**pipeline.yaml**

apiVersion: tekton.dev/v1beta1

kind: Pipeline

metadata:

name: simple-ci-pipeline

spec:

workspaces:

- name: shared-workspace

tasks:

- name: install-requirements

taskSpec:

steps:

- name: pip-install

image: python:3.9

workingDir: /workspace/shared-workspace

script: |

pip install -r requirements.txt

**PipelineRun**

apiVersion: tekton.dev/v1

kind: PipelineRun

metadata:

generateName: ci-sandbox-run-

spec:

pipelineRef:

name: simple-ci-pipeline

timeouts:

pipeline: 15m

workspaces:

- name: shared-workspace

volumeClaimTemplate:

spec:

accessModes: ["ReadWriteOnce"]

resources:

requests:

storage: 1Gi

**Big picture (what happens end-to-end)**

1. **A Pipeline** called simple-ci-pipeline defines *what* to do: run one task that installs Python dependencies from requirements.txt.
2. **A PipelineRun** is the *instance* of that pipeline: it provides runtime details (timeouts; a scratch PVC; and the binding of the workspace the task will use).
3. When you trigger the run (via LightSpeed/your CI UI):
   * Tekton **creates a temporary PersistentVolumeClaim (PVC)** from your volumeClaimTemplate and mounts it as a **workspace** at /workspace/shared-workspace inside the step’s container.
   * (In your LightSpeed setup) the **repo contents are made available** in that workspace (this is why the file is found without adding an explicit git-clone task).
   * Tekton starts a **container** from the python:3.9 image and executes the provided shell script.
   * The script runs pip install -r requirements.txt. If that file resolves, your step succeeds.
   * When the PipelineRun finishes, Tekton cleans up the pod; the PVC is ephemeral unless you told it otherwise.

**Line-by-line: pipeline.yaml**

apiVersion: tekton.dev/v1beta1

* This is the Kubernetes API version for **Pipeline** resources. (Some clusters already support tekton.dev/v1 for Pipelines; v1beta1 is the widely compatible one.)

kind: Pipeline

* The Kubernetes object type you’re creating.

metadata:

name: simple-ci-pipeline

* Metadata for the object; the pipeline’s name is how your PipelineRun refers to it.

spec:

* The desired behavior of the pipeline begins here.

workspaces:

- name: shared-workspace

* Declares a **named workspace** the pipeline will require. Think of a workspace as a shared disk volume that tasks can read/write.

tasks:

* The list of tasks to run (and their order/dependencies).

- name: install-requirements

* A single task named install-requirements.

taskSpec:

* Instead of referencing a pre-existing Task object (taskRef), you embed the task definition **inline**.

steps:

* A task is made of one or more **steps**. Each step is a container with a command/script.

- name: pip-install

image: python:3.9

* The step runs inside a container created from the official Docker image **python:3.9** (which already has Python and pip installed).

workingDir: /workspace/shared-workspace

* Sets the step’s current directory. Tekton automatically mounts the pipeline workspace at /workspace/<workspace-name>.  
  Here, the workspace named shared-workspace is mounted at /workspace/shared-workspace. That’s where your repo files show up.

script: |

pip install -r requirements.txt

* A small shell script that gets executed inside the container.  
  By default Tekton uses /bin/sh -c to run it.  
  It runs **pip** in the working directory and installs all dependencies listed in requirements.txt.

If you ever need more commands, just add more lines under script:.

**Line-by-line: PipelineRun**

apiVersion: tekton.dev/v1

kind: PipelineRun

* This is the runtime object that **executes** a Pipeline.

metadata:

generateName: ci-sandbox-run-

* Kubernetes will create a unique name starting with this prefix (e.g., ci-sandbox-run-abc123). Useful when you trigger multiple runs.

spec:

pipelineRef:

name: simple-ci-pipeline

* Points to the Pipeline you defined above by name.

timeouts:

pipeline: 15m

* Caps the total runtime of the PipelineRun at 15 minutes. (Prevents stuck runs.)

workspaces:

- name: shared-workspace

volumeClaimTemplate:

spec:

accessModes: ["ReadWriteOnce"]

resources:

requests:

storage: 1Gi

* **Binds** the pipeline’s required workspace to an **ephemeral PVC** created specifically for this run.
* volumeClaimTemplate tells Tekton to provision a 1Gi PVC on the fly and mount it into the pipeline/steps at /workspace/shared-workspace.
* ReadWriteOnce means the volume will be mounted read/write by a single node (standard for CI pods).

In clusters without LightSpeed’s auto-checkout, you would add an explicit git-clone task that writes into this workspace. In your environment the platform injects repo contents for you, which is why requirements.txt is present.

**Execution flow in plain English**

1. The PipelineRun starts → a pod is created with an **init container** (which sometimes emits those .docker warnings) and your step container.
2. The PVC is provisioned and mounted at /workspace/shared-workspace.
3. Your step container boots from python:3.9, workingDir is set, and the script runs.
4. pip installs the dependencies.
5. Step finishes → task succeeds → pipeline completes.

**Where to go next**

* **Add safety**: change the script to fail if the file doesn’t exist.
* script: |
* set -euo pipefail
* test -f requirements.txt
* pip install -r requirements.txt
* **Add unit tests**: introduce a second step that runs pytest.
* **Cache wheels**: mount a second workspace for pip cache to speed up subsequent runs.
* **Pin Python**: use python:3.11 if your project targets 3.11.

If you paste your updated goals (e.g., run pytest, lint, build a wheel, or build a Spark image), I’ll extend this pipeline with the right extra tasks and explain those too.

**1. What is a PersistentVolumeClaim (PVC)?**

* In Kubernetes, a **PVC** is a request for storage.
* Think of it as: *“please give me a 1Gi disk I can mount into my container at a path”*.
* The cluster satisfies that request using whatever storage it has available (local disks, NFS, cloud disks like AWS EBS, GCP PD, etc.).

In Tekton:

* Workspaces (shared directories) are **backed by PVCs** so steps/tasks can read/write files between them.

**2. Ephemeral PVCs in Pipelines**

* Normally, a PVC is created and kept around until you delete it.
* But Tekton Pipelines often only need storage **just for the run**.
* That’s why Tekton supports **ephemeral PVCs** — automatically created at the start of a PipelineRun and deleted after it finishes.

This is what your YAML does:

workspaces:

- name: shared-workspace

volumeClaimTemplate:

spec:

accessModes: ["ReadWriteOnce"]

resources:

requests:

storage: 1Gi

**3. Line by line: what volumeClaimTemplate means**

* **workspaces:**  
  Declares the binding for the workspace (shared-workspace).
* **volumeClaimTemplate:**  
  Says: *“Don’t bind this workspace to an existing PVC, create a new one for me on-the-fly.”*
* **accessModes: ["ReadWriteOnce"]**
  + Only one node can mount this volume at a time (read/write).
  + That’s enough since your Pipeline runs on a single pod.
* **resources.requests.storage: 1Gi**
  + Request a volume with 1 GiB capacity.
  + The storage provisioner in your cluster (e.g., AWS EBS, Azure Disk, GCP PD) fulfills this.

**4. How Tekton uses this**

1. At the start of your PipelineRun, Tekton creates a temporary PVC from this template.  
   Example name: pipelinerun-ci-sandbox-run-xyz-pvc.
2. Tekton mounts that PVC into the pod at /workspace/shared-workspace.
3. Any step writing files there (like logs, test reports, build artifacts) can be read later by other steps/tasks.
4. After the PipelineRun ends, Tekton cleans up the PVC automatically.

**5. Why is this needed?**

* Each step in a task is a **separate container**. By default, they don’t share a filesystem.
* Workspaces + PVCs allow:
  + **Sharing code** (checkout in one step, run tests in another).
  + **Passing artifacts** (build → test → publish).
  + **Persistence across tasks** (task1 writes → task2 reads).

1. **git-clone**
   * Clones the exact commit/branch that triggered the run into the workspace (PVC).
   * Output: your repo’s files on disk for later steps.
2. **python-build**
   * Uses a Python image to:
     + print Python version, upgrade pip
     + run pip install -r requirements.txt (this is where your earlier numpy typo broke things)
     + optionally run unit tests if configured (often via pytest or a command from pipeline.yaml).
   * Output: a populated virtualenv/site-packages on the workspace + test reports/coverage files (if enabled).
3. **sonar-scan**
   * Runs Sonar scanner against the checked-out sources to compute code quality metrics (bugs, smells, coverage).
   * Inputs: repo files and (optionally) test reports from **python-build**.
   * Output: analysis published to your Sonar project.
4. **github-sonar-binding**
   * Associates the Sonar analysis with the GitHub Pull Request so you see quality status/checks back in GitHub.
   * Output: status checks/PR decorations.
5. **snyk-scan**
   * Runs Snyk (or org-equivalent) to scan dependencies for known vulnerabilities.
   * Inputs: requirements.txt/installed wheels from **python-build**.
   * Output: a security report and pass/fail gate.
6. **heimdall**
   * Collects build metadata (commit, branch, versions, timestamps, test results) into a standardized artifact.
   * Output: a build.info/manifest + metadata bundle on the workspace.
7. **heimdall-s3-upload**
   * Uploads the Heimdall bundle (and any declared artifacts like test reports) to your central artifact store/S3 bucket.
   * Output: immutable build artifacts you can reference later.
8. **gcr-ci** (sometimes labeled *container build/publish*)
   * If present in your template, this builds and optionally pushes a container image (or publishes a wheel).
   * Inputs: repo + installed deps; Output: image in the registry or built package.
   * (If your minimal pipeline doesn’t build images, this may be skipped.)
9. **deployment-trigger**
   * Kicks off a downstream “deploy” or notifies CD tooling.
   * In a “CI-only” run it may no-op or just post a status.
10. **finalizer**

* Always-runs step that gathers logs, marks the run result, and cleans up transient files.
* It does not erase the workspace PVC during the run; cleanup of the temporary PVC happens once the run finishes.