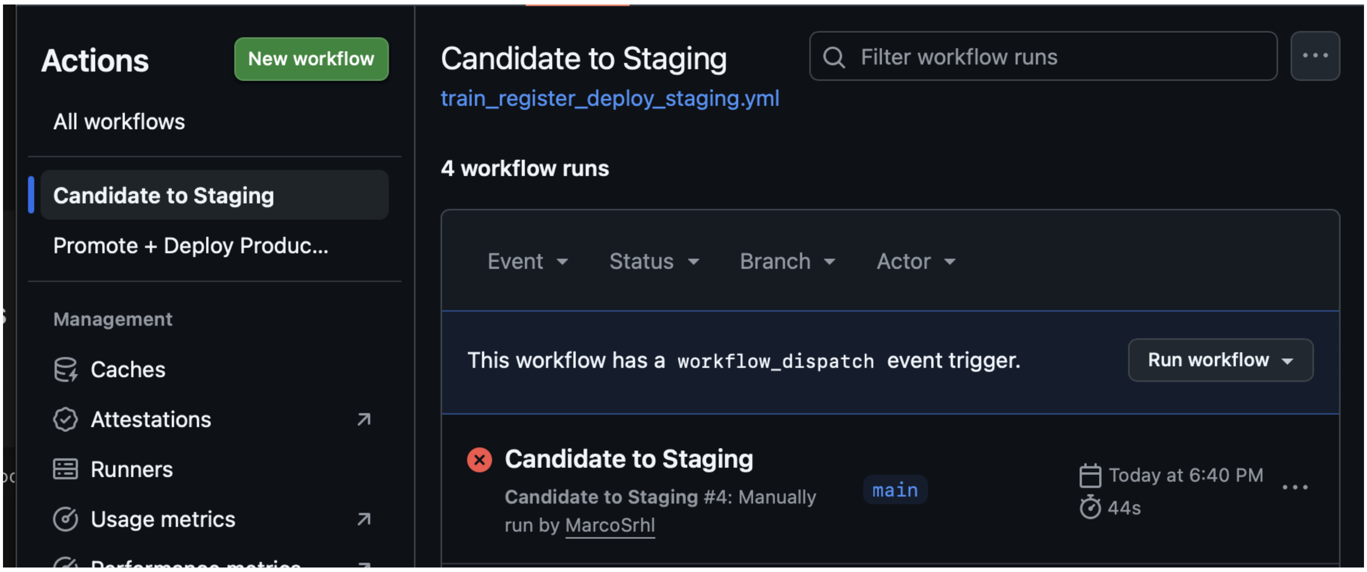
Exercise: Model Promotion Pipeline (Staging -> Production)

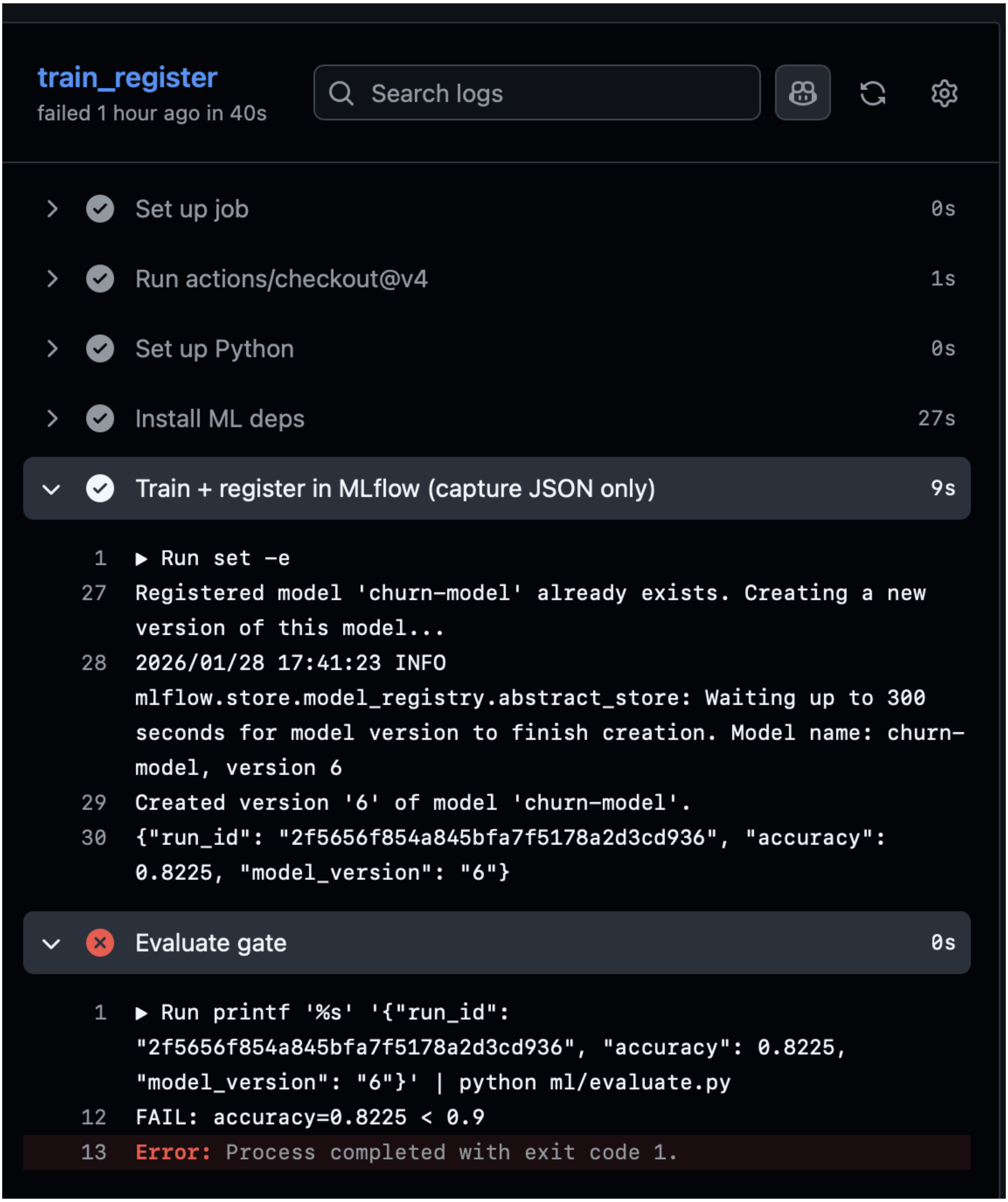
Github link: https://github.com/MarcoSrhl/MLopsExercise Marco-Naji Serhal

Task 1: Run Candidate -> Staging workflow

Trigger Candidate to Staging workflow (workflow_dispatch)

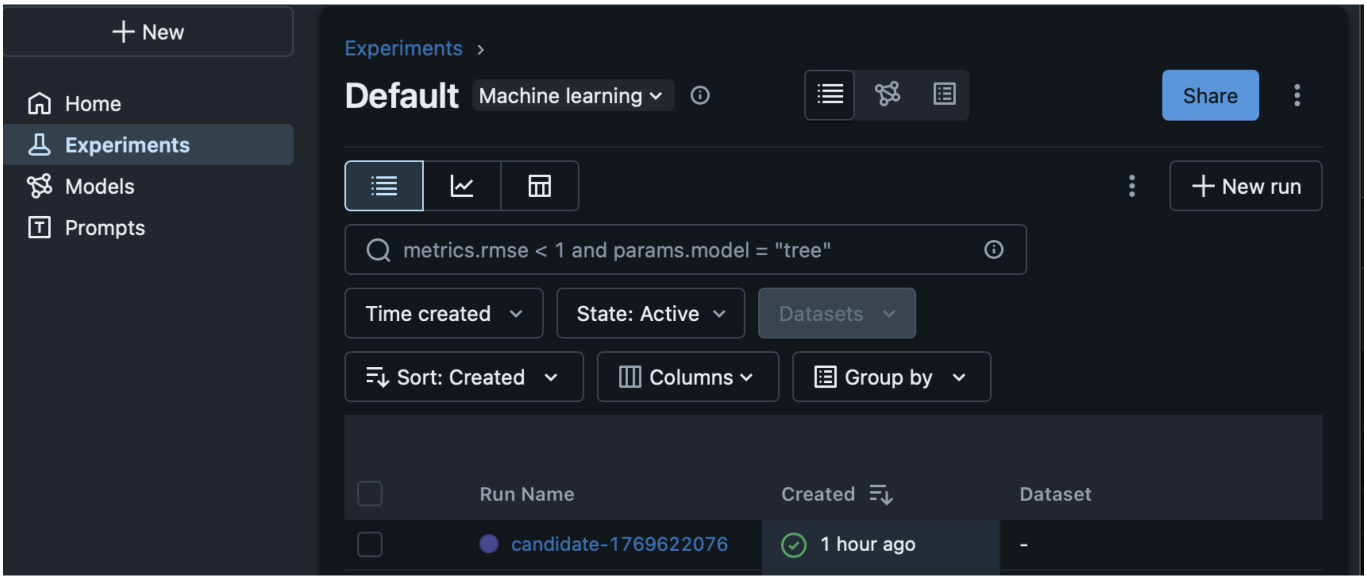


After Running the workflow we can see that it failed.

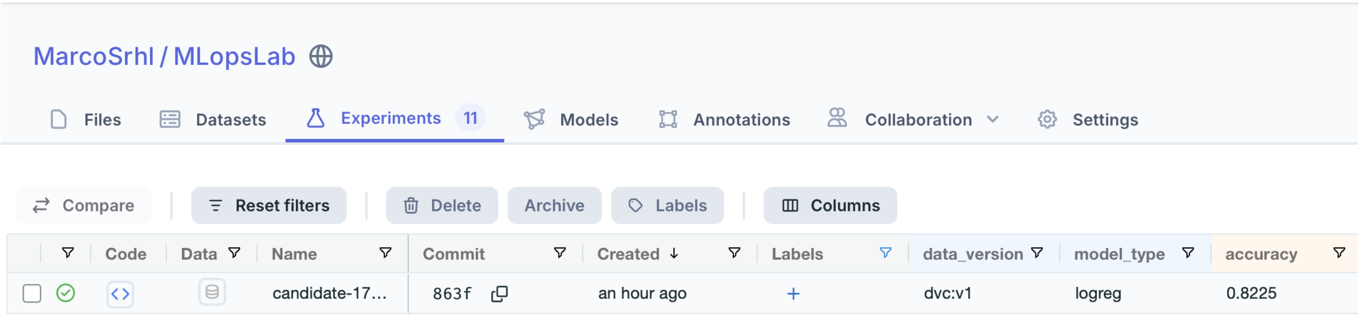


As we can see, the pipeline correctly trained the Logestic Regression model, computed the accuracy, logged the metrics in MLFlow and also registered a new model version.

The gate checks allow an accuracy >= .90, but in the observed results we can see 0.8225 so the gate failed and the deployment was aborted.



Although, we still can see that the model has been registered in MLFlow.



the MLFlow expirement was created. We can see from all the above:

- Model version: 6
- Accuracy: 0.8225
- Gate did not pass (failed)

Task 2: Explain what "staging" proves

What staging tests that offline evaluation does not

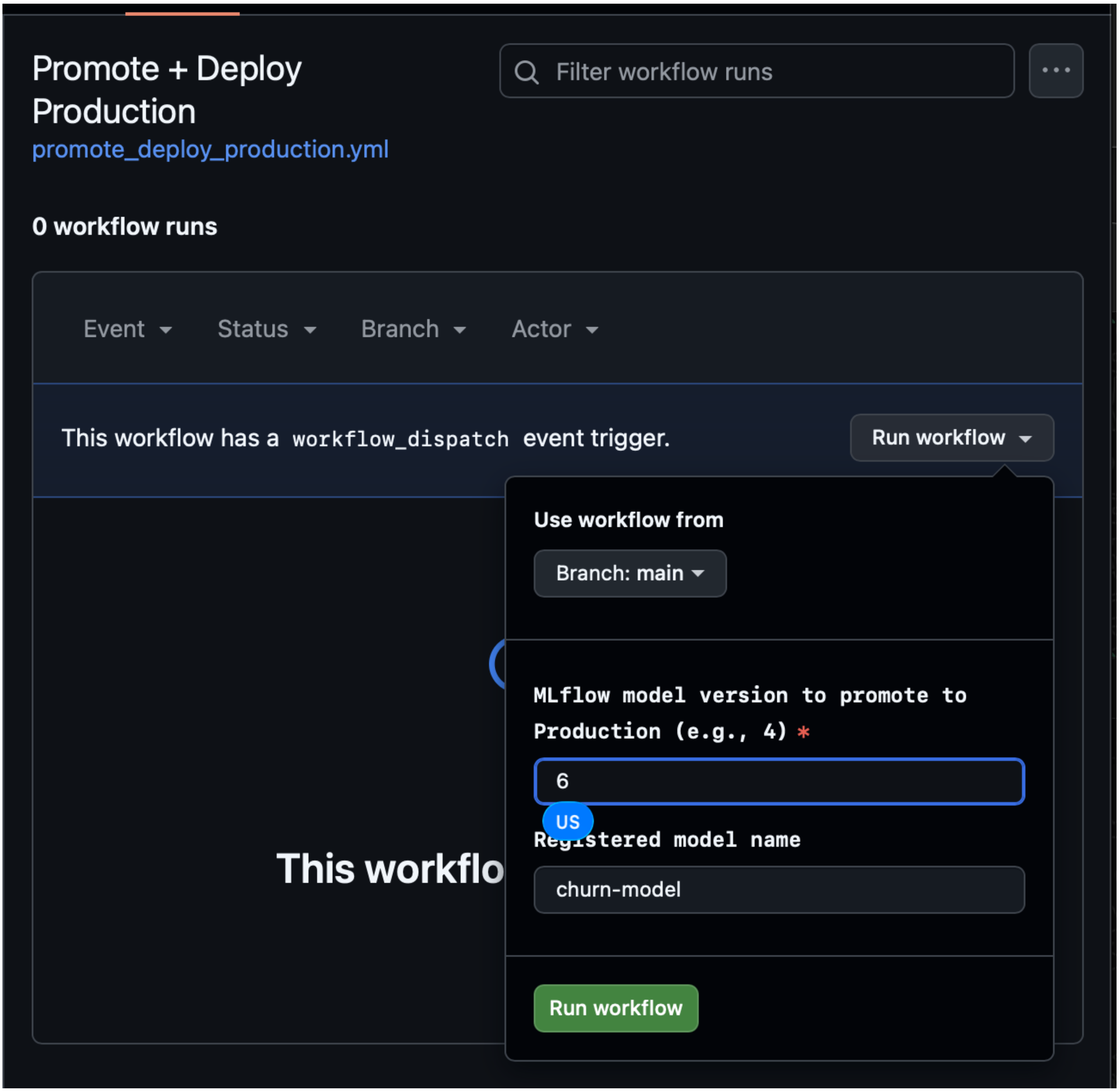
Offline evaluation only shows metrics like accuracy on a dataset. Staging additionally tests things like:

- The model can be loaded from the registry by stage (models:/churn-model/Staging) with the real credentials and artifacts
- The serving code/container starts and runs with the correct dependencies (no missing packages, version mismatches)
- The API server starts correctly
- The API contract works end-to-end (/health, /predict), including input format, preprocessing expectations, and output shape
- Runtime behavior is acceptable (latency, memory), and failures show up as real errors/logs
- Integration with surrounding pieces (Docker, env vars, secrets, network) is correct

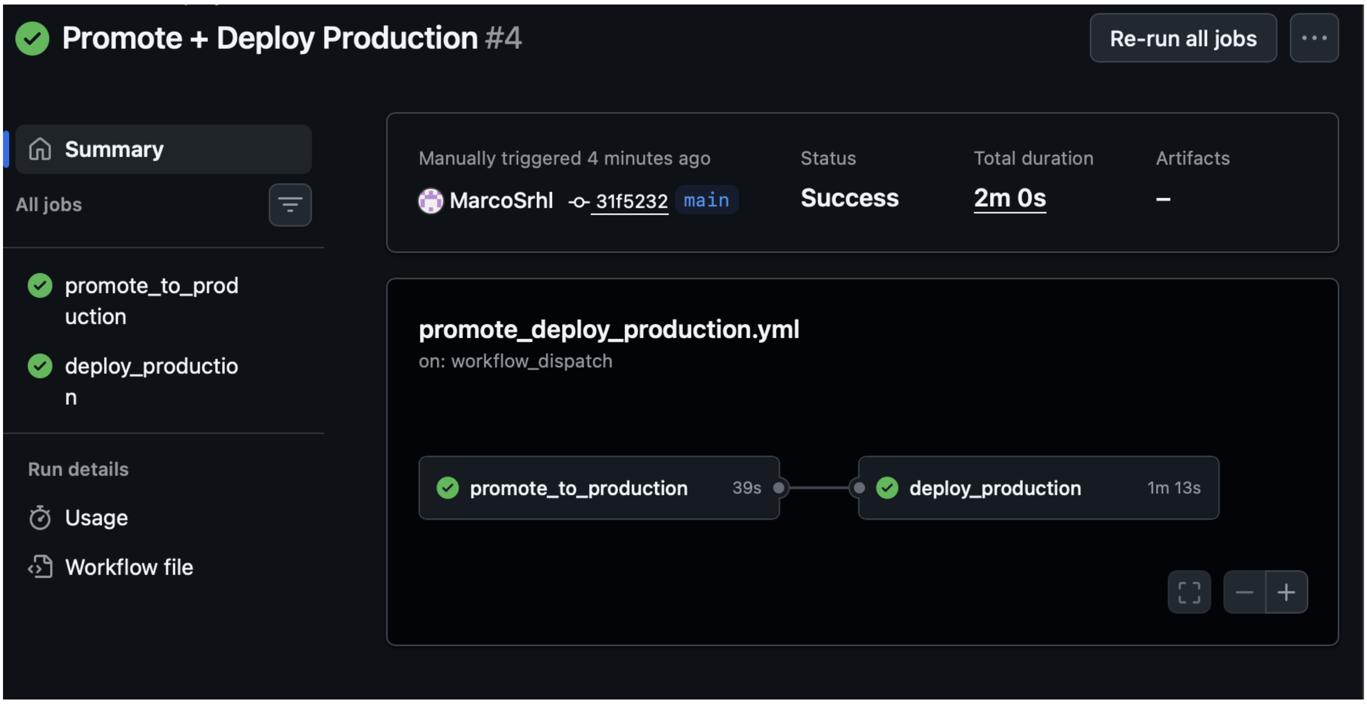
In short, staging tests the real deployment, in real production conditions.

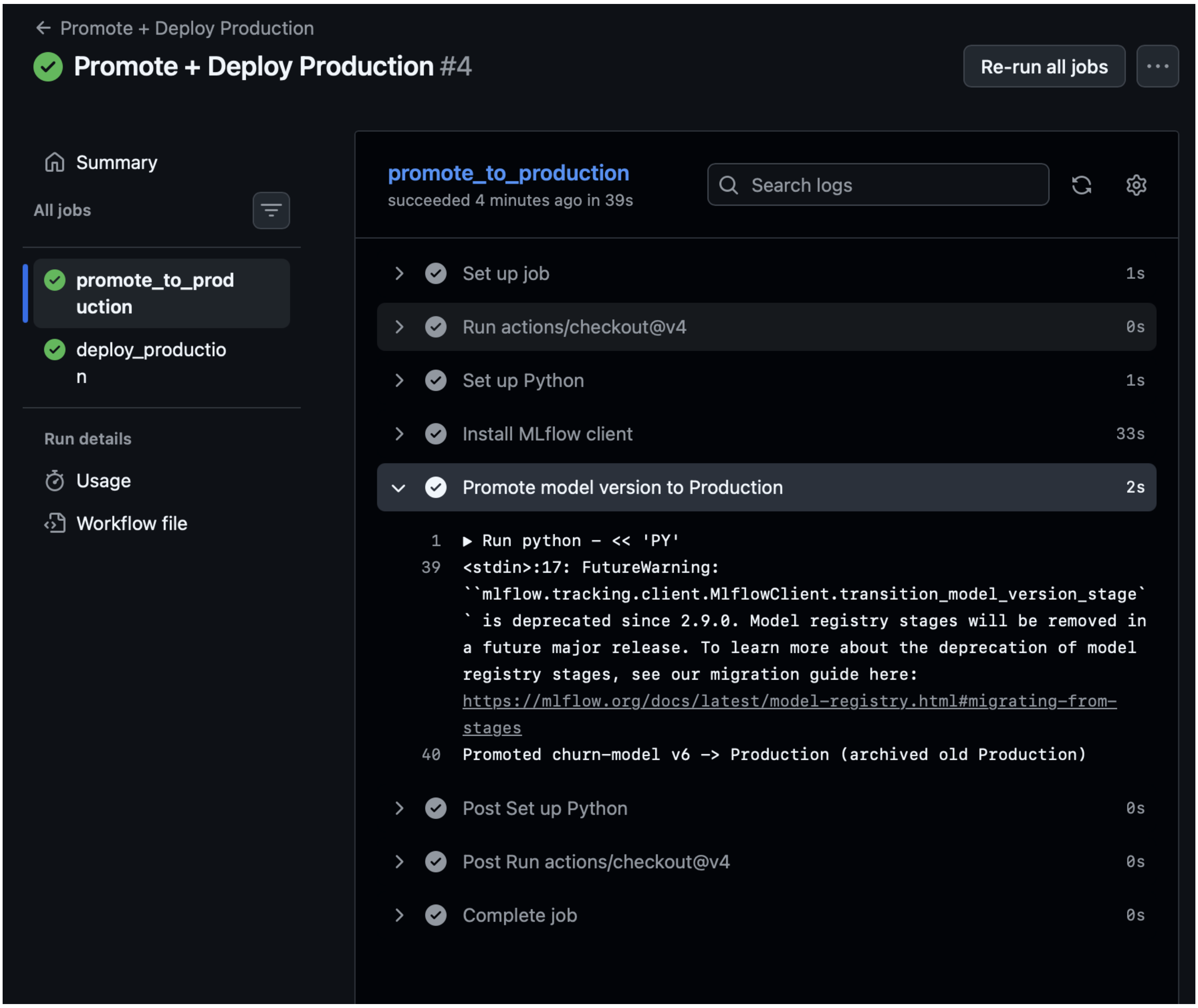
Task 3: Promote to production

- Trigger Promote to Production
- Provide model_version from Task 1
- Observe: MLflow stage transition to Production deployment job Deliverable: screenshot of the promotion log line.

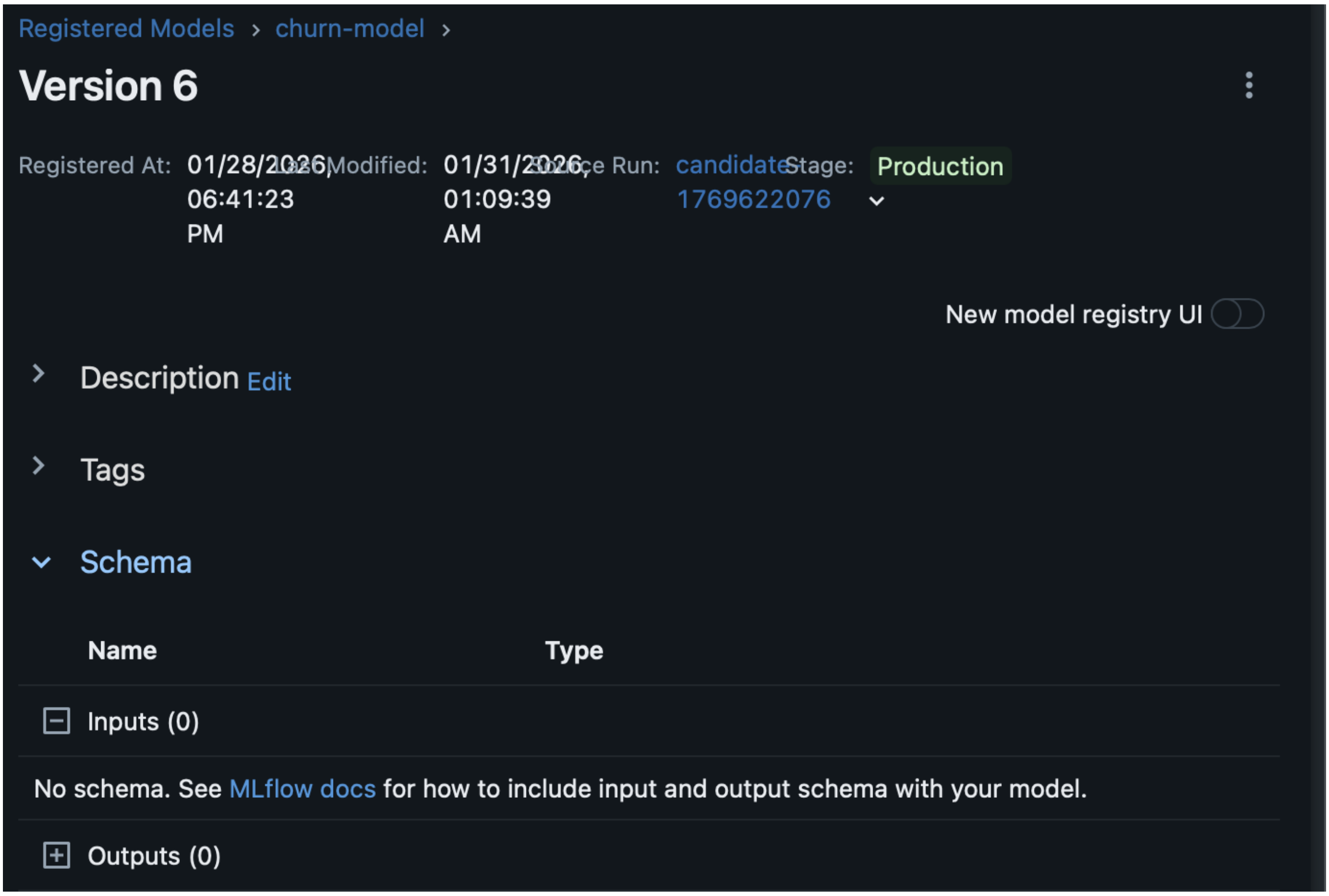


we are using the model_version = 6

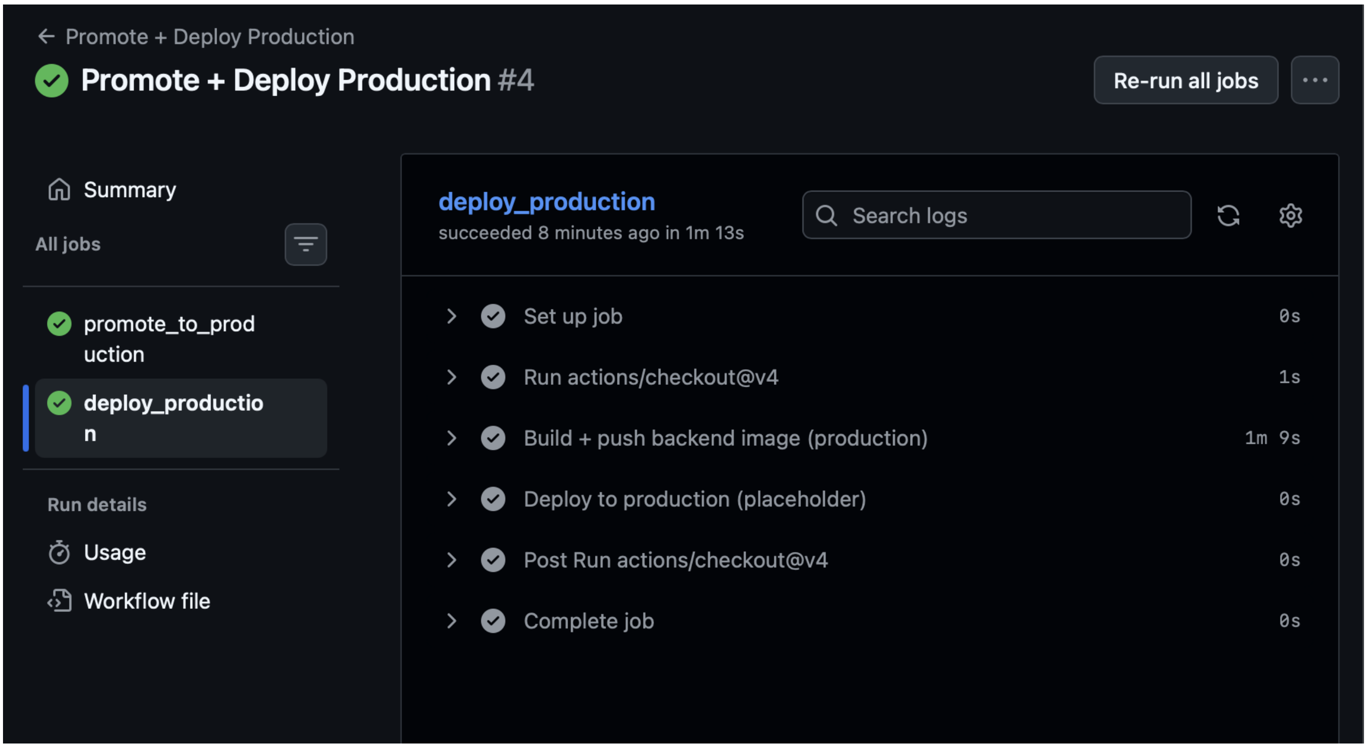




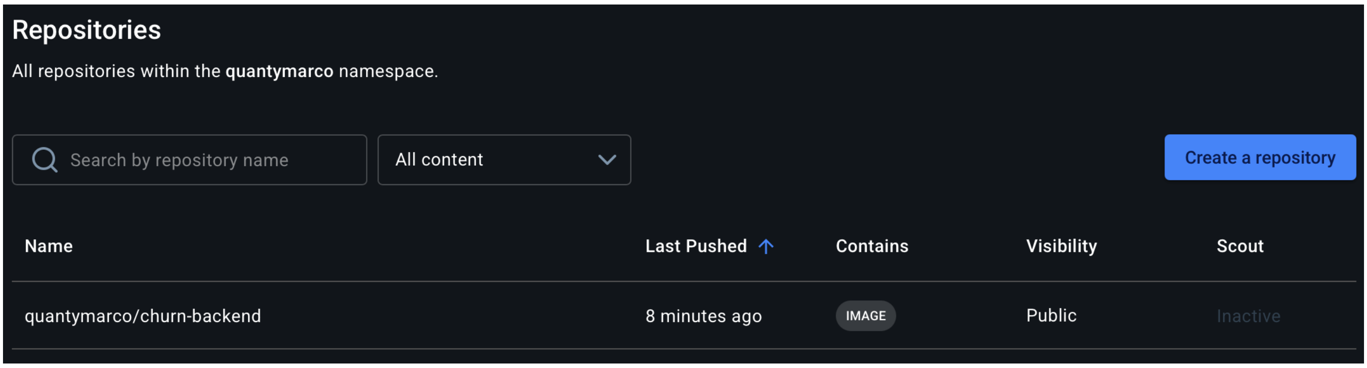
As we can see, the worflow updated the MLFlow registery, and this model is now the offical prodiction one.



Below we can see the deployment job using the production model, confirming the deployment step was executed successfully.

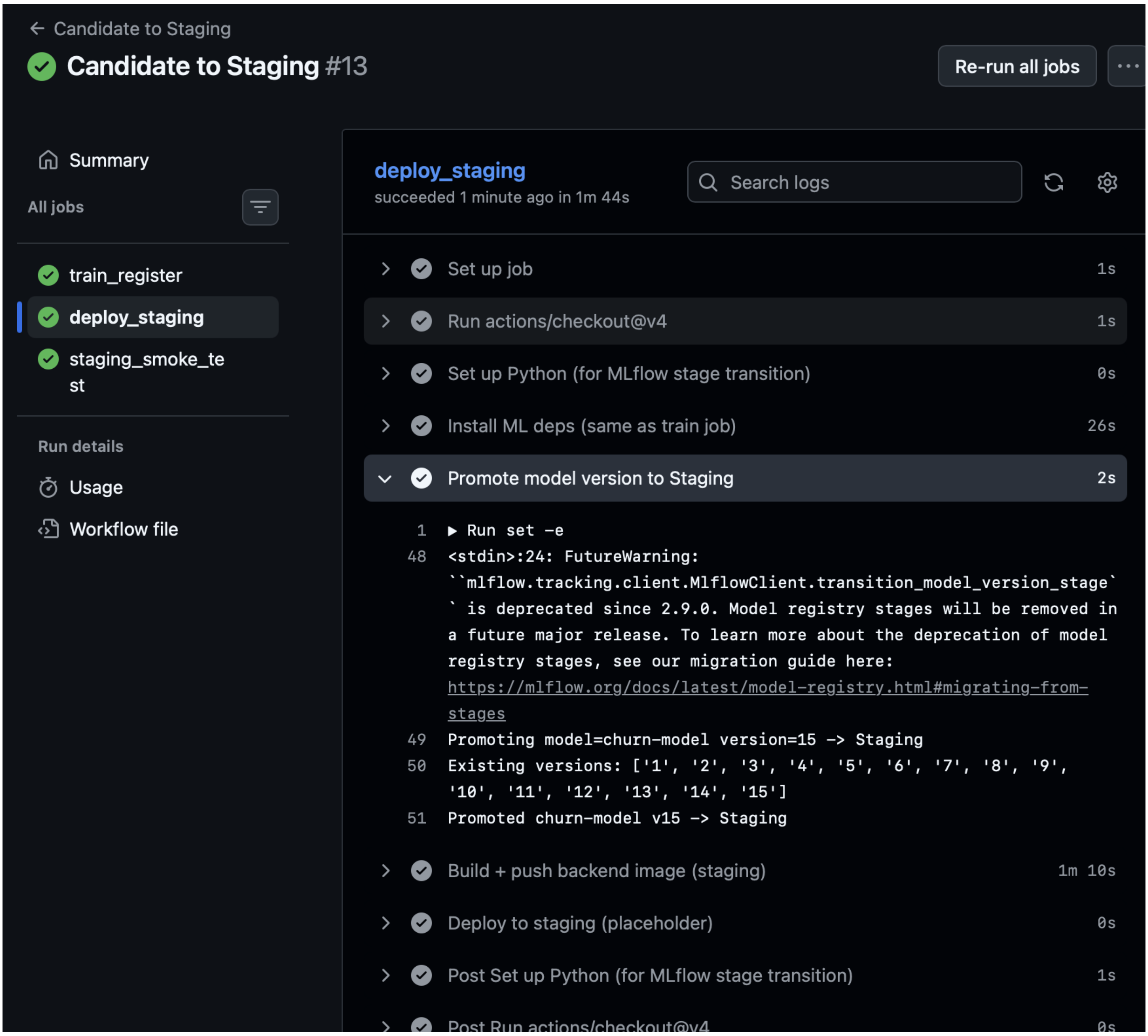


The docker image also has been created in DockerHub.

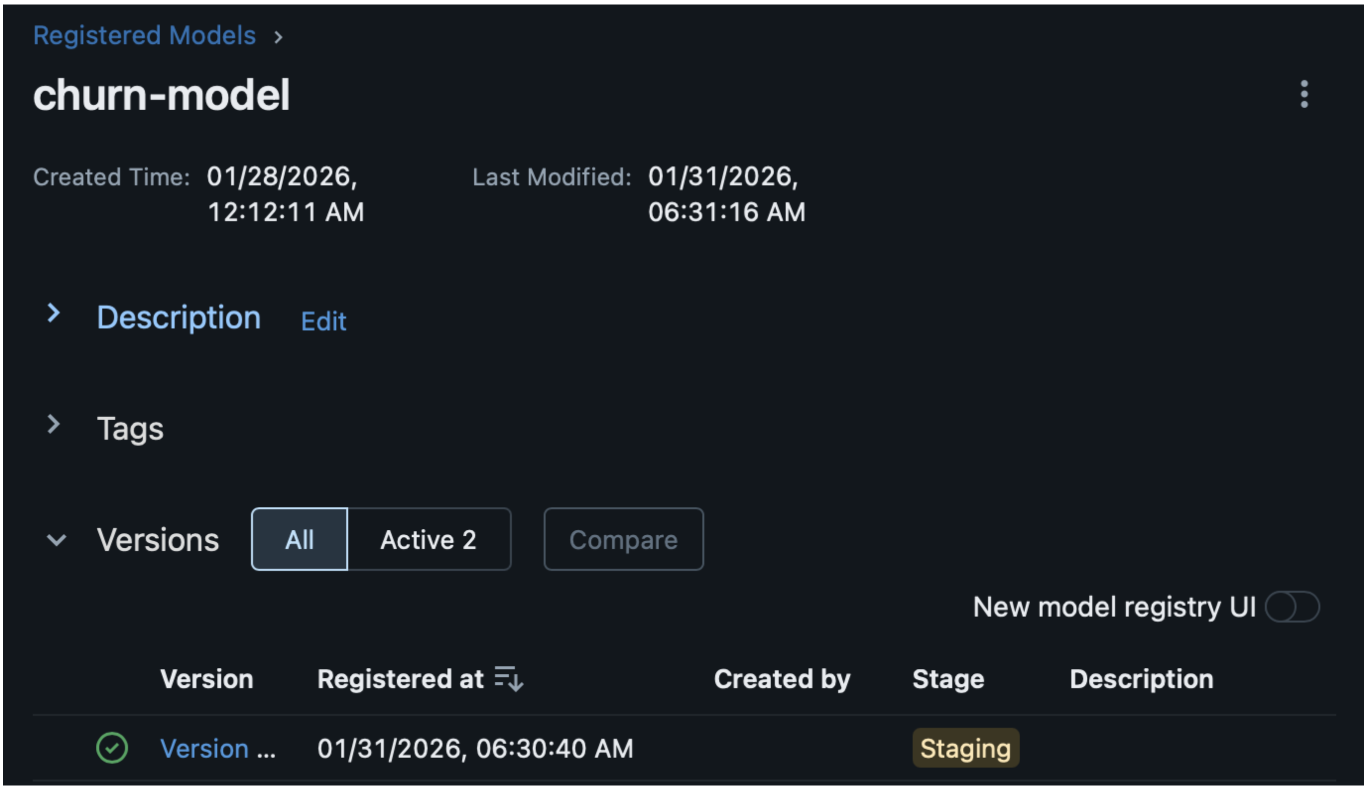


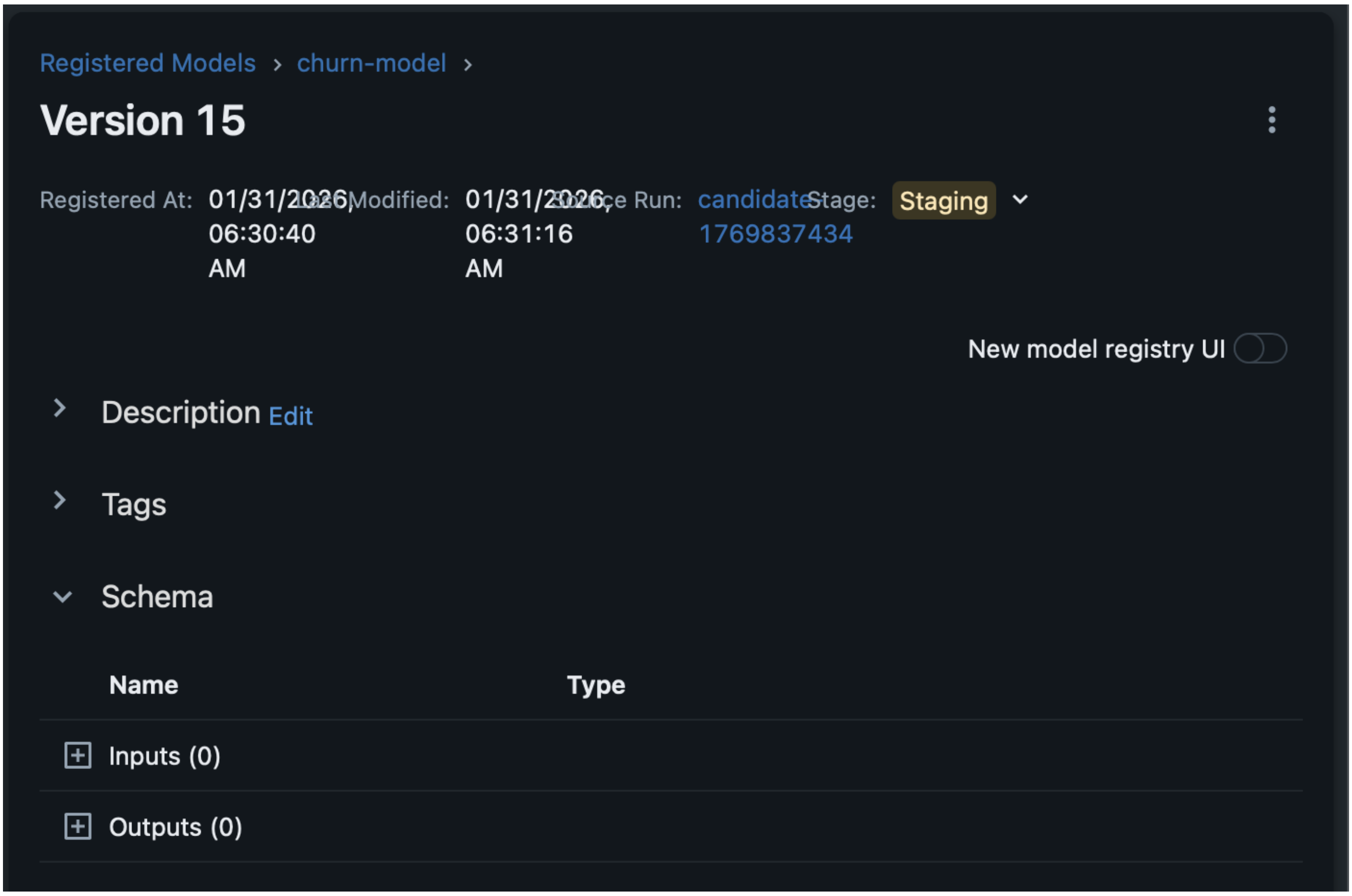
Task 4: Prove production uses registry stage, not "latest code"

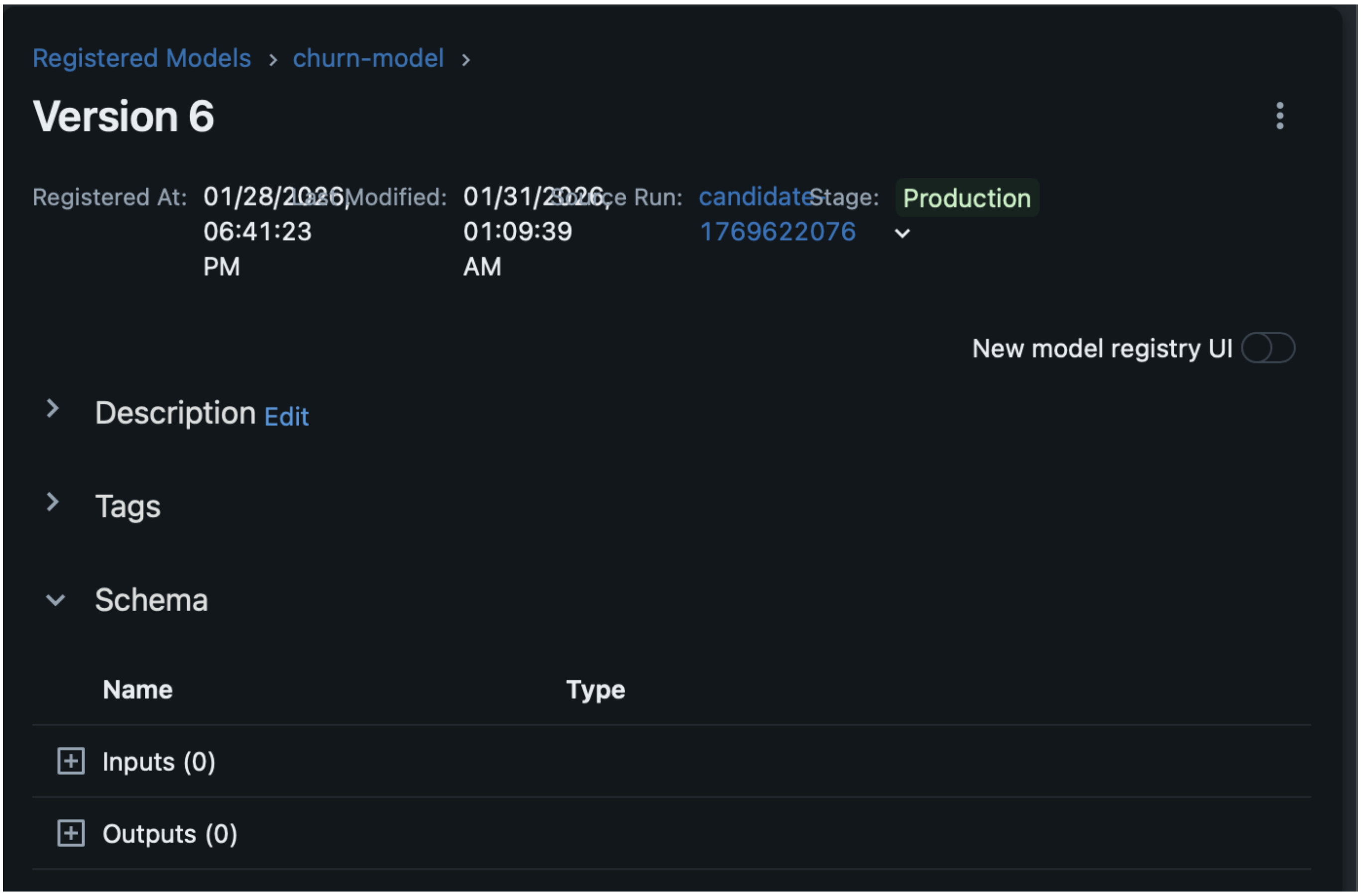
Locally: Run staging backend reading "Staging" and prod backend reading "Production" Verify /health returns correct stage



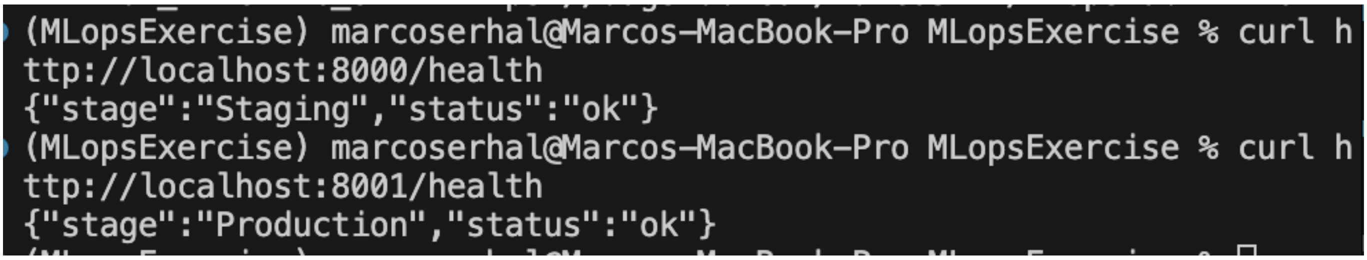
Above, we modified the accuracy threshold (from .90 to .80) to automatically allow one version to pass the gate and be set to staging. We could have also done it manually on MLFlow (I played a lot with it to understand the logic around it).


Registered Models ›
churn-model
Created Time: 01/28/2026, 12:12:11 AM
Last Modified: 01/31/2026, 06:31:16 AM
Description
Edit
Tags
Versions
All
Active 2
Compare
New model registry UI
Version
Registered at
Created by
Stage
Description
Version ...
01/31/2026, 06:30:40 AM
Staging

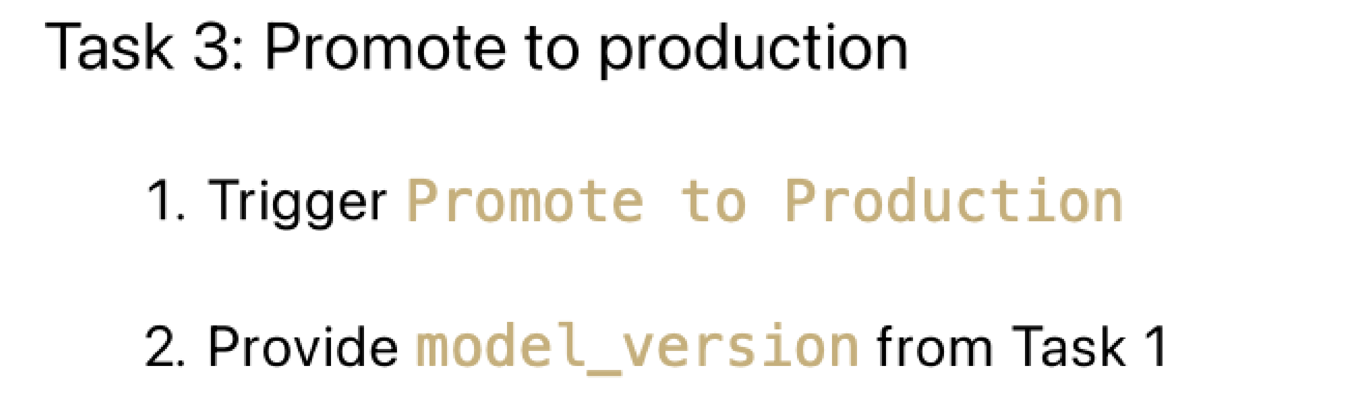

Registered Models › churn-model ›
Version 15
Registered At: 01/31/2026, 06:30:40 AM
Modified: 01/31/2026, 06:31:16 AM
Run: candidate 1769837434
Stage: Staging
New model registry UI
Description
Edit
Tags
Schema
Name
Type
Inputs (0)
Outputs (0)



We now have both versions of model in the registry and can then run both production and staging workflows and make sure that they are using the correct versions from above.



Clarification on gate vs stage transitions In the first “Candidate -> Staging” run, the pipeline successfully registered a new MLflow model version (v6) but the quality gate failed (accuracy < threshold as we saw). A gate failure does not prevent the version from existing in the registry—it only prevents the automatic transition to the “Staging” stage and any downstream automated promotion steps. For Task 3, the exercise required demonstrating a 'manual override' promotion to Production, so we promoted a specific version directly to “Production” to show the MLflow stage transition and the production deployment workflow



Later, we lowered the accuracy threshold and re-ran “Candidate -> Staging”. This produced a new model version (new run/artifact) that passed the gate and could be transitioned cleanly to “Staging”. Even if the training code and data generation were unchanged between runs, a re-run still creates a new MLflow version because it logs a new run and artifact; the only difference here was the gate threshold.

1. Why is it dangerous to deploy "whatever just merged to main" as the model?

It is dangerous because the main reflects code changes, not a validated and reproducible model artifact. It means that we can accidentally ship an untested model (or a model trained on different data), lose traceability (which data/params produced it), and make rollbacks hard because the main would move constantly.

2. What does the registry stage give you that a Git tag does not?

A registry stage points to a specific model artifact or version with its metrics, lineage, and lifecycle state (Staging/Production), not just code. It enables controlled promotion/rollback of models without changing Git history, and multiple model versions can coexist as the Production pointer moves.

3. If staging passes but production fails, what could be the causes?

The causes could be environment or runtime differences (different env vars/secrets, network access, permissions, data), different infrastructure constraints (CPU/memory limits), dependency mismatch, missing model/data access in prod (artifact store auth), config differences (different stage/URI, ports), or production-only traffic/schema edge cases that aren't covered in the staging tests.

4. Where should DVC fit in a serious pipeline:

DVC should be used to version all key datasets to make sure that experiments are comparable and reproductible. Training data snashot ensures that models can be retrained, the evaluation datast snapshot guarantees consistent comparisons and the drift reference dataset is used for monitoring, to detect when production data changes over time.

5. What should be added to the gate beyond accuracy?

Accuracy is good to check but not enough to show that we have a good production model. We should add latency checks to make sure the model is fast enough in real-time use. The schema checks are here to prevent crashes from input format change. Fairness constraints is like its name suggests, reduce biais but also unethical behavior. Adversarial or robustness tests ensure that the model remains stable if we use noisy inputs or like its name says adversarial inputs.

Extensions (optional)

'use ngrok to add a public-facing endpoint that you can use to automatically run deployments locally'

We run only our backend locally with 'docker compose -f deploy/docker-compose.staging.yml up --build'. Then we expose it using 'ngrok http 8000'

```
ngrok                                                    (Ctrl+C to quit)

  One gateway for every AI model. Available in early access *now*: h

Session Status                online
Account                       Marco (Plan: Free)
Version                       3.35.0
Region                        Europe (eu)
Latency                       23ms
Web Interface                 http://127.0.0.1:4040
Forwarding                    https://carmen-tumid-cultivatedly.ngrok

Connections                   ttl     opn     rt1     rt5     p50
                              0       0       0.00    0.00    0.00
```

We can see that /health returns same response for both locally and using the ngrok public URL, which confirms that the local deployment is reachable externally, enabling remote triggers/tests against the local environment.

```
(MLopsExercise) marcoserhal@Marcos-MacBook-Pro MLopsExercise % curl https://carm
en-tumid-cultivatedly.ngrok-free.dev/health
{"stage":"Staging","status":"ok"}
(MLopsExercise) marcoserhal@Marcos-MacBook-Pro MLopsExercise %
```