Using a Monorepo for Data Science Projects on GitHub

Table of Contents

What is a Monorepo and Why Use One in Data Science?

A **monorepo** (monolithic repository) is a single Git repository that contains multiple distinct projects or codebases within it. This is in contrast to a **multirepo**, where each project or service lives in its own separate repository. For example, a data science team might have a monorepo that includes multiple Python projects (e.g. a machine learning model, an ETL pipeline, and a data analysis app) all in one GitHub repo . Monorepos have been popularized by large tech companies (Google, Facebook, etc.) for managing large codebases, but their adoption in data science is growing as well .

Why use a monorepo for data science? There are several compelling reasons:

- **Centralized codebase and collaboration:** All data science projects live together, making it easier for teams to share code and collaborate. Team members can work on different parts of the same project or across projects in the same repository, which can improve communication and knowledge sharing . In a multirepo, you might have to switch between repositories or coordinate across multiple repos to make changes that affect several projects.
- **Code reuse and standardization:** A monorepo allows you to reuse components (like common data loading functions, data schemas, or utility libraries) across projects. This reduces duplication of code and boilerplate. For instance, a data engineering team might consolidate code for data processing pipelines,

ensuring new features are easily accessible to all projects . It also makes it easier to enforce consistent coding standards, testing frameworks, and tooling across all projects .

- **Streamlined workflows and consistency:** With all data science projects in one repo, you can maintain a unified set of workflows and CI/CD pipelines. This means **one build system and one dependency management strategy** for the entire codebase, simplifying the onboarding of new team members . Teams can share tooling and configuration, and changes to the tooling (like updating a linter or adding a new test) can be applied once and propagate to all projects. This consistency is especially important as projects evolve together.
- **Atomic changes and deployments:** A monorepo lets you make changes that span multiple projects and deploy them together atomically. For example, if you fix a bug in a data pipeline and also update a model that uses it, you can commit both changes in one PR and ensure they are deployed in sync. This can be hard to manage in a multirepo, where you might have to deploy multiple repos sequentially . In short, a monorepo enables coordinated changes and releases across your data science stack.
- **Improved CI/CD efficiency:** A monorepo can simplify continuous integration (CI) setups. Many CI systems can be configured to run jobs only on files that change (path filters), so you can trigger tests and builds only for the projects affected by a change . Additionally, caching dependencies and build artifacts can be more effective in a monorepo because the same cache can be used for multiple projects . This can speed up CI runs. GitHub Actions, for instance, supports monorepos by allowing workflows to run jobs only when specific paths change .

Overall, the idea is that a monorepo can **unify your data science projects**, making them easier to manage, collaborate on, and evolve together. Many data engineering teams have reported benefits like faster onboarding, reduced code duplication, and more coordinated development by adopting a monorepo . However, it's important to weigh these benefits against the challenges, which we'll discuss next.

Benefits of a Monorepo for Data Science

Using a monorepo in a data science context offers several concrete benefits:

- **Improved Collaboration:** All projects and code live in one place, reducing friction between teams and enabling easier cross-team collaboration. Developers can more easily understand how different parts of the data pipeline (e.g. ETL, model, app) interact. If a data scientist is working on a model and a data engineer is working on a pipeline, they can collaborate within the same repository without switching contexts or dealing with multiple repos . Code sharing is encouraged since components are right there.
- **Code Reuse and Reduced Duplication:** A monorepo allows you to share code across projects. Common data processing functions, schema definitions, or helper modules can be placed in one location and used by all projects. This dramatically reduces code duplication. For example, if multiple models use similar data loading logic, that logic can be factored into a shared module in the monorepo. This improves maintainability and makes it easier to update shared code once . It also ensures consistency – all projects can use the same set of libraries or versions of code.

- **Unified Workflow and Standardization:** Teams can enforce consistent tooling and practices across the board. All projects can use the same testing framework, linters, and data validation tools. You can have a single `Makefile` or `requirements.txt` that all projects follow, making it simpler to manage dependencies. As one data team noted, consolidating tooling in a monorepo made it easier to introduce improvements and updates . This consistency extends to development practices and documentation as well, since everyone is working in the same repository.
- **One Repository for All Artifacts:** In a monorepo, all project artifacts (code, notebooks, datasets, etc.) are in one place. This means you can track the full context of a project's development in a single version history. It also means you can version-control your data and configuration files alongside your code, which can be important for reproducibility. Many data science projects now include data files or configuration in the repo (with proper size considerations). A monorepo makes this more straightforward.
- **Streamlined CI/CD and Deployment:** With all projects in one repo, you can set up **one CI/CD pipeline** that covers multiple projects. For instance, you might have a workflow that runs tests for all projects on every commit, or deploys all projects when they are updated. This can reduce the complexity of managing multiple pipelines. Moreover, CI systems can be optimized for monorepos – you can use path filters to only run tests for the changed projects, and use caching across jobs for common dependencies . In practice, this leads to faster builds and more efficient use of CI resources . GitHub Actions, for example, can be configured to trigger workflows only when specific files change . This can greatly speed up CI for large projects.
- **Atomic Changes and Releases:** Because all code is in one repo, you can make changes that span multiple projects and deploy them together. If, say, you fix a bug in a data pipeline and also update a model that depends on it, you can commit both changes in a single pull request and ensure they are deployed in sync. This atomicity ensures that you don't have mismatched versions of related components (a common problem in multirepos where one repo might lag behind another). It also enables coordinated releases – you can version and release multiple projects together under a single version number, ensuring everything is consistent .
- **Enhanced Developer Experience:** A monorepo can simplify the developer experience by providing a single entry point for most tasks. Data scientists and engineers don't have to switch between different repos; everything is in one place. This reduces context switching and cognitive load. It also means onboarding new team members is easier – they can clone one repo and immediately access all projects and their dependencies. Many data science teams have reported improved onboarding and reduced learning curve when moving to a monorepo .

In summary, a well-structured monorepo can **improve collaboration, code reuse, and consistency** for data science projects, while also simplifying CI/CD and deployment. These benefits have led many teams to adopt monorepos as a way to scale their data science efforts more effectively.

Challenges of a Monorepo in Data Science

While monorepos offer significant advantages, they also come with challenges, especially in a data science context:

- **Large Repository Size and Complexity:** A monorepo can grow very large, containing many projects, large datasets, and extensive commit history. This can slow down operations like cloning, checking out, and diffing the repository. Git performance can degrade as the repository grows (more commits, more history) . For example, a single commit that touches many files or projects might take longer to process. Some teams find that their monorepo becomes unwieldy – developers struggle to search through or understand the entire codebase, and it's harder to work on only a subset of the code . To mitigate this, teams must implement strategies like sparse checkout or efficient build tools (discussed later) to keep performance manageable.
- **Performance and CI Scalability:** With many projects in one repo, CI pipelines can become very slow if not optimized. If every pull request triggers a full build of all projects and their dependencies, it can take a long time. GitHub, for instance, had to invest in improvements (Project Cyclops) to handle extremely large monorepos with thousands of contributors, driving push failure rates near zero by optimizing how large repositories are maintained . Teams need to set up efficient CI workflows – using path filters to only run jobs on changed projects, parallelizing tests across projects, and caching dependencies . Without these optimizations, a monorepo's CI can become a bottleneck, especially as the number of projects and code increases.
- **Complex Dependency Management:** Managing dependencies across multiple projects can be tricky in a monorepo. If different projects have different requirements or versions of libraries, you need a strategy to handle this. In a multirepo, each project can manage its own dependencies, but in a monorepo you might end up with conflicting versions or shared dependencies that are hard to update. One approach is to use a single dependency lockfile for the entire repo (common in JavaScript monorepos with tools like Yarn or npm workspaces) , but in Python this is less common. Another challenge is handling **private dependencies** – if some projects depend on internal libraries or packages, you need to ensure those are accessible. Tools like `uv` (discussed later) can help manage this by allowing you to specify local paths or private package sources in a unified way . Overall, dependency management in a monorepo requires careful coordination to avoid version mismatches and ensure all projects have the correct environment.
- **Isolation and Testing:** Ensuring that changes to one project don't accidentally break another can be harder in a monorepo. Because all projects share the same repository, a bug in one part of the code could affect others that use that code. For example, a change in a data processing module might break multiple models that use it. This means you need robust testing and a clear understanding of dependencies. Teams often need to run comprehensive tests that cover all projects when major changes are made, which can slow down development. On the other hand, some testing can be optimized by running only the tests for the specific project or package that changed. GitHub Actions can help by identifying changed files and only running relevant jobs , but it's important to set this up correctly. Additionally, isolating work can be challenging – if one team is working on a project, they might inadvertently break another project if they don't have proper isolation. Some monorepo tools provide ways to partition work (e.g. feature branches for specific projects within the monorepo), but this requires discipline.
- **Access Control and Permissions:** With a single repo, controlling access to different parts of the code can be more complex. In a multirepo, you can easily restrict access to a particular repository. In a monorepo, you can't easily restrict access to just a subdirectory. This means that if certain projects are sensitive or handled by specific teams, everyone with access to the monorepo can see all projects. Some teams address this by using code ownership or GitHub's CODEOWNERS feature to ensure that only relevant people review or merge

changes in certain directories , but ultimately, a monorepo requires a high level of trust and governance. You must be careful to not accidentally expose sensitive data or code in the monorepo. Another aspect is permissions for CI: if different projects in the monorepo need different permissions (e.g. one project deploying to a staging environment and another to production), you need to manage secrets and permissions carefully.

- **Learning Curve and Tooling:** Adopting a monorepo can require learning new tools and practices. Teams that are used to separate repos might find it hard to navigate a large monorepo. There can be a learning curve for developers to understand the structure and find where different components live. Additionally, some monorepo management tools (like Nx, Turborepo, etc.) can be complex and require configuration. While `uv` aims to be simple for Python, it's still a new tool in the Python ecosystem, so there might be a learning curve for those not familiar with it. Teams need to invest time in training and setup to effectively use a monorepo. It's also worth noting that not all data science workflows are well-suited to monorepos – if you have very disparate projects that don't interact much, a monorepo might be overkill and introduce unnecessary complexity .
- **Large-Scale Maintenance and Reputation:** As a monorepo grows, maintaining it can become a full-time job. You might need to do things like garbage collection of old data, or periodic refactors to keep the repo manageable. There's also a reputational risk – a single bug in a huge monorepo can have far-reaching consequences. Some developers worry that monorepos encourage tight coupling and can be harder to manage at scale, potentially leading to slower development cycles . It's important to monitor the health of the monorepo and consider splitting it into smaller repos if it becomes too big or unwieldy .

Despite these challenges, many teams have successfully navigated them by implementing best practices (discussed later) and using appropriate tooling. The key is to be aware of these issues and plan accordingly.

Setting Up a Monorepo in GitHub for Data Science

Setting up a monorepo on GitHub is straightforward – it's just a matter of creating a new repository and pushing all your data science projects into it. However, to manage it effectively, you should follow some organizational guidelines:

1. **Create the Repository:** Start by creating a new GitHub repository (you might name it something like `data-science-monorepo`). Initialize it with a README if desired, but you can add it later.

2. **Add Your Projects:** Add all your data science projects as subdirectories in the repo. For example, you might have directories like `model_1/`, `etl_pipeline/`, `data_analysis/`, etc., each containing the code for that project. It's a good idea to have a clear naming convention for these directories (e.g. all project directories start with `project_` or `service_` to distinguish them).

3. **Add Metadata and Documentation:** Include a `README.md` in the root of the repo that documents the structure and purpose of the monorepo. Explain what each subdirectory contains and any dependencies or setup instructions. You might also have a `CONTRIBUTING.md` file that outlines how to contribute to the monorepo (e.g. branching strategies, code review guidelines). Having a well-organized README will help new team members get up to speed quickly.

4. **Initialize Git Submodules (Optional):** If you have external projects or libraries that you want to include but don't want to duplicate in the monorepo, you can use Git submodules. For instance, if you have a common data processing library that's developed in another repo, you can add it as a submodule. This allows you to keep a reference to that repo's code in your monorepo. However, submodules can be tricky to manage (they require extra commands to update, etc.), so use them sparingly and only if needed .

5. **Set Up Gitignore:** Create a `.gitignore` in the root to ignore common data science artifacts (like virtual environment directories, large data files, etc.). For example, you might ignore `*.pyc`, `__pycache__/`, `.venv/`, `data/` (if you're tracking data in the repo but want to exclude large data files), etc. This helps keep the repository clean and avoid committing unnecessary files.

6. **Initial Commit:** Commit all your files to the main branch. If you already had individual repos for each project, you can migrate them by either adding the existing project directories to the new monorepo or by deleting the old repos and moving the code. GitHub provides ways to move repositories or import code from other sources, but in practice, adding the code as subdirectories and then deleting the old repos works well.

Once your monorepo is set up on GitHub, you can start using it for collaboration. All team members can clone the repo and work on different projects within it. You should establish a branching strategy (like trunk-based development or feature branches) to manage concurrent work, but now everything is in one repo.

It's also a good idea to set up GitHub Actions for CI/CD right away. You can create a `.github/workflows` directory and start writing workflows that cover all your projects. We'll discuss workflows in more detail later, but for now, just know that you can have a single workflow that runs tests and deploys for all projects, or multiple workflows if you prefer.

Organizing Your Data Science Monorepo

An important part of a successful monorepo is a clear and consistent project structure. Here are some recommendations for organizing a data science monorepo:

- **Group Projects by Type or Domain:** Decide on a structure that makes sense for your team. A common approach is to group projects by their type or domain. For example:
- `data_pipelines/` – for ETL pipelines, data cleaning scripts, etc.
- `models/` – for machine learning models and training code
- `apps/` – for web applications, dashboards, or other front-end apps that use the data
- `tools/` – for utility libraries, shared modules, or helper scripts
- `notebooks/` – for Jupyter notebooks (though some prefer to keep notebooks in a separate directory and reference them from the code)
- **Use a Common Library Structure:** If you have a `tools/` or `shared/` directory, consider following a standard library structure for each project within it. For example, each library project might have a `src/` directory containing Python modules, a `tests/` directory for tests, and a `README.md` and `pyproject.toml` for metadata. This consistency makes it easier for developers to navigate any project in the monorepo.

- **Centralize Configuration and Requirements:** You might create a `config/` directory for shared configuration files (like environment variables or config for CI/CD), or a `requirements.txt` or `pyproject.toml` at the root if you have common dependencies. If you're using a monorepo tool like `uv`, you can use a `pyproject.toml` at the root to define workspaces . In general, try to avoid duplication of configuration. For example, if all projects use the same Python version or linter settings, keep those in one place.
- **Data and Artifacts:** If you are including data in the monorepo (which is not always recommended for large datasets due to Git size limits), organize it in a `data/` directory. You might further sub-divide by data source or type. However, be cautious with large data files – consider using Git LFS or external storage. Another approach is to keep data in separate repositories or in cloud storage and just reference it in the monorepo. In any case, have a clear way to handle data (versioning, cleaning scripts, etc.).
- **Documentation and Readme:** Each project should have a `README.md` that describes what it does, how to run it, and any dependencies. The root README should link to each project's README and explain the overal architecture. This documentation can be as simple as a table of contents linking to each project's directory.
- **Separate Staging Areas:** Some teams use a `staging/` or `work-in-progress/` directory for new projects or experimental code that isn't ready to be merged into the main projects. This keeps the main directories clean and organized.
- **Adopt a Consistent Naming:** Use consistent naming for directories, files, and even conventions for module names. For example, all data processing scripts could be in `data_pipelines/` and named with a `etl_*` prefix. This consistency helps everyone find things quickly.

Here's an example structure of a data science monorepo:

```
data-science-monorepo/
├── README.md
├── CONTRIBUTING.md
├── data_pipelines/
│   ├── etl_1/
│   │   ├── src/
│   │   │   └── etl_1/
│   │   │       ├── __init__.py
│   │   │       └── pipeline.py
│   │   ├── tests/
│   │   │   └── test_pipeline.py
│   │   ├── pyproject.toml
│   │   └── README.md
│   └── etl_2/
│       └── ...
├── models/
│   ├── model_1/
│   │   ├── src/
│   │   │   └── model_1/
│   │   │       ├── __init__.py
│   │   │       └── train.py
```

```
│   │   ├── tests/
│   │   │   └── test_model.py
│   │   ├── pyproject.toml
│   │   └── README.md
│   └── model_2/
│       └── ...
├── apps/
│   ├── dashboard/
│   │   ├── src/
│   │   │   └── dashboard/
│   │   │       └── app.py
│   │   ├── tests/
│   │   │   └── test_app.py
│   │   ├── pyproject.toml
│   │   └── README.md
│   └── api/
│       └── ...
├── tools/
│   ├── data_utils/
│   │   ├── src/
│   │   │   └── data_utils/
│   │   │       └── __init__.py
│   │   ├── tests/
│   │   │   └── test_data_utils.py
│   │   ├── pyproject.toml
│   │   └── README.md
│   └── plot_utils/
│       └── ...
└── .github/
    └── workflows/
        └── ci.yml
```

This structure separates projects into logical categories (data pipelines, models, apps, tools) and each project has its own directory with a `src` for code, `tests` for tests, and metadata files. The root contains high-level documentation and the CI workflow. This kind of structure is flexible – you can adjust it based on your team's needs. The key is that it's clear and consistent.

With a well-organized monorepo, developers can quickly find where to look for a particular piece of code. It also makes it easier to apply changes across projects (for example, if you add a new data validation step, you can add it in `tools/data_utils` and update all projects that use data utils to use the new function).

Monorepo Workflows in GitHub Actions

GitHub Actions is a powerful CI/CD tool that can be used to manage monorepos effectively. Here are some strategies and best practices for setting up workflows in a monorepo:

- **Use Path Filters to Trigger Workflows Only on Relevant Changes:** One of the biggest benefits of a monorepo is that you can avoid running the full CI suite for every pull request. GitHub Actions allows you to configure workflows to run only when certain files change. For example, you can have a workflow that runs tests only if files in the `models/` directory are modified . This can be done using the `on: [push]` or `on: [pull_request]` trigger with a `paths` filter. For instance:

```
on:
  push:
    paths:
      - 'models/**'
```

This will make the workflow run only when files in the `models/` directory (or any subdirectory) are pushed. You can have multiple such filters for different projects. This way, a change in a model project won't trigger tests for the data pipeline projects, and vice versa . It's a good practice to set up these path filters so that your CI remains fast even as the monorepo grows.

- **Matrix Builds for Parallel Testing:** If you do need to run tests for multiple projects, you can use a matrix strategy to parallelize the work. For example, if you have projects `projectA` , `projectB` , and `projectC` , you can create a matrix job that runs tests for each project in parallel:

```
jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        project: [projectA, projectB, projectC]
    steps:
      - uses: actions/checkout@v4
      - name: Install dependencies for ${{ matrix.project }}
        run: |
          cd ${{ matrix.project }}
          pip install -r requirements.txt
      - name: Run tests for ${{ matrix.project }}
        run: |
          cd ${{ matrix.project }}
          pytest tests/
```

This will create separate jobs for each project, allowing them to run in parallel. GitHub Actions will handle this efficiently. You can also combine this with path filters – for example, if a pull request only changes `projectA` , the matrix will only include `projectA` , saving time . Matrix builds are a great way to ensure that all projects are tested without having to run everything sequentially.

- **Reusable Workflows to Avoid Duplication:** If you have common steps across projects (like installing dependencies, running linters, etc.), consider creating **reusable workflows**. A reusable workflow is a

workflow file that can be called by other workflows. This helps avoid duplicating the same setup code in multiple workflow files . For example, you could have a `.github/workflows/reusable_install.yml` that installs dependencies, and then each project's workflow can use `uses: ./.github/workflows/reusable_install.yml` to call it. This keeps your CI configuration DRY (Don't Repeat Yourself) and easier to maintain.

- **Cache Dependencies:** Caching is crucial in monorepos to speed up CI. You can cache Python dependencies (e.g. using `actions/cache` to cache `~/.cache/pip` or the `uv` cache directory) so that each run doesn't have to reinstall everything from scratch. For instance, if you use `uv`, you can cache the `.venv` and `~/.cache/uv` directories between runs . GitHub's caching system can be used to achieve this. By caching, you can drastically reduce the time it takes to install dependencies for each project in the monorepo .

- **Environment Variables and Secrets Management:** When deploying or running different projects in different environments (e.g. one project to staging, another to production), you can use GitHub's environment features. Define environments for staging and production in the repository settings, and use them in your workflow jobs. This allows you to set environment-specific secrets and variables (like API keys or database URLs) . For example, you might have a job that only runs on the `main` branch and deploys to production, and it can access production secrets, whereas a job on a feature branch deploys to staging and uses staging secrets. This way, you keep production and staging separate even in a monorepo.

- **CI Job Organization:** Structure your GitHub Actions workflow jobs logically. You might have separate jobs for **linting**, **testing**, **type checking**, and **deployment** for each project. For example, you could have a job that runs `flake8` or `pylint` on all projects, another that runs unit tests for all projects, and a final job that deploys projects when appropriate. You can use the matrix or path filters to determine which jobs to run. By breaking it down, you can parallelize where possible and also see at a glance which part of the CI failed. GitHub's UI will show each job's status, making it easier to diagnose issues.

- **Workflow Examples:** To illustrate, here's a simplified example of a GitHub Actions workflow that runs linting and tests for all projects when any project's code changes:

```
name: Monorepo CI

on:
  push:
    paths:
      - '**/*.py'  # Run on any Python file change

jobs:
  lint:
    name: Lint
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Set up Python
        uses: actions/setup-python@v5
        with:
```

```
          python-version: '3.12'
      - name: Install dependencies
        run: pip install flake8
      - name: Run flake8 on all projects
        run: find . -name "*.py" | xargs flake8

  test:
    name: Test
    runs-on: ubuntu-latest
    strategy:
      matrix:
        project: [projectA, projectB, projectC]
    steps:
      - uses: actions/checkout@v4
      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.12'
      - name: Install dependencies for ${{ matrix.project }}
        run: |
          cd ${{ matrix.project }}
          pip install -r requirements.txt
      - name: Run tests for ${{ matrix.project }}
        run: |
          cd ${{ matrix.project }}
          pytest tests/
```

This workflow has two jobs: `lint` and `test`. The `lint` job runs `flake8` on all Python files in the repo. The `test` job uses a matrix to run tests for each project in parallel. Each job will only run if there are relevant changes (due to the `paths` filter) . If a PR only changes `projectA`, the `test` matrix will only include `projectA`, and the `lint` job will still run on all files (since linting is not project-specific).

For deployment, you might have a separate workflow that runs on `push to main` and uses environment-specific steps:

```
name: Monorepo Deploy

on:
  push:
    branches:
      - main

jobs:
  deploy-staging:
    name: Deploy to Staging
    runs-on: ubuntu-latest
    environment:
      name: staging
```

```
    steps:
      - uses: actions/checkout@v4
      - name: Deploy projectA to staging
        run: |
          cd projectA
          # deployment commands for projectA to staging

  deploy-production:
    name: Deploy to Production
    runs-on: ubuntu-latest
    environment:
      name: production
    needs: [deploy-staging]
    steps:
      - uses: actions/checkout@v4
      - name: Deploy projectA to production
        run: |
          cd projectA
          # deployment commands for projectA to production
      - name: Deploy projectB to production
        run: |
          cd projectB
          # deployment commands for projectB to production
```

This workflow deploys projects to staging on every main branch push and then to production after staging deployment succeeds. The `needs` keyword ensures production deployment happens only after staging is done. Each deployment job runs in its respective environment, so it can use environment-specific secrets. For example, the production deployment can access production secrets, whereas staging uses staging secrets. This setup keeps CI/CD fast, modular, and easier to maintain .

By following these practices, you can leverage GitHub Actions to handle a monorepo effectively. The key is to design workflows that are efficient and tailored to the monorepo structure. With careful configuration, you can avoid the pitfalls of monorepo CI and reap the benefits of unified pipelines.

Using the `uv` Library in a Monorepo

`uv` is a modern Python package manager and environment builder that has been gaining popularity as an alternative to traditional tools like `pip` and `poetry`. It's written in Rust for speed and offers a **monorepo-friendly** approach to dependency management. Here's why `uv` is useful and how to use it in a monorepo:

Why Use `uv` in a Monorepo?

- **Speed and Efficiency:** `uv` is extremely fast – it can install dependencies **10–100 times faster** than `pip` in many cases . This is crucial for CI/CD pipelines and for large projects. In a monorepo with many projects and dependencies, this speed can save significant time.

- **Unified Dependency Management:** `uv` allows you to manage dependencies for multiple projects in a single repository with a consistent workflow. It uses a `pyproject.toml` file at the root to define workspaces and each project can have its own `pyproject.toml` with its dependencies . The result is a single lockfile that tracks all dependencies for the entire monorepo, ensuring consistency across projects . This is similar to how JavaScript monorepos use a single `yarn.lock` or `package-lock.json` . In Python, `uv` 's approach helps avoid issues where different projects might require different versions of the same library.
- **Modern Python Project Structure:** `uv` encourages using `pyproject.toml` for all metadata and source code in a dedicated package directory (e.g. `src/`). This aligns with modern Python best practices and makes it easier to treat each project in the monorepo as a distinct Python package. It also supports **workspaces**, which is a key feature for monorepos .
- **Lightweight and Minimalist:** `uv` is a single binary that handles installation, virtual environment creation, and dependency resolution. It doesn't come with extra features like project scaffolding or publishing (it focuses on being a package manager) . This simplicity means it has a small footprint and is easy to integrate into existing workflows.
- **Support for Private Dependencies:** `uv` can be configured to use private package repositories. For example, if you have internal Python packages or a private PyPI, you can specify those in your `pyproject.toml` and `uv` will fetch them. This is useful if your monorepo has projects that depend on internal libraries – `uv` can be set up to use a private index, ensuring those dependencies are installed correctly .
- **CI-Friendly:** Because `uv` is fast and produces a reproducible lockfile, it's well-suited for CI/CD. It can cache installed packages and reuse them across jobs. Many data engineering teams have switched to `uv` for its speed and reliability in CI . It's also been noted to be "optimized for CI/CD" and is very well documented .

Setting Up `uv` in a Monorepo:

To use `uv` in your monorepo, you'll need to have `uv` installed (you can install it via the official installer or package managers). Once installed, you can initialize a workspace and manage your projects:

1. **Create a `pyproject.toml` at the root:** If you don't have one already, create a `pyproject.toml` in the root of the monorepo. In this file, you define your workspace. For example:

```
[tool.uv.workspace]
members = ["projectA", "projectB", "projectC"]
```

This tells `uv` that the monorepo has projects `projectA` , `projectB` , and `projectC` as members of the workspace. The root `pyproject.toml` can also have a `[project]` section if you consider the monorepo itself a "project" (though it might not be necessary if the root is just metadata).

2. **Set up each project with `pyproject.toml` :** For each project directory (like `projectA/`), create a `pyproject.toml` file that defines the project's metadata and dependencies. For example, `projectA/pyproject.toml` might look like:

```
[project]
name = "projectA"
version = "0.1.0"
description = "Description for projectA"
authors = ["Your Name "]
dependencies = [
    "numpy >=1.26",
    "pandas >=2.1",
    # Add other dependencies
]
```

Each project should also have a `src/` directory containing its source code. The `src` directory should have the package structure (e.g. `src/projectA/__init__.py`).

3. **Initialize the workspace:** From the root of the monorepo, run `uv init`. This will create the `uv.lock` file (the lockfile) and set up the virtual environment. The `uv.lock` will list all the exact versions of dependencies for the entire workspace . It's important to commit `uv.lock` to version control so that all developers and CI systems use the same dependencies.

4. **Install dependencies:** Run `uv sync` from the root. This will install all dependencies for all projects into a single virtual environment. If the `uv.lock` is already present, `uv sync` will ensure that the environment matches exactly. If not, it will create the lockfile and install the dependencies. This is the equivalent of `pip install -r requirements.txt` but for all projects together.

5. **Install a dependency for a specific project:** If you need to add a dependency to one project, you can do so by running `uv add -w projectA "new-package";` from the root. The `-w` flag specifies the workspace member (project) to add the dependency to. This will update `projectA/pyproject.toml` and the `uv.lock`. If you are in the `projectA` directory, you can also run `uv add "new-package"` (the `-w` isn't needed in that case).

6. **Run a project's code:** To run a script or the code of a project, you can use `uv run`. For example, from the root, run `uv run projectA:script.py` to execute `script.py` in the `projectA` environment. Or if you're in `projectA`, run `uv run script.py`. `uv run` will use the virtual environment created by `uv sync` and run the command with the project's dependencies.

7. **Testing:** To run tests for a project, you can use `uv run pytest tests/` in that project's directory (or `uv run -w projectA pytest tests/` from root). This ensures the tests run with the project's dependencies.

8. **Using `uv` with IDEs:** Many IDEs (like PyCharm) support using `uv` by pointing them to the virtual environment created by `uv`. You can find the virtual environment path in the output of `uv sync` (it's usually in `.venv` in the root). You can configure your IDE to use this environment for the project. Alternatively, you can use `uv` to manage the environment and have the IDE use it, which ensures consistency between development and CI.

Example Workflow: Suppose you have a monorepo with projects `etl_pipeline`, `model_training`, and `data_analysis`. Here's how you might use `uv`:

1. **Root `pyproject.toml`:**

```
[tool.uv.workspace]
members = ["etl_pipeline", "model_training", "data_analysis"]
```

2. **Each project's `pyproject.toml` (e.g. `etl_pipeline/pyproject.toml`):**

```
[project]
name = "etl_pipeline"
version = "0.1.0"
dependencies = [
    "pandas",
    "numpy",
    "requests",
]
```

3. **Initialize and sync:** From root, run `uv init` (creates `uv.lock`) and `uv sync` (installs dependencies in `.venv`).

4. **Install a new dependency for `model_training`:** From root, run `uv add -w model_training scikit-learn`. This adds `scikit-learn` to `model_training/pyproject.toml` and updates the lockfile.

5. **Run a script in `etl_pipeline`:** From root, run `uv run etl_pipeline:scripts/etl.py` to execute that script with `etl_pipeline`'s dependencies.

6. **Run tests for `data_analysis`:** From root, run `uv run -w data_analysis pytest tests/` to run tests in the `data_analysis` project's environment.

By using `uv`, you have a unified way to manage dependencies across all projects. The lockfile ensures that everyone and every CI run uses the same versions, preventing dependency drift. The speed of `uv` means you'll spend less time waiting for installs, which is great for large projects. And because `uv` is designed for reproducibility, you can be confident that the environment in development, staging, and production will be identical.

It's worth noting that `uv` is relatively new, but it's gaining adoption rapidly. Many data engineering teams are switching to `uv` for its performance and simplicity . If you're already using tools like `poetry` or `pip-tools`, you can try `uv` in your monorepo to see if it improves your workflow.

Useful Tools and Ecosystem for Monorepos in Data Science

While `uv` is a key tool for monorepos in Python, there are several other tools and practices that can enhance the monorepo experience:

- **Monorepo Management Tools:** There are various tools designed to manage and optimize monorepos, especially for JavaScript/TypeScript projects. For Python, however, many of these tools aren't directly

applicable. However, some concepts from these tools can be useful:

- **Bazel:** A Google-developed build tool that supports monorepos by enabling incremental builds and dependency analysis. It's used by companies like Google and Twitter for large codebases. Bazel can be used with Python projects (there are Python rules available) and can efficiently build and test only the parts of the monorepo that changed . It's more complex to set up but can be powerful for very large or complex monorepos.
- **Gradle:** A build tool primarily for Java/Kotlin, but it has plugins and can be used for Python projects as well. Gradle supports multi-project builds and has features like incremental builds. It's more common in Java projects but some data science teams use it for building and testing Python code in a monorepo.
- **Nx:** A monorepo toolkit developed by Nrwl (acquired by Google) that is very popular in the Angular and JavaScript communities. Nx provides smart build caching, task orchestration, and code generation for monorepos. It can detect which projects are affected by a change and only run relevant tasks. Nx has plugins fo many frameworks and can be used to manage Python projects in a monorepo as well . It's a more comprehensive solution but requires learning its specific commands and configuration . Nx is often chosen for JavaScript/TypeScript monorepos due to its rich ecosystem, but it's also possible to use it for Python by setting up custom executors.
- **Turborepo:** Developed by Vercel, Turborepo is a high-performance build system for JavaScript/TypeScript monorepos that focuses on task orchestration and caching . It can parallelize tasks and cache results to avoid redundant work. Turborepo can be used with Python as well – you can use it to run scripts for each project in the monorepo. It's known for its simplicity and is easy to integrate into CI pipelines. Many data engineering teams have found Turborepo useful for speeding up their CI when they have multiple Python projects in a monorepo . It's a good choice if you want a tool that's optimized for fast builds and has a straightforward setup
- **Pants:** An open-source build system for Python and other languages that is monorepo-oriented. Pants can manage multi-language codebases and uses a "hermetic" approach to builds (each build is isolated). It can automatically detect affected targets and has features like remote caching. Pants is used by some data science teams (e.g. some Netflix teams) to manage their Python monorepos. It's more complex to learn but offers strong performance and scalability for large projects .
- **Rush:** Developed by Microsoft, Rush is a monorepo toolchain for JavaScript/TypeScript. It provides a way to manage dependencies across projects and can be used for Python by running scripts via Rush. It's less common in Python but might be used if you have a mixed-language monorepo.
- **PNPM:** A fast npm package manager that supports workspaces, making it suitable for monorepos. If you have JavaScript projects in your monorepo, PNPM can manage shared dependencies efficiently. However, it's less relevant for pure Python projects, but could be useful if you have a hybrid codebase.
- **Lerna:** A tool for managing JavaScript projects with multiple packages. Lerna has been succeeded by Nx, but if you have a legacy JavaScript monorepo, Lerna might still be used. It's not relevant for Python, but it's worth noting as a concept.

- **Data Engineering and Orchestration Tools:** When working on data science projects, you might use data engineering tools like **Apache Airflow** for pipelines, **DVC** for data versioning, or **MLflow** for model tracking. These can be part of the monorepo (for code and configuration) or separate. If they are separate, you can still reference them in the monorepo's documentation. It's important to ensure that these tools are configured to work with the monorepo structure (for example, MLflow projects might be stored in a directory and run via MLflow, etc.).

- **Linters and Formatters:** Tools like **flake8**, **pylint**, **black**, **ruff** (the fast Python linter), and **isort** are essentia for code quality. In a monorepo, you can set up a single configuration for these tools and run them across all projects. For instance, you might have a `pyproject.toml` with `ruff` configuration that applies to all Python code. You can run a command like `ruff check .` from the root to check all files in the repo. Similarly, `black .` can format all Python files. This ensures consistency and makes it easy to enforce coding standards.
- **Testing Frameworks:** Use a unified testing framework for all projects. **pytest** is the most popular choice in Python, and you can configure it to run tests across all projects. For example, you could have a root `conftest.py` and a `pytest.ini` that define shared test configurations. Running `pytest` from the root wil discover and run tests in all project `tests/` directories. This is convenient, but you should also ensure that tests are isolated and that a failure in one project doesn't break another (which is why isolation and good design are important).
- **Continuous Integration Services:** We already discussed GitHub Actions, but other CI services like GitLab CI or Azure Pipelines can also be used with monorepos. The principles are similar – use path filters and matrix jobs. Many CI systems have documentation on handling monorepos (for example, GitLab CI has features like `rules:changes` to trigger jobs on file changes). If you use a different CI, you can apply the same strategies of running only relevant jobs.
- **Documentation and Collaboration Tools:** Use tools like **GitHub Wiki** or **readthedocs** to document the monorepo and its projects. A clear wiki or README can help developers understand how everything fits together. For collaboration, GitHub's features like **CODEOWNERS** can be used to ensure that certain files or directories are reviewed by the right people . This is especially important in a monorepo to manage ownership and prevent one team's changes from accidentally breaking another's code. Additionally, using feature branches for each project's work (even within the monorepo) can help isolate changes and make reviews easier.
- **Monitoring and Monitoring:** If you deploy your data science applications or models, consider using monitoring tools. For example, you might deploy a model API and use a tool like **Prometheus** or **Datadog** to monitor its performance. These tools can be configured in the monorepo's deployment configuration or separate repos. The monorepo can include the configuration files for these tools.

In summary, while `uv` is a key tool for dependency management, the monorepo ecosystem for data science also includes build tools (like Bazel, Gradle, Nx, Turborepo, Pants), data engineering tools (Airflow, DVC, MLflow), and various quality and collaboration tools. The choice of tools depends on your specific needs and stack. Many teams use a combination of these – for example, using `uv` for Python package management, Nx or Turborepo for build and test orchestration, and Airflow or MLflow for data pipelines and models. The goal is to find the right mix that works for your team and makes the monorepo effective.

Best Practices for Collaboration and Monorepo Management

To successfully use a monorepo for data science projects, it's important to follow best practices that address the challenges and maximize the benefits. Here are some key best practices:

- **Clear Ownership and CODEOWNERS:** Define clear ownership of different parts of the monorepo. Use GitHub's **CODEOWNERS** file to specify which team or person is responsible for each directory or file . This ensures that when changes are made, the right people are notified for review. In a data science context, you might have CODEOWNERS like:

```
/data_pipelines @data-team
/models @model-team
/apps @app-team
/tools @data-engineering-team
```

This way, if someone modifies a data pipeline file, the data team is automatically added as a reviewer. This helps in managing code ownership and prevents accidental merges without the right expertise. Additionally, enforce code review policies (like requiring a certain number of approvals or that all tests pass) to maintain code quality .

- **Separate Branches for Isolation:** Even though everything is in one repo, it's still good practice to use feature branches for development. Encourage developers to create feature branches (e.g. `feature/new-model`, `feature/etl-improvement`) for each significant change. This way, changes are isolated until ready to merge. When a feature branch is ready, it can be reviewed and merged into the main branch. This approach helps prevent conflicts and ensures that main stays stable. In a monorepo, you can still use trunk-based development or Gitflow as appropriate – just apply it within the single repo.
- **Continuous Integration and Testing:** Set up a robust CI/CD pipeline as discussed. Ensure that all tests (unit tests, integration tests, etc.) are run for every change. Use path filters and matrix jobs to keep CI fast . It's also a good idea to run linting and static type checking for Python code in CI (using tools like `flake8`, `pylint`, or `mypy`) to catch issues early. Fail the build if any of these checks fail. Consider running tests in different environments (e.g. different Python versions) if your projects support it. Regularly monitor CI performance and adjust the pipeline as needed to avoid bottlenecks.
- **Versioning and Releases:** Decide on a versioning strategy for your monorepo. If all projects are part of the same product or release cycle, you might version them together. You could use a single `VERSION` file at the root that all projects reference, or use `uv`'s workspace versioning (which isn't automatic but can be done manually). When releasing, you can tag the monorepo with a version and deploy all projects at that version. If projects are independent, you might version them separately – but even then, consider using a common approach (like Semantic Versioning) to keep things consistent. Using tools like `uv` can help manage versions across projects. Also, ensure that you document the version history for each project (maybe with a `CHANGELOG.md` in each project directory) to track changes.
- **Documentation and Onboarding:** Maintain thorough documentation. Have a README for the monorepo that explains the structure, how to run each project, and any dependencies. Include installation instructions and setup steps. Create `CONTRIBUTING.md` that outlines the workflow (like branching, PR process, etc.). When onboarding new team members, provide them with this documentation and a quick tour of the monorepo structure. Having clear docs will make it easier for new members to contribute without causing disruptions.

- **Use of Git Submodules or External Repos:** If you have very large or external projects that you don't want to duplicate in the monorepo, use **Git submodules** sparingly. As mentioned, submodules allow you to reference another repo's code in your monorepo. However, they require extra care – team members must remember to update submodules and handle any changes. An alternative is to keep external projects in separate repos and reference them via Git submodules or just via links. For data, consider using data versioning tools (like DVC or Git LFS) instead of putting large data files in the repo. For code libraries, if they are developed in-house, you might consider hosting them in a private repo and adding them as a submodule. But be cautious: submodules can complicate things, so use them only when necessary and ensure your team understands how to manage them.
- **Monitor and Refactor:** Regularly monitor the health of the monorepo. Check for any performance issues (like slow clones or builds) and address them by pruning old data, optimizing workflows, or splitting the repo if needed. As the monorepo grows, you might find that certain projects or code can be split into their own repositories for better manageability. If a project is no longer active or is very independent, consider removing it from the monorepo or archiving it. Also, refactor the monorepo structure periodically to keep it clean – for example, if a directory becomes too large or if projects diverge, consider reorganizing. The goal is to keep the monorepo as efficient as possible.
- **Security and Compliance:** Since the monorepo contains all code, be mindful of security. Ensure that sensitive data or credentials are not committed (use environment variables or secrets for those). Use GitHub's secret management for any API keys or database passwords used in the monorepo. If your team deals with compliance (like GDPR or HIPAA), ensure that the monorepo's code and data don't violate any policies. Regularly audit the code for any sensitive information. Also, consider using tools like **CodeQL** or **SonarQube** to scan the monorepo for security vulnerabilities and code quality issues. These tools can be integrated into CI to catch issues early.
- **Communication and Governance:** Establish clear communication channels and governance for the monorepo. If possible, have a "monorepo owner" or a group that is responsible for managing the repository, merging changes, and enforcing policies. This can be a small team or just a set of individuals who are familiar with the monorepo. Regularly hold meetings or syncs to discuss any issues with the monorepo (like a project that's growing too large, or a dependency conflict). Encourage developers to reach out if they encounter problems in the monorepo – early communication can prevent bigger issues down the line.
- **Backups and Disaster Recovery:** Treat the monorepo like any other critical codebase. Ensure you have backups of the repository (GitHub should handle this, but it's good to be aware). If using GitHub, you can take advantage of GitHub's features for backup and versioning. Consider using **git clone --mirror** to create a backup mirror. For data stored in the repo, ensure you have external backups as well (like storing large data files in cloud storage). In case of a disaster, you want to be able to recover the monorepo and all its data.

By following these best practices, you can mitigate many of the challenges of monorepos and ensure a smooth collaboration environment. The key is to treat the monorepo as a unified codebase that requires governance and care, just like any other large project. With the right practices, a monorepo can become a powerful asset for your data science team, enabling efficient development, testing, and deployment of all your projects.

Summary

Using a monorepo for data science projects on GitHub can greatly enhance collaboration, code reuse, and efficiency. It allows multiple data science projects to live together in one repository, simplifying workflows and enabling coordinated changes. Key advantages include improved collaboration, reduced code duplication, unified tooling, and streamlined CI/CD. However, monorepos also present challenges like large repository size, performance, and dependency management, which must be addressed through strategies like path-based workflows, caching, and careful configuration.

To set up a monorepo, create a new GitHub repository and organize your projects into subdirectories. Use a clear structure (e.g. separate directories for pipelines, models, apps, tools) and maintain documentation. Leverage GitHub Actions for CI, using features like path filters and matrix jobs to optimize your workflows . Tools like `uv` can be invaluable for managing dependencies in a monorepo, providing speed and consistency in dependency resolution . Other tools and practices (like monorepo management tools, linters, and testing frameworks) should be integrated to ensure the monorepo runs smoothly.

By adhering to best practices such as clear ownership, feature branching, thorough testing, and good documentation, your team can overcome the challenges of monorepos and fully realize their benefits. A well-managed monorepo can lead to faster development cycles, easier maintenance, and better alignment between data science projects. In the end, the decision to use a monorepo should be weighed against the specific needs of your team and projects, but for many data science teams, it represents a practical and powerful approach to managing their codebase.

Reference

[1] Monorepo in Data Science Teams — A Practical Starting Point from ...
https://medium.com/clarityai-engineering/monorepo-in-data-science-teams-892fe64a9ef0

[2] Notes on the Monorepo Pattern - DEV Community
https://dev.to/david_whitney/notes-on-the-monorepo-pattern-5egc

[3] Monorepo in Data Science Teams — A Practical Starting Point from ...
https://medium.com/clarityai-engineering/monorepo-in-data-science-teams-892fe64a9ef0

[4] Code Reviews at Scale: CODEOWNERS & GitHub Actions Guide
https://www.aviator.co/blog/code-reviews-at-scale/

[5] Monorepos: Are They a Good Thing? - Visual Studio Live!
https://vslive.com/blogs/news-and-tips/2024/05/monorepos-pros-cons.aspx

[6] Best Architecture for Dev Collaboration: Monorepo vs. Multi-Repo
https://www.gitkraken.com/blog/monorepo-vs-multi-repo-collaboration

[7] Monorepo in Data Science Teams — A Practical Starting Point from ...
https://medium.com/clarityai-engineering/monorepo-in-data-science-teams-892fe64a9ef0

[8] What are the best practices for organizing large monorepos using ...
https://github.com/orgs/community/discussions/158727

[9] Monorepo Mayhem? Tame Your CI/CD with GitHub Actions the ...

https://lalits77.medium.com/monorepo-mayhem-tame-your-ci-cd-with-github-actions-the-right-way-2adacbdd33c6

[10] Monorepos: A Comprehensive Guide with Examples | by Md Julakadar
https://medium.com/@julakadaredrishi/monorepos-a-comprehensive-guide-with-examples-63202cfab711

[11] Benefits and challenges of monorepo development practices - CircleCI
https://circleci.com/blog/monorepo-dev-practices/

[12] Monorepos with Turborepo. Managing multiple repositories in a…
https://medium.com/@ignatovich.dm/monorepos-with-turborepo-6aa0852708ee

[13] Why You Should Use a Monorepo (and Why You Shouldn't)
https://lembergsolutions.com/blog/why-you-should-use-monorepo-and-why-you-shouldnt

[14] alexeagleson/monorepo-example - GitHub
https://github.com/alexeagleson/monorepo-example

[15] Monorepo in Data Science Teams — A Practical Starting Point from ...
https://medium.com/clarityai-engineering/monorepo-in-data-science-teams-892fe64a9ef0

[16] Monorepos: Please don't! - Medium
https://medium.com/@mattklein123/monorepos-please-dont-e9a279be011b

[17] Monorepos: Please don't! - Medium
https://medium.com/@mattklein123/monorepos-please-dont-e9a279be011b

[18] Ultimate guide to uv library in Python - Deepnote
https://deepnote.com/blog/ultimate-guide-to-uv-library-in-python

[19] Google, Meta, and others use monorepos | Sahn Lam | 19 comments
https://www.linkedin.com/posts/sahnlam_google-meta-and-others-use-monorepos-activity-7135519039124115457-ZnZ-

[20] What are the best practices for organizing large monorepos using ...
https://github.com/orgs/community/discussions/158727

[21] Monorepos: Please don't! - Medium
https://medium.com/@mattklein123/monorepos-please-dont-e9a279be011b

[22] Managing Python Projects With uv: An All-in-One Solution
https://realpython.com/python-uv/

[23] An example starting point monorepo for data sicence teams - GitHub
https://github.com/clarityai-eng/datascience-monorepo-example

[24] uv workspaces in a monorepo - thoughts on change-only testing ...
https://github.com/astral-sh/uv/issues/6356

[25] Nx vs Turborepo: A Comprehensive Guide to Monorepo Tools
https://www.wisp.blog/blog/nx-vs-turborepo-a-comprehensive-guide-to-monorepo-tools

[26] Monorepo in Data Science Teams — A Practical Starting Point from ...
https://medium.com/clarityai-engineering/monorepo-in-data-science-teams-892fe64a9ef0

[27] When (and when not) to use a monorepo - Graphite
https://graphite.dev/guides/when-to-use-monorepo

[28] Nx: A Feature-Based Philosophy - by Ferdi Sancak - Medium
https://medium.com/@ferdi_48628/nx-a-feature-based-philosophy-917529204ff6

[29] Monorepo vs. multi-repo: Different strategies for ... - Thoughtworks
https://www.thoughtworks.com/en-us/insights/blog/agile-engineering-practices/monorepo-vs-multirepo

[30] alexeagleson/monorepo-example - GitHub
https://github.com/alexeagleson/monorepo-example

[31] Monorepo vs Multi-Repo: Pros and Cons of Code Repository ...
https://kinsta.com/blog/monorepo-vs-multi-repo/

[32] An example CI/CD setup for a monorepo using vanilla GitHub Actions
https://www.generalreasoning.com/blog/software/cicd/2025/03/22/github-actions-vanilla-monorepo.html

[33] A curated list of awesome Monorepo tools, software and architectures.
https://github.com/korfuri/awesome-monorepo

[34] Scaling monorepo maintenance - The GitHub Blog
https://github.blog/open-source/git/scaling-monorepo-maintenance/

[35] Monorepo Mayhem? Tame Your CI/CD with GitHub Actions the ...
https://lalits77.medium.com/monorepo-mayhem-tame-your-ci-cd-with-github-actions-the-right-way-2adac
bdd33c6

[36] Handling CI/CD in a Mono Repo With Multiple Python Packages
https://www.kazis.dev/blogs/python-packages-mono-repo-ci

[37] Create reusable workflows in GitHub Actions
https://resources.github.com/learn/pathways/automation/intermediate/create-reusable-workflows-in-githu
b-actions/

[38] Using uv in GitHub Actions - Astral Docs
https://docs.astral.sh/uv/guides/integration/github/

[39] Optimizing uv in GitHub Actions: One Global Cache to Rule Them All
https://szeyusim.medium.com/optimizing-uv-in-github-actions-one-global-cache-to-rule-them-all-9c64b42
aee7f

[40] What are the best practices for organizing large monorepos using ...
https://github.com/orgs/community/discussions/158727

[41] What are the best practices for organizing large monorepos using ...
https://github.com/orgs/community/discussions/158727

[42] Python: Monorepo with UV - Medium
https://medium.com/@life-is-short-so-enjoy-it/python-monorepo-with-uv-f4ced6f1f425

[43] why monorepos?? : r/devops - Reddit
https://www.reddit.com/r/devops/comments/1nvy3if/why_monorepos/

[44] uv workspaces in a monorepo - thoughts on change-only testing ...
https://github.com/astral-sh/uv/issues/6356

[45] Anyone using uv for package management instead of pip in their ...
https://www.reddit.com/r/dataengineering/comments/1ok9dj2/anyone_using_uv_for_package_manageme
nt_instead_of/

[46] Using workspaces | uv - Astral Docs
https://docs.astral.sh/uv/concepts/projects/workspaces/

[47] astral-sh/uv: An extremely fast Python package and project ... - GitHub
https://github.com/astral-sh/uv

[48] Monorepo Explained
https://monorepo.tools/

[49] Monorepo Explained
https://monorepo.tools/

[50] Monorepo in Data Science Teams — A Practical Starting Point from ...
https://medium.com/clarityai-engineering/monorepo-in-data-science-teams-892fe64a9ef0

[51] Who is using Nx in production? · nrwl nx · Discussion #5483 - GitHub
https://github.com/nrwl/nx/discussions/5483

[52] Top 5 Monorepo Tools for 2025 | Best Dev Workflow Tools - Aviator
https://www.aviator.co/blog/monorepo-tools/

[53] Can Turborepo be used for a monorepo that also includes a Django ...
https://github.com/vercel/turborepo/discussions/1077

[54] A curated list of awesome Monorepo tools, software and architectures.
https://github.com/korfuri/awesome-monorepo

[55] Mastering Nx: The Complete Guide to Modern Monorepo ...
https://dev.to/mcheremnov/mastering-nx-the-complete-guide-to-modern-monorepo-development-5573

[56] A curated list of awesome Monorepo tools, software and architectures.
https://github.com/korfuri/awesome-monorepo

[57] A curated list of awesome Monorepo tools, software and architectures.
https://github.com/korfuri/awesome-monorepo

[58] monorepo · GitHub Topics
https://github.com/topics/monorepo

[59] Git Monorepos with Github Actions | by Sam Reghenzi - Medium
https://medium.com/@SammyRulez/git-monorepos-with-github-actions-e47f56d8793b

[60] Monorepo CI best practices - Buildkite
https://buildkite.com/resources/blog/monorepo-ci-best-practices/

[61] 10 Common monorepo problems and how your team can solve them
https://digma.ai/10-common-problems-of-working-with-a-monorepo/

[62] Benefits and challenges of monorepo development practices - CircleCI
https://circleci.com/blog/monorepo-dev-practices/

[63] Git Monorepos with Github Actions | by Sam Reghenzi - Medium
https://medium.com/@SammyRulez/git-monorepos-with-github-actions-e47f56d8793b

[64] uv workspaces in a monorepo - thoughts on change-only testing ...
https://github.com/astral-sh/uv/issues/6356

[65] Deploy individual services from a monorepo using github actions

https://stackoverflow.com/questions/58136102/deploy-individual-services-from-a-monorepo-using-github-actions

[66] The Ultimate Tips for Working With Large Git Monorepos - Ken Muse
https://www.kenmuse.com/blog/tips-for-large-monorepos-on-github/

[67] Poetry vs UV. Which Python Package Manager should you use in ...
https://medium.com/@hitorunajp/poetry-vs-uv-which-python-package-manager-should-you-use-in-2025-4212cb5e0a14

[68] Document best practices for a monorepo · Issue #10960 · astral-sh/uv
https://github.com/astral-sh/uv/issues/10960

[69] Choosing monorepo tooling: nx.dev vs Turborepo for a green field ...
https://medium.com/@knidarkness/nx-dev-vs-turborepo-for-a-green-field-projects-in-2022-c73dd858b687

[70] Monorepo Explained
https://monorepo.tools/

[71] Best CI/CD Triggering Strategies for a Microservices "Monorepo"?
https://www.reddit.com/r/github/comments/1im6f71/best_cicd_triggering_strategies_for_a/