "Authorization: Bearer <root-token>" \
-H "Content-Type: application/json" \
-d '{"name": "acme-corp"}'
```

# Initial Administrator

When a tenant is created, you usually create a `tenant-admin` user immediately after, or use the CLI which can do both.

## 2. CLI

The `pangolin-admin` CLI provides convenient commands for tenant management.

### List Tenants

```
pangolin-admin list-tenants
```

### Create Tenant

You can create a tenant and an optional admin user in one go.

```
# Create tenant only
pangolin-admin create-tenant --name "acme-corp"

# Create tenant AND admin user
pangolin-admin create-tenant \
  --name "acme-corp" \
  --admin-username "admin" \
  --admin-password "securePass123"
```

### Switch Context

To perform operations on a specific tenant using the CLI:

```
pangolin-admin use "acme-corp"
```

This sets the context for subsequent commands (like creating users or warehouses).

## 3. Python SDK (`pypangolin`)

### List Tenants

```
from pypangolin import PangolinClient

client = PangolinClient(uri="http://localhost:8080", username="root",
password="root_password")

tenants = client.tenants.list()
for t in tenants:
    print(t.name, t.id)
```

# Create Tenant

```
new_tenant = client.tenants.create(name="acme-corp")
print(f"Created tenant: {new_tenant.id}")
```

# Switch Context

```
# Switch client context to specific tenant
client.tenants.switch("acme-corp")

# Subsequent calls now apply to acme-corp
client.users.list()
```

## 4. UI

**Log in** as the **Root User**.

Navigate to **Tenants** in the sidebar.

**List:** You will see a table of all existing tenants.

**Create:** Click the "**Create Tenant**" button.

Enter the Tenant Name.

(Optional) Enter Admin Username/Password to auto-provision an admin.

Click **Save**.

**Switch Context:** Click the "**Switch**" (or log-in) icon next to a tenant to switch your active session to that tenant.

## Source: tokens.md

## Tokens Reference

Pangolin uses **JSON Web Tokens (JWT)** for user authentication.

### 1. API

**Base Endpoint:** `/api/v1/tokens`

### Create Token (Login)

**Method:** `POST`

**Path:** `/api/v1/users/login`

**Body:**

```
{
  "username": "alice",
  "password": "...",
  "tenant_id": "optional-uuid"
}
```

## Revoke Current Token (Logout)

**Method:** `POST`**Path:** `/api/v1/tokens/revoke`**Auth:** Bearer Token**Body:** `{}`

## Revoke Specific Token

**Method:** `POST`**Path:** `/api/v1/tokens/revoke/{token\_id}`**Auth:** Admin

## List User Tokens

**Method:** `GET`**Path:** `/api/v1/users/{user\_id}/tokens`

## 2. CLI

### List User Tokens

```
pangolin-admin list-user-tokens --user-id <uuid>
```

### Revoke Token

```
# Revoke your current session
```

```
pangolin-admin revoke-token
```

```
# Revoke a specific token by ID
```

```
pangolin-admin revoke-token-by-id --id <token-uuid>
```

## 3. Python SDK (`pypangolin`)

### List Tokens

```
tokens = client.tokens.list_user_tokens("user-uuid")
for t in tokens:
    print(t.token_id, t.is_valid)
```

### Revoke Token

```
client.tokens.revoke_by_id("token-uuid")
```

## 4. UI

**Log in as Tenant Admin.**

Navigate to **Users**.

Select a specific user (click name or "Edit").

Navigate to the **Sessions/Tokens** tab.

**List:** You will see active sessions.

**Revoke:** Click **Revoke** next to a session to invalidate it immediately.

Source: users.md

## Users Reference

Users are identities that can log in and interact with Pangolin resources. Users (except Root) are scoped to a specific Tenant.

### 1. API

**Base Endpoint:** `/api/v1/users`

### List Users

**Method:** `GET`

**Path:** `/api/v1/users`

**Auth:** Tenant Admin Token (or Root with Tenant Context)

**Headers:** `X-Pangolin-Tenant: <tenant-uuid>` (if using Root token)

```
curl -X GET http://localhost:8080/api/v1/users \
-H "Authorization: Bearer <token>"
```

## Create User

**Method:** `POST`

**Path:** `/api/v1/users`

**Body:**

```
{
  "username": "alice",
  "email": "alice@example.com",
  "password": "password123",
  "role": "tenant-user",
  "tenant_id": "<optional-uuid>"
}
```

**Note:** `role` must be `tenant-user`, `tenant-admin`, or `root`.

```
curl -X POST http://localhost:8080/api/v1/users \
-H "Authorization: Bearer <token>" \
-H "Content-Type: application/json" \
-d '{"username": "alice", "email": "alice@ex.com", "password": "pw", "role": "tenant-user"}'
```

## 2. CLI

Ensure you have selected a tenant first using `pangolin-admin use <tenant>`.

### List Users

```
pangolin-admin list-users
```

## Create User

```
# Create a standard user
pangolin-admin create-user alice \
  --email alice@example.com \
  --role tenant-user \
  --password "securePass123"

# Create a tenant admin
pangolin-admin create-user bob-admin \
  --email bob@example.com \
  --role tenant-admin
```

## Delete User

```
pangolin-admin delete-user alice
```

## 3. Python SDK (`pypangolin`)

### List Users

```
users = client.users.list()
for u in users:
    print(u.username, u.role)
```

### Create User

```
# Create standard user
user = client.users.create(
    username="alice",
    email="alice@example.com",
    role="tenant-user", # Case-sensitive: tenant-user or tenant-admin
    password="password123"
)
print(f"Created user {user.id}")
```

## 4. UI

**Log in** as a **Tenant Admin** (or Root switched to a tenant).

Navigate to **Users** in the sidebar.

**List:** You will see a table of users in the current tenant.

**Create:** Click "**Create User**".

Fill in Username, Email, Password.

Select Role: **Tenant User** or **Tenant Admin**.

Click **Create**.

**Manage:** Click the "Edit" icon to update details or "Trash" icon to delete.

Source: `version_control.md`

## Version Control Reference

Pangolin operates like Git for data. You can Branch, Tag, and Merge data across

namespaces.

# 1. API

## Create Branch

**Method:** `POST`

**Path:** `/api/v1/branches`

**Body:**

```
{
  "catalog": "analytics",
  "name": "feature-branch",
  "source_ref": "main"
}
```

## Create Merge Operation

**Method:** `POST`

**Path:** `/api/v1/merges`

**Body:**

```
{
  "catalog": "analytics",
  "source": "feature-branch",
  "target": "main"
}
```

## List Conflicts

**Method:** `GET`

**Path:** `/api/v1/merges/{merge\_id}/conflicts`

## Resolve Conflict

**Method:** `POST`

**Path:** `/api/v1/merges/conflicts/{conflict\_id}/resolve`

**Body:** `{ "strategy": "KeepSource" }` (or `KeepTarget`)

## Complete Merge

-

**Method:** `POST`

**Path:** `/api/v1/merges/{merge\_id}/complete`

## 2. CLI

### Branching

```
pangolin-admin create-branch \  
  --catalog analytics \  
  --name dev \  
  --from main  
  
pangolin-admin list-branches --catalog analytics
```

### Merging

```
# Start Merge  
pangolin-admin create-merge \  
  --catalog analytics \  
  --source dev \  
  --target main  
  
# Check Conflicts  
pangolin-admin list-merge-conflicts --id <merge-id>  
  
# Resolve  
pangolin-admin resolve-merge-conflict \  
  --conflict-id <uuid> \  
  --strategy KeepSource  
  
# Complete  
pangolin-admin complete-merge --id <merge-id>
```

## 3. Python SDK (`pypangolin`)

### Branching

```
client.branches.create(  
    catalog="analytics",  
    name="experiment",  
    source="main"  
)
```

### Merging

```
# 1. Start
merge_op = client.merge_operations.create(
    catalog="analytics",
    source="experiment",
    target="main"
)

# 2. Check Conflicts
if merge_op.conflicts:
    conflicts = client.merge_operations.list_conflicts(merge_op.id)
    for c in conflicts:
        client.merge_operations.resolve_conflict(c.id, "KeepSource")

# 3. Complete
client.merge_operations.complete(merge_op.id)
```

## 4. UI

### Branches:

Navigate to **Catalogs**.

Select a catalog.

Go to **Branches** tab.

Click "**New Branch**".

### Merging:

Go to **Merge Requests** (or similar tab in Catalog).

Click "**New Merge**".

Select Source/Target.

**Review:** UI shows diffs/conflicts.

**Resolve:** Use UI buttons ("Keep Source" / "Keep Target") for each conflict.

**Complete:** Click **Merge**.

Source: warehouses.md

## Warehouses Reference

Warehouses define the underlying object storage (S3, GCS, Azure, etc.) where catalogs store data.

## 1. API

**Base Endpoint:** `/api/v1/warehouses`

## List Warehouses

**Method:** `GET`

**Path:** `/api/v1/warehouses`

**Auth:** Tenant Admin Token

```
curl -X GET http://localhost:8080/api/v1/warehouses \
-H "Authorization: Bearer <token>"
```

## Create Warehouse

**Method:** `POST`

**Path:** `/api/v1/warehouses`

**Body:**

```
{
  "name": "s3-warehouse",
  "storage_config": {
    "s3.bucket": "my-bucket",
    "s3.region": "us-east-1",
    "s3.access-key-id": "AKIA...",
    "s3.secret-access-key": "secret..."
  },
  "vending_strategy": {
    "AwsStatic": {
      "access_key_id": "AKIA...",
      "secret_access_key": "secret..."
    }
  }
}
```

```
curl -X POST http://localhost:8080/api/v1/warehouses \
-H "Authorization: Bearer <token>" \
-H "Content-Type: application/json" \
-d '...'
```

## Delete Warehouse

**Method:** `DELETE`

**Path:** `/api/v1/warehouses/{name}`

## 2. CLI

## List Warehouses

```
pangolin-admin list-warehouses
```

## Create Warehouse (S3)

```
pangolin-admin create-warehouse s3-warehouse \  
  --type s3 \  
  --bucket my-data-bucket \  
  --region us-east-1 \  
  --access-key "AKIA..." \  
  --secret-key "secret..."
```

## Delete Warehouse

```
pangolin-admin delete-warehouse s3-warehouse
```

## 3. Python SDK (`pypangolin`)

### List Warehouses

```
warehouses = client.warehouses.list()  
for w in warehouses:  
    print(w.name)
```

### Create Warehouse (S3)

```
wh = client.warehouses.create_s3(  
    name="s3-prod",  
    bucket="my-prod-bucket",  
    region="us-east-1",  
    access_key="AKIA...",  
    secret_key="secret...",  
    vending_strategy="AwsStatic"  
)  
print(f"Created warehouse: {wh.name}")
```

## 4. UI

**Log in** as a **Tenant Admin**.

Navigate to **Warehouses** in the sidebar.

**List:** Validates current warehouses.

**Create:** Click "**Create Warehouse**".

**Name:** Unique identifier.

**Type:** Select S3 (MinIO provided by default config).

**Credentials:** Enter keys if not using IAM roles.

Click **Save**.

**Manage:** Edit or delete warehouses.

Source: README.md

## Pangolin Management UI

Welcome to the documentation for the Pangolin Management UI. This web interface is built with SvelteKit and provides a comprehensive suite of tools for managing your data lakehouse.

### Navigation Map

#### General Overview

Start here to understand the interface layout, theme switching, and the core dashboard experience.

#### Data Management

Learn how to browse your data, manage branches (Git-for-Data), and handle merge operations.

Data Explorer

Branching & Tagging

Merge Operations & Conflict Resolution

Table Maintenance

#### Discovery & Governance

Tools for finding data and auditing system activity.

Data Discovery Portal

Access Request Workflow

Audit Log Viewer

## Administration

Configure the foundational resources of your Pangolin instance.

Multi-Tenant Management

Warehouse & Catalog Configuration

Service Users & API Keys

## Security

Manage who can access what.

Users & Roles

Granular Permissions UI

Token Management (JWT)

## ⚙ Configuration & Development

### Local Development Setup

When running the UI locally against a backend services, ensure your `vite.config.ts` proxy matches your backend port (default: 8080).

**Common Pitfall:** The backend defaults to port `8080`. If `vite.config.ts` proxies to `8085`, you may encounter 500 errors or connection refusals. Ensure the proxy target matches:

```
server: {
  proxy: {
    '/api': 'http://localhost:8080', // Match backend port
    '/v1': 'http://localhost:8080',
    '/oauth': 'http://localhost:8080'
  }
}
```

Source: administration.md

## Administration

Administrative tools in Pangolin are split between global system owners (Root) and individual team lead (Tenant Admins).

### Root-Level Management

Root users have access to the **Root Dashboard** for cross-tenant operations.

# Tenant Management

**Lifecycle:** Create, Update, and Delete tenants.

**Isolation:** Each tenant represents a fully isolated environment with its own users and catalogs.

## System Settings

Configure global properties like JWT secrets, allowed email domains for OAuth, and system-wide maintenance windows.

## Tenant-Level Management

Tenant Admins manage the resources within their specific project or organization.

### 1. Warehouse Management

Warehouses define the connection and authentication to your storage (S3, GCS, Azure, etc.).

**Vending Strategy:** Choose between ``AwsStatic``, ``AwsSts``, ``AzureSas``, or ``GcpDownscoped`` for credential management.

**Connection Test:** Verify connectivity to your storage bucket directly from the creation form.

### 2. Catalog Management

Connect warehouses to your namespace hierarchies.

**Internal Catalogs:** Managed by Pangolin's metadata store.

**Federated Catalogs:** Proxy external REST catalogs (e.g., Tabular, Snowflake). The UI allows you to configure secret headers for these connections.

### 3. Service Users

For machine-to-machine integrations (CI/CD, internal tools).

**API Keys:** Generate long-lived API keys for non-human users. Keys are displayed **once** upon creation or rotation.

**Key Rotation:** Securely rotate keys to invalidate old credentials without deleting the user.

**Scoped Identity:** Service users inherit specific roles just like regular users.

## UI Dashboard

The **Dashboard** tab provides quick metrics:

**Tenant Health:** Total catalogs and active users.

**Storage Metrics:** (Feature plan) Usage statistics per warehouse.

**Recent Operations:** A subset of the audit logs showing recent management actions.

Source: `data_management.md`

## Data Management

This guide covers how to browse, version, and maintain your data assets through the Pangolin UI.

### Data Explorer

The Explorer is the heart of the Pangolin user experience, providing a hierarchical "Finder-like" view of your lakehouse.

### Hierarchy Navigation

**Catalogs:** Expand to see namespaces.

**Namespaces:** Recursive folders containing tables, views, or further sub-namespaces.

**Assets:** Tables and Views are represented with distinct icons.

### Table Detail View

Clicking an asset opens a comprehensive detail pane:

**Schema:** Columns, types, and nullability.

**Snapshots:** A chronological list of table commits.

**Properties:** Key-value pairs defining table behavior.

**Maintenance:** One-click actions for table health (e.g., Compaction, Snapshot Expiration).

### Bulk Operations

**Asset List:** View all tables and namespaces within a catalog in a searchable list.

**Bulk Delete:** Select multiple assets to delete them in a single batch operation with progress tracking.

### Smart Validation

-

**Name Validation:** Creation forms (Catalogs, Warehouses) automatically check for duplicate names in real-time, preventing errors before submission.

## Versioning (Git-for-Data)

Pangolin brings version control to your data via a dedicated UI for branches and tags.

## Branch Management

**List Branches:** See all branches for a specific catalog.

**Create Branch:** Spin up a new isolated environment (e.g., `feature/cleanup`) from any existing commit or branch.

**Delete Branch:** Remove experimental environments once they are no longer needed.

## Tag Management

Create permanent, named aliases for specific commits (e.g., `v1\_analytics\_gold`) to ensure reproducibility in reporting.

## Merge Operations

Manage complex data integration workflows with the Merge UI.

### Initiate Merge

Trigger a merge from one branch into another (e.g., `ingest` -> `main`).

### Conflict Resolution

If Pangolin detects changes to the same metadata in both branches, the UI enters **Resolution Mode**:

**Conflict List:** See exactly which assets or properties are in conflict.

**Visual Diff:** Compare values from the Source, Target, and Base (3-way merge).

**Resolver:** Select a winner (Source or Target) or apply a manual fix.

## Table Maintenance

Ensure your lakehouse stays performant without manual scripting.

**Compaction:** Merge small files to improve query performance.

**Manifest Rewriting:** Optimize metadata structures.

**Snapshot Expiration:** Remove old data versions to save storage costs.

\*All maintenance tasks can be triggered manually from the Table Detail header or scheduled (future feature).\*

## Source: discovery\_governance.md

# Discovery & Governance

Pangolin provides a transparent and searchable view of your data estate, combined with rigorous auditing and access governance.

## Discovery Portal

The Discovery view is designed for data consumers to find and understand assets without needing deep technical knowledge of the catalog hierarchy.

## Search & Filtering

**Fuzzy Search:** Search by asset name, description, or owner.

**Tag Filtering:** Filter assets by business tags (e.g., `PII`, `Financial`, `Certified`).

**Full Qualified Names:** Search results display the full `catalog.namespace.table` for clarity.

## Asset Cards

Each search result presents a card with:

**Status Indicators:** Indicates if the data is currently accessible to you.

**Business Metadata:** Descriptions and ownership info.

**Schema Preview:** Quick glimpse at the top columns.

## Access Request Workflow

For assets you discover but do not have permissions to read, Pangolin provides a self-service workflow.

**Request Access:** From the Discovery card, click "Request Access".

**Justification:** Provide a reason for the request.

**Admin Review:** Tenant Admins see a notification and a dedicated "Requests" dashboard to Approve or Deny.

**Approval:** Upon approval, Pangolin automatically grants the necessary RBAC permissions.

## Audit Log Viewer

The Audit Log provides a complete, immutable history of all actions performed in both the UI and via the API.

## Features

**Global View:** Admins can see events across the entire tenant.

**Details Modal:** Click an event to see the raw JSON payload, Actor ID, Client IP, and Timestamp.

**Filtering:** (Upcoming) Filter by Action Type (e.g., `Asset\_Create`) or User ID.

## Logged Actions

Pangolin logs everything from login attempts to table deletions and permission grants, ensuring compliance and security.

Source: overview.md

## General Overview

The **Pangolin Management UI** is a professional, high-performance web interface designed to provide total visibility and control over your data lakehouse.

### Design & Theme

Pangolin features a modern, clean aesthetic tailored for data engineers and administrators.

**Dark/Light Mode:** Toggle between themes using the icon in the top header. The UI defaults based on your system preference.

**Glassmorphism:** Subtle transparency and backdrop blurs are used for modals and sidebars to maintain context.

**Responsive Design:** The sidebar collapses on smaller screens to maximize the data viewing area.

### Interface Layout

The interface is divided into three main zones:

#### 1. Unified Sidebar (Left)

Your primary navigation hub. It dynamically updates based on your **User Role**:

**Root:** System-wide management (Tenants, System Settings).

**Admin:** Tenant resources (Users, Catalogs, Warehouses).

**User:** Data focused (Explorer, Discovery).

## 2. Global Header (Top)

Provides constant access to:

**Global Search:** Instantly find assets (Tables, Views, Namespaces) across all catalogs with type-ahead suggestions.

**Notifications:** Toast alerts for operation statuses (e.g., "Table Created Successfully").

**User Profile:** Access to your tokens, settings, and logout.

**Theme Toggle:** Switch between light and dark modes.

## 3. Workspace Area (Center)

The main area where you interact with data, forms, and logs.

### Dashboard Views

#### Root Dashboard

\*Accessible only to system owners.\*

Provides high-level metrics on system health.

**Stats Cards:** Total Tenants, Warehouses, Catalogs, Namespaces, and Tables across the platform.

#### Tenant Dashboard

\*The home screen for most users.\*

Displays recent activity and resource usage.

**Resource Stats:** Real-time counts of Catalogs, Warehouses, Namespaces, and Tables within your tenant.

### Interaction Patterns

**Toasts:** Non-intrusive popups in the top right confirm successful actions or explain errors.

**Modals:** Used for focused tasks like creating a new branch or inviting a user.

**Optimistic UI:** Many actions (like renaming a catalog) update the UI immediately while the backend completes the task, providing a snappy experience.

Source: security.md

# Security & Identity

This guide covers how to manage identities and access control within the Pangolin UI.

## User Management

Manage the lifecycle of human users in your tenant.

**Invite Users:** Create accounts with a default role.

**Profile Details:** View a user's role, home tenant, and recent activity.

**Revocation:** Delete or deactivate users (Root/Admin only).

## Role-Based Access Control (RBAC)

Pangolin uses a dynamic RBAC model to grant precise permissions.

### Roles

The UI provides a dedicated **Roles** dashboard where you can:

**Define Roles:** Create a "Data Scientist" or "Auditor" role.

**Assign Roles to Users:** One user can hold multiple roles simultaneously.

### Permissions Interface

When creating or editing a role, the UI provides a granular selector for:

**Action:** ``READ``, ``WRITE``, ``CREATE``, ``DELETE``, ``LIST``, ``ALL`` (Full Access), ``MANAGE_DISCOVERY``.

**Scope Type:** ``System``, ``Tenant``, ``Catalog``, ``Namespace``, ``Table``, ``View``.

**Target:** The specific resource (e.g., the ``finance`` namespace).

## Token Management

manage your session and programmatic access via JWT tokens.

### Personal Tokens

From the **Profile** menu, users can:

**List Sessions:** See all active web and API sessions.

**Revoke Tokens:** Manually sign out other devices or rotate your current session.

### Admin Token Control

Tenant Admins can view and revoke tokens for **any user** within their tenant to mitigate security incidents or during user offboarding.

## Authentication & Identity Providers

Pangolin's UI seamlessly handles multiple authentication flows:

### 1. No Auth Mode (Local Development)

For local testing without an identity provider.

**Trigger:** Set ``PANGOLIN_NO_AUTH=true`` and ``VITE_NO_AUTH=true``.

**Login:** Use username ``root`` and password ``root``.

**Behavior:** Simulates a Root user session with full system access. Sessions persist across refreshes via local storage.

### 2. Native Auth

Username and Bcrypt-hashed password management for standard deployments.

### 3. OAuth

One-click login with Google, GitHub, or Microsoft (configured in [System Settings](#)).

Source: README.md

## Utilities

This directory contains utility documentation for maintaining and working with the Pangolin project.

## Available Guides

### Regenerating OpenAPI Documentation

Complete guide for regenerating the OpenAPI specification files (JSON and YAML) after making changes to API handlers or models.

#### Quick Commands:

```
# Generate JSON
cd pangolin && cargo run -p pangolin_api --bin export_openapi json 2>/dev/null >
../docs/api/openapi.json

# Generate YAML
cd pangolin && cargo run -p pangolin_api --bin export_openapi yaml 2>/dev/null >
../docs/api/openapi.yaml
```

# Future Utilities

Additional utility guides will be added here as needed:

Database migrations

Test data generation

Performance benchmarking

Deployment scripts

Backup and restore procedures

## Source: regenerating-openapi.md

# Regenerating OpenAPI Documentation

This guide explains how to regenerate the OpenAPI specification files after making changes to API handlers or models.

## Overview

The Pangolin API uses utoipa to automatically generate OpenAPI 3.0 specifications from Rust code annotations. The OpenAPI spec is available in two formats:

**JSON:** ``docs/api/openapi.json`` (4734 lines)

**YAML:** ``docs/api/openapi.yaml`` (3051 lines)

## When to Regenerate

Regenerate the OpenAPI documentation when you:

Add new API endpoints

Modify existing endpoint parameters or responses

Add or update request/response models

Change API descriptions or metadata

Update security schemes

## Prerequisites

Ensure you have:

Rust toolchain installed

All dependencies up to date (``cargo update``)

-

The `pangolin\_api` crate compiles successfully

## Regeneration Commands

### Generate JSON Format

```
cd pangolin
cargo run -p pangolin_api --bin export_openapi json 2>/dev/null >
../docs/api/openapi.json
```

**Or** from the project root:

```
cd pangolin && cargo run -p pangolin_api --bin export_openapi json 2>/dev/null >
../docs/api/openapi.json
```

### Generate YAML Format

```
cd pangolin
cargo run -p pangolin_api --bin export_openapi yaml 2>/dev/null >
../docs/api/openapi.yaml
```

**Or** from the project root:

```
cd pangolin && cargo run -p pangolin_api --bin export_openapi yaml 2>/dev/null >
../docs/api/openapi.yaml
```

### Generate Both Formats

```
cd pangolin

# Generate JSON
cargo run -p pangolin_api --bin export_openapi json 2>/dev/null >
../docs/api/openapi.json

# Generate YAML
cargo run -p pangolin_api --bin export_openapi yaml 2>/dev/null >
../docs/api/openapi.yaml
```

## Verification

After regeneration, verify the output:

### Check File Sizes

```
wc -l docs/api/openapi.json docs/api/openapi.yaml
```

Expected output (approximate):

```
4734 docs/api/openapi.json
3051 docs/api/openapi.yaml
7785 total
```

## Validate JSON

```
cat docs/api/openapi.json | jq . > /dev/null && echo "✓ Valid JSON"
```

## View YAML Header

```
head -20 docs/api/openapi.yaml
```

Should show:

```
openapi: 3.0.3
info:
  title: Pangolin Catalog API
  description: Multi-tenant Apache Iceberg REST Catalog...
  ...
```

## Testing the Swagger UI

After regeneration, test the Swagger UI to ensure all endpoints appear correctly:

**Start the API server:**

```
cd pangolin
cargo run -p pangolin_api
```

**Open Swagger UI** in your browser:

```
http://localhost:8080/swagger-ui
```

**Verify:**

All 13 API tags are present

All 67 endpoints are listed

Schemas are properly rendered

-

"Try it out" functionality works

## Troubleshooting

### Issue: Empty or Invalid Output

**Symptom:** Generated file is empty or contains compilation errors

**Solution:** Check that stderr is redirected properly:

```
# Correct (redirects stderr to /dev/null)
cargo run -p pangolin_api --bin export_openapi json 2>/dev/null >
../docs/api/openapi.json

# Incorrect (stderr mixed with stdout)
cargo run -p pangolin_api --bin export_openapi json > ../docs/api/openapi.json
```

### Issue: Compilation Errors

**Symptom:** Export command fails with compilation errors

**Solution:**

Ensure all code compiles:

```
cargo check -p pangolin_api
```

Fix any compilation errors before regenerating

Ensure all models referenced in handlers have ``#[derive(ToSchema)]``

### Issue: Missing Endpoints

**Symptom:** Some endpoints don't appear in the generated spec

**Solution:**

Verify the handler has ``#[utoipa::path(...)]`` annotation

Ensure the handler is imported in ``openapi.rs``:

```
use crate::your_handlers::*;
```

Add the handler function name to the ``paths(...)`` list in ``openapi.rs``

### Issue: Missing Schemas

**Symptom:** Request/response types show as empty objects

## Solution:

Add ``#[derive(ToSchema)]`` to the struct:

```
use utoipa::ToSchema;

#[derive(Serialize, Deserialize, ToSchema)]
pub struct YourStruct {
    // fields...
}
```

Import the type in ``openapi.rs``:

```
use crate::your_module::YourStruct;
```

Add it to the ``components(schemas(...))`` list in ``openapi.rs``

## Adding New Endpoints

When adding a new API endpoint, follow these steps:

### 1. Annotate the Handler

Add ``#[utoipa::path(...)]`` above your handler function:

```
use utoipa::ToSchema;

#[derive(Deserialize, ToSchema)]
pub struct CreateItemRequest {
    pub name: String,
}

#[derive(Serialize, ToSchema)]
pub struct ItemResponse {
    pub id: Uuid,
    pub name: String,
}

#[utoipa::path(
    post,
    path = "/api/v1/items",
    tag = "Items",
    request_body = CreateItemRequest,
    responses(
        (status = 201, description = "Item created", body = ItemResponse),
        (status = 400, description = "Bad request"),
        (status = 500, description = "Internal server error")
    ),
    security(("bearer_auth" = []))
)]
pub async fn create_item(
    State(store): State<AppState>,
```

```

    Json(payload): Json<CreateItemRequest>,
) -> impl IntoResponse {
    // implementation...
}

```

## 2. Update openapi.rs

Add imports:

```
use crate::item_handlers::*;
```

Add to paths:

```

#[openapi(
    paths(
        // ... existing paths ...
        create_item,
        list_items,
        // etc.
    ),

```

Add schemas:

```

components(
    schemas(
        // ... existing schemas ...
        CreateItemRequest,
        ItemResponse,
    )
)

```

Add tag (if new category):

```

tags(
    // ... existing tags ...
    (name = "Items", description = "Item management endpoints"),
)

```

## 3. Regenerate

```

cd pangolin
cargo run -p pangolin_api --bin export_openapi json 2>/dev/null >
../docs/api/openapi.json
cargo run -p pangolin_api --bin export_openapi yaml 2>/dev/null >
../docs/api/openapi.yaml

```

## 4. Verify

Check Swagger UI to ensure your new endpoint appears correctly.

## Export Utility Source

The export utility is located at:

```
pangolin/pangolin_api/src/bin/export_openapi.rs
```

It supports both JSON and YAML formats via command-line argument:

```
`cargo run bin export_openapi json` - Outputs JSON
```

```
`cargo run bin export_openapi yaml` - Outputs YAML
```

## Current API Statistics

**Total Handlers:** 67

**API Tags:** 13

**Core Models:** 35+

**OpenAPI Version:** 3.0.3

**JSON Size:** ~4734 lines

**YAML Size:** ~3051 lines

## Related Documentation

[API Documentation](#)

[Authentication Guide](#)

[utoipa Documentation](#)

[OpenAPI 3.0 Specification](#)

## Automation

Consider adding these commands to your CI/CD pipeline or Makefile:

```
# Makefile
.PHONY: openapi-json openapi-yaml openapi-all

openapi-json:
cd pangolin && cargo run -p pangolin_api --bin export_openapi json 2>/dev/null >
../docs/api/openapi.json

openapi-yaml:
```

```
cd pangolin && cargo run -p pangolin_api --bin export_openapi yaml 2>/dev/null >
../docs/api/openapi.yaml

openapi-all: openapi-json openapi-yaml
@echo "✓ OpenAPI documentation regenerated"
```

Then simply run:

```
make openapi-all
```

## Source: README.md

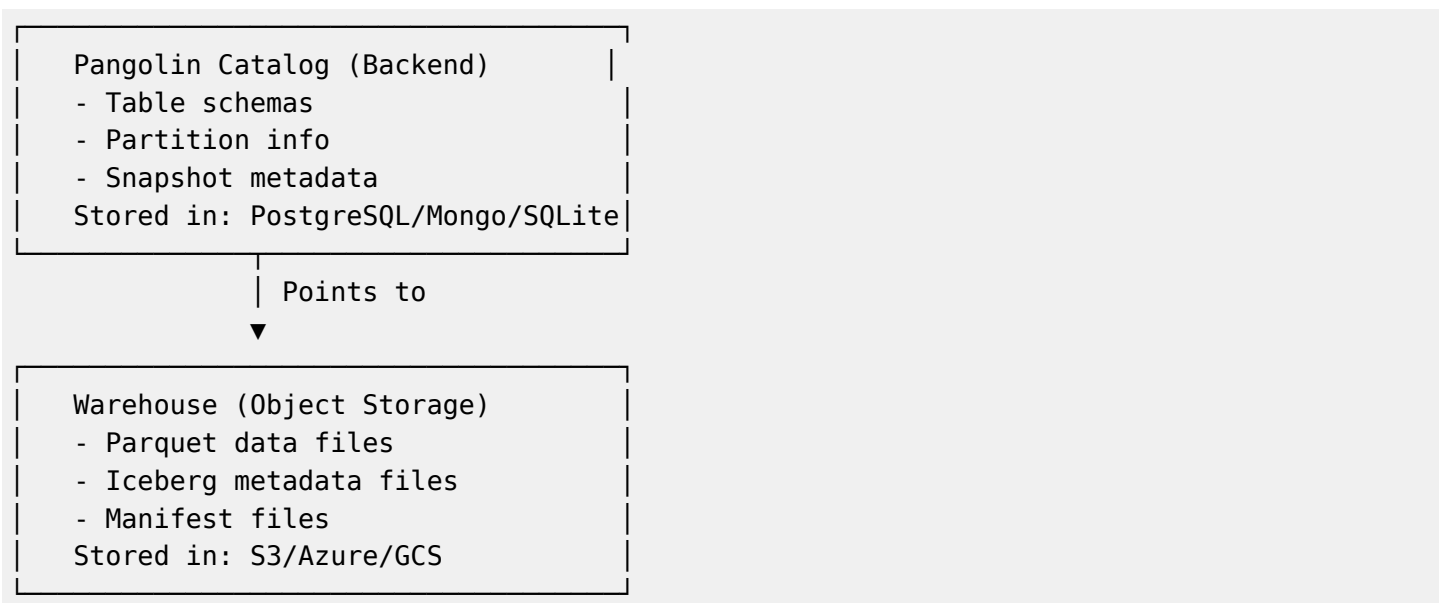
## Warehouse Storage

Warehouses in Pangolin define where your **actual data files** are stored (Parquet, Avro, ORC files), separate from the catalog metadata.

## Warehouse vs Backend Storage

It's important to understand the distinction:

Component	Purpose	Storage	Examples
<b>Backend Storage</b>	Catalog metadata	PostgreSQL, MongoDB, SQLite	Table schemas, partitions, snapshots
<b>Warehouse Storage</b>	Actual data files	S3, Azure Blob, GCS	Parquet files, metadata.json



## Warehouse Concept

A **warehouse** in Pangolin is a named configuration that specifies:

**Storage type:** S3, Azure Blob Storage, or Google Cloud Storage

**Location:** Bucket/container and path prefix

**Credentials:** How to authenticate (static credentials or STS/IAM roles)

**Region:** Geographic location of storage

[!TIP]

**Flat Key Support:** As of v0.1.0, `storage\_config` supports flat keys (e.g., `"s3.bucket": "mybucket"`) as an alternative to nested objects. This is often easier to pass via CLI or environment-driven scripts.

## Example Warehouse (Flat Keys)

```
{
  "name": "production-s3",
  "storage_config": {
    "s3.bucket": "my-company-datalake",
    "s3.region": "us-east-1"
  },
  "vending_strategy": {
    "type": "AwsSts",
    "role_arn": "arn:aws:iam::123456789:role/PangolinDataAccess"
  }
}
```

## Warehouse Patterns

### Pattern 1: Warehouse Attached to Catalog

The catalog configuration includes a warehouse reference:

```
{
  "name": "analytics",
  "type": "local",
  "warehouse": "production-s3",
  "properties": {}
}
```

#### **Benefits:**

Centralized credential management

Consistent storage configuration

Automatic credential vending to clients

Easier to manage and audit

**Client Configuration:** Minimal - Pangolin vends credentials automatically

## Pattern 2: Catalog Without Warehouse

The catalog has no warehouse attached:

```
{
  "name": "analytics",
  "type": "local",
  "warehouse": null,
  "properties": {}
}
```

### Benefits:

Clients control their own storage access

Flexible for multi-cloud scenarios

Useful when clients have their own credentials

**Client Configuration:** Clients must configure storage themselves

## Authentication Methods

### VendingStrategy Enum

Pangolin uses the `vending\_strategy` field to configure credential vending. The `use\_sts` field is deprecated but kept for backward compatibility.

### Available Strategies:

#### 1. AwsSts - AWS STS Temporary Credentials (Recommended)

```
{
  "name": "prod-s3",
  "storage_config": {
    "bucket": "my-datalake",
    "region": "us-east-1",
    "s3.role-arn": "arn:aws:iam::123456789:role/PangolinDataAccess"
  },
  "vending_strategy": {
    "AwsSts": {
      "role_arn": "arn:aws:iam::123456789:role/PangolinDataAccess"
    }
  }
}
```

[!NOTE]

STS Vending requires `PANGOLIN\_STS\_ROLE\_ARN` server configuration.

## 2. AwsStatic - AWS Static Credentials

```
{
  "name": "dev-s3",
  "storage_config": {
    "bucket": "my-dev-datalake",
    "region": "us-east-1",
    "s3.access-key-id": "AKIAIOSFODNN7EXAMPLE",
    "s3.secret-access-key": "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
  },
  "vending_strategy": "AwsStatic"
}
```

**Pros:** Simple, works everywhere (Development)

**Cons:** Less secure, credentials don't expire

## 3. Client Provided (No Vending)

```
{
  "name": "client-provided",
  "storage_config": {
    "bucket": "my-datalake"
  },
  "vending_strategy": "None"
}
```

Clients must provide their own credentials via environment variables or Spark/Iceberg config.

## Deprecated: use\_sts Field

The `use\_sts` boolean field is deprecated. Use `vending\_strategy` instead.

**Old Format** (Deprecated):

```
{
  "use_sts": true,
  "role_arn": "arn:aws:iam::123:role/Access"
}
```

**New Format** (Current):

```
{
  "vending_strategy": {
    "type": "AwsSts",
    "role_arn": "arn:aws:iam::123:role/Access",
    "external_id": null
  }
}
```

# Supported Storage Types

Storage	Status	Best For
----- ----- -----		
<a href="#">AWS S3</a>	Production	Most common, excellent performance
<a href="#">Azure Blob</a>	Production	Azure-native deployments
<a href="#">Google Cloud Storage</a>	Production	GCP-native deployments
<a href="#">Local Filesystem</a>	△ Dev/Test	Local development & testing

## Quick Start

### 1. Create a Warehouse

```
curl -X POST http://localhost:8080/api/v1/warehouses \
-H "X-Pangolin-Tenant: my-tenant" \
-H "Content-Type: application/json" \
-d '{
  "name": "production-s3",
  "storage_config": {
    "bucket": "my-datalake",
    "region": "us-east-1"
  },
  "vending_strategy": {
    "type": "AwsSts",
    "role_arn": "arn:aws:iam::123456789:role/DataAccess",
    "external_id": null
  }
}'
```

### 2. Create a Catalog with Warehouse

```
curl -X POST http://localhost:8080/api/v1/catalogs \
-H "X-Pangolin-Tenant: my-tenant" \
-H "Content-Type: application/json" \
-d '{
  "name": "analytics",
  "type": "local",
  "warehouse": "production-s3"
}'
```

### 3. Use from Pylceberg

```
from pyiceberg.catalog import load_catalog

# Pangolin vends credentials automatically
catalog = load_catalog(
    "pangolin",
    **{
```

```

        "uri": "http://localhost:8080/api/v1/catalogs/analytics",
        "warehouse": "s3://my-datalake/analytics/"
    }
)

# Create table - Pangolin handles storage access
catalog.create_table(
    "db.table",
    schema=schema
)

```

## Client Configuration

### With Warehouse (Recommended)

When a catalog has a warehouse attached, Pangolin automatically vends credentials to clients via the `X-Iceberg-Access-Delegation` header.

**PyIceberg:** No storage configuration needed

**PySpark:** No storage configuration needed

### Without Warehouse

When a catalog has no warehouse, clients must configure storage themselves.

**PyIceberg:** Configure S3/Azure/GCS credentials

**PySpark:** Configure Hadoop filesystem properties

See individual storage guides for details:

[S3 Client Configuration](#)

[Azure Client Configuration](#)

[GCS Client Configuration](#)

## Best Practices

### Security

**Use STS/IAM Roles:** Prefer temporary credentials over static keys

**Least Privilege:** Grant minimum required permissions

**Separate Warehouses:** Use different warehouses for dev/staging/prod

**Audit Access:** Enable CloudTrail/Azure Monitor/GCS audit logs

### Performance

**Regional Colocation:** Place warehouse in same region as compute

**Bucket Naming:** Use descriptive, hierarchical names

**Lifecycle Policies:** Archive old data to cheaper storage tiers

**Compression:** Use Snappy or Zstd for Parquet files

## Organization

**Naming Convention:** `{environment}-{region}-{purpose}`

Examples: `prod-us-east-1-analytics`, `dev-eu-west-1-ml`

**Path Structure:** `s3://bucket/{catalog}/{namespace}/{table}/`

**Multi-Tenant:** Use separate buckets or prefixes per tenant

## Troubleshooting

### Permission Denied

Error: Access Denied to s3://my-bucket/path/

#### Solutions:

Check IAM role permissions

Verify bucket policy

Check STS assume role permissions

Verify warehouse configuration

### Credential Vending Not Working

Error: No credentials provided

#### Solutions:

Ensure catalog has warehouse attached

Check warehouse `use\_sts` setting

Verify IAM role ARN

Check Pangolin server has permission to assume role

### Slow Performance

## Solutions:

Check region - ensure compute and storage are colocated

Enable S3 Transfer Acceleration

Use larger instance types for compute

Check network bandwidth

## Next Steps

[S3 Warehouse Configuration](#)

[Azure Blob Warehouse Configuration](#)

[GCS Warehouse Configuration](#)

[Backend Storage Options](#)

Source: [azure.md](#)

# Azure Blob Storage Warehouse Configuration

Configure Azure Blob Storage (ADLS Gen2) as a warehouse for storing Iceberg table data in Pangolin.

## Overview

Azure Blob Storage with ADLS Gen2 provides:

Hierarchical namespace for better performance

Integration with Azure ecosystem

Enterprise-grade security

Cost-effective storage tiers

Global availability

## Prerequisites

Azure Storage Account with ADLS Gen2 enabled

Storage account name and access key or SAS token

Container created for Iceberg tables

## Warehouse Configuration

# AzureSas - Storage Account Key

```
curl -X POST http://localhost:8080/api/v1/warehouses \
-H "X-Pangolin-Tenant: my-tenant" \
-H "Content-Type: application/json" \
-d '{
  "name": "azure-prod",
  "storage_config": {
    "account_name": "mystorageaccount",
    "container": "iceberg-tables"
  },
  "vending_strategy": {
    "type": "AzureSas",
    "account_name": "mystorageaccount",
    "account_key": "your-account-key-here=="
  }
}'
```

## None - Client-Provided Credentials

For scenarios where clients provide their own Azure credentials:

```
curl -X POST http://localhost:8080/api/v1/warehouses \
-H "X-Pangolin-Tenant: my-tenant" \
-H "Content-Type: application/json" \
-d '{
  "name": "azure-client-creds",
  "storage_config": {
    "account_name": "mystorageaccount",
    "container": "iceberg-tables"
  },
  "vending_strategy": {
    "type": "None"
  }
}'
```

**Note:** Managed Identity support via Azure AD is not yet implemented in the VendingStrategy enum. Use AzureSas for credential vending or None for client-provided credentials.

## Client Configuration

### Scenario 1: Catalog WITH Warehouse (Recommended)

Pangolin vends credentials automatically.

#### Pylceberg

```
from pyiceberg.catalog import load_catalog
```

```

# Pangolin vends Azure credentials automatically
catalog = load_catalog(
    "pangolin",
    **{
        "uri": "http://localhost:8080/api/v1/catalogs/analytics",
        "warehouse":
"abfss://iceberg-tables@mystorageaccount.dfs.core.windows.net/analytics/"
    }
)

# Create table - Pangolin handles Azure access
from pyiceberg.schema import Schema
from pyiceberg.types import NestedField, StringType, IntegerType

schema = Schema(
    NestedField(1, "id", IntegerType(), required=True),
    NestedField(2, "name", StringType(), required=True)
)

table = catalog.create_table("db.users", schema=schema)

# Write data
import pyarrow as pa
data = pa.table({"id": [1, 2, 3], "name": ["Alice", "Bob", "Charlie"]})
table.append(data)

# Read data
df = table.scan().to_pandas()
print(df)

```

## PySpark

```

from pyspark.sql import SparkSession

# Pangolin vends credentials automatically
spark = SparkSession.builder \
    .appName("Pangolin Iceberg Azure") \
    .config("spark.sql.catalog.pangolin", "org.apache.iceberg.spark.SparkCatalog") \
    .config("spark.sql.catalog.pangolin.catalog-impl",
"org.apache.iceberg.rest.RESTCatalog") \
    .config("spark.sql.catalog.pangolin.uri",
"http://localhost:8080/api/v1/catalogs/analytics") \
    .config("spark.sql.catalog.pangolin.warehouse",
"abfss://iceberg-tables@mystorageaccount.dfs.core.windows.net/analytics/") \
    .getOrCreate()

# Create table
spark.sql("""
    CREATE TABLE pangolin.db.users (
        id INT,
        name STRING

```

```

    ) USING iceberg
    """)

# Write data
data = [(1, "Alice"), (2, "Bob"), (3, "Charlie")]
df = spark.createDataFrame(data, ["id", "name"])
df.writeTo("pangolin.db.users").append()

# Read data
spark.table("pangolin.db.users").show()

```

## Scenario 2: Catalog WITHOUT Warehouse

Clients configure Azure access themselves.

### PyIceberg

```

from pyiceberg.catalog import load_catalog

catalog = load_catalog(
    "pangolin",
    **{
        "uri": "http://localhost:8080/api/v1/catalogs/analytics",
        "warehouse":
"abfss://iceberg-tables@mystorageaccount.dfs.core.windows.net/analytics/",
        # Client-side Azure configuration
        "adls.account-name": "mystorageaccount",
        "adls.account-key": "your-account-key-here==",
        "adls.connection-string":
"DefaultEndpointsProtocol=https;AccountName=mystorageaccount;AccountKey=your-key;EndpointSuffix=core.windows.net"
    }
)

# Use catalog normally
table = catalog.load_table("db.users")
df = table.scan().to_pandas()

```

### PySpark

```

from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("Pangolin Iceberg Azure") \
    .config("spark.sql.catalog.pangolin", "org.apache.iceberg.spark.SparkCatalog") \
    .config("spark.sql.catalog.pangolin.catalog-impl",
"org.apache.iceberg.rest.RESTCatalog") \
    .config("spark.sql.catalog.pangolin.uri",
"http://localhost:8080/api/v1/catalogs/analytics") \

```

```

        .config("spark.sql.catalog.pangolin.warehouse",
"abfss://iceberg-tables@mystorageaccount.dfs.core.windows.net/analytics/") \
        .config("spark.hadoop.fs.azure.account.key.mystorageaccount.dfs.core.windows.net",
"your-account-key-here==") \
        .getOrCreate()

# Use Spark normally
spark.table("pangolin.db.users").show()

```

## Azure Best Practices

### Performance

**Use Premium Block Blobs** for high-performance workloads

**Enable Hierarchical Namespace** (ADLS Gen2) for better performance

**Use Azure CDN** for global distribution

**Collocate compute and storage** in same region

### Security

**Use Managed Identities** instead of storage keys

**Enable Storage Encryption** (enabled by default)

**Use Private Endpoints** for network isolation

**Enable Storage Analytics Logging**

**Use Azure RBAC** for fine-grained access control

### Cost Optimization

**Use Lifecycle Management** to tier data to Cool/Archive

**Enable Soft Delete** for data protection

**Use Azure Storage Reserved Capacity** for discounts

**Monitor with Azure Cost Management**

## Troubleshooting

### Access Denied

```
Error: This request is not authorized to perform this operation
```

**Solutions:**

Check storage account key is correct

Verify container exists

Check managed identity permissions

Ensure storage account allows public access (if needed)

## Connection Issues

**Solutions:**

Verify storage account name

Check endpoint URL format

Ensure firewall rules allow access

Check network connectivity

## Additional Resources

[Azure Blob Storage Documentation](#)

[ADLS Gen2 Documentation](#)

[Iceberg Azure Configuration](#)

## Next Steps

[S3 Warehouse](#)

[GCS Warehouse](#)

[Warehouse Concept](#)

Source: `gcs.md`

# Google Cloud Storage Warehouse Configuration

Configure Google Cloud Storage (GCS) as a warehouse for storing Iceberg table data in Pangolin.

## Overview

Google Cloud Storage provides:

Multi-regional and dual-regional storage

Strong consistency

Integration with Google Cloud ecosystem

Automatic encryption at rest

Cost-effective storage classes

## Prerequisites

GCS bucket created

Service account with appropriate permissions

Service account key JSON file

## Warehouse Configuration

### GcpDownscoped - Service Account Credentials

```
curl -X POST http://localhost:8080/api/v1/warehouses \
-H "X-Pangolin-Tenant: my-tenant" \
-H "Content-Type: application/json" \
-d '{
  "name": "gcs-prod",
  "storage_config": {
    "bucket": "my-iceberg-tables",
    "project_id": "my-gcp-project"
  },
  "vending_strategy": {
    "type": "GcpDownscoped",
    "service_account_email": "iceberg@my-gcp-project.iam.gserviceaccount.com",
    "private_key": "-----BEGIN PRIVATE
KEY-----\nMIIEvQIBADANBgkqhkiG9w0BAQE...\n-----END PRIVATE KEY-----\n"
  }
}'
```

### None - Client-Provided Credentials

For scenarios where clients provide their own GCS credentials:

```
curl -X POST http://localhost:8080/api/v1/warehouses \
-H "X-Pangolin-Tenant: my-tenant" \
-H "Content-Type: application/json" \
-d '{
  "name": "gcs-client-creds",
  "storage_config": {
    "bucket": "my-iceberg-tables",
    "project_id": "my-gcp-project"
  },
  "vending_strategy": {
    "type": "None"
  }
}'
```

```
}  
}'
```

**Note:** Workload Identity (GKE) support is not yet implemented in the VendingStrategy enum. Use GcpDownscoped for credential vending or None for client-provided credentials.

## Client Configuration

### Scenario 1: Catalog WITH Warehouse (Recommended)

Pangolin vends credentials automatically.

#### PyIceberg

```
from pyiceberg.catalog import load_catalog  
  
# Pangolin vends GCS credentials automatically  
catalog = load_catalog(  
    "pangolin",  
    **{  
        "uri": "http://localhost:8080/api/v1/catalogs/analytics",  
        "warehouse": "gs://my-iceberg-tables/analytics/"  
    }  
)  
  
# Create table - Pangolin handles GCS access  
from pyiceberg.schema import Schema  
from pyiceberg.types import NestedField, StringType, IntegerType  
  
schema = Schema(  
    NestedField(1, "id", IntegerType(), required=True),  
    NestedField(2, "name", StringType(), required=True)  
)  
  
table = catalog.create_table("db.users", schema=schema)  
  
# Write data  
import pyarrow as pa  
data = pa.table({"id": [1, 2, 3], "name": ["Alice", "Bob", "Charlie"]})  
table.append(data)  
  
# Read data  
df = table.scan().to_pandas()  
print(df)
```

#### PySpark

```

from pyspark.sql import SparkSession

# Pangolin vends credentials automatically
spark = SparkSession.builder \
    .appName("Pangolin Iceberg GCS") \
    .config("spark.sql.catalog.pangolin", "org.apache.iceberg.spark.SparkCatalog") \
    .config("spark.sql.catalog.pangolin.catalog-impl",
"org.apache.iceberg.rest.RESTCatalog") \
    .config("spark.sql.catalog.pangolin.uri",
"http://localhost:8080/api/v1/catalogs/analytics") \
    .config("spark.sql.catalog.pangolin.warehouse", "gs://my-iceberg-tables/analytics/") \
    .getOrCreate()

# Create table
spark.sql("""
    CREATE TABLE pangolin.db.users (
        id INT,
        name STRING
    ) USING iceberg
""")

# Write data
data = [(1, "Alice"), (2, "Bob"), (3, "Charlie")]
df = spark.createDataFrame(data, ["id", "name"])
df.writeTo("pangolin.db.users").append()

# Read data
spark.table("pangolin.db.users").show()

```

## Scenario 2: Catalog WITHOUT Warehouse

Clients configure GCS access themselves.

### Pylceberg

```

from pyiceberg.catalog import load_catalog
import os

# Set GCS credentials
os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = "/path/to/service-account-key.json"

catalog = load_catalog(
    "pangolin",
    **{
        "uri": "http://localhost:8080/api/v1/catalogs/analytics",
        "warehouse": "gs://my-iceberg-tables/analytics/",
        # Client-side GCS configuration
        "gcs.project-id": "my-gcp-project",
        "gcs.service-account-file": "/path/to/service-account-key.json"
    }
)

```

```
)

# Use catalog normally
table = catalog.load_table("db.users")
df = table.scan().to_pandas()
```

## PySpark

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("Pangolin Iceberg GCS") \
    .config("spark.sql.catalog.pangolin", "org.apache.iceberg.spark.SparkCatalog") \
    .config("spark.sql.catalog.pangolin.catalog-impl",
"org.apache.iceberg.rest.RESTCatalog") \
    .config("spark.sql.catalog.pangolin.uri",
"http://localhost:8080/api/v1/catalogs/analytics") \
    .config("spark.sql.catalog.pangolin.warehouse", "gs://my-iceberg-tables/analytics/") \
    .config("spark.hadoop.google.cloud.auth.service.account.enable", "true") \
    .config("spark.hadoop.google.cloud.auth.service.account.json.keyfile",
"/path/to/service-account-key.json") \
    .getOrCreate()

# Use Spark normally
spark.table("pangolin.db.users").show()
```

## GCS Best Practices

### Performance

**Use Regional buckets** for best performance

**Use Multi-regional** for global access

**Enable Turbo Replication** for faster replication

**Collocate compute and storage** in same region

### Security

**Use Workload Identity** (GKE) instead of service account keys

**Enable Uniform Bucket-Level Access**

**Use Customer-Managed Encryption Keys** (CMEK) if required

**Enable Audit Logging**

-

**Use IAM Conditions** for fine-grained access

## Cost Optimization

**Use Lifecycle Policies** to transition to Nearline/Coldline/Archive

**Enable Object Versioning** selectively

**Use Committed Use Discounts**

**Monitor with Cloud Billing**

## IAM Permissions

Service account needs these permissions:

```
{
  "roles": [
    "roles/storage.objectAdmin"
  ],
  "resources": [
    "projects/my-project/buckets/my-iceberg-tables"
  ]
}
```

Or custom role with specific permissions:

`storage.objects.create`

`storage.objects.delete`

`storage.objects.get`

`storage.objects.list`

`storage.buckets.get`

## Troubleshooting

### Access Denied

Error: 403 Forbidden

#### **Solutions:**

Check service account has correct permissions

Verify bucket name is correct

Check project ID

-

Ensure service account key is valid

## Connection Issues

### **Solutions:**

Verify bucket exists

Check network connectivity

Ensure firewall rules allow access

Check service account key format

## Additional Resources

[Google Cloud Storage Documentation](#)

[GCS Best Practices](#)

[Iceberg GCS Configuration](#)

## Next Steps

[S3 Warehouse](#)

[Azure Warehouse](#)

[Warehouse Concept](#)

## Source: local.md

## Local Filesystem Warehouse

Pangolin supports using the local filesystem for warehouse storage. This is ideal for development, testing, and single-node deployments where the Pangolin server and the compute engine (e.g., Spark, Pylceberg) run on the same machine or share a mounted volume.

## Configuration

To use a local filesystem warehouse, set the `type` to `local` (or omit specific config keys usually required for S3/Azure/GCS) and provide a path.

## API Example

```
curl -X POST http://localhost:8080/api/v1/warehouses \
-H "Authorization: Bearer <token>" \
-H "Content-Type: application/json" \
-d '{
```

```

    "name": "local-warehouse",
    "storage_config": {
      "type": "local",
      "path": "/tmp/pangolin-data"
    },
    "vending_strategy": {
      "type": "None"
    }
  }
}'

```

**Note:** The `path` must be accessible/writable by the Pangolin server process.

## Usage with Pylceberg

When using a local warehouse, Pylceberg needs to be able to access the files directly.

```

from pyiceberg.catalog import load_catalog

catalog = load_catalog(
    "local",
    **{
        "type": "rest",
        "uri": "http://localhost:8080/api/v1/catalogs/my_catalog/iceberg",
        "token": "...",
        # No specific storage credentials needed for local files
    }
)

```

## Credential Vending

The Local Filesystem warehouse type supports the `None` vending strategy. It does **not** vend temporary credentials because file access is managed by the operating system's file permissions.

## Limitations

**Scalability:** Limited by the local disk size and IO.

**Concurrency:** Local filesystems may not handle high concurrent write loads as well as object stores.

**Sharing:** Data is only accessible to processes that can read the specific directory. Not suitable for distributed compute clusters unless a shared network file system (NFS/EFS) is used.

Source: s3.md

## AWS S3 Warehouse Configuration

Configure AWS S3 as a warehouse for storing Iceberg table data in Pangolin.

## Overview

AWS S3 is the most common object storage for Iceberg tables, offering:

High durability (99.999999999%)

Excellent performance

Global availability

Rich ecosystem and tooling

Cost-effective storage tiers

## Warehouse Configuration

### Option 1: AwsStatic - Static Credentials (Development)

```
curl -X POST http://localhost:8080/api/v1/warehouses \
-H "X-Pangolin-Tenant: my-tenant" \
-H "Content-Type: application/json" \
-d '{
  "name": "dev-s3",
  "storage_config": {
    "bucket": "my-dev-datalake",
    "region": "us-east-1",
    "s3.access-key-id": "AKIAIOSFODNN7EXAMPLE",
    "s3.secret-access-key": "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
  },
  "vending_strategy": "AwsStatic"
}'
```

#### Use When:

Local development

Testing

Simple deployments

**Security Note:** Static credentials don't expire. Use AwsSts for production.

### Option 2: AwsSts - IAM Role with STS (Production - Recommended)

```
curl -X POST http://localhost:8080/api/v1/warehouses \
-H "X-Pangolin-Tenant: my-tenant" \
-H "Content-Type: application/json" \
-d '{
  "name": "prod-s3",
```

```

"storage_config": {
  "bucket": "my-prod-datalake",
  "region": "us-east-1",
  "s3.role-arn": "arn:aws:iam::123456789012:role/PangolinDataAccess"
},
"vending_strategy": {
  "AwsSts": {
    "role_arn": "arn:aws:iam::123456789012:role/PangolinDataAccess"
  }
}
}'

```

## Use When:

Production deployments  
 Need temporary credentials  
 Want fine-grained permissions

## IAM Role Setup

### Create IAM Role:

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::123456789012:role/PangolinServerRole"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}

```

### Attach Policy:

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
        "s3:PutObject",
        "s3:DeleteObject",
        "s3:ListBucket"
      ],
      "Resource": [
        "arn:aws:s3::my-prod-datalake/*",

```

```

        "arn:aws:s3:::my-prod-datalake"
    ]
}
]
}

```

### Grant Pangolin Server Permission to Assume Role:

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "sts:AssumeRole",
      "Resource": "arn:aws:iam::123456789012:role/PangolinDataAccess"
    }
  ]
}

```

## Client Configuration

### Scenario 1: Catalog WITH Warehouse (Recommended)

Pangolin automatically vends credentials to clients. No client-side S3 configuration needed!

### Pylceberg

```

from pyiceberg.catalog import load_catalog

# Pangolin vends credentials automatically via X-Iceberg-Access-Delegation header
catalog = load_catalog(
    "pangolin",
    **{
        "uri": "http://localhost:8080/api/v1/catalogs/analytics",
        "warehouse": "s3://my-prod-datalake/analytics/"
    }
)

# Create namespace
catalog.create_namespace("db")

# Create table - Pangolin handles S3 access
from pyiceberg.schema import Schema
from pyiceberg.types import NestedField, StringType, IntegerType

schema = Schema(
    NestedField(1, "id", IntegerType(), required=True),
    NestedField(2, "name", StringType(), required=True)
)

```

```

)

table = catalog.create_table(
    "db.users",
    schema=schema
)

# Write data - credentials vended automatically
import pyarrow as pa

data = pa.table({
    "id": [1, 2, 3],
    "name": ["Alice", "Bob", "Charlie"]
})

table.append(data)

# Read data
df = table.scan().to_pandas()
print(df)

```

## PySpark

```

from pyspark.sql import SparkSession

# Pangolin vends credentials automatically
spark = SparkSession.builder \
    .appName("Pangolin Iceberg") \
    .config("spark.sql.catalog.pangolin", "org.apache.iceberg.spark.SparkCatalog") \
    .config("spark.sql.catalog.pangolin.catalog-impl",
"org.apache.iceberg.rest.RESTCatalog") \
    .config("spark.sql.catalog.pangolin.uri",
"http://localhost:8080/api/v1/catalogs/analytics") \
    .config("spark.sql.catalog.pangolin.warehouse", "s3://my-prod-datalake/analytics/") \
    .getOrCreate()

# Create table
spark.sql("""
    CREATE TABLE pangolin.db.users (
        id INT,
        name STRING
    ) USING iceberg
""")

# Write data
data = [(1, "Alice"), (2, "Bob"), (3, "Charlie")]
df = spark.createDataFrame(data, ["id", "name"])
df.writeTo("pangolin.db.users").append()

# Read data
spark.table("pangolin.db.users").show()

```

## Scenario 2: Catalog WITHOUT Warehouse

Clients must configure S3 access themselves.

### Pylceberg

```
from pyiceberg.catalog import load_catalog
import os

# Set AWS credentials
os.environ["AWS_ACCESS_KEY_ID"] = "AKIAIOSFODNN7EXAMPLE"
os.environ["AWS_SECRET_ACCESS_KEY"] = "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
os.environ["AWS_REGION"] = "us-east-1"

catalog = load_catalog(
    "pangolin",
    **{
        "uri": "http://localhost:8080/api/v1/catalogs/analytics",
        "warehouse": "s3://my-prod-datalake/analytics/",
        # Client-side S3 configuration
        "s3.region": "us-east-1",
        "s3.access-key-id": "AKIAIOSFODNN7EXAMPLE",
        "s3.secret-access-key": "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
    }
)

# Use catalog normally
table = catalog.load_table("db.users")
df = table.scan().to_pandas()
```

### PySpark

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("Pangolin Iceberg") \
    .config("spark.sql.catalog.pangolin", "org.apache.iceberg.spark.SparkCatalog") \
    .config("spark.sql.catalog.pangolin.catalog-impl", "org.apache.iceberg.rest.RESTCatalog") \
    .config("spark.sql.catalog.pangolin.uri", "http://localhost:8080/api/v1/catalogs/analytics") \
    .config("spark.sql.catalog.pangolin.warehouse", "s3://my-prod-datalake/analytics/") \
    .config("spark.hadoop.fs.s3a.access.key", "AKIAIOSFODNN7EXAMPLE") \
    .config("spark.hadoop.fs.s3a.secret.key", "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY") \
    .config("spark.hadoop.fs.s3a.endpoint.region", "us-east-1") \
    .getOrCreate()

# Use Spark normally
spark.table("pangolin.db.users").show()
```

# S3 Best Practices

## Performance

**Use S3 Transfer Acceleration** for cross-region access

**Enable S3 Intelligent-Tiering** for cost optimization

**Use Requester Pays** for shared datasets

**Collocate compute and storage** in same region

## Security

**Enable S3 Bucket Encryption** (SSE-S3 or SSE-KMS)

**Use VPC Endpoints** to avoid internet traffic

**Enable S3 Access Logging** for audit trails

**Use Bucket Policies** to restrict access

**Enable Versioning** for data protection

## Cost Optimization

**Use Lifecycle Policies** to transition old data to Glacier

**Enable S3 Intelligent-Tiering** for automatic cost optimization

**Use S3 Storage Lens** to analyze usage

**Compress Parquet files** with Snappy or Zstd

## Organization

**Bucket Structure:** `s3://bucket/{catalog}/{namespace}/{table}/`

**Naming Convention:** Use descriptive, hierarchical names

**Separate Buckets:** Use different buckets for dev/staging/prod

**Cross-Region Replication:** For disaster recovery

## Troubleshooting

### Access Denied

Error: Access Denied (Service: Amazon S3; Status Code: 403)

**Solutions:**

Check IAM role permissions

Verify bucket policy

Check STS assume role permissions

Ensure bucket exists and is in correct region

## Slow Performance

**Solutions:**

Check region - ensure compute and S3 are colocated

Enable S3 Transfer Acceleration

Use larger instance types

Check network bandwidth

Use S3 Select for filtering

## Credential Vending Not Working

Error: No credentials provided

**Solutions:**

Ensure catalog has warehouse attached

Check warehouse `use\_sts` setting

Verify IAM role ARN is correct

Check Pangolin server has permission to assume role

## Monitoring

### CloudWatch Metrics

Monitor these S3 metrics:

`BucketSizeBytes` - Total bucket size

`NumberOfObjects` - Object count

`AllRequests` - Request rate

`4xxErrors` - Client errors

`5xxErrors` - Server errors

# CloudTrail

Enable CloudTrail to audit S3 access:

```
aws cloudtrail create-trail \  
  --name pangolin-s3-audit \  
  --s3-bucket-name my-audit-bucket
```

## Additional Resources

[AWS S3 Documentation](#)

[S3 Best Practices](#)

[S3 Performance Optimization](#)

[Iceberg S3 Configuration](#)

## Next Steps

[Azure Blob Warehouse](#)

[GCS Warehouse](#)

[Warehouse Concept](#)

[Backend Storage](#)