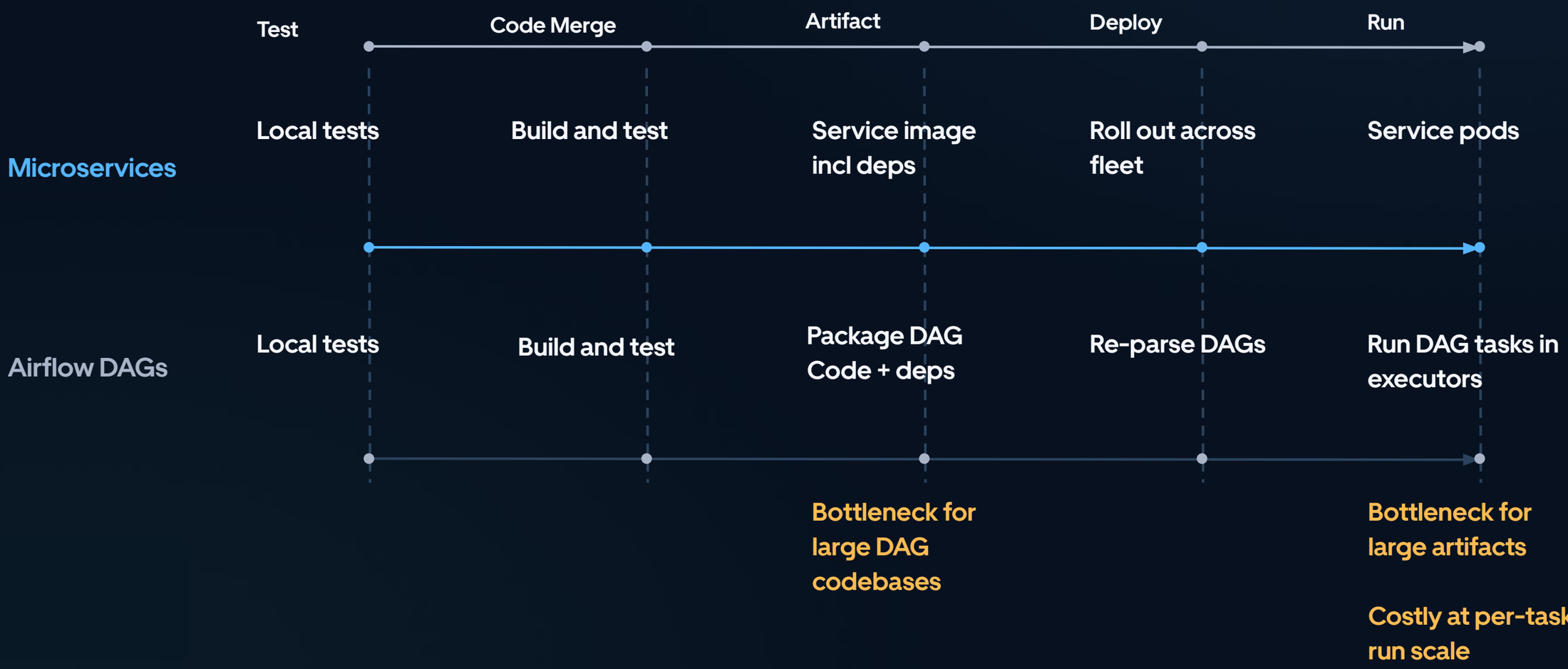


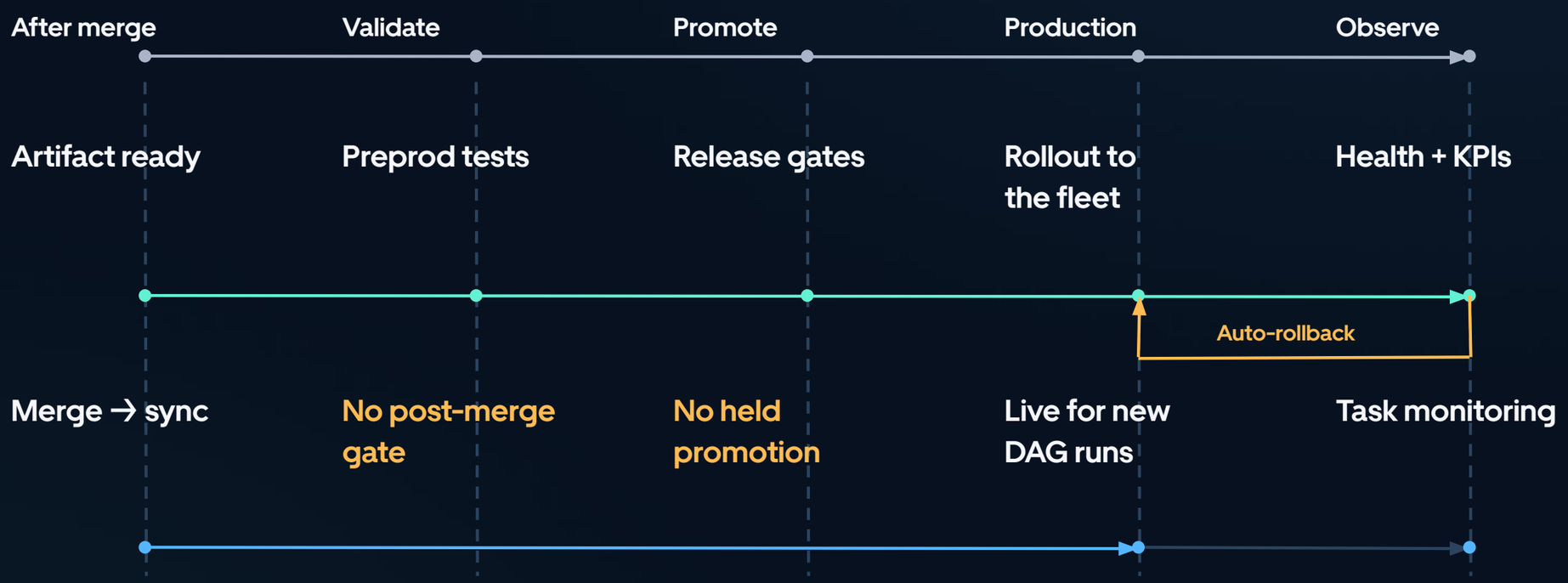
Microservice-Grade Delivery and Release for Airflow DAGs

Piyush Maheshwari | Sameer Raj
(Software Engineers at Uber)

Delivery: Authoring Code → Execution



Release: Merged code → Production rollout



Merge-to-prod baseline

Two problems to solve

01 · FIRST

Delivery

Delivering DAGs from a large codebase

Dependency-aware slim bundles

02 · THEN

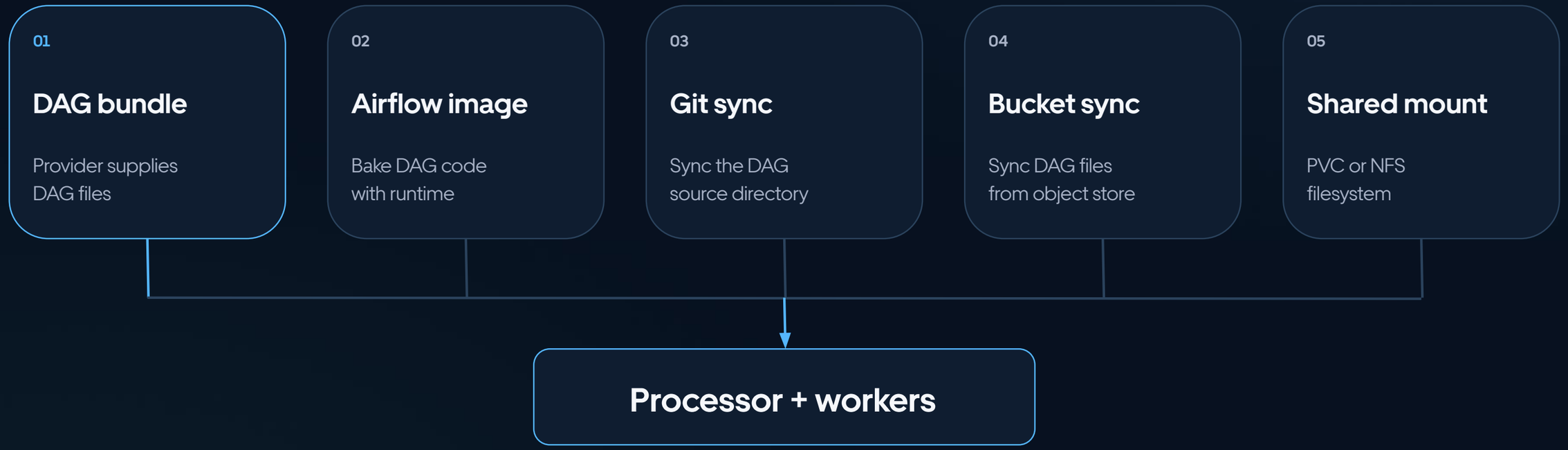
Release

Control how code goes live

Version pinning + deployment gates

How Airflow deployments ship DAG code

Source sync, images and bundles make DAG files available to Airflow.



Preparing Airflow for Uber's workloads

~1M

Daily task runs

The workload scale Airflow is being prepared for.

~2 GB

Monorepo artifact

Many DAGs and shared code in one snapshot.

>3.5k

Monthly monorepo commits

Repository commits, not DAG deployments.

OUR CHOICE · Fetch versioned bundles from blob storage

Decouple DAG code from worker-image rollouts; keep exact versions retrievable.

Why the other approaches were not our fit

Worker image

Build / rollout coupling

Git sync

Large-monorepo sync cost

NFS mount

Simultaneous multi-version checkouts can be expensive

PVC mount

Lifecycle management overhead and cardinality bloat with commits

One global bundle serves processor and workers

GLOBAL DAG TAR · REPRESENTATIVE

~2 GB

dag_alpha	SQL operator	sql/daily.sql
shared_utils.py	generated client	dag_bravo
dag_charlie	dag_delta	other DAGs

■ dag_alpha needs ■ Unrelated DAGs

BUNDLE REFRESH + PERIODIC PARSING · DAG PROCESSOR

Refresh code and reparse DAGs

Fetch, extract bundle → Parse local DAGs → Serialize

Reparse DAGs to show new changes to users

ON A COLD START · TASK WORKER

Load code before execution

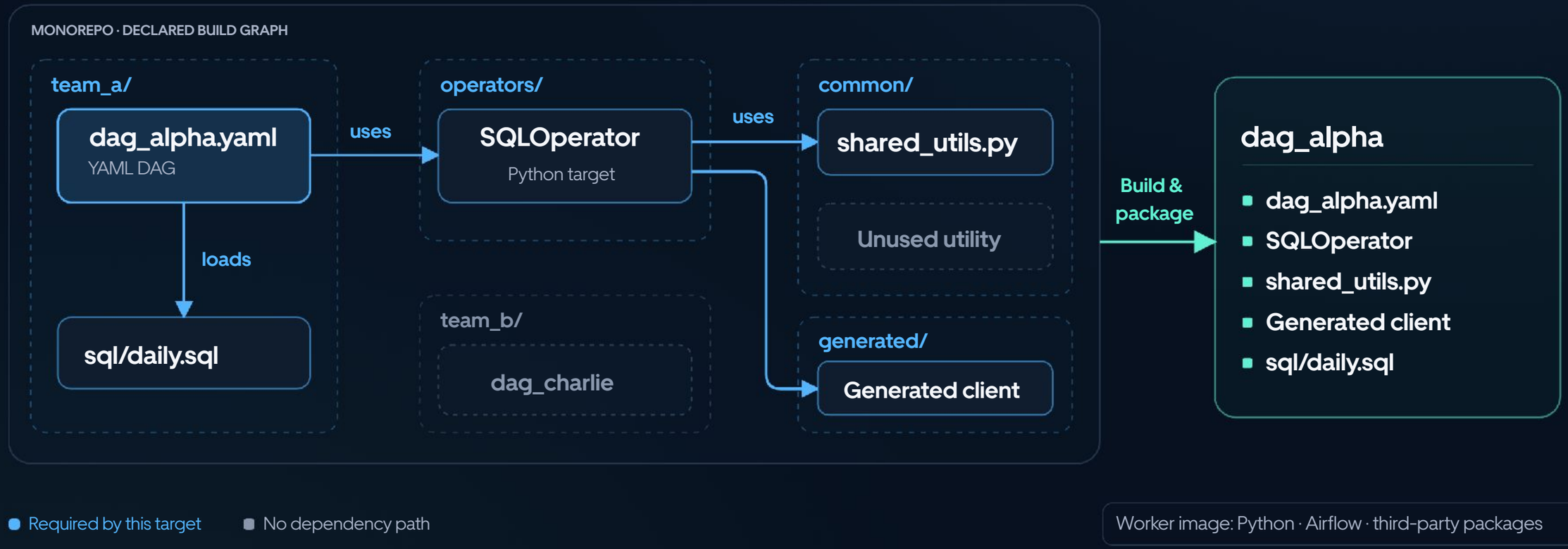
Provision worker → Fetch, extract bundle → Run dag_alpha

Loading unrelated DAG code postpones useful task work

2 GB TAR Bundle POC took ~ 2 mins for loading code

Use a build graph to follow DAG dependencies

Bazel is an open source build tool. A DAG target declares Python code and runtime files, even across directories.



Required by this target

No dependency path

Worker image: Python · Airflow · third-party packages

Forming Build Graph

dag.yaml · ILLUSTRATIVE

```
dag_alpha:
  tasks:
    query:
      operator:
        example.ops.SqlOperator
      sql_file: sql/daily.sql
    transform:
      python_callable:
        example.jobs.normalise
```

GAZELLE · CONFIGURED KEY PATHS

tasks.operator

pyref → deps

Resolve to the owning Python target.

tasks.python_callable

pyref → deps

Use the same Python target index.

tasks.sql_file

data → data

Existing local path → runtime file.

GENERATED BUILD.bazel · SIMPLIFIED

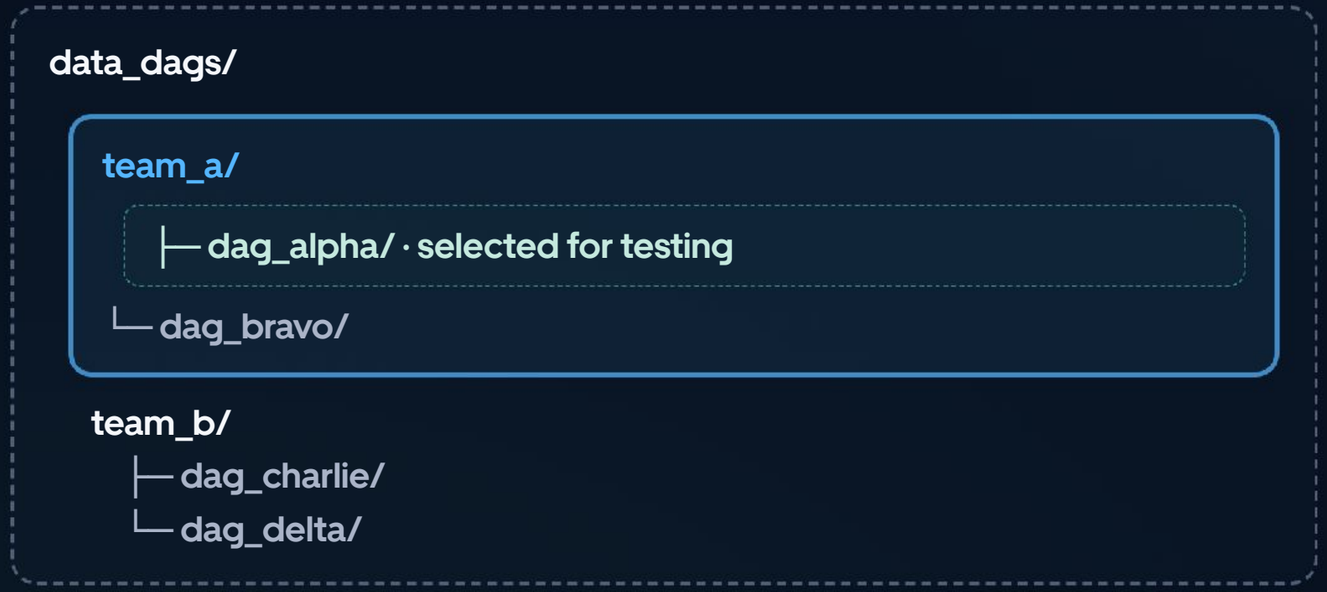
```
dag_yaml(
  name = "dag_alpha",
  deps = [
    "//example/ops:py_default_library",
    "//example/jobs:py_default_library",
  ],
  data = ["sql/daily.sql"],
)
```

— Python code references → **deps** — Runtime file references → **data**

Gazelle generates **BUILD.bazel** from configured YAML fields: Python references become **deps**; files become **data**.

Flexible bundle scopes from DAG targets

YAML DAG TARGETS | dag_alpha | dag_bravo → **team_a production bundle** or dag_alpha alone for testing



Before · global bundle

All DAGs and shared code in one large artifact.

Production · team scope

Group targets by owner; split an oversized team further.

Testing · selected DAGs

Build one requested DAG or group with its dependencies.

Targets let us choose the DAG group; the build graph still supplies its runtime dependencies.

Smaller scopes mean less to fetch per worker, but more artifacts and potentially duplicated shared code.

Find what changed; rebuild the complete scope

LANDED MONOREPO CHANGE

shared_utils.py

– old helper logic

+ updated helper logic

The change can originate outside the DAG directory.

REVERSE DEPENDENCY ANALYSIS

DAG target graph

dag_alpha	Affected
dag_bravo	Unchanged
dag_charlie	Unchanged

SELECTED BUNDLE SCOPE

team_a — New complete snapshot

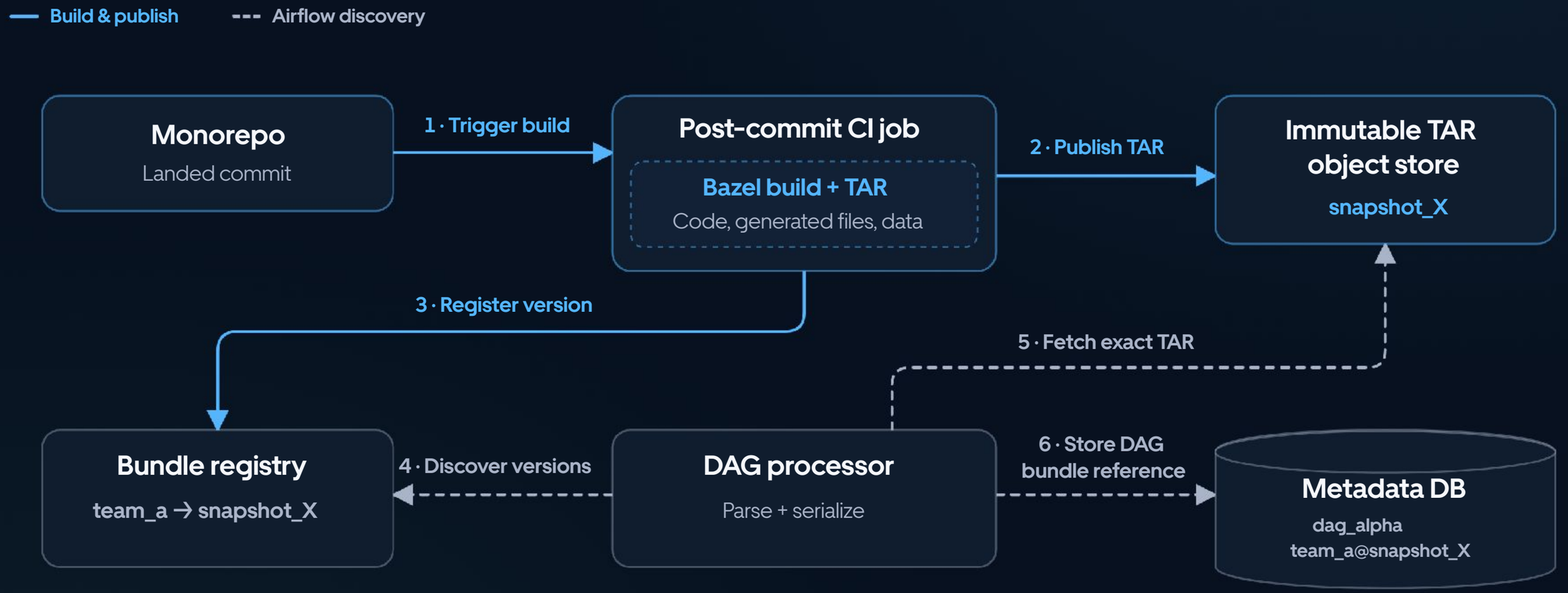
dag_alpha	changed
dag_bravo	unchanged

+ complete runtime dependencies

First-party code, generated outputs and declared resources.

UNRELATED SCOPE **team_b** dag_charlie and dag_delta are unchanged — **No rebuild**

Build and publish versioned DAG TARs



Exact lookup: name/version → blob hash → TAR

Building on top of AIP: 63 DAG Versioning and AIP 66: DAG Bundles & Parsing

Impact of designed DAG delivery

USER EXPERIENCE

TASK START-UP

~5x

Faster task start-up

Less time from queueing to useful task execution.

RELEASE TO PROD

~2.5x

Lower time to production

From code merge to DAG code available.

PLATFORM EFFICIENCY

NETWORK BANDWIDTH

99% less

Bundle download bandwidth

Bundle download traffic at target task volume.

BUNDLE RETENTION

10x

Higher bundle retention

More bundle versions retained with the same storage budget.

The code is available.
How does it go live safely?

Delivery

Build a complete, smaller artifact.
Make each version retrievable.

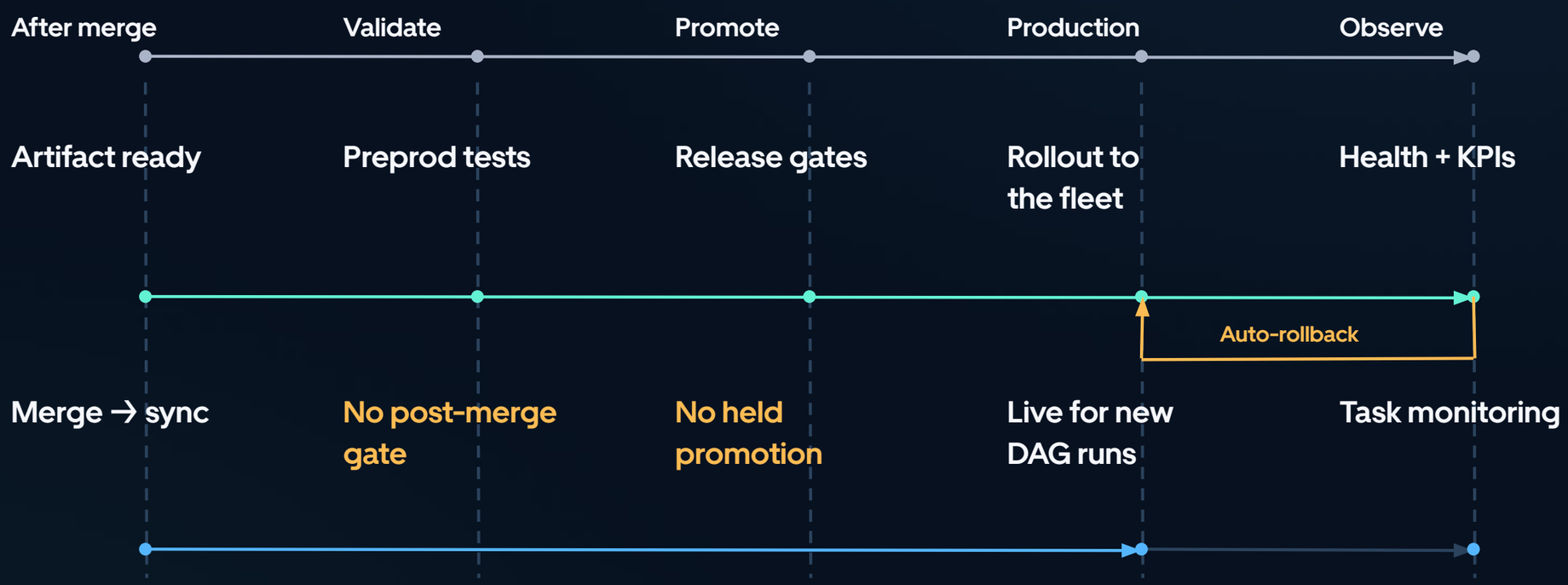


Release

Validate a candidate.
Advance DAGs gradually through release gates.

A bundle is the distribution unit. **A DAG is the release-control unit.**

Recap: Microservice release analogy



Shared blast radius across DAGs

shared_utils.py

New version distributed and parsed.

→

- dag_alpha
- dag_delta
- dag_charlie
- dag_echo

Different DAGs, different release needs

Ownership

Different teams / owners

Criticality

Different impact of failure

Failure tolerance

Different tolerance for a bad rollout

Release policy

Different gates and cutover timing

Change in a single shared utility should not force the release decision on all consumers.

Rolling back bad changes

1 · CHANGE
Prepare a revert

2 · VERIFY
Run CI checks

3 · DELIVER
Merge, build, publish

4 · DISCOVER
Wait for processing

5 · ACTIVATE
New runs select it

Inter-commit dependencies: returning to older logic may need not be a simple revert if there are dependent commits.

Coarse grained rollback: reverting a shared helper reverts all its consumers together.

How to make DAG releases safer

CODE MERGE != GO LIVE

Decouple delivery from release

SHARED CODE

Per DAG release control

CANDIDATE RISK

Validate candidate before prod promotion

SLOW REVERTS

Faster recovery without rebuilding

IN-FLIGHT RUNS

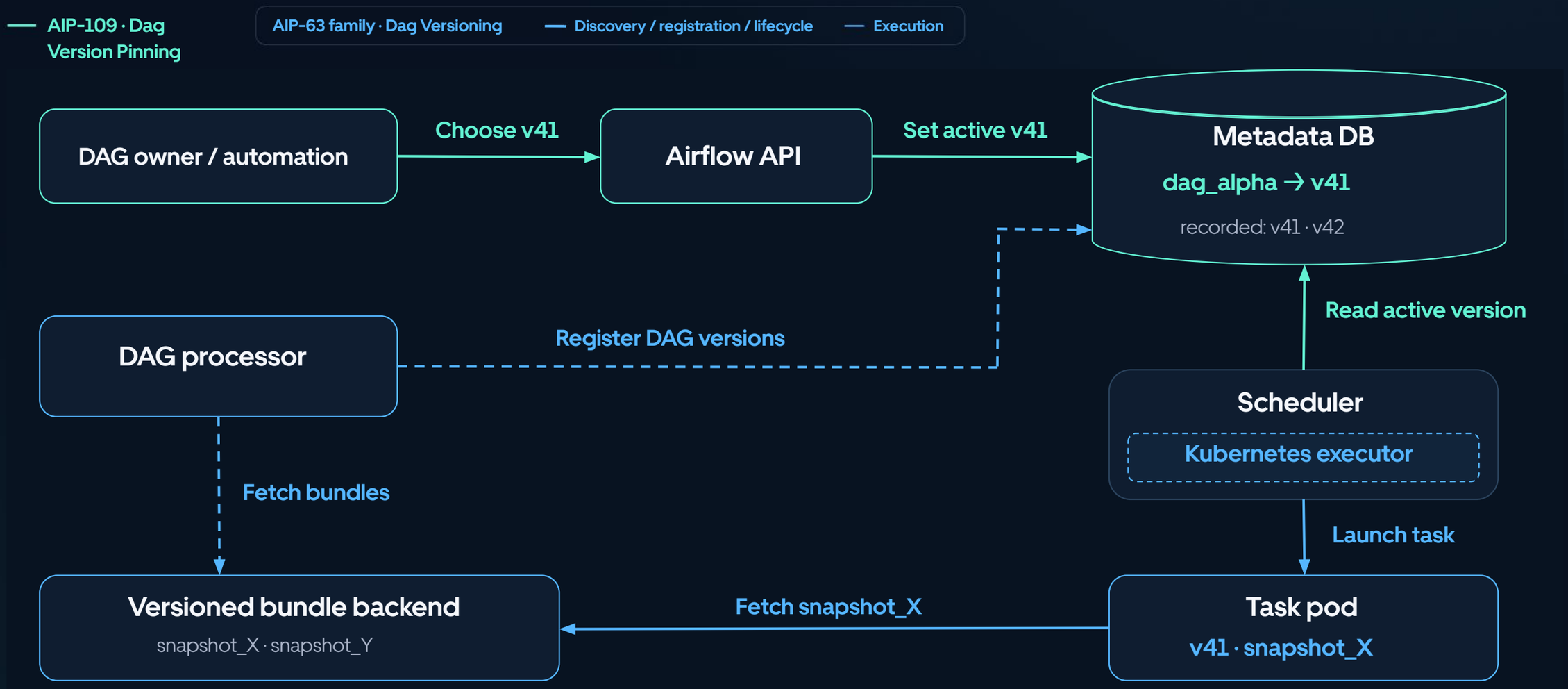
No disruption to inflight runs

TEAM OWNERSHIP

Owning team can configure the release policy

> The release decision must be explicit, and execution must follow that choice.

Version selection and execution



Release across DAGs independently

ONE SHARED CHANGE

shared_utils.py

Helper, operator or DAG factory

Some DAGs depend on this helper. Others do not.

deps

Affected DAGs

Dependency on shared_utils.py

dag_alpha — Gate passed

v41 active · v42 ready

dag_delta — Hold current

v17 active · v18 held

dag_charlie — Release in progress

v10 active · health gate running

dag_echo — Rolled back

v23 restored · v24 rejected

Unaffected DAGs

No dependency on this change

dag_foxtrot — Unchanged

v8 remains active

dag_golf — Unchanged

v6 remains active

The primitive enables richer release policies

Preprod regression tests

Test the exact candidate.

Soak and acknowledgement

Observe, then approve.

Deployment windows

Choose when to promote.

Automatic recovery

Restore known-good code.

Tiered rollout

Promote by risk tier.

Policies run externally to Airflow; pinning controls activation.

Continuous deployment pipeline view

Continuous deployment / dag_alpha

PRODUCTION

Adapted, illustrative view

ACTIVE DAG VERSION
v41 · known good

SELECTED CANDIDATE
v42 · rolled back

RELEASE UNIT
One DAG

DAG version history

v43	**Remediated candidate** New version · awaiting its own gates	Pending
v42	**Selected release** Production observation failed	Rejected
v41	**Previous known-good** Production pointer restored here	Active
v40	**Earlier version** Retained rollback history	Retained

Candidate v42 · release detail

- **Build**
  Slim bundle · generated code · CI tests
- **Preprod gate**
  Graph checks · regression DAG runs · data checks
- **Promotion**
  Production pin advanced to v42
- **Observation**
  Production health check failed
- **Automated rollback**
  Production pin restored to v41

A remediation may appear as v43, a new candidate.

Release control shortens the rollback path

CODE-REVERT ROLLBACK PATH

>30 min

Validation, rebuild, distribution and processing delay.

→

VERSION-SELECTION ROLLBACK

<1 min

Restore the retained known-good version through the control plane.

Safer DAG releases

Per-DAG gates and selective activation.

Configurable release automation

Configurable rollout and health policies.

Automated rollback

Known-good selection on a health breach.

Ship the required code.
Control each DAG's release.

Dependency-aware slim bundles for Airflow DAGs.

Per-DAG version pinning for deployment gating, promotion and rollback.





OPEN SOURCE AIP-109 Dag Version Pinning contributes the version pinning primitive back to Apache Airflow

Questions & discussion