

# Git, GitHub, Python & Datalake Tooling

A Beginner's Guide

---

Texas A&M Transportation Institute

2026-02-10

# Before We Start: Installation Checklist

## Required Tools

1. **Git** - Version control
  - Download: <https://git-scm.com>
2. **GitHub Account**
  - Sign up: <https://github.com>
3. **UV** - Package manager
  - Install: [UV installation guide](#)
5. **VS Code** (Recommended)
  - Download: <https://code.visualstudio.com>
6. **cookie cutter** - Code template
  - user guide: <https://github.com/cookiecutter/cookiecutter>

## Verify Installation

Open terminal and run:

```
git --version
python --version
uv --version
```

**Python Packages** (install during workshop):

```
uv add duckdb
uv add ruff --dev
uv add vulture --dev
```

## TTI Tools:

- datalake-sync (will be provided)

# Why Version Control?

## The Problem

- “Which version is the latest?”
- “Who changed this and why?”
- “I broke something, how do I go back?”
- “How do we work on the same file?”
- `report_final_v2_FINAL_revised.docx`

## Without Git:

- Manual backups everywhere
- Emailing files back and forth
- Overwriting each other's work
- No history of changes

## The Solution: Git

- Complete history of every change
- Know who changed what and when
- Revert to any previous version
- Work in parallel without conflicts
- One source of truth

## Benefits:

- Reproducible research
- Safe experimentation (branches)
- Seamless collaboration
- Backup in the cloud (GitHub)

# What is Git?

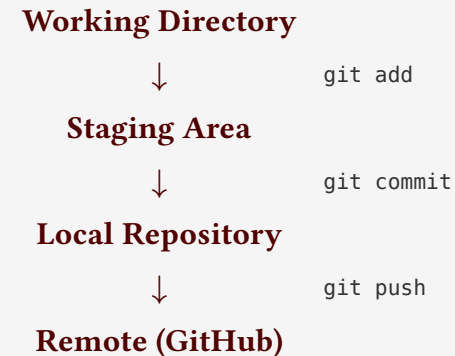
**Git** is a distributed version control system that tracks changes in your code.

## Why use it?

- Track every change you make
- Collaborate with others without conflicts
- Revert to previous versions anytime
- Work on features in isolation (branches)

```
git init          # Start a new repository
git clone <url>   # Clone a repository
git status        # Check what's changed
```

### How Git Works



# What is Git?

**Git** is a distributed version control system that tracks changes in your code.

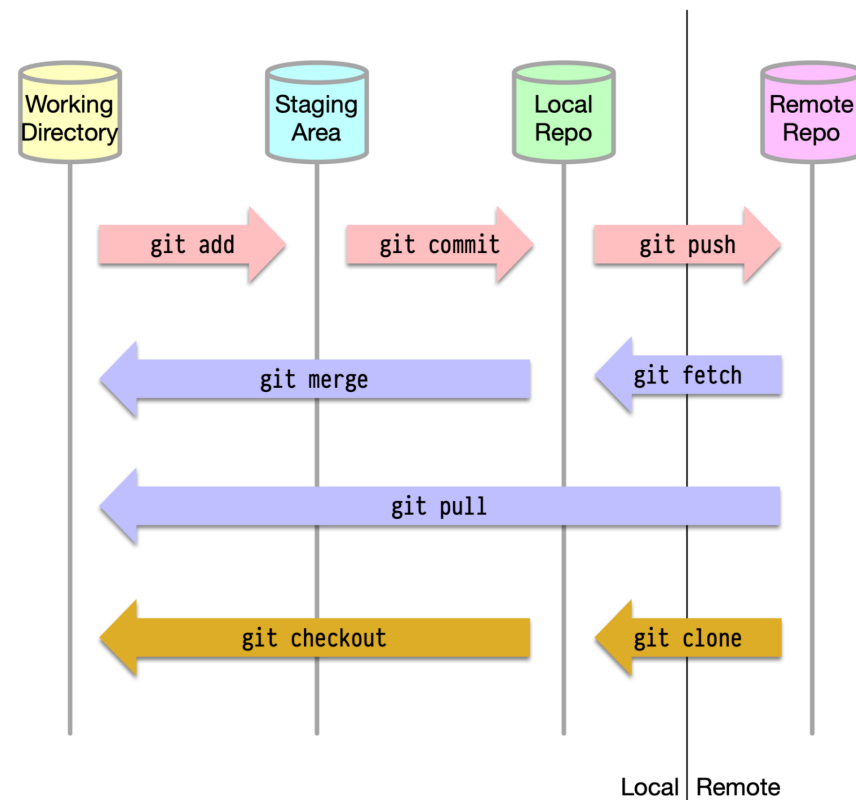
## Why use it?

- Track every change you make
- Collaborate with others without conflicts
- Revert to previous versions anytime
- Work on features in isolation (branches)

```
git init           # Start a new repository
git clone <url>    # Clone a repository
git status         # Check what's changed
```

## How Git Commands work

ByteByteGo.com



# Git Workflow & Branching

## Basic Workflow

# Stage your changes

```
git add .
```

# Commit with a message

```
git commit -m "Add new feature"
```

# Push to remote

```
git push
```

# Pull latest changes

```
git pull
```

## Branching

# Create a new branch

```
git branch feature-name
```

# Switch to a branch

```
git checkout feature-name
```

# Create & switch (shortcut)

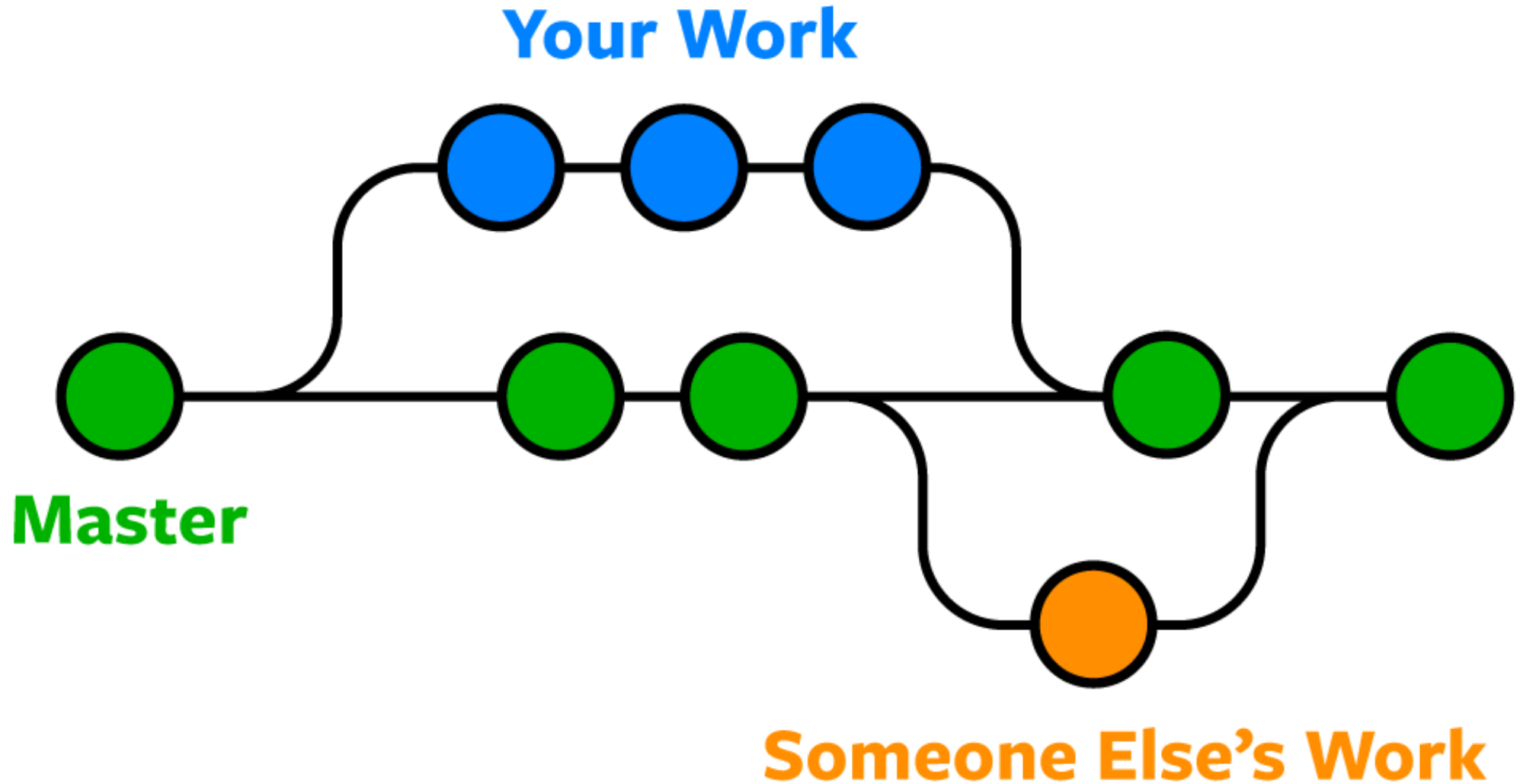
```
git checkout -b feature-name
```

# Merge branch into main

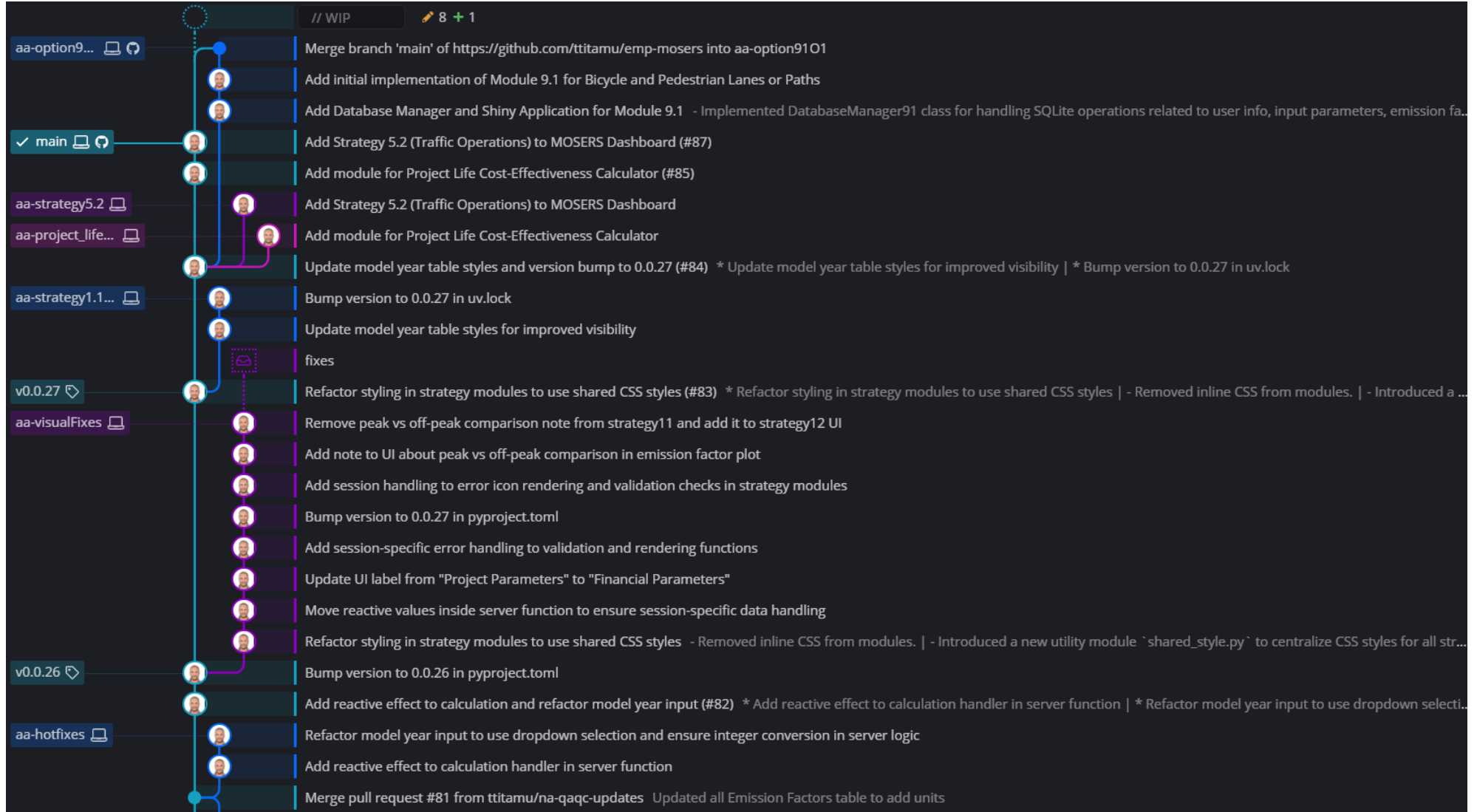
```
git checkout main
```

```
git merge feature-name
```

**Branching Flow:** main → feature-branch → *work on feature* → merge back to main



# Git Workflow & Branching





# Handling Merge Conflicts

## What Are Conflicts?

Occur when Git can't auto-merge changes:

- Two people edit the same line
- File deleted in one branch, modified in another

## Conflict Markers:

```
<<<<<< HEAD
```

```
Your changes here
```

```
=====
```

```
Their changes here
```

```
>>>>>> branch-name
```

## How to Resolve

1. Open the conflicted file
2. Choose which code to keep
3. Delete conflict markers
4. Stage the resolved file
5. Complete the commit

```
git add resolved-file.py
```

```
git commit -m "Resolve conflicts"
```

**Pro tip:** VS Code has a built-in merge editor!

## Setting Up

1. Create a repository on GitHub
2. Connect your local repo:

```
# Add remote origin
```

```
git remote add origin <url>
```

```
# Push for the first time
```

```
git push -u origin main
```

**The -u flag** sets upstream tracking, so future pushes just need `git push`.

## Typical Workflow

1. **Clone** the repo
2. Create a **branch**
3. Make changes
4. **Commit** your work
5. **Push** to GitHub
6. Open a **Pull Request**
7. Review & **Merge**

# GitHub: Creating a New Repository

## Create a new repository

Repositories contain a project's files and version history. Have a project elsewhere? [Import a repository](#).  
Required fields are marked with an asterisk (\*).

1

### General

Owner \*

Choose an owner ▾

Repository name \*

Great repository names are short and memorable. How about [cautious-octo-funicular?](#)

Description

0 / 350 characters

2

### Configuration

Choose visibility \*

Choose who can see and commit to this repository

Public ▾

Start with a template

Templates pre-configure your repository with files.

No template ▾

Add README

READMEs can be used as longer descriptions. [About READMEs](#)

Off

☐

Add .gitignore

.gitignore tells git which files not to track. [About ignoring files](#)

No .gitignore ▾

Add license

Licenses explain how others can use your code. [About licenses](#)

No license ▾

Create repository

# Python Tooling: UV, Ruff & Vulture

## UV

*Fast Python package manager*

# Create new project

```
uv init .
```

# Add a package

```
uv add pandas
```

# Run a script

```
uv run main.py
```

# Sync dependencies

```
uv sync
```

## Ruff

*Lightning-fast linter & formatter*

# Lint your code

```
ruff check .
```

# Auto-fix issues

```
ruff check --fix .
```

# Format code

```
ruff format .
```

Replaces flake8, black, isort!

## Vulture

*Find dead/unused code*

# Scan for dead code

```
vulture .
```

# Set confidence level

```
vulture --min-confidence 80 .
```

# Check specific files

```
vulture src/
```

Clean up your codebase!

# Why DuckDB?

## Traditional Data Analysis Pain Points

- Loading large CSVs crashes Python
- Pandas is slow on big datasets
- Need a database server for SQL
- Converting between formats is tedious
- Memory limitations

## Common Workarounds:

- Chunk processing (complex code)
- Spin up PostgreSQL/MySQL (overkill)
- Use Spark (heavy setup)

## DuckDB Solves This

- Query files directly (no loading)
- Blazing fast on large data
- Zero setup, just `import duckdb`
- SQL on DataFrames instantly
- Handles larger-than-memory data

## Perfect For:

- Data exploration and analysis
- ETL pipelines
- Quick aggregations on big files
- Replacing slow Pandas operations

# DuckDB: In-Process SQL Database

## What is DuckDB?

- Fast, in-process SQL database
- No server setup required
- Reads files directly (CSV, Parquet, JSON)
- Perfect for data analysis

## Basic Usage

```
import duckdb
```

```
# In-memory database  
con = duckdb.connect()
```

```
# Persistent database  
con = duckdb.connect("my.db")
```

## Read Files Directly

```
# CSV  
duckdb.sql("SELECT * FROM 'data.csv'")
```

```
# Parquet  
duckdb.sql("SELECT * FROM  
'data.parquet'")
```

```
# JSON  
duckdb.sql("SELECT * FROM  
'data.json'")
```

```
# Multiple files with glob  
duckdb.sql("SELECT * FROM 'data/  
*.csv'")
```

# DuckDB: Pandas Integration

## Query DataFrames with SQL

```
import pandas as pd
import duckdb

df = pd.read_csv("sales.csv")

# Query DataFrame directly
result = duckdb.sql("""
    SELECT region, SUM(amount)
    FROM df
    GROUP BY region
""")
```

DuckDB auto-detects DataFrames by name!

## Convert Results

```
# To DataFrame
result.df()
```

```
# To Polars
result.pl()
```

```
# To Arrow
result.arrow()
```

```
# To Python list
result.fetchall()
```

## Useful Operations

```
# Describe table
duckdb.sql("DESCRIBE df")
```

# Why datalake-sync?

## Data Lake Challenges

- Manual file transfers to Azure
- Duplicate shared datasets across projects
- Complex format conversions
- Missing metadata and documentation

## Common Workarounds:

- Manually copy files to/from Azure
- Store same dataset in multiple places
- Manual format conversions (error-prone)
- Share data between team members

## datalake-sync Solution

- Git-like workflow for data
- Automatic sync with Azure ADLS Gen2
- Copy-by-reference (no duplication)
- Built-in format conversion (DuckDB)

## Perfect For:

- Data tracking and reproducibility
- Team collaboration on datasets
- Shared data management
- Quick data format conversions
- Backup in the cloud (Azure)



# datalake-sync: Core Features

## Key Features

- Bidirectional Azure sync
- Shared data references (manifest-based)
- Format conversion (DuckDB-powered)
- Metadata validation
- Project initialization and management

## Basic Commands

```
# Initialize project
```

```
datalake init . -c CLIENT \  
  -y YEAR -p PROJECT_ID
```

```
datalake sync      # Sync with Azure
```

```
datalake status   # Check status
```

## Shared Data Workflow

### Typical Workflow

1. Initialize project
2. Add shared data references
3. Fetch shared data to local
4. Work with local files
5. Sync project data changes to Azure
6. Convert formats as needed

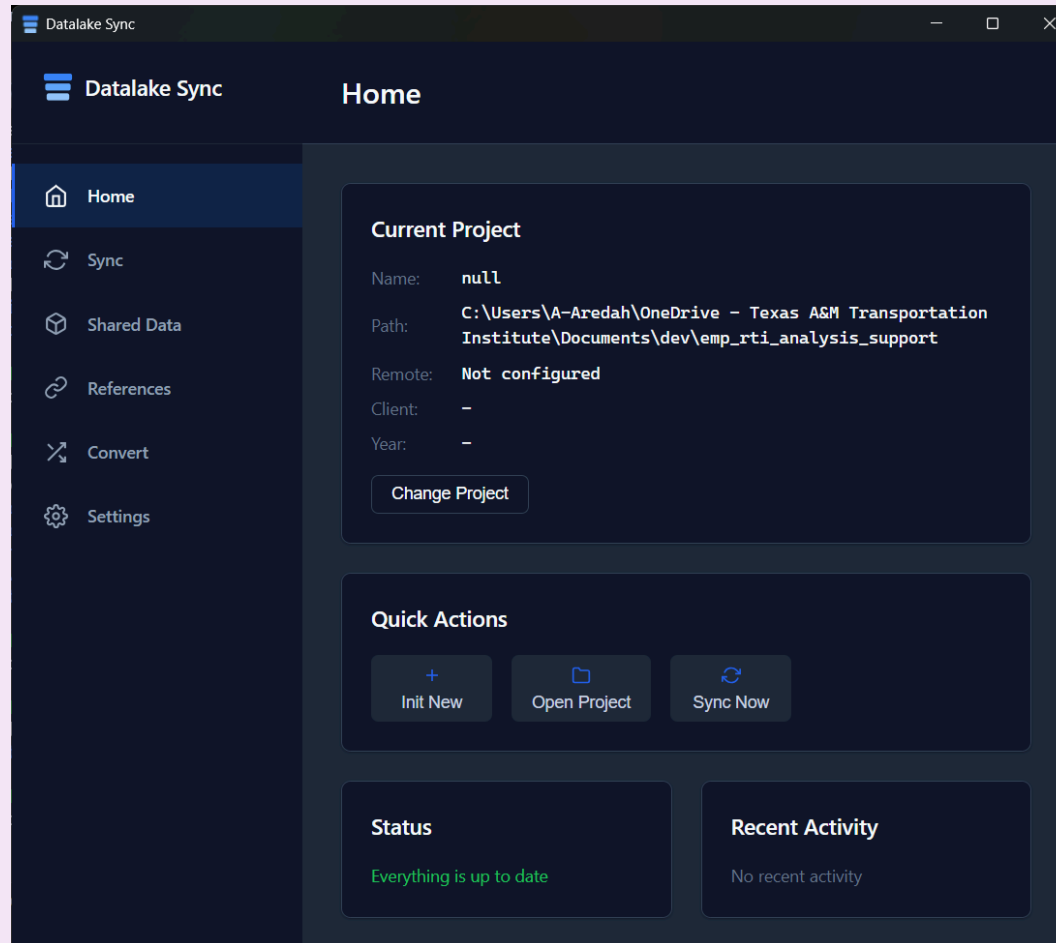
### Example Commands

```
# Add shared reference  
datalake refs add shared/inputs/census
```

```
# Fetch all shared data  
datalake shared fetch --all
```

```
# Convert to Parquet  
datalake convert file data.csv \  
  --format parquet
```

# datalake-gui: Visual Interface



# Quick Reference

## Git Commands

<code>git init</code>	Start repo
<code>git clone &lt;url&gt;</code>	Copy repo
<code>git status</code>	Check changes
<code>git add .</code>	Stage all
<code>git commit -m ""</code>	Commit
<code>git push</code>	Upload
<code>git pull</code>	Download
<code>git branch</code>	Create branch
<code>git checkout</code>	Switch branch
<code>git merge</code>	Merge branch

## Python Tools

<code>uv init .</code>	Create project
<code>uv add &lt;pkg&gt;</code>	Add package
<code>uv run &lt;file&gt;</code>	Run script
<code>uv sync</code>	Sync deps
<code>ruff check .</code>	Lint code
<code>ruff format .</code>	Format code
<code>vulture .</code>	Dead code

## DuckDB

<code>duckdb.connect()</code>	Connect
<code>duckdb.sql()</code>	Run query
<code>FROM 'file.csv'</code>	Read CSV
<code>FROM 'file.parquet'</code>	Read Parquet
<code>FROM df</code>	Query DataFrame
<code>.df()</code>	To Pandas
<code>COPY ... TO</code>	Export file

## datalake-sync

<code>datalake init</code>	Create project
<code>datalake sync</code>	Sync data
<code>datalake status</code>	Check status
<code>datalake refs add</code>	Add reference
<code>datalake shared fetch</code>	Fetch data
<code>datalake convert</code>	Convert format
<code>datalake config</code>	Configure

# Hands-On Exercises Overview

## What We'll Practice Today

- |   |        |
|---|--------|
| 1. <b>Git Basics:</b> Create repo, commit, push to GitHub     | 20 min |
| 2. <b>Python Environment:</b> Set up UV project, add packages | 15 min |
| 3. <b>Code Quality:</b> Run Ruff and Vulture on code          | 10 min |
| <i>Break</i>  | 5 min  |
| 4. <b>DuckDB:</b> Query CSV files, convert to Parquet         | 25 min |
| 5. <b>datalake-sync:</b> Initialize project, sync with Azure  | 25 min |

Reference guides available in notes/ folder for detailed commands!

# Common Issues & Quick Fixes

## Git Issues

Error: fatal: not a git repository

**Fix:** Run `git init` in your project folder

Error: Authentication failed

**Fix:** Set up GitHub personal access token or SSH key

Error: UV: command not found

**Fix:** Restart terminal or check PATH

## Python/Package Issues

Error: ModuleNotFoundError: No module named 'duckdb'

**Fix:** `uv add duckdb`

Error: ruff: command not found

**Fix:** `uv add ruff --dev` or use `uv run ruff`

## Reference Guides

- `notes/git.md` - Git commands
- `notes/uv.md` - UV commands
- `notes/ruff.md` - Ruff usage
- `notes/vulture.md` - Vulture usage
- `notes/duckdb.md` - DuckDB CLI
- `notes/duckdb_python.md` - DuckDB Python
- `notes/datalake-sync.md` - TTI data tool

## Practice & Learning

### Next Steps:

1. Practice Git daily with your projects
2. Use UV for all new Python projects
3. Run Ruff before every commit
4. Try DuckDB for data analysis tasks
5. Use datalake-sync for TTI data

**Questions? Let's discuss!**