

Environment-Sensitive Staging in Beam Python

Tarun Annapareddy(tannapareddy@google.com)

(Last Updated: 20 Oct 2026)

Summary:

Beam stages per-environment artifacts into a single namespace: the worker-local file name (`staged_name`, e.g. `requirements.txt`) is reused as the GCS object name by the Dataflow runner and as the [on-disk path](#) inside SDK containers, which on Dataflow share one host directory.

When a pipeline has two or more Python environments (Beam YAML `pythonPackage` providers, `config: dependencies:`, multiple Python external transforms), each environment stages its own `requirements.txt`. The uploads collide; the last writer wins. Every environment still records the SHA-256 of *its own* file in `pipeline.pb`, so the losing environment fails the staged-file integrity check ([materialize.go](#)) at startup and crash-loops. Before integrity checks existed, the same collision silently installed the wrong dependency set.

Java does not face this issue, as staged jars are always hash appended and Java uses [per-container directory](#). Go also uses unique names for staged files. For Python it is most pronounced because only the Python stager emits fixed names (`requirements.txt`, `pickled_main_session`, `extra_packages.txt`, `workflow.tar.gz`), so any job with two Python environments collides. In the collided state the provider environment ends up installing the main environment's requirements; pipelines have only worked when the main list happened to be a superset of the provider's.

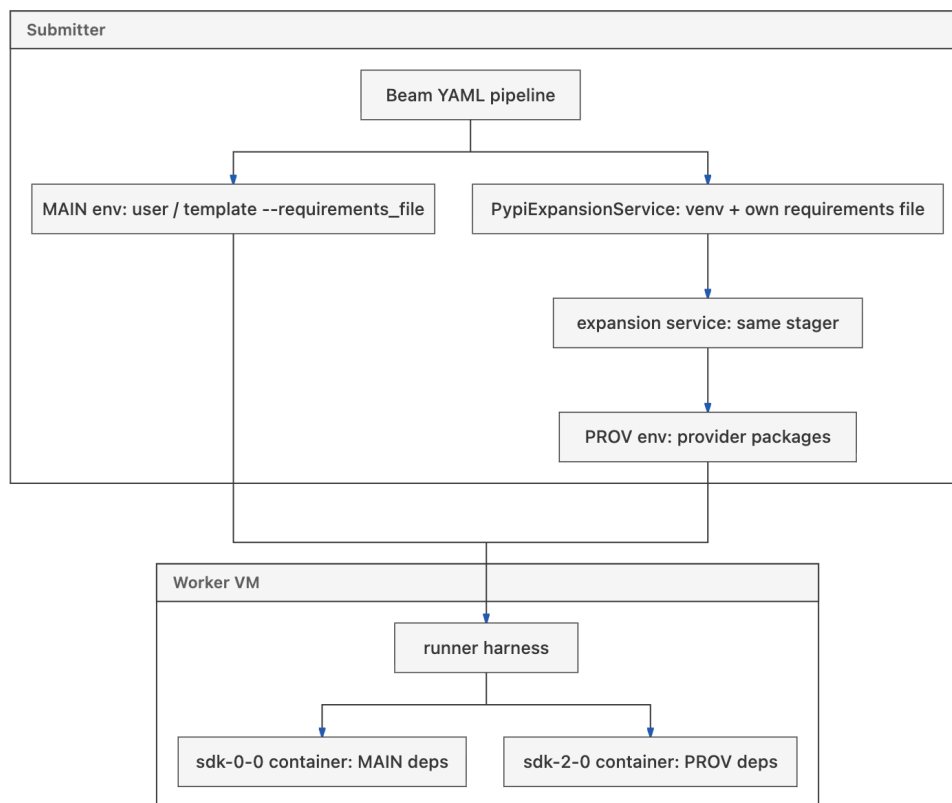
Proposal: Make staging environment-sensitive - content-address(hash) the remote object name and isolate the on-disk staging root per environment.

Context:

Multi-environment Pipelines:

Multi-environment is how many cross-language IO works, and Beam YAML extends it to Python-in-Python isolation ([Python providers](#)). All these providers/external transforms start a [PypiExpansionService](#): it creates a venv on the submitter, pip-installs the provider's packages into it, writes its [own requirements file](#) and launches `expansion_service_main` from that venv with the requirements file ([here](#)). The expansion service runs the same

[stager.py](#) as the main pipeline, and result is a second Python environment whose dependency list also contains `STAGING_TO staged_name=requirements.txt`.



Each SDK container receives only its own environment's dependency list through [ProvisionInfo](#) whose `dependencies` are that environment's only. The container's [boot.go](#) then materializes exactly those artifacts ([artifact.Materialize](#)), pip-installs them ([installSetupPackages](#) → [piplInstallRequirements](#)), and runs only the transforms tagged with its environment.

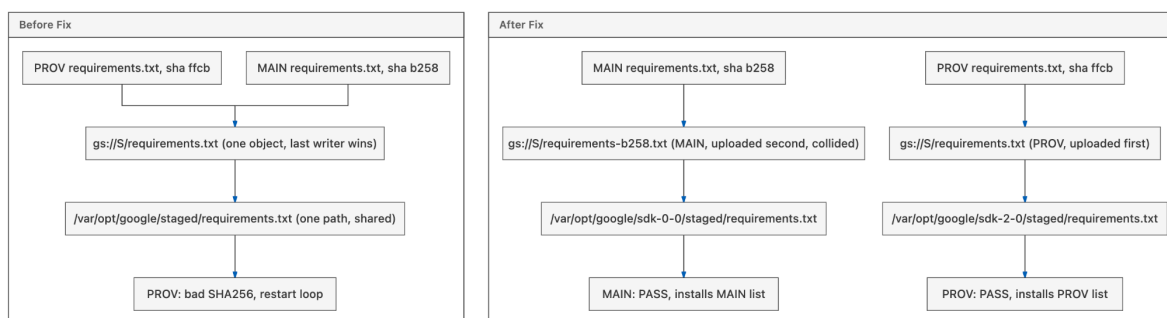
Issue:

1. Both MAIN and PROV stages requirement files; [stager.py](#) tags both `STAGING_TO staged_name=requirements.txt`. But the contents differ.
2. [apiclient._stage_resources](#) uses `staged_name` as the GCS object name, so both upload to `gs://<staging>/requirements.txt`. Environments are processed in sorted-ID order — PROV first, MAIN second — and the [dedup](#) only catches identical content, so MAIN's upload replaces PROV's.
3. Each dependency is [rewritten](#) to `URL + sha256` with its own environment's hash. `pipeline.pb` ends up with two environments pointing at one URL with two different hashes.
4. On the worker, [materialize.go](#) downloads content from the URL and compares the hashes. MAIN's container matches. PROV's container gets an integrity hash mis-match and fails to start.

Notes: Separately, all SDK containers on a Dataflow VM share one host directory and the Python container stages into [<semi_persist_dir>/staged](#), and both write [/var/opt/google/staged/requirements.txt](#). So even with distinct GCS objects the on-disk file would be overwritten between one container's download and its pip install.

Fix:

The high level idea is no two shared artifacts for a job can share the same GCS object name or the on-disk path, unless they are byte identical.



1. **Collision-safe remote names:** In [stage_resources](#), when a second artifact maps to an already-used [remote_name](#) with a different [sha256](#), upload it under [<stem>-<sha>.<ext>](#) instead (e.g. [requirements-<sha>.txt](#), ex: Java's [createStagingFileName](#)). Existing names are unchanged unless there is a conflict, so single-environment jobs are unaffected. With two environments each staging a [requirements.txt](#), every environment ends up with its own URL and hash in [pipeline.pb.role_payload.staged_name](#) is left untouched, so [materialize.go](#) still writes the file as [requirements.txt](#) and [boot.go](#) still finds it by that literal name for [pipInstallRequirements](#).
2. **Per-environment staging directory:** Match the Java container ([semiPersistDir/<id>/staged](#)): in Python [boot.go](#) and Go [boot.go](#), set directory to [semiPersistDir/<id>/staged](#) before [artifact.Materialize](#). Each environment then has its own directory for downloaded dependencies on the shared host volume.