# Building and Publishing Python Packages

# Gratitude

Big ***thank you*** for attending!

# Presentation Outline

1. 📦 Python Packages
2. 🐍 Understanding PyPI
3. 🏗️ Builds with Poetry
4. 🚀 GitHub Releases
5. 🤖 Further Automation

# Why?

Packaging is a matter of design and architecture.

> "Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change."

- Grady Booch, Software Architecture Zen

# Python Packaging - Definitions

```
my_package/
│   __init__.py
│   module_a.py
│   module_b.py
```

- A Python **package** is a collection of **modules** (`.py` files) that usually include an "initialization file" `__init__.py`.

# Python Packaging - Definitions

```
my_package/
    __init__.py
    module_a.py
    module_b.py
```

Package(s)

Packaging

- Python *"**packaging**"* is a broader term indicating formalization of code with publishing intent.

# Python Packaging - Definitions

***But wait, I don't intend to publish some of my code for anyone! Why does packaging even matter?***

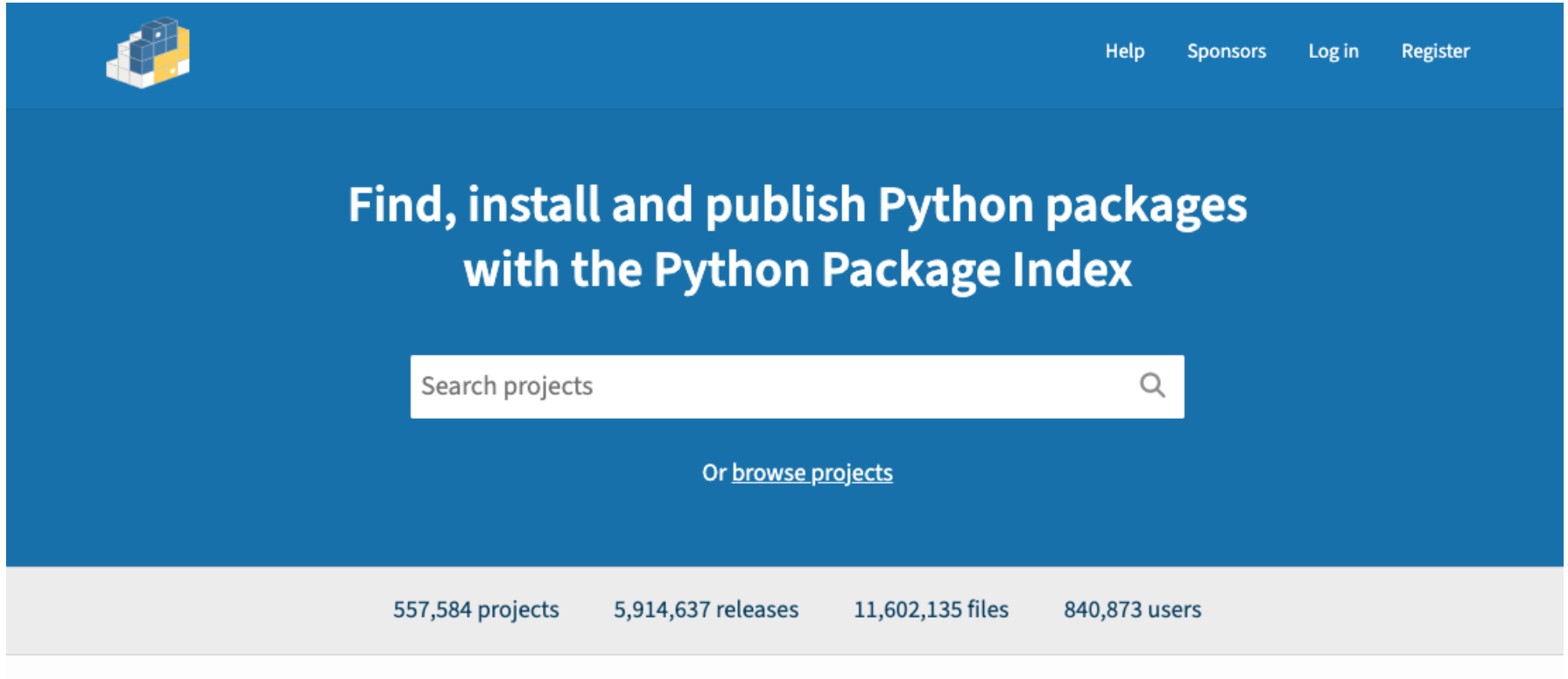
- Packaging practices provide a common way to organize your code so other developers may understand your project.
- Good packaging practices will help you understand other people's code as a result.
- What if you change your mind and you'd like to publish your work later on (no time like the present)?

# Python Packaging - Definitions

- Python packages are commonly installed from **PyPI** (Python Package Index, https://pypi.org).

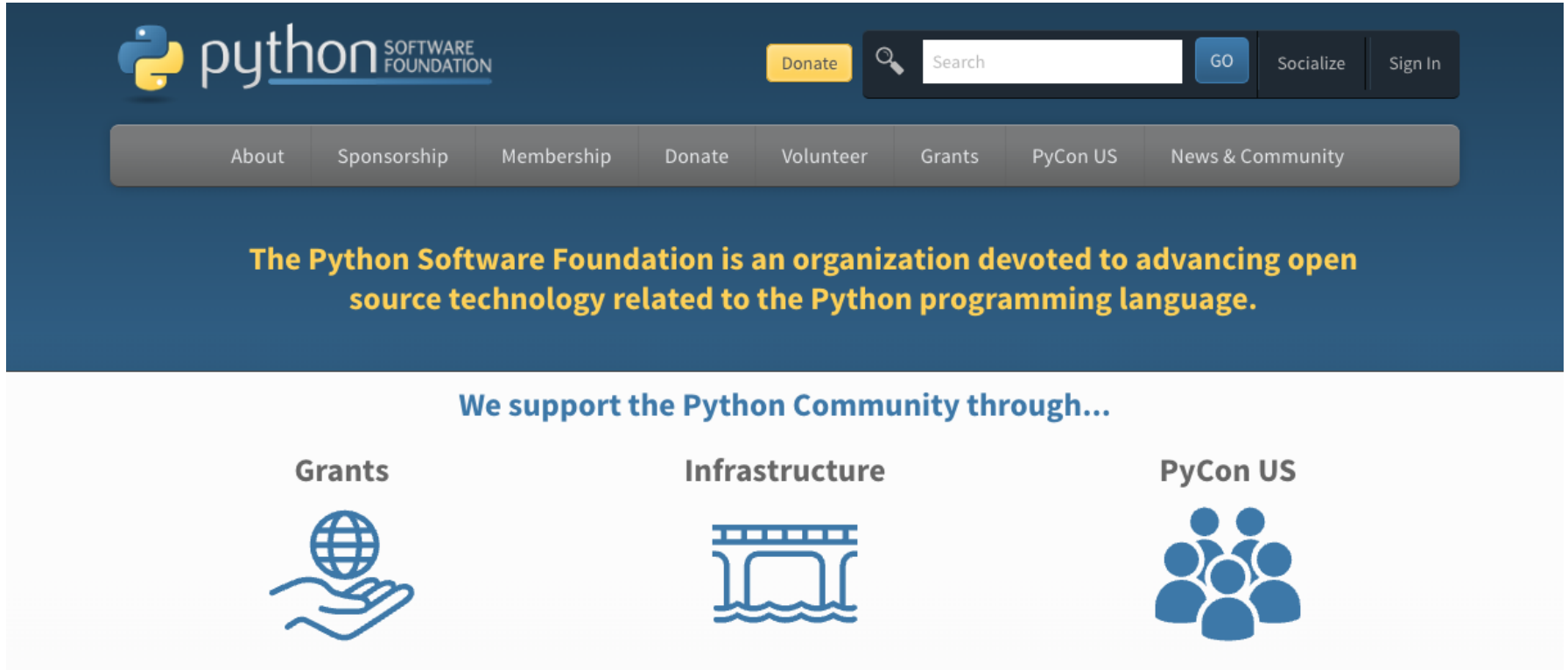
> For example: `pip install pandas` references PyPI by default to install for the `pandas` package.

# PyPI: Python Package Index



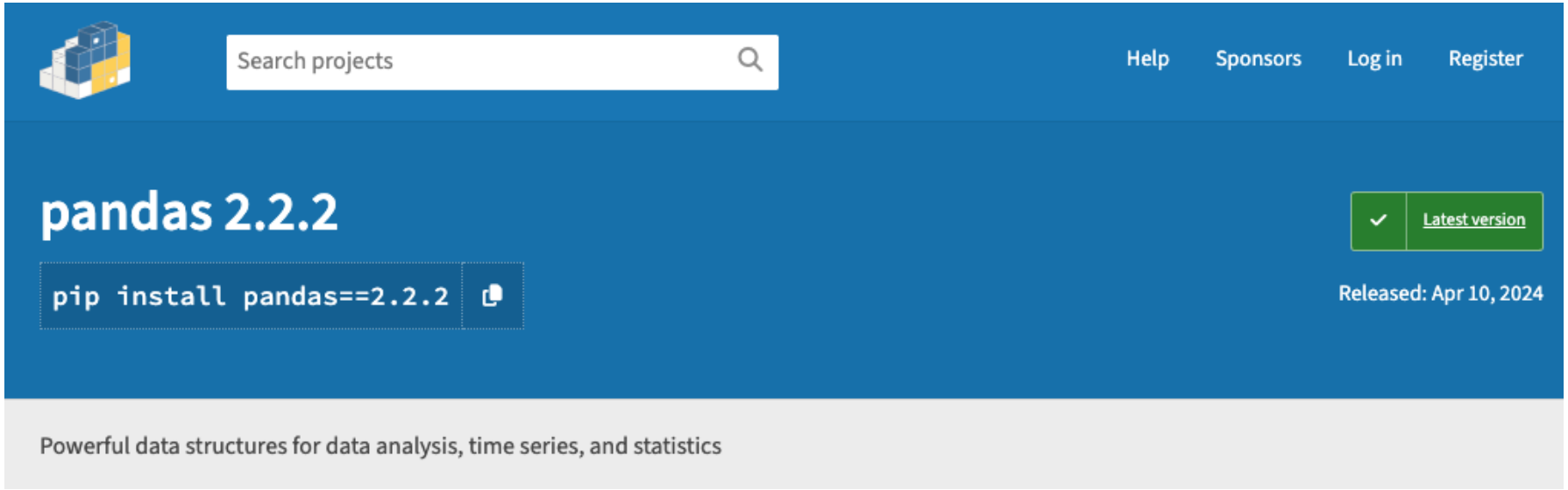

The **Python Package Index (PyPI)** (pypi.org) is the official repository for Python packages. It hosts thousands of projects, making it easy for developers to find, install, and share Python code.

# Python Software Foundation




PyPI is maintained by the **Python Software Foundation** (python.org) a 501(c)(3) non-profit which also supports Python documentation, runs the PyCon US conference, and distributes grants for Python-related development.

# What's in a PyPI package?




Let's look at Pandas on PyPI to learn about PyPI packages.

- Note the release number and ✅ latest version checkmark.
- Pandas uses semantic versioning schemes to distinguish between different versions.

# Semantic Versioning

4.2.1

**MAJOR** *Minor* patch

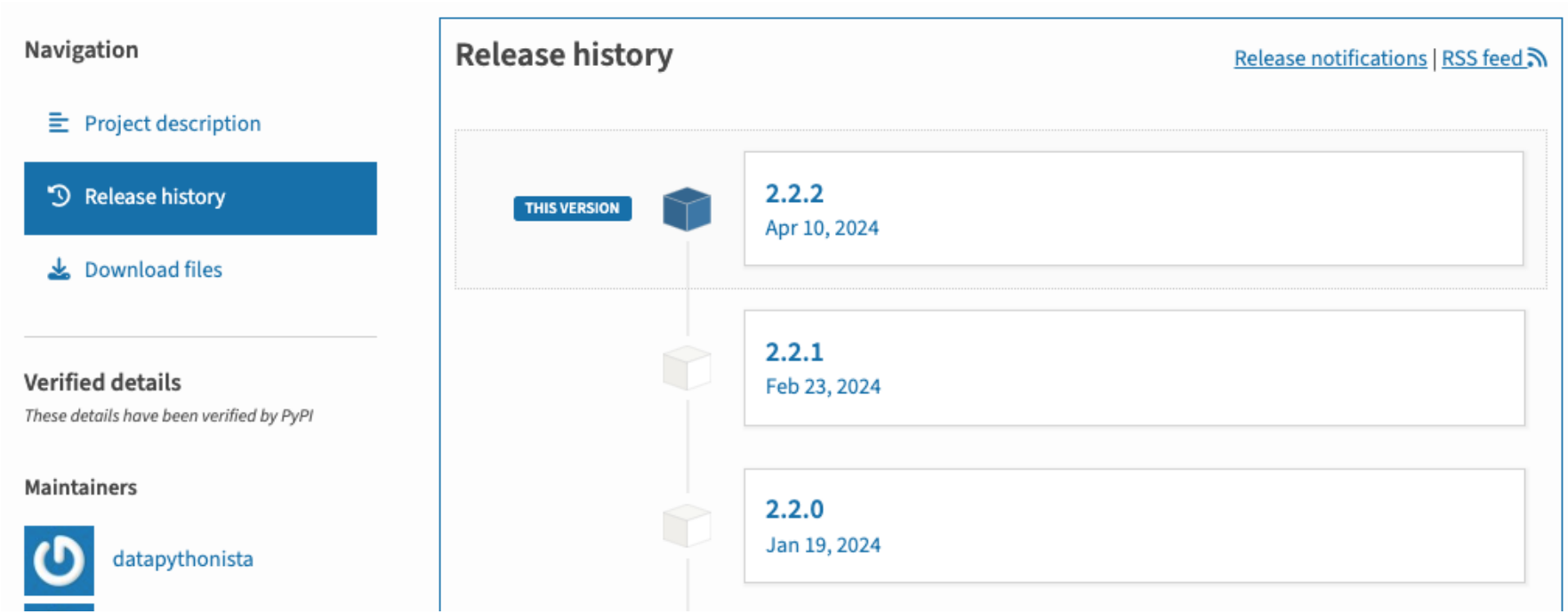
Semantic Versioning distinguishes releases of software.

```
Given a version number MAJOR.MINOR.PATCH, increment the:

MAJOR version when you make incompatible API changes
MINOR version when you add functionality in a backward compatible manner
PATCH version when you make backward compatible bug fixes

(from https://semver.org/)
```

# PyPI Releases




PyPI can provide multiple releases of packages.

# PyPI Release Downloads

Navigation

Project description

Release history

Download files

Verified details

*These details have been verified by PyPI*

Maintainers

datapythonista

jbrockmendel

jorisvandenbossche

jreback

## Download files

Download the file for your platform. If you're not sure which to choose, learn more about installing packages.

### Source Distribution

pandas-2.2.2.tar.gz (4.4 MB view hashes)

Uploaded Apr 10, 2024 `Source`

### Built Distributions

pandas-2.2.2-cp312-cp312-win_amd64.whl (11.5 MB view hashes)

Uploaded Apr 10, 2024 `CPython 3.12` `Windows x86-64`

pandas-2.2.2-cp312-cp312-musllinux_1_1_x86_64.whl (13.4 MB view hashes)

Uploaded Apr 10, 2024 `CPython 3.12` `musllinux: musl 1.1+ x86-64`

pandas-2.2.2-cp312-cp312-musllinux_1_1_aarch64.whl (15.9 MB view hashes)

Uploaded Apr 10, 2024 `CPython 3.12` `musllinux: musl 1.1+ ARM64`

pandas-2.2.2-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (12.7 MB view hashes)

Uploaded Apr 10, 2024 `CPython 3.12` `manylinux: glibc 2.17+ x86-64`

PyPI releases include source code and ***wheels***.

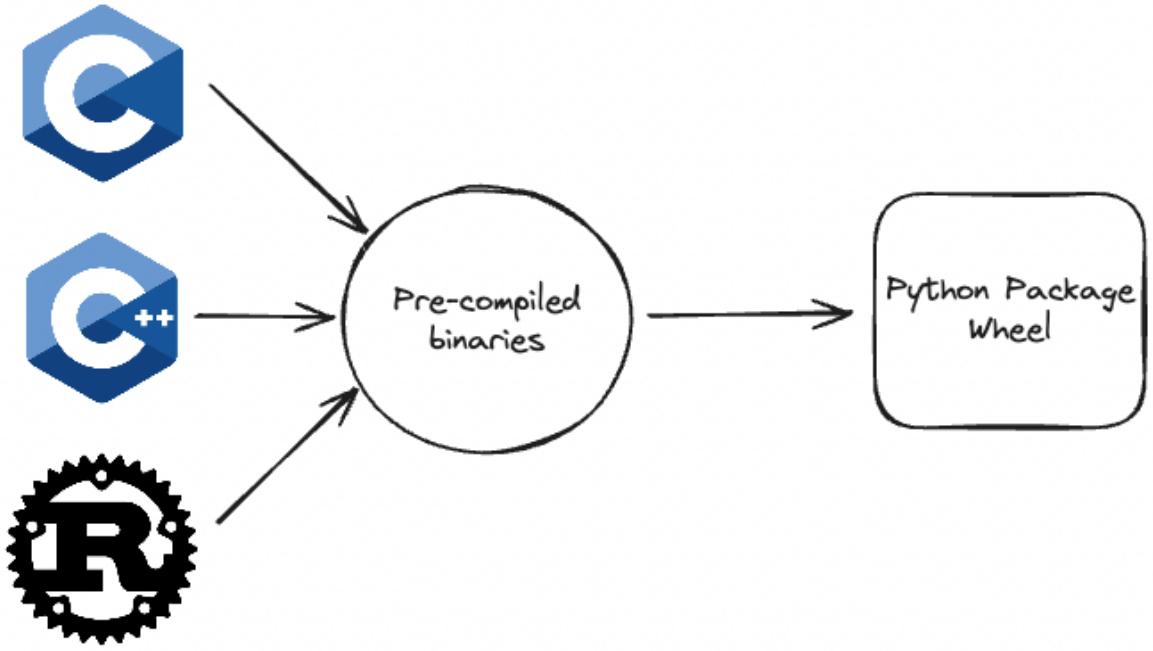
# What are Python wheels (`.whl`)?

```
$ unzip pandas-2.2.2-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64
inflating: ...

$ tree pandas-2.2.2
pandas-2.2.2
├── pandas
│   ├── __init__.py
│   ...
├── pandas-2.2.2.dist-info
│   ├── LICENSE
│   ├── METADATA
│   ├── RECORD
│   ├── WHEEL
│   └── entry_points.txt
```

- A Python wheel (`.whl`) is a zip file with package data.
- Wheels can be managed through the `wheel` package.
- Includes metadata about packages and optional pre-compiled extensions (sometimes OS-specific).

# Why pre-compile extensions?

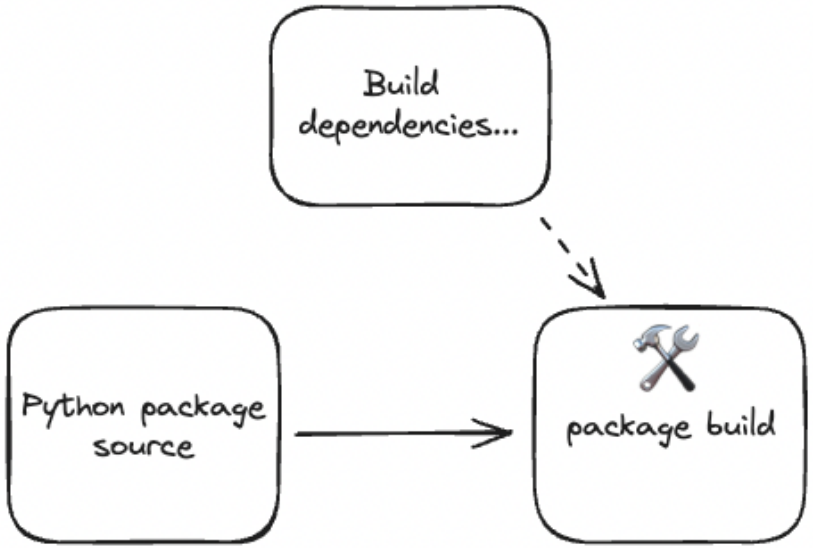



Pre-compilation allows us to remove dependencies or system-specific build procedures for languages like C, C++, or Rust.

# What about Python eggs (`.egg`)?

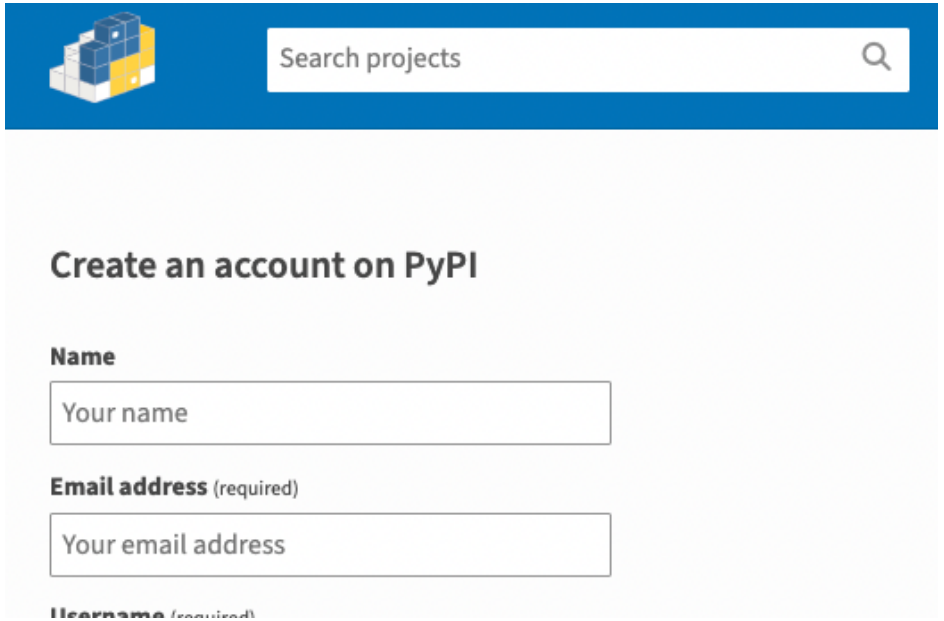
- Python eggs (`.egg`) predate the wheel (`.whl`) format.
- The wheel format solves various challenges with the egg format.
- Wheels are an accepted standard via PEP-0427 (eggs are generally considered legacy).

# What happens when there's no wheel?



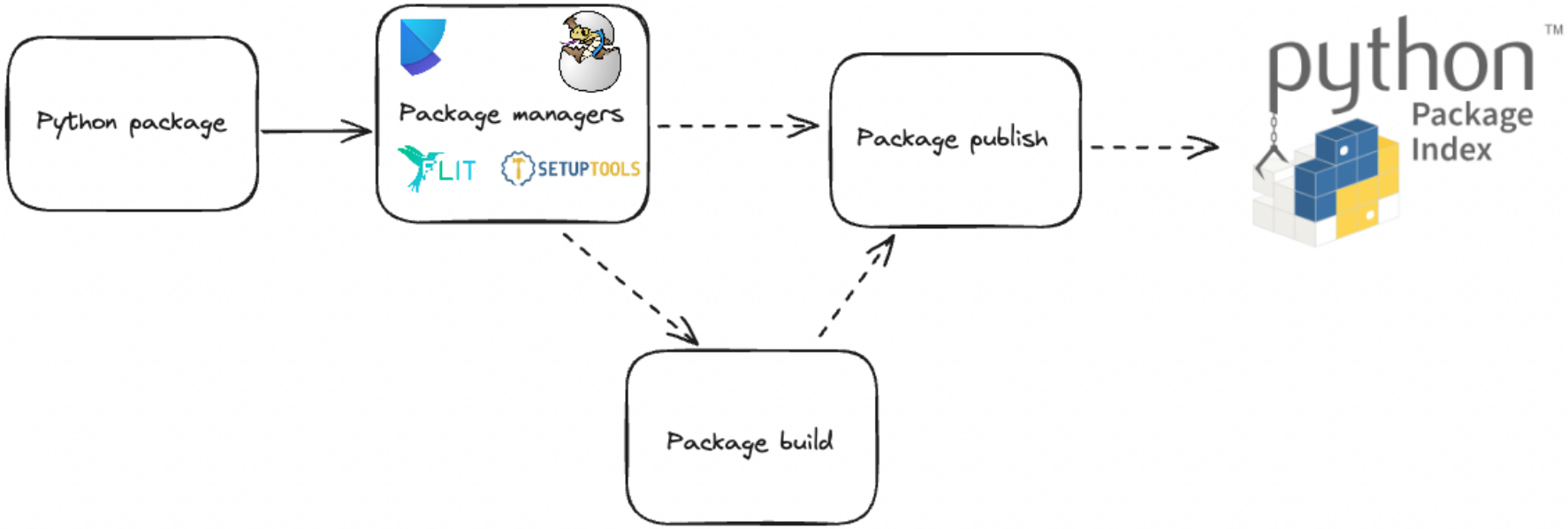

- When installing a package with no wheel (or egg) package management tools will attempt to build from source.
- Generally this can involve additional build dependencies which may need manual configuration ahead of time.

# How can I use PyPI?



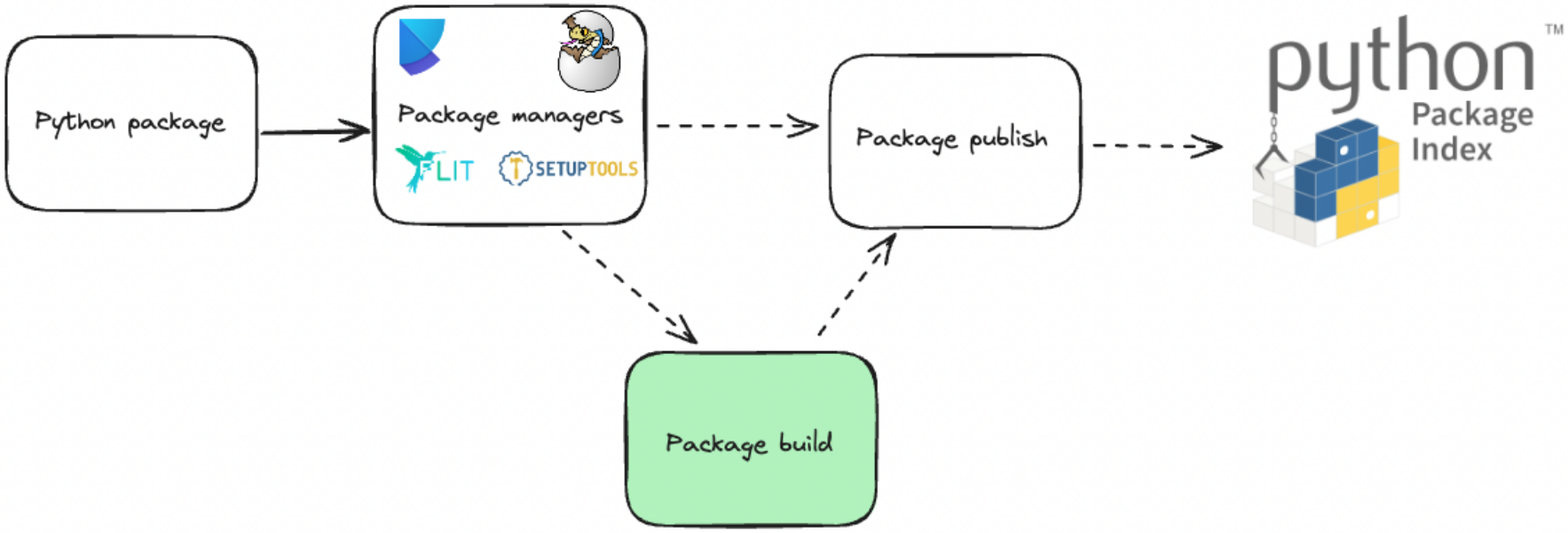

- You must have a user account (free) in order to manage packages on PyPI.
- Registration: https://pypi.org/account/register/
- Be ready with a 2FA (2-factor authentication) method (required).

# How do I send my package to PyPI?




- We can use Python package managers like Poetry, Hatch, Flit, or setuptools to help build and publish our packages to PyPI.

# Package builds




- We'll cover the package build process with Poetry first.

# Poetry Package Example Structure

```
user@machine % poetry new --name=package_name --src .
Created package package_name in .

user@machine % tree .
.
├── README.md
├── pyproject.toml
├── src
│   └── package_name
│       └── __init__.py
└── tests
    └── __init__.py
```

After installation, Poetry gives us the ability to initialize a directory structure similar to what we presented earlier by using the `poetry new ...` command.

# Poetry Package `pyproject.toml`

```toml
# pyproject.toml
[tool.poetry]
name = "package-name"
version = "0.1.0"
description = ""
authors = ["username <email@address>"]
readme = "README.md"
packages = [{include = "package_name", from = "src"}]

[tool.poetry.dependencies]
python = "^3.11"

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

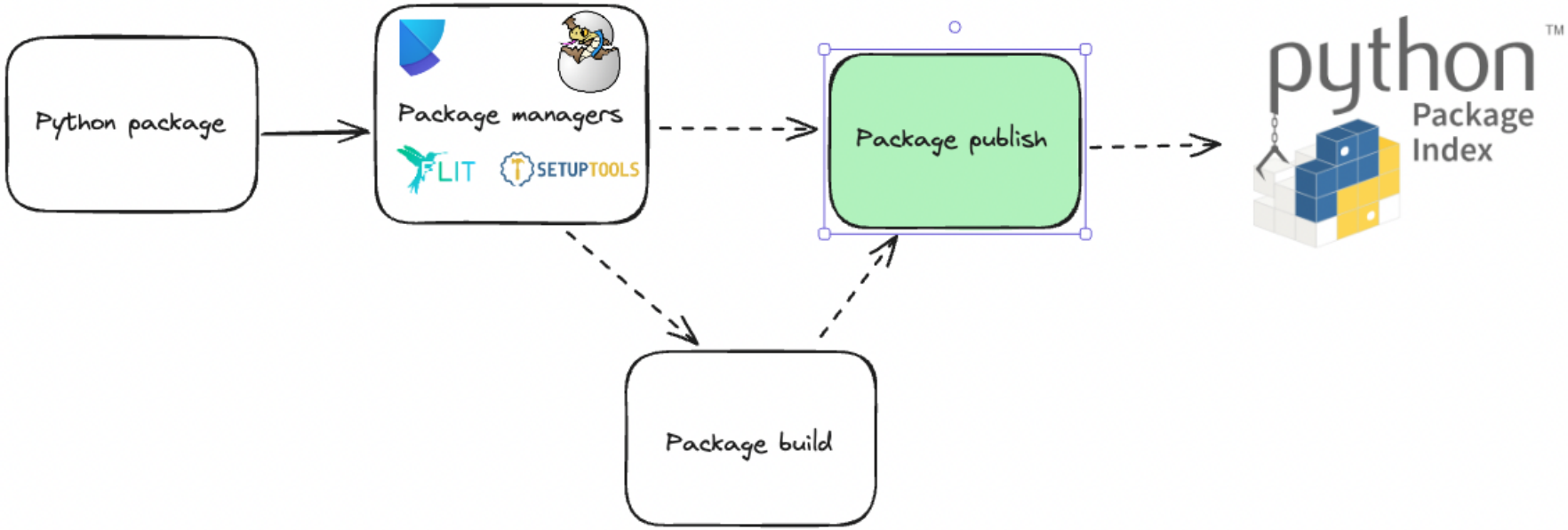
Using this command also initializes the content of our `pyproject.toml` file with details.

# Poetry's `build` command

```
$ poetry build
Building package-name (0.1.0)
  - Building sdist
  - Built package_name-0.1.0.tar.gz
  - Building wheel
  - Built package_name-0.1.0-py3-none-any.whl
$ tree .
.
├── README.md
├── dist
│   ├── package_name-0.1.0-py3-none-any.whl
│   └── package_name-0.1.0.tar.gz
├── pyproject.toml
├── src
│   └── package_name
│       └── __init__.py
└── tests
    └── __init__.py
```

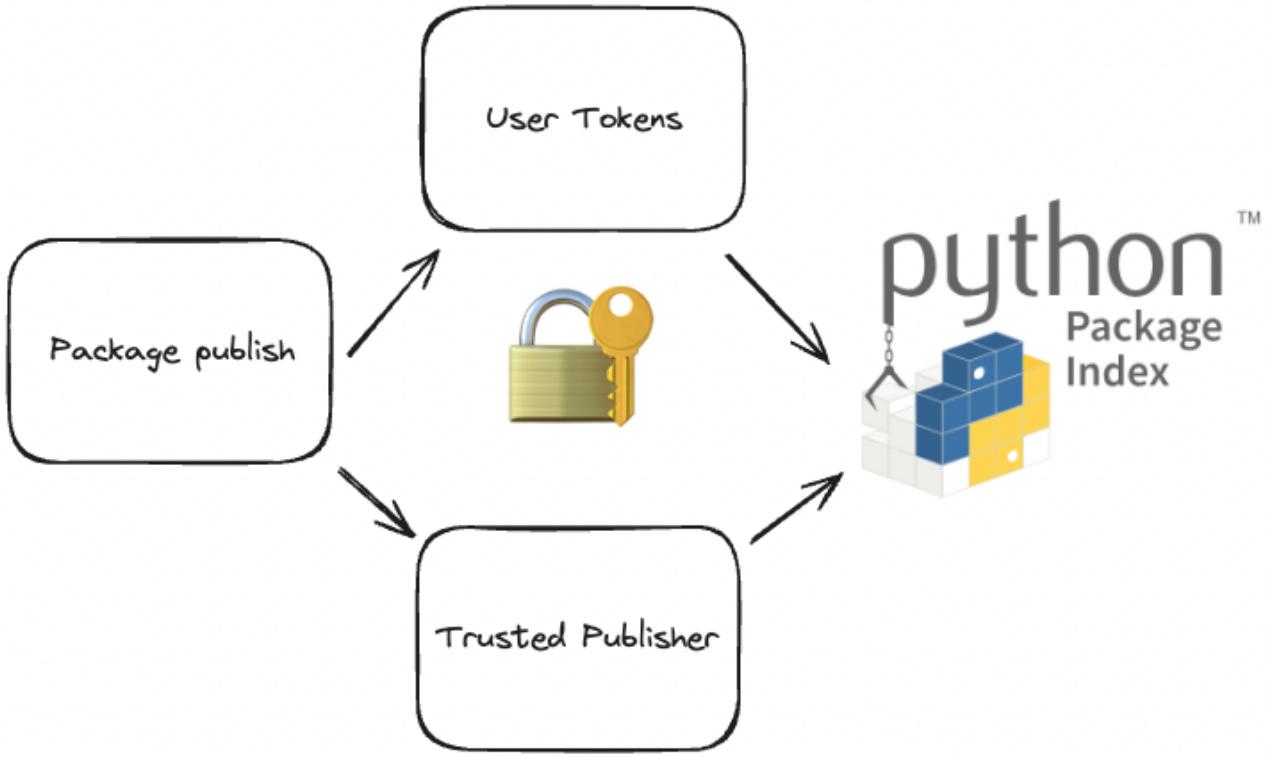
Poetry's `build` command will create a source code distribution tarball (`.tar.gz`) and wheel (`.whl`) files for the package. The build is based on the `pyproject.toml` file.

# Package Publishing to PyPI



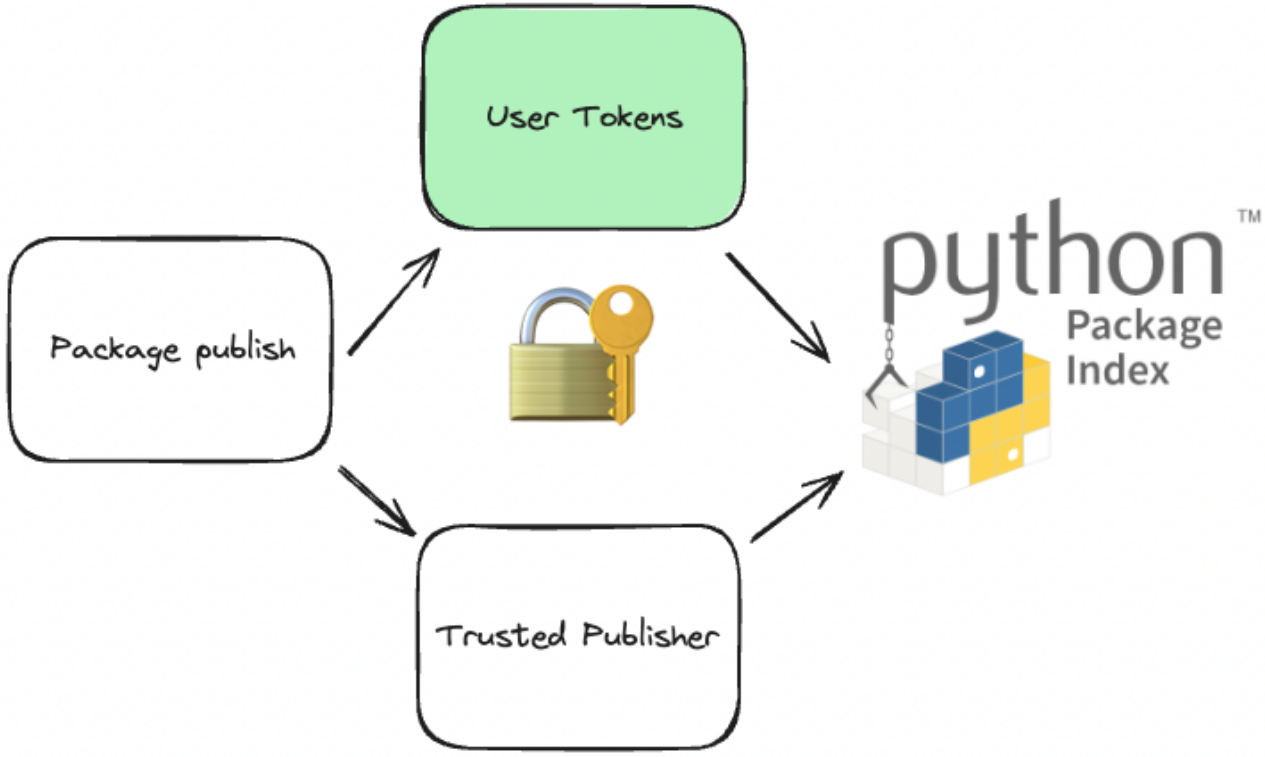

- Next we'll cover the package publishing process, where we send the built package to PyPI.

# PyPI Publishing Security



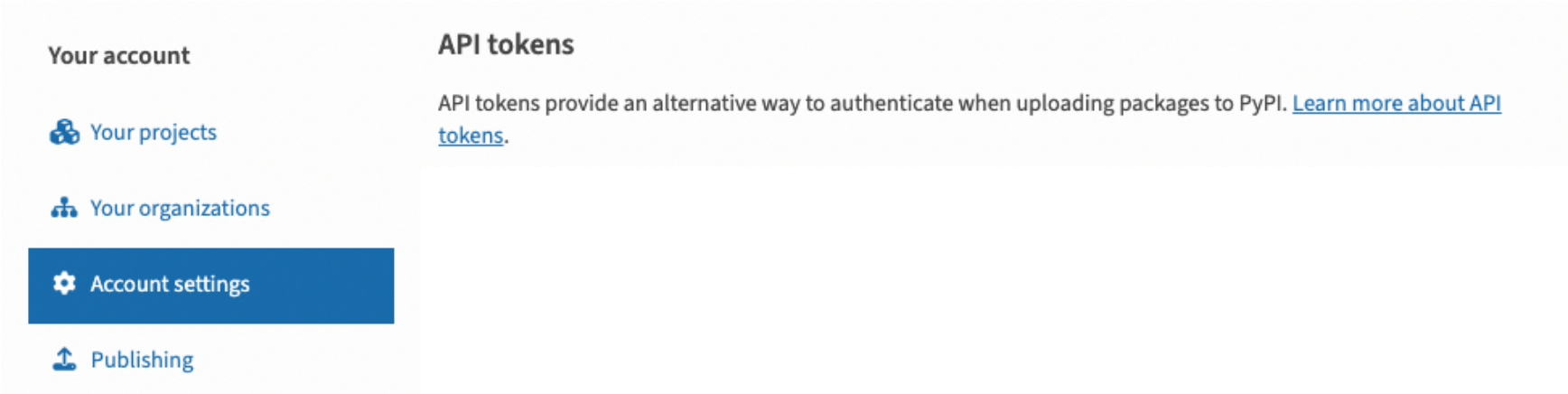

PyPI requires user tokens or trusted publisher authorization (more on this later) in order to publish packages.

# PyPI Publishing Security




We'll talk about token-based publishing through Poetry first.

# PyPI publishing Tokens




First, you need to create a scoped token from PyPI from your account settings.
You'll copy the contents of the token for use within Poetry's configuration.

# Poetry configuration with tokens

```
$ poetry config pypi-token.pypi <my-token>
```

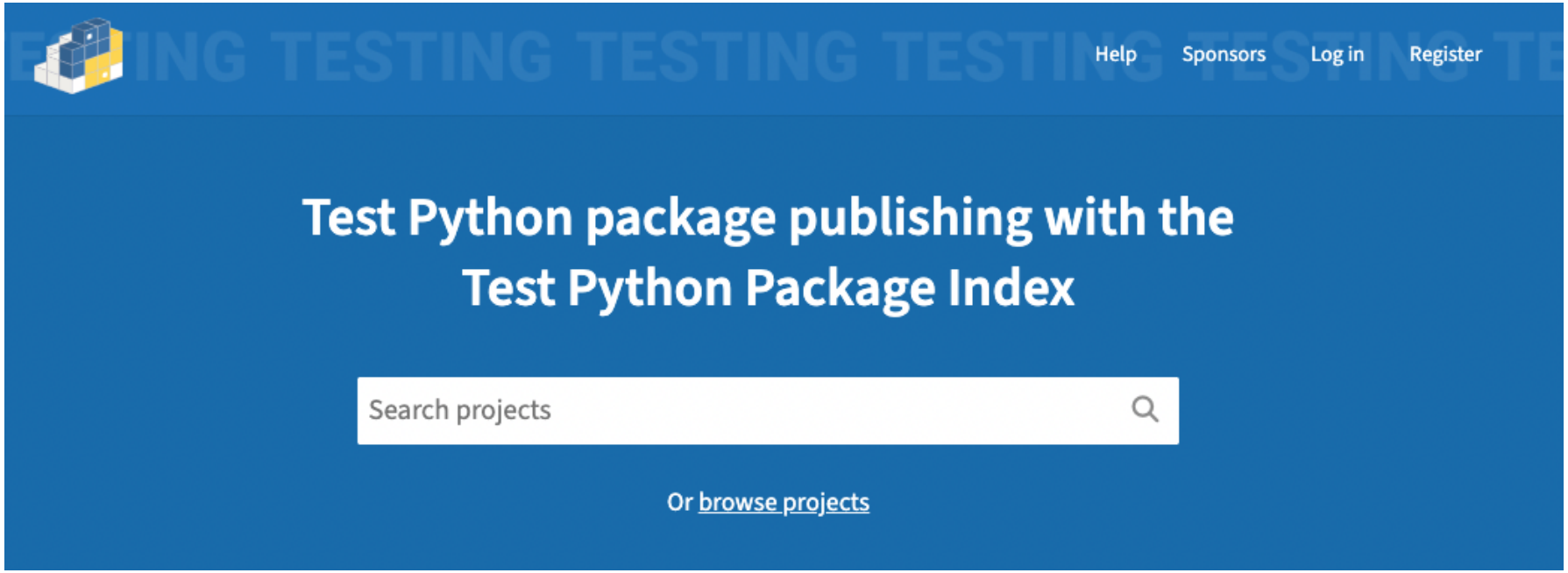
You then can add the token for Poetry to access using the `config` command.

# Poetry's `publish` command

```
$ poetry publish
Publishing package-name (0.1.0) to PyPI
 - Uploading package-name-0.1.0.tar.gz 100%
 - Uploading package-name-0.1.0-py3-none-any.whl 100%
```

- After the token has been configured, you can use the `publish` command to push the build to PyPI.
- Doing this for the first time for a package is what creates the package on PyPI.

# PyPI's Test Instance



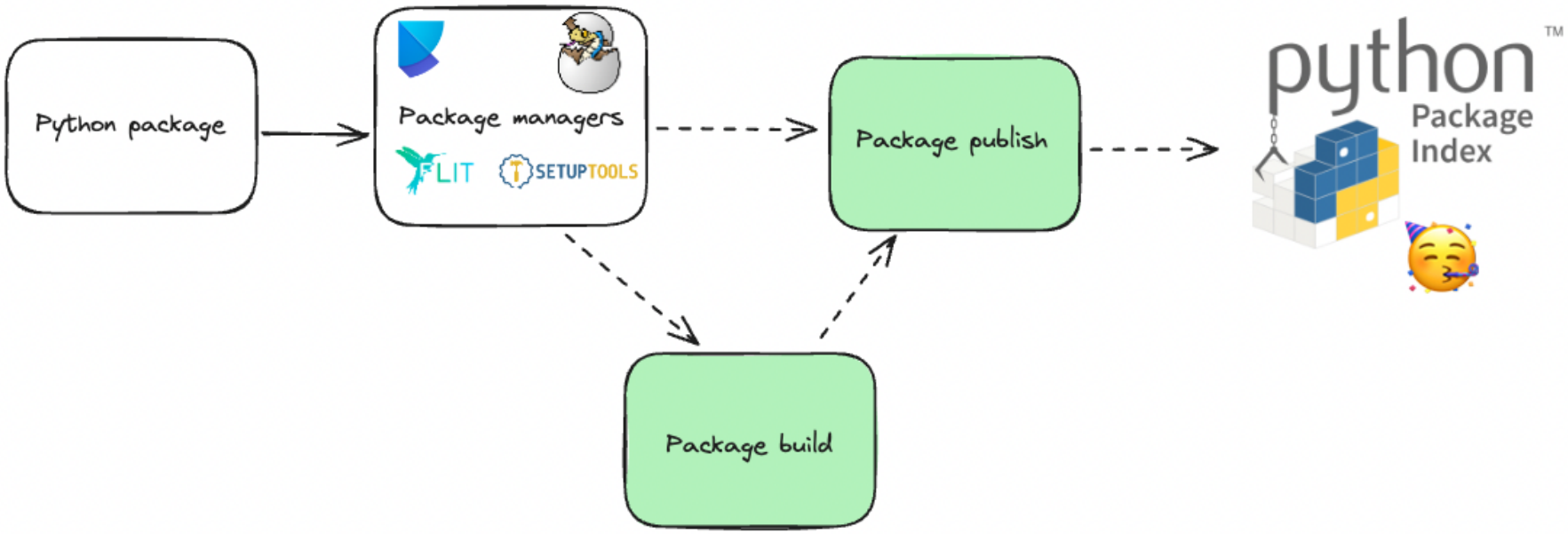

Note: there's also a test instance for PyPI (https://test.pypi.org/) which can be used to test your build and publish procedures before "production" changes. This can help avoid surprises in production! 😅

# PyPI's Test Instance Usage

```
# add the test-pypi token to poetry configuration
$ poetry config pypi-token.testpypi <your-token>
# publish to test-pypi from poetry
$ poetry publish -r test-pypi
```

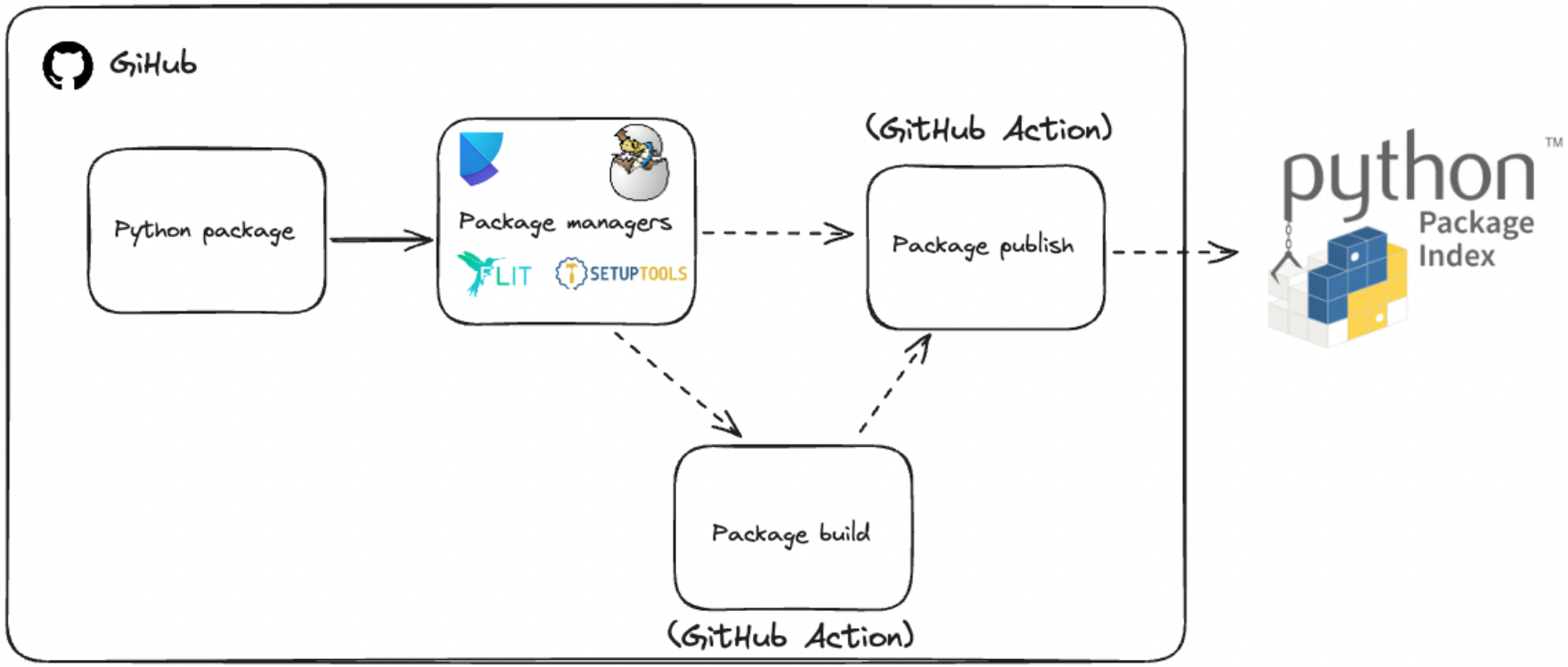
The test instance of PyPI requires similar setup steps including separate token, Poetry configuration, and an additional `publish` argument ("production" PyPI is the default for publishing).

# PyPI's Test Instance Usage



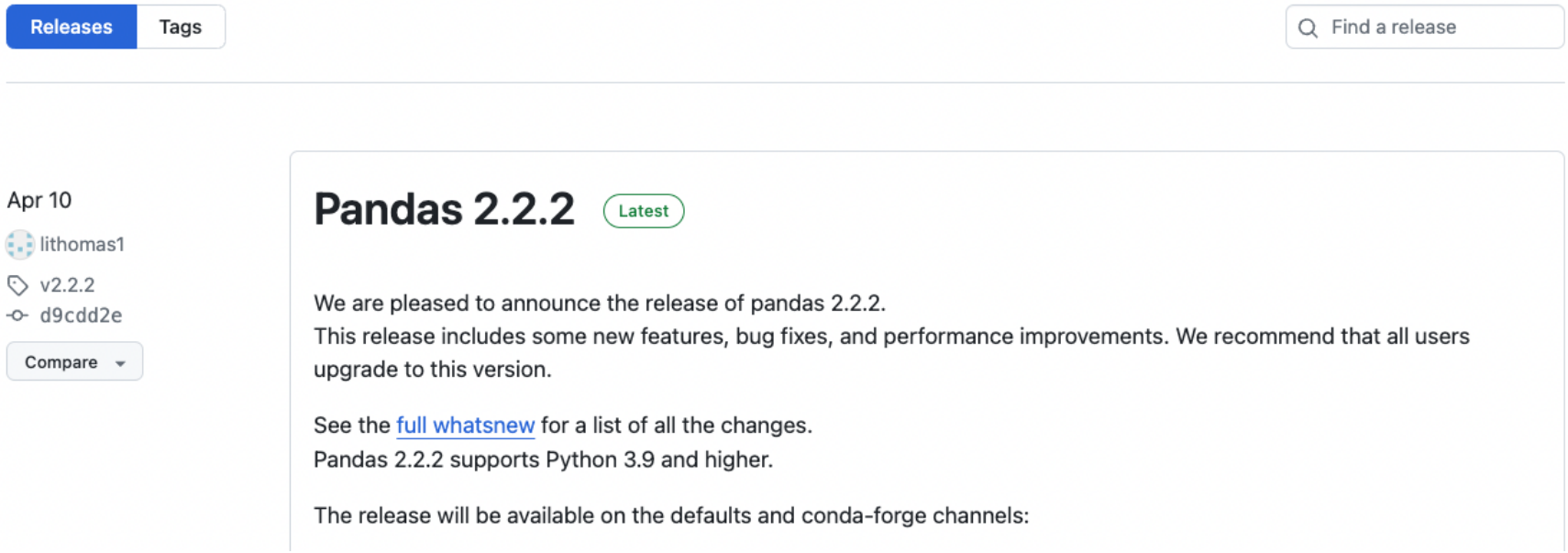

Hopefully at this point during your publishing process you can celebrate - the package is now on PyPI! (be sure to check!)

# GitHub Publishing Tools



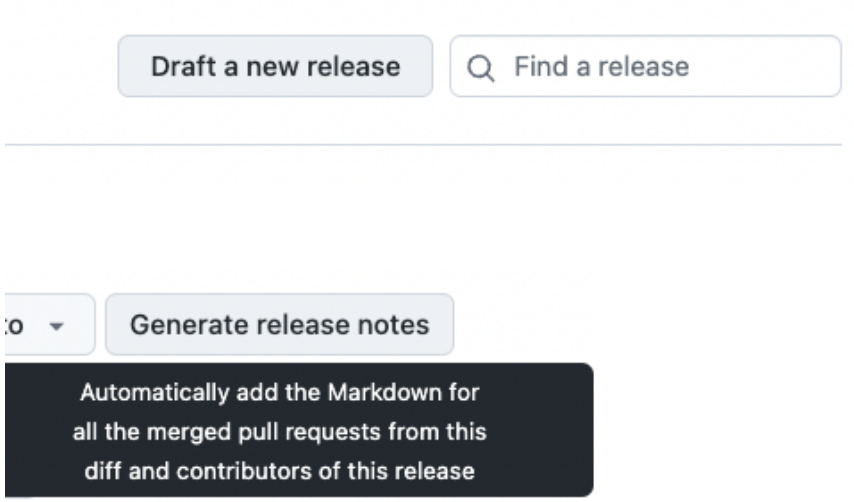

GitHub includes tools which can help to automate this process from your repository (removing some of the manual steps).

# GitHub Releases



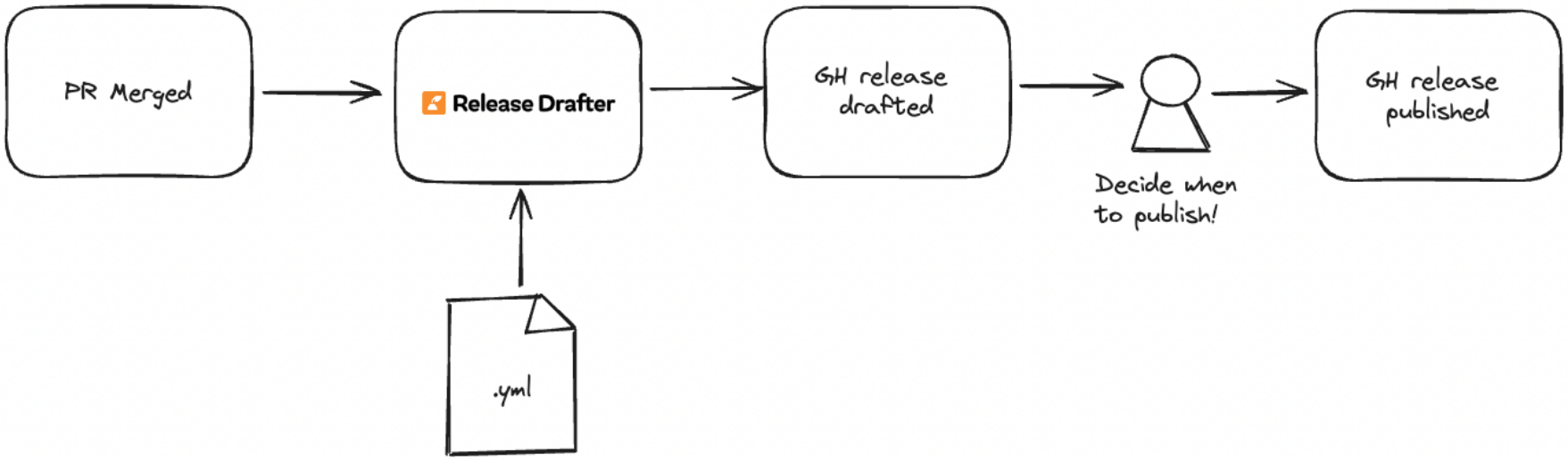

- GitHub releases are a way to package and distribute software releases from your repository.
- Each release will generally be tied to a commit, include source code, and release notes for the community.

# GitHub Releases



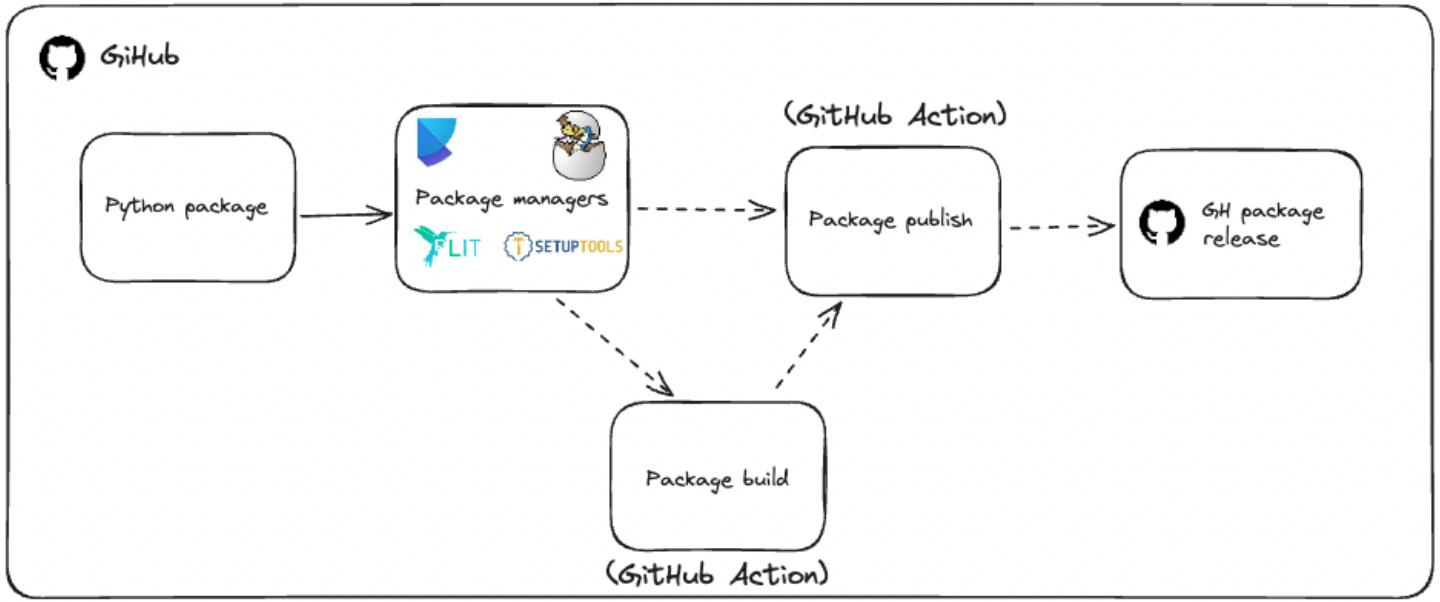

- GitHub releases can be drafted manually using the “Draft new release” button.
- You can select a tag (usually a semantic version) and generate release notes based on changes using the “Generate release notes” button.

# Automation with `release-drafter`



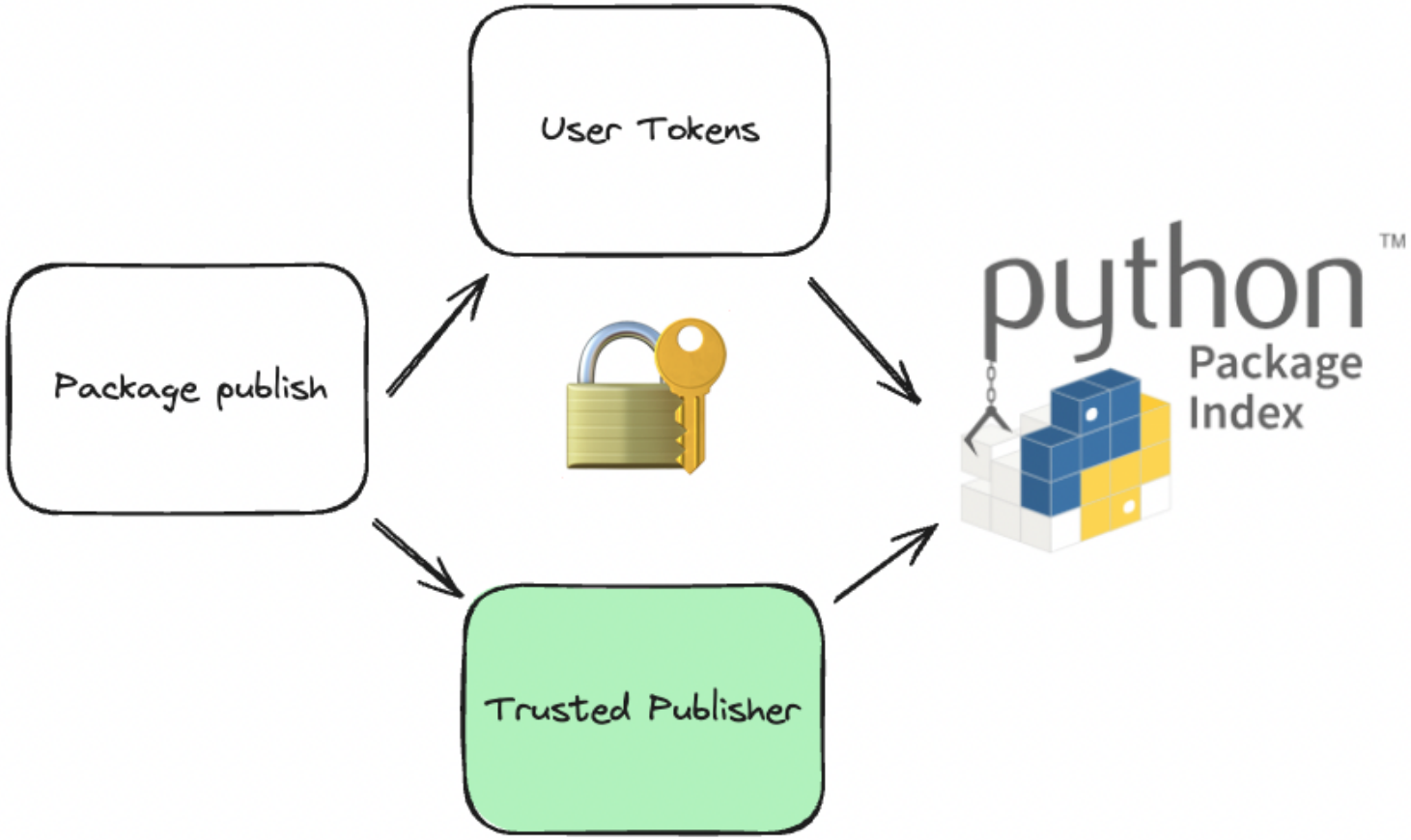

GitHub releases can automatically be drafted after pull requests are merged using `release-drafter`.
It uses `.yaml` configuration and GitHub Actions.

# GitHub Releases vs PyPI



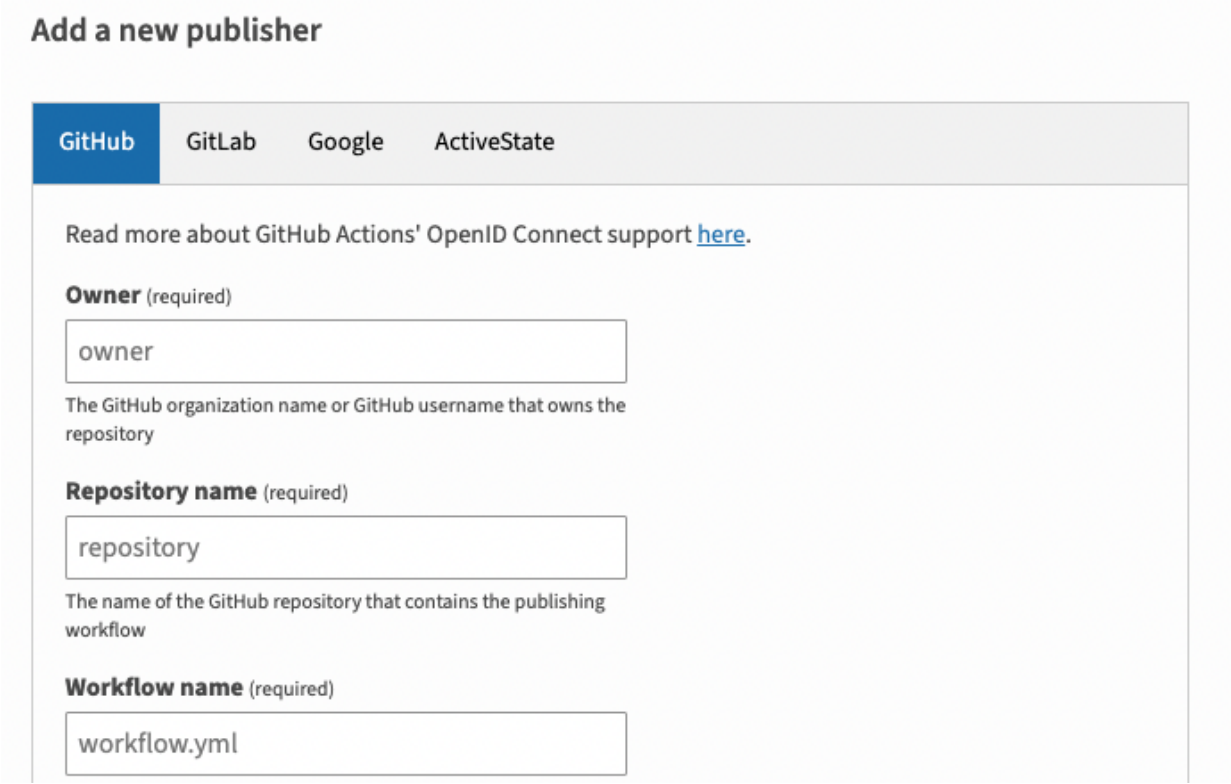

Note: GitHub releases are separate from PyPI releases. Generally both are expected and good to have with a package.

# Publishing to PyPI from GitHub



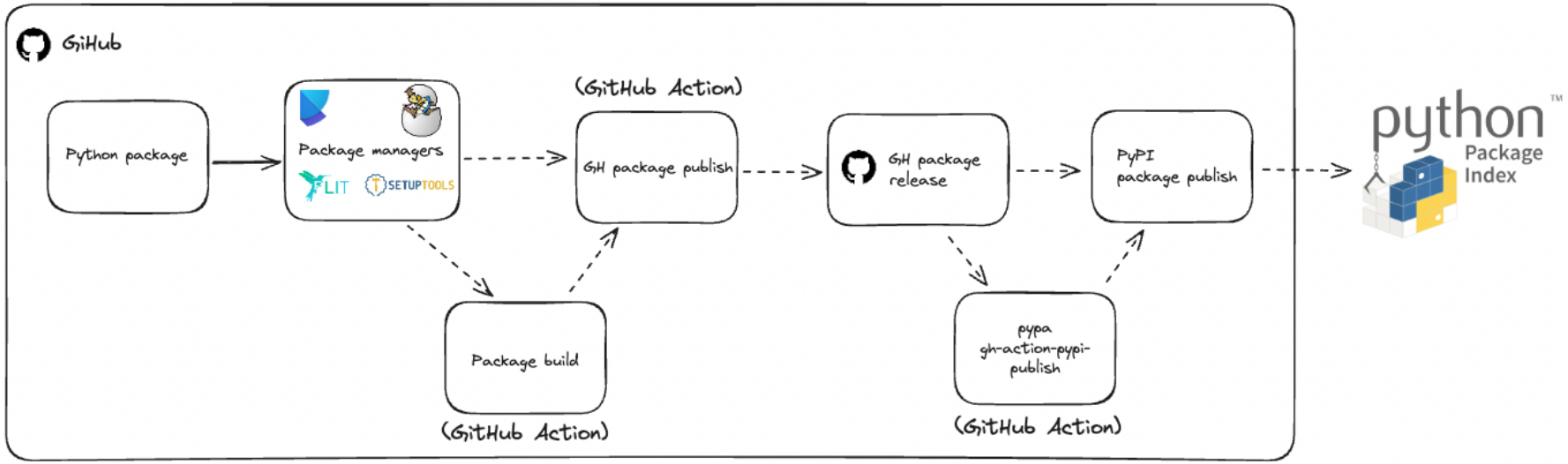

We can publish to PyPI from GitHub using the preferred "Trusted Publisher" authentication.

# Publishing to PyPI from GitHub




We configure trusted publishing settings with various details using the PyPI package webpage.

# Publishing to PyPI from GitHub




After trusted publishing settings are configured, one needs to add related source code files in alignment with the trusted publishing settings (workflow name, etc).

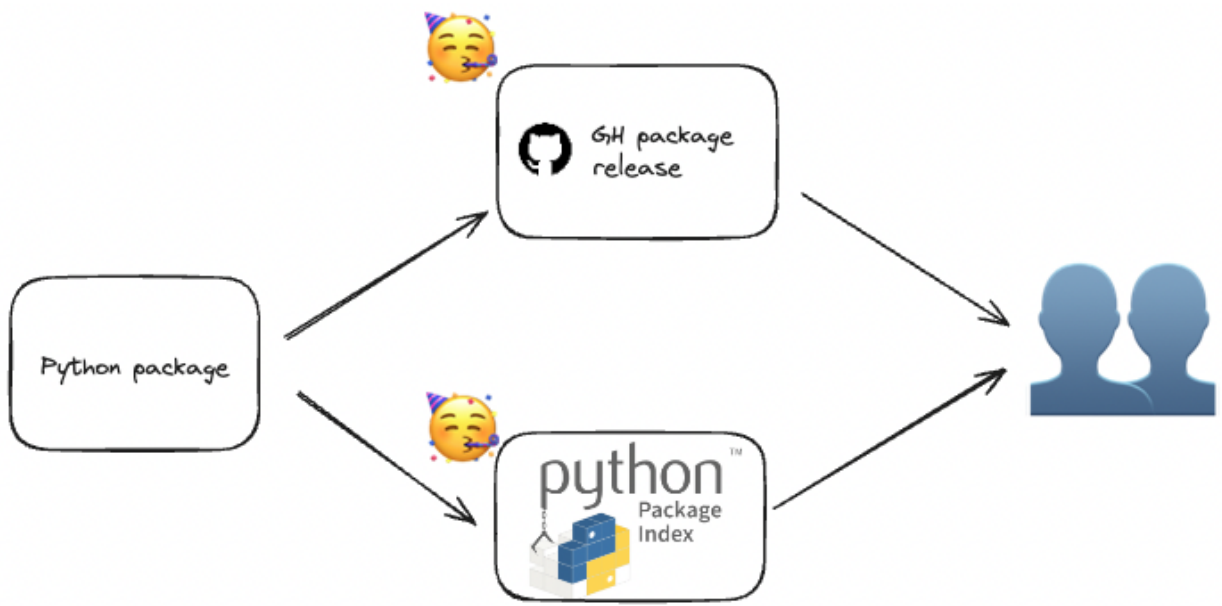
# Publishing to PyPI from GitHub

```yaml
jobs:
  pypi-publish:
    name: upload release to PyPI
    runs-on: ubuntu-latest
    # Specifying a GitHub environment is optional, but strongly encouraged
    environment: release
    permissions:
      # IMPORTANT: this permission is mandatory for trusted publishing
      id-token: write
    steps:
      # retrieve your distributions here

      - name: Publish package distributions to PyPI
        uses: pypa/gh-action-pypi-publish@release/v1
```

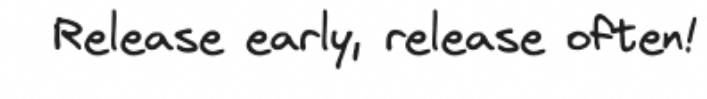
Trusted publishing leverages the `pypa/gh-action-pypi-publish` GitHub Action with little additional configuration beyond the files.

# Celebrate your Releases!



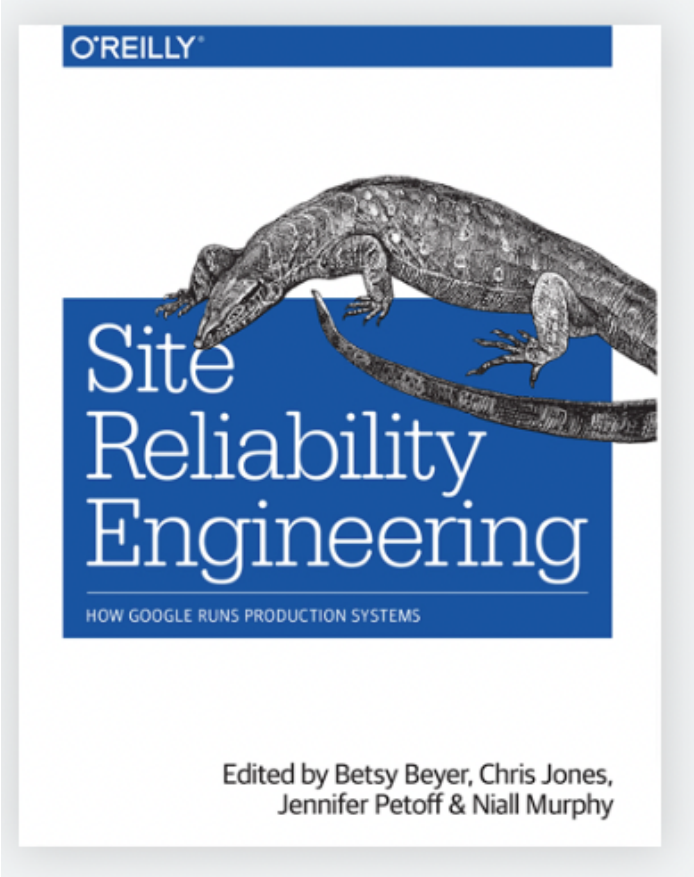

Celebrate your wins! They help connect your software with others.

# Additional thoughts




A common mantra: ***release early, release often!***. Source: Eric S. Raymond, *The Cathedral and the Bazaar* (link).

# Additional thoughts




Engineers at Google wrote about a ***"Release Engineer"*** role in *Site Reliability Engineering* (link).

# Thank you!

Thank you for attending! Questions / comments?

Please don't hesitate to reach out!

- </> CU Anschutz DBMI SET Team
- @d33bs