Fearless Automation that runs Anywhere

With Python

Padraic Calpin - 🇮🇪/🇬🇧, living in 🇸🇮, working in 🇧🇪

What are we automating?

- Everything outside the application code

  More specifically, everything that happens after git push

- ‘Pipelines’ applied to every code change

  Steps like linting, testing, building, deploying

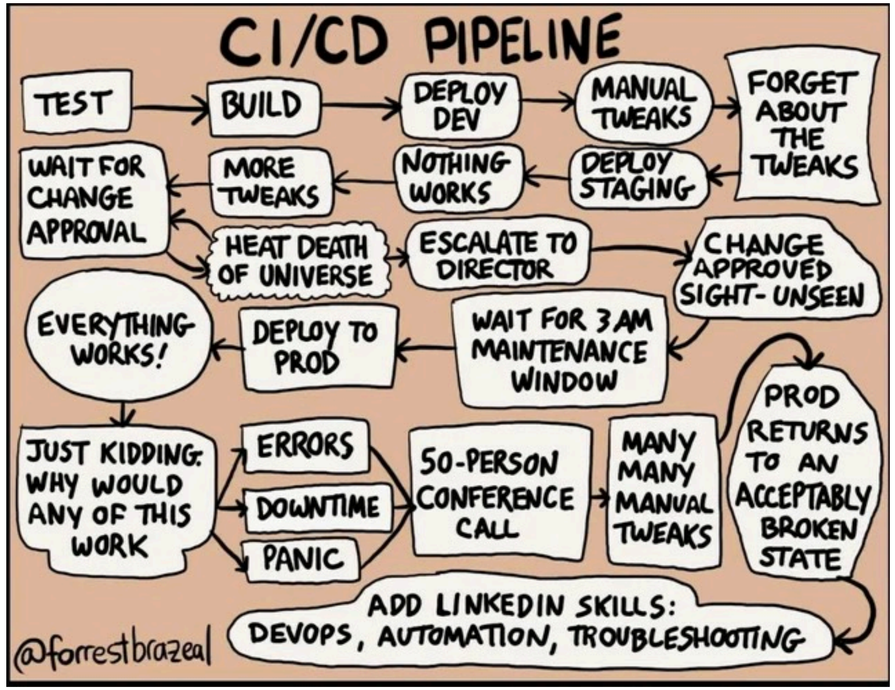
  Typically happening in remote runtimes

What are we automating?

CI/CD Pipelines - Everyone's favourite!

- By reputation: complex, flakey, slow, opaque
- Arises from a mix of

  'Single purpose' / one-off code chunks



What are we automating?

CI/CD Pipelines - Everyone's favourite!

- By reputation: complex, flakey, slow, opaque
- Arises from a mix of

  'Single purpose' / one-off code chunks

  Configuring the CI/CD runtime (often in YAML)

What are we automating?

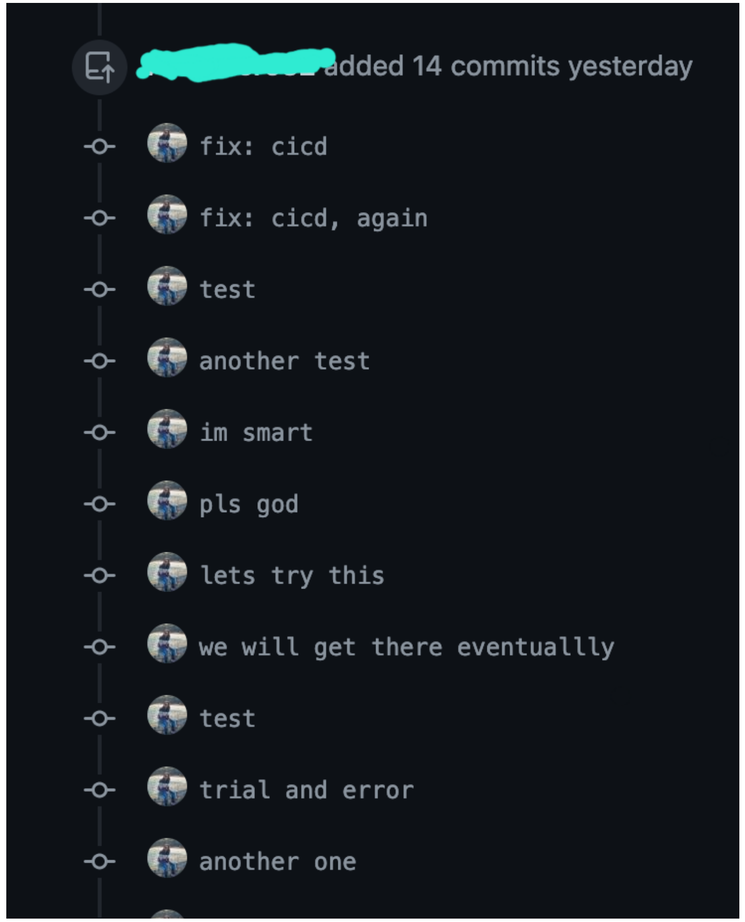
CI/CD Pipelines - Everyone's favourite!

- By reputation: complex, flakey, slow, opaque
- Arises from a mix of

  'Single purpose' / one-off code chunks

  Configuring the CI/CD runtime (often in YAML)

  'Black box' runtimes and plugins



zensor®

What are we automating?

CI/CD Pipelines - Everyone's favourite!

- By reputation: complex, flakey, slow, opaque
- Arises from a mix of

  'Single purpose' / one-off code chunks

  Configuring the CI/CD runtime (often in YAML)

  'Black box' runtimes and plugins
- Induces a 'Push and Pray' mentality

Against ‘Push and Pray’

Ensuring Portability and Reproducibility

- ‘Elevate’ our pipeline code
  - Treat pipelines holistically
  - Use good practices: Testing, code-reuse…
- Decouple ourselves from the Runtime
  - Code that can be executed anywhere
  - Quicker Feedback
  - Redundancy

zensor®

Against ‘Push and Pray’

Containerisation

- Bundle workflow steps into containers

  Isolated from underlying system

- Programmatic API to define and link steps
- Code re-use and portability are explicit goals

Build your modern software factory.

Define software delivery workflows and dev environments with reusable components — including LLMs — and run them anywhere. Built by the creators of Docker.

Against ‘Push and Pray’

Containerisation isn’t needed?

- Bundle workflow steps into containers
  - Isolated from underlying system
- Programmatic API to define and link steps
- Code re-use and portability are explicit goals
- However:
  - Glue code still required
  - Containers often replicate the underlying environment

Build your modern software factory.

Define software delivery workflows and dev environments with reusable components — including LLMs — and run them anywhere. Built by the creators of Docker.

zensor®

Let’s do Python!

Python pipelines that run almost identically in any runtime

Let’s do Python!

- Cross-platform language with powerful scripting capabilities
- Huge ecosystem of existing libraries and solutions available to leverage

Let's do Python!

Making version management ~~painless~~ less painful

- Cross-platform language with powerful scripting capabilities
- Huge ecosystem of existing libraries and solutions available to leverage
- **Easier than ever to ensure consistent environments**

  Easy installation via `python-build-standalone`

  Modern "workflow" tooling abstracts away dependency and virtual environment management

Let's do Python!

Principles for Creating Pipelines in Code

- **Keep pipeline code isolated from application code**

  e.g. `pipelines` package in the repository root, with executable modules

  Allows easy code re-use between pipelines

  Single, consistent location vs scattered scripts with PEP723

Let's do Python!

Principles for Creating Pipelines in Code

- **Delegate Workflow Management to Tools**

  Project Tool e.g. `hatch`, `uv`, `poetry`

  Invocation, dependency management

  Command Runner e.g. `just`, `make`

  Short aliases and linking workflows

```
uv run --env-file .env python -m pipelines.test
```

```
test:
    uv run --env-file .env python -m pipelines.test
just test
```

Let's do Python!

Principles for Creating Pipelines in Code

- **Keep as much logic as possible inside Python itself**

  Use high-level abstractions Python provides like `pathlib`

  Prefer native libraries over subprocess calls e.g. `gitpython` vs. `git`

  Leverage our tools maximally e.g. controlling test execution using marks and hooks in `pytest`

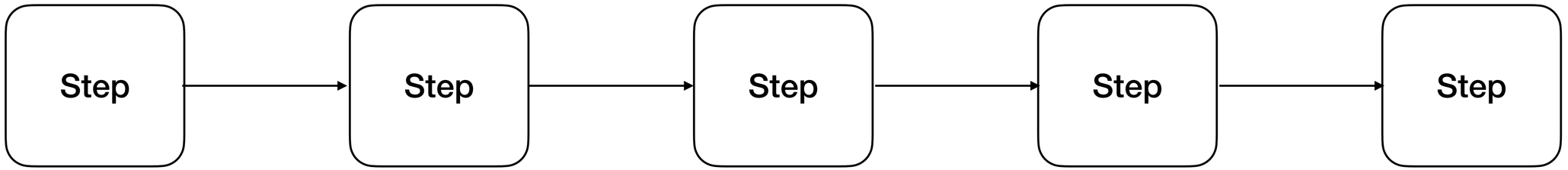
Against ‘Push and Pray’

Ensuring Portability and Reproducibility

- ‘Elevate’ our pipeline code - **In Python**
  - Treat pipelines holistically
  - Use good practices: Testing, code-reuse…
- Decouple ourselves from the Runtime - **??**
  - Code that can be executed anywhere
  - Quicker Feedback
  - Redundancy

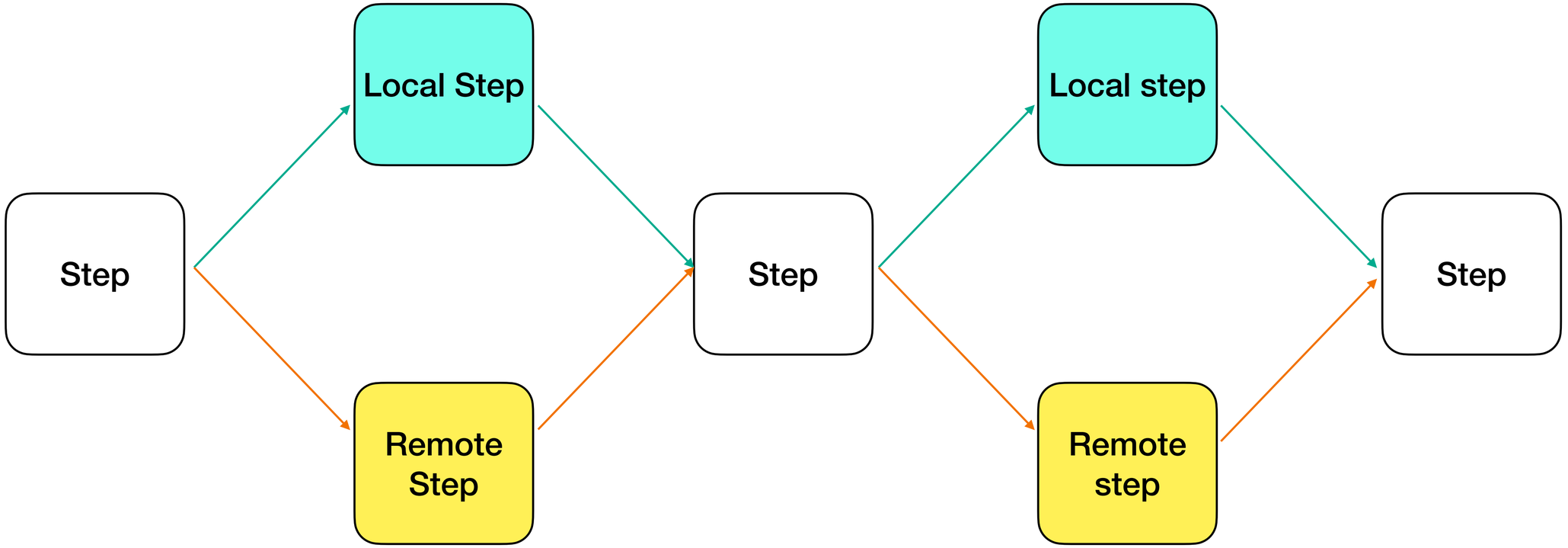
zensor®

Creating Robust Pipelines



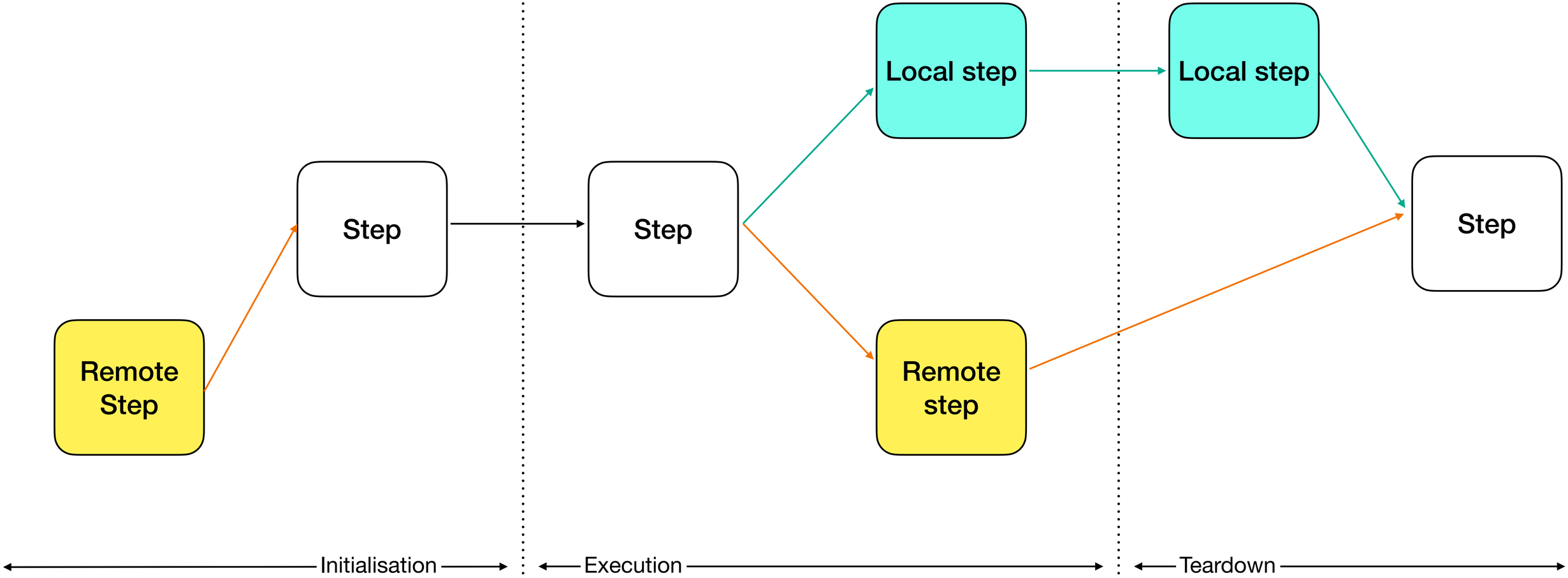
Creating Robust Pipelines

Branching for Local and Remote



Creating Robust Pipelines

Branching for Local and Remote



Creating Robust Pipelines

Branching for Local and Remote

- Branches handle irreconcilable differences between local and remote cases

  Use them to reconcile these differences

- Keep branches as short as possible

  Minimise divergences to minimise the code surface

- Conditionals and tests should be simple and foolproof

Creating Robust Pipelines

Defining your runtime environments

- What assumptions can you make about where your pipelines will run?

  E.g. Python version, OS, shell, installed programmes, auth methods…

- **Locally**: Defined by IT, dev guides, project `CONTRIBUTING.md` etc.
- **Remotely:** A function of your runtime provider

Creating Robust Pipelines

Defining your remote environment

- Treat as distributed runner, integrated with VCS
- Be selective what features you use

  Eschew workflow plugins, secret storage, etc.
- **Limit remote-only steps to setup**

  Bring remote runtime into parity with local case

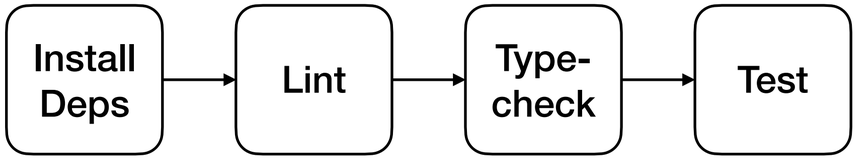
```yaml
name: "Check and Deploy"

on:
  workflow_call:

env:
  <<: *COMMON_ENV

jobs:
  test:
    name: Run tests
    runs-on: ubuntu-latest
    steps:
      - *SETUP_STEPS
      - name: Run the test pipeline
        run: "just test"
  deploy:
    name: Deploy
    runs-on: ubuntu-latest
    needs: test
    permissions:
      id-token: write
      contents: read
    steps:
      - *SETUP_STEPS
      - name: Run the build pipeline
        run: "just build"
      - name: Run the deploy pipeline
        run: "just deploy"
```

Creating Robust Pipelines

Outlining workflows

Begin with a high level description e.g.

Testing:

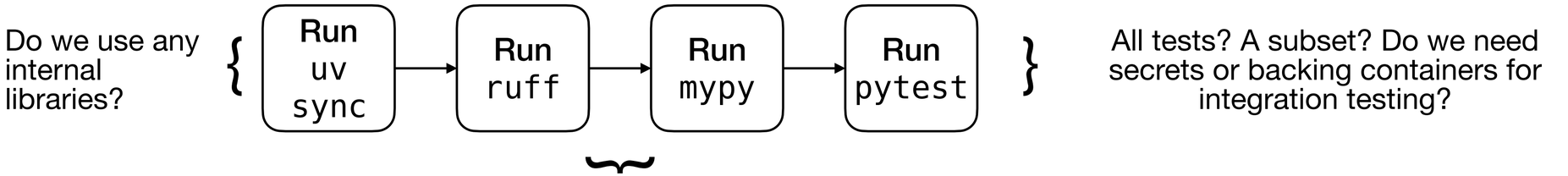


Creating Robust Pipelines

Outlining workflows and requirements

Decompose steps, adding technical detail **and requirements**

Testing:

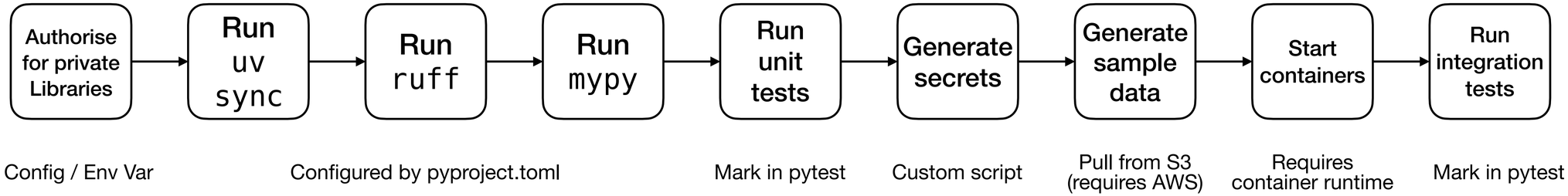


Creating Robust Pipelines

Outlining workflows and requirements

Decompose steps, adding technical detail **and requirements**

Testing



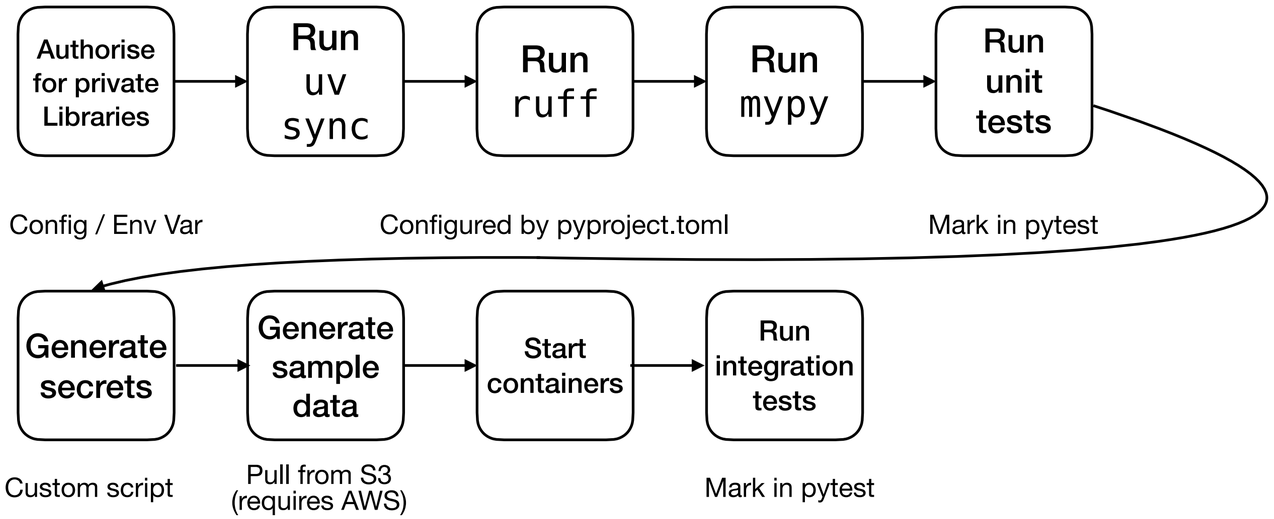
Creating Robust Pipelines

Defining your runtime environment - Example

Requirement	Local Assumption	Remote Assumption
Python	At least 3.12 present	3.13, installed via CI/CD plugin
Dependency Management	UV, globally installed	UV, setup via CI/CD plugin
AWS Auth	SSO via AWS CLI, pre-configured	Uses OIDC via vendor plugin
Git Auth	Uses SSH Keys	Uses OIDC via vendor plugin
Tool config e.g. Ruff	N.A. (in repository code)	
Container Runtime	Installed on dev machine	Present in base runtime

zensor®

Examples



Requirement	Local Assumption	Remote Assumption
Python	At least 3.12 present	3.13, installed via CI/CD plugin
Dependency Management	UV, globally installed	UV, setup via CI/CD plugin
AWS Auth	SSO via AWS CLI, pre-configured	Uses OIDC via vendor plugin
Git Auth	Uses SSH Keys	Uses OIDC via vendor plugin
Tool config e.g. Ruff	N.A. (in repository code)	
Container Runtime	Installed on dev machine	Present in base runtime

Examples

Private PyPI Authentication

Requirement	Local Assumption	Remote Assumption
Dependency Management	UV, globally installed	UV, setup via CI/CD plugin

‘Extra Indices’ configurable via file or environment variable

Locally: Create a per-user config file

Remote: Use a CI/CD ‘secret’

```yaml
name: "Lint and Test"

on:
  workflow_call:

env:
    UV_EXTRA_INDEX_URL: ${{ secrets.PYPI_URL }}
    PIP_EXTRA_INDEX_URL: ${{ secrets.PYPI_URL }}/simple

jobs:
  ...
```

In Pipeline: uv automatically includes the index

Examples

Cloud Vendor Authentication

Requirement	Local Assumption	Remote Assumption
AWS Auth	SSO via AWS CLI, pre-configured	Uses OIDC via vendor plugin

Authenticating calls to AWS, made from Python via boto3

Locally: boto3 natively understands SSO, no work needed

Remotely: Call out to vendor plugin, which sets required environment variables

```yaml
name: "Lint and Test"

on:
  workflow_call:

env:
    AWS_DEFAULT_REGION: ${{ secrets.AWS_REGION }}

jobs:
  test:
    name: Run Project Tests
    runs-on: ubuntu-latest
    steps:
      ...
      - name: Configure AWS Credentials
        uses: aws-actions/configure-aws-credentials@v4
        with:
          role-to-assume: ${{ secrets.project_deployment_role }}
          aws-region: ${{ env.AWS_DEFAULT_REGION }}
```

zensor®

Examples

Cloud Vendor Authentication

Requirement	Local Assumption	Remote Assumption
AWS Auth	SSO via AWS CLI, pre-configured	Uses OIDC via vendor plugin

Authenticating calls to AWS, made from Python via boto3

Locally: boto3 natively understands SSO, no work needed

Remotely: Call out to vendor plugin, which sets required environment variables

```
import boto3

# Rely on internal mechanisms to automatically
# identify credentials in the right way
SSM_CLIENT = boto3.client("ssm")
...
```

In Pipelines: Call boto3 directly as needed!

Examples

Conditional Test Execution

Requirement	Local Assumption	Remote Assumption
Dependency Management	UV, globally installed	UV, setup via CI/ CD plugin
Tool config e.g. Ruff	N.A. (in application code)	

Running steps based on changed files

Combining test config with GitPython, to identify changed files

```
def find_changed_files(pytest_dir: Path) -> set[str]:
    repo = Repo(pytest_dir.parent)
    current_branch = repo.active_branch
    if current_branch == 'master':
        diff = repo.index.diff("HEAD~1")
    else:
        diff = repo.index.diff("master")
    return {
        change.a_path for change in diff
    }
```

zensor®

Examples

Conditional Test Execution

Requirement	Local Assumption	Remote Assumption
Dependency Management	UV, globally installed	UV, setup via CI/ CD plugin
Tool config e.g. Ruff	N.A. (in application code)	

Running steps based on changed files

Combining test config with GitPython, to identify changed files

```python
SKIP_MARK = "unchanged_files"
WATCHED_FILES = {...}

def pytest_collection_modifyitems(config: pytest.Config,
                                  items: list[pytest.Item]):
    changed_files = find_changed_files(config.rootpath)
    if not WATCHED_FILES & changed_files:
        for test in filter(lambda i: SKIP_MARK in i.keywords, items):
            test.add_marker(SKIP_MARK)
```

Examples

Conditional Test Execution

Requirement	Local Assumption	Remote Assumption
Dependency Management	UV, globally installed	UV, setup via CI/ CD plugin
Tool config e.g. Ruff	N.A. (in application code)	

Running steps based on changed files

Combining test config with GitPython, to identify changed files

Locally: No extra config needed

Remotely: Ensure full project history is checked-out

In Pipeline: Pytest automatically skips tests during ‘test discovery’ phase of execution

```python
def find_changed_files(pytest_dir: Path) -> set[str]:
    repo = Repo(pytest_dir.parent)
    current_branch = repo.active_branch
    if current_branch == 'master':
        diff = repo.index.diff("HEAD~1")
    else:
        diff = repo.index.diff("master")
    return {
        change.a_path for change in diff
    }



SKIP_MARK = "unchanged_files"
WATCHED_FILES = {...}

def pytest_collection_modifyitems(config: pytest.Config,
                                  items: list[pytest.Item]):
    changed_files = find_changed_files(config.rootpath)
    if not WATCHED_FILES & changed_files:
        for test in filter(lambda i: SKIP_MARK in i.keywords, items):
            test.add_marker(SKIP_MARK)
```

Examples

Interacting with Containers

Requirement	Local Assumption	Remote Assumption
Container Runtime	Installed on dev machine	Present in base runtime

Switch between libraries and `subprocess` calls, based on use-case

E.g. Backing services for integration tasting

Leverage the `testcontainers` package to setup and teardown containers in Python code

Possible to integrate directly with e.g. `pytest`

```python
from testcontainers.core.container import DockerContainer
from testcontainers.generic import ServerContainer
from testcontainers.redis import RedisContainer


class StatpingContainer(ServerContainer):

    ...

class ValkeyContainer(RedisContainer):

    ...

@contextmanager
def zmart_services():
    containers = [
        ValkeyContainer(
            VALKEY_IMAGE,
            password=os.getenv('VK_PASSWORD')
        ),
        StatpingContainer(port=8080, image=STATPING_IMAGE)
    ]
    try:
        for c in containers:
            c.start()
        yield
    finally:
        for c in containers:
            c.stop()
```

zensor®

Examples

Interacting with Containers

Requirement	Local Assumption	Remote Assumption
Container Runtime	Installed on dev machine	Present in base runtime

Switch between libraries and `subprocess` calls, based on use-case

E.g. Building containers via buildkit

- No libraries (that I'm aware of)
- Build commands and dispatch

```python
@dataclass
class DockerBuildService():
    dockerfile: Path = field(default_factory=lambda: Path() / "Dockerfile")
    image: str = ""
    version: str = ""
    platform: set[Platform] = field(default_factory=_default_platforms)
    context_dir: Path = PROJECT_DIR
    target: str = ""
    build_args: dict[str, str] = field(default_factory=dict)
    builder: str = ""

    @property
    def _docker_args(self) -> list[str]:
        args = []
        if self.builder:
            args += [
                "--builder",
                self.builder
            ]
        ...
        for k, v in self.build_args.items():
            args += [
                "--build-arg",
                f"{k}={v}"
            ]
        return args
```

Examples

Interacting with Containers

Requirement	Local Assumption	Remote Assumption
Container Runtime	Installed on dev machine	Present in base runtime

Switch between libraries and `subprocess` calls, based on use-case

E.g. Building containers via buildkit

- No libraries (that I'm aware of)
- Build commands and dispatch

```python
def build(self, context: Path, *, publish: bool):
    command = [
        "docker",
        "buildx",
        "build",
    ] + self._docker_args
    command += [
        "--tag",
        f"{self.image}:{self.version}",
        "-f",
        str(self.dockerfile),

    ]
    if publish:
        command += ["--push"]
    else:
        command += ["--load"]
    command += [str(context)]
    subprocess.run(
        command,
        check=True
    )
```

Conclusion

- How technical assumptions inform your pipeline design
- How to design for both local and remote runtimes in code

  Remote only steps end up in our runtime config…

  Local-only steps leverage our ‘assumed environment’
- How you can access the full power of Python to model common pipeline steps

Thank You For Listening!

zensor®

BACKUP

Examples

Pulling Extra Code

Requirement	Local Assumption	Remote Assumption
Git Auth	Uses SSH Keys	Uses OIDC via vendor plugin

Different methods of interacting with VCS

E.g. Custom Ansible collection

Local: Branch in code, delegating to installing over SSH

Remote: Use the CI/CD Primitives to pre-clone the requirement

In Pipeline: Ansible can access the collection

```python
import subprocess


from pipelines.env import PROJECT_DIR

COLLECTION_REPO_URL = "..."
ANSIBLE_DIR = PROJECT_DIR / "ansible"
COLLECTION_REPO_PATH = (
    ANSIBLE_DIR /
    "ansible-zensor-collections" /
    "zensor"
)

def ensure_ansible_collection():
    target = COLLECTION_REPO_URL
    if COLLECTION_REPO_PATH.exists():
        target = COLLECTION_REPO_PATH
    subprocess.check_output([
        "ansible-galaxy", "collection",
        "install", target
    ])
```

zensor®

Examples

Pulling Extra Code

Requirement	Local Assumption	Remote Assumption
Git Auth	Uses SSH Keys	Uses OIDC via vendor plugin

Different methods of interacting with VCS

E.g. Custom Ansible collection

Local: Branch in code, delegating to installing over SSH

Remote: Use the CI/CD Primitives to pre-clone the requirement

In Pipeline: Ansible can access the collection

```yaml
name: "Check and Deploy"

on:
  push:

jobs:
  deploy:
    name: Deploy
    runs-on: ubuntu-latest
    needs: lint
    permissions:
      id-token: write
      contents: read
    steps:
      - ...
      - name: Authorise to clone the Ansible collection
        id: generate_token
        uses: actions/create-github-app-token@v1
        with:
          app-id: ${{ secrets.APP_ID }}
          private-key: ${{ secrets.APP_PRIVATE_KEY }}
          owner: "ACME"
          repositories: "ansible-spam"
      - ...
```

Examples

Non-Python Tools

Requirement	Local Assumption	Remote Assumption
Terraform	>= 1.12 installed locally	1.12, setup via vendor Plugin
Git Auth	Uses SSH Keys	Uses OIDC via vendor plugin

Depends on Technical Assumptions

Consider: Terraform, with custom modules

Remote: Clone repo w/ plugin

Local: If repo absent, pipeline clones via SSH with GitPython

In Pipeline:

- Terraform looks for module on disk
- Invocation handled with subprocess

Examples

Non-Python Tools

Requirement	Local Assumption	Remote Assumption
Container Runtime	Installed on dev machine	Present in base runtime

Depends on Technical Assumptions

Consider: Terraform, with custom modules

Alternatively:

In Pipeline:

- Pull custom docker image including Terraform and dependencies
- Invoke via `docker-py`, `dagger` or similar tool

zensor®

Examples

Private PyPI Authentication

Requirement	Local Assumption	Remote Assumption
Dependency Management	UV, globally installed	UV, setup via CI/CD plugin

‘Extra Indices’ configurable via file or environment variable

Alternatively: Fetch secret value and export to environment variables

```
ifndef PYPI_INDEX_URL
export PYPI_INDEX_URL := $(shell aws ssm get-parameter \
    --name /MY_PARAM --with-decryption --query "Parameter.Value" \
    --output text)
)
endif
```

Cons:

- Re-triggered on every command
- Requires extra system tools e.g. `aws-cli`
- For local work, manual commands might break

zensor®

Examples

Package Publishing

Requirement	Local Assumption	Remote Assumption
Dependency Management	UV, globally installed	UV, setup via CI/ CD plugin

Most Python tools also include wrappers for publishing

Here, we can either branch like for pulling packages, or else inject settings via Env Vars

- Publishing here might not require a virtual environment, so can be simpler e.g. a script using PEP723 or uv's "`uv run --with`" function

Small helper script handles building and publishing

```python
import os
import subprocess
import boto3

SSM = boto3.client('ssm')

def get_pypi_url() -> str:
    p = SSM.get_parameter(Name='/credentials/PYPI_URL',
                          WithDecryption=True)
    return p['Parameter']['Value']

def build_dist():
    subprocess.run(
        ["uv", "build", "-c"],
        capture_output=True, check=True
    )

def upload_package():
    pypi_url = get_pypi_url()
    build_dist()
    _ = subprocess.run(
        ["uv", "publish"],
        env={
            'UV_PUBLISH_URL': pypi_url,
            **os.environ
        },
        check=True
    )

if __name__ == '__main__':
    upload_package()
```

zensor®