# GitLab CI Pipeline Failure Runbook Pack

**A senior engineer's incident runbook for a red pipeline — from a failed job log to runners, executors, variables, and deploys.** DevOps AI ToolKit · devopsaitoolkit.com

A GitLab CI pipeline is a graph of jobs grouped into stages, dispatched by the coordinator to registered runners. When a pipeline goes red, the pipeline view only tells you *which* job failed — the real signal is in that job's log: the commands GitLab ran, their output, and the final exit code. Work outside-in: read the failing job first, decide whether the failure is *configuration* (bad `.gitlab-ci.yml`, no eligible runner, missing variable) or *execution* (the runner ran your script and it exited non-zero). Those two classes have completely different fixes.

> Examples use sanitized names (`registry.example.com`, `group/project`). Commands assume a Linux runner and GitLab 15+. Read-only checks first; change CI/CD settings or restart runners with a change record.

## 1. Read the Failure: Pipeline vs Job vs Exit Code

Start at the pipeline, but the answer is always in a single job.

```
# From a clone, see recent pipeline/job status without the UI (needs a token)
glab ci status                      # current branch's pipeline
glab ci view                        # interactive stage/job graph
glab ci trace <job-id>              # stream a specific job's full log
```

- **Pipeline "failed" vs job "failed":** the pipeline is failed because *at least one* non-`allow_failure` job failed. Find that job — ignore the green ones.
- **Read the very end of the job log.** GitLab prints `ERROR: Job failed: exit code 1` (script error) vs `ERROR: Job failed (system failure)` (runner/executor problem — image pull, timeout, lost connection). These are different §s below.
- `exit code 1` = *your* command failed; the runner did its job. `system failure` = the environment failed before/around your script.
- Note the **stage** and whether earlier stages passed — a failure in `test` after `build` succeeded narrows scope fast.

## 2. Invalid `.gitlab-ci.yml` (Lint, Syntax, Stages, Rules)

If the pipeline shows "yaml invalid" or never created expected jobs, it's config, not code.

```
# Validate the config for a project against the live server (authoritative lint)
glab ci lint .gitlab-ci.yml
# Or via API: POST the file to the project-scoped CI lint endpoint
curl -s --header "PRIVATE-TOKEN: $TOKEN" \
  "https://gitlab.example.com/api/v4/projects/<id>/ci/lint" \
  --data-urlencode "content@.gitlab-ci.yml" | jq '.valid, .errors'
```

- Use the **project-scoped** lint (`/projects/:id/ci/lint`), not the global one — it resolves `include:`, variables, and `rules` the way the pipeline actually will.
- Every job must reference a stage that exists in the top-level `stages:` list (or the built-in `.pre` / `build` / `test` / `deploy` / `.post`). A typo'd stage name silently drops the job or errors.
- `rules:` and `only/except` are mutually exclusive **per job** — mixing them is a lint error. See §9.
- Common breakers: tabs instead of spaces, an unquoted `:` inside a `script` line, a `script:` that's a map instead of a list.

## 3. No Runners Available / Jobs Stuck Pending

A job stuck on "This job is stuck because you don't have any active runners..." is a *matching* problem, not a code problem.

```
# On a runner host: is the runner process up and registered?
sudo gitlab-runner verify             # confirms each config is still valid on the server
sudo gitlab-runner list               # registered runners in config.toml
systemctl status gitlab-runner

# Project → Settings → CI/CD → Runners shows online/paused and tags in the UI/API:
curl -s --header "PRIVATE-TOKEN: $TOKEN" \
  "https://gitlab.example.com/api/v4/projects/<id>/runners" | jq '.[] | {description, online, status, tag_list}'
```

- **Tag mismatch** is the #1 cause: a job with `tags: [docker, linux]` only runs on a runner registered with *both* those tags. Either add the tags to a runner or relax the job's `tags:`.
- An **untagged job** runs only on runners that have "Run untagged jobs" enabled. If all your runners are tag-restricted, untagged jobs hang forever.
- Check **online status** (`online: true`) and that the runner isn't **paused**. A runner that stopped heartbeating shows `status: offline` after ~1 min.
- **Protected branches:** a runner marked "protected" only picks up jobs on protected branches/tags. A job on a feature branch won't

match it.

## 4. Docker Executor Failures (Image Pull, DinD, Services)

The `docker` executor runs each job in a container from `image:`; failures cluster around pulling that image or reaching services.

```
# Correct DinD pattern for building images inside CI
build:
  image: docker:24.0
  services:
    - docker:24.0-dind
  variables:
    DOCKER_HOST: tcp://docker:2376
    DOCKER_TLS_CERTDIR: "/certs"          # empty string ("") disables TLS if 2375 is used
  script:
    - docker info
    - docker build -t registry.example.com/group/project:$CI_COMMIT_SHORT_SHA .
```

- **`image pull failed` / `manifest unknown`:** wrong tag or the runner can't reach the registry. Test the runner's own pull path; for a private registry, jobs need `DOCKER_AUTH_CONFIG` (a CI/CD variable holding a docker `config.json`).
- **`Cannot connect to the Docker daemon`:** DinD service not up or `DOCKER_HOST`/`DOCKER_TLS_CERTDIR` mismatch. The `docker:dind` service and the `docker` client must agree on TLS (`/certs`) vs plain `tcp://docker:2375`.
- **Services unreachable:** a service like `postgres:15` is reachable at the hostname `postgres` (the image name), not `localhost`. Job and service share a network namespace only for `localhost` when the runner uses `services_limit`/`FF_NETWORK_PER_BUILD`.
- **`no space left on device`:** the runner host's Docker storage filled with layers — prune on the host, not in the job.

## 5. Shell Executor Failures (Host Env, Permissions, Tools)

The `shell` executor runs your `script` directly on the runner host as the `gitlab-runner` user — no isolation, so the host's state *is* the build environment.

```
# Reproduce as the runner user on the host
sudo -u gitlab-runner -i bash -lc 'which node && node --version && env | sort'
```

- **`command not found`:** the tool isn't installed for the `gitlab-runner` user, or isn't on that user's non-login `PATH`. Shell executor doesn't source your interactive profile the same way your SSH session does.
- **Permission denied writing/deploying:** `gitlab-runner` isn't in the needed group (e.g. `docker`, or a deploy dir's group). Fix with group membership, not `chmod 777`.
- **State bleed between jobs:** shell executor shares the host — a previous job's leftover files, running processes, or env can poison the next. Prefer `docker`/`kubernetes` for isolation; if you must use shell, clean up explicitly.
- **`fatal: unable to access` on clone:** the host's git/CA trust or a stale `GIT_STRATEGY` build dir. Try `GIT_STRATEGY: clone` to force a fresh checkout.

## 6. Kubernetes Executor Failures

The `kubernetes` executor spawns a pod per job (a `build` container plus any `services`, and a `helper`). Failures are pod-scheduling and image problems in disguise.

```
# Watch the transient CI job pods in the runner's namespace
kubectl -n gitlab-runner get pods -w
kubectl -n gitlab-runner describe pod <runner-job-pod>     # Events explain most failures
kubectl -n gitlab-runner logs <pod> -c build               # the actual script output
```

- **`ImagePullBackOff`:** wrong `image:` or missing registry pull secret. Set `[runners.kubernetes]` `image_pull_secrets` or provide `DOCKER_AUTH_CONFIG`.
- **Pod `Pending` → job stuck/timeout:** no node has the requested CPU/memory. Check `describe pod` Events for `Insufficient cpu/memory` and tune `[runners.kubernetes]` requests/limits or `node_selector`/`tolerations`.
- **`context deadline exceeded` / helper can't attach:** `poll_timeout` too low for slow image pulls, or the runner→apiserver path is flaky.
- **RBAC:** the runner's ServiceAccount needs create/list/delete on pods/exec in its namespace, or every job fails at pod creation.

## 7. CI/CD Variables & Protected/Masked Secrets

An empty variable at runtime is almost always a *scoping* rule, not a missing value.

```
deploy:
  script:
    - test -n "$DEPLOY_TOKEN" || { echo "DEPLOY_TOKEN is empty"; exit 1; }
    - echo "len=${#DEPLOY_TOKEN}"        # print length, never the value
```

- **Protected variables** are only injected on **protected branches/tags**. A pipeline on a feature branch sees them as empty — this is the single most common "works on main, empty on MR" bug. Either unprotect the variable or run the job on a protected ref.
- **Masked variables** are hidden in job logs but still fully available to the script. Masking requires the value to meet rules (base64 alphabet, ≥8 chars, single line) — a value that can't be masked either fails to save masked or leaks; check the variable's settings.
- **Precedence:** job/pipeline `variables:` and `include`d ones can override project settings. Trigger/manual pipeline variables and `dotenv` artifacts (from `needs`) layer on top. When a value is "wrong," dump `echo "len=${#VAR}"` to prove which layer won.
- **Environment scope:** a variable scoped to `production` won't exist in a job whose `environment:` is `staging`.

---

## 8. Cache & Artifact Upload/Download Failures

Cache is best-effort speedup; artifacts are contractually passed between jobs. Confuse them and jobs "lose" files.

```
build:
  script: [ "npm ci", "npm run build" ]
  cache:                                  # keyed reuse, may be absent — never rely on it
    key: ${CI_COMMIT_REF_SLUG}
    paths: [ node_modules/ ]
  artifacts:                              # guaranteed hand-off to later stages
    paths: [ dist/ ]
    expire_in: 1 week
```

- **`Uploading artifacts... ERROR: too large`:** the path exceeds the instance `max artifact size`. Narrow `paths:`, raise the limit (admin), or use `expire_in` + external storage.
- **Downstream job can't find files:** artifacts flow **forward by stage** automatically, or explicitly via `needs:` / `dependencies:`. If you set `dependencies: []`, you opt *out* of all artifacts. Files from `cache:` are **not** guaranteed downstream — use `artifacts:` for hand-offs.
- **`WARNING: … no matching files`:** the `paths:` glob didn't match — the build wrote somewhere else, or a prior step failed silently. Assert the file exists at the end of `script`.
- **Cache never hits:** the `key:` differs per branch/commit, or object storage (S3/MinIO) creds in `config.toml` are wrong — check the runner's cache config, not the job.

---

## 9. rules/only/except & needs/dependency Errors

Whether a job exists, and in what order, is decided before any script runs.

```
test:
  rules:
    - if: '$CI_PIPELINE_SOURCE == "merge_request_event"'
    - if: '$CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH'
      when: always
    - when: never                         # explicit default: don't run otherwise
```

- `rules` **replace** `only/except` **— don't mix them in one job** (lint error). Prefer `rules:`; it's evaluated top-down and the **first match wins**.
- **Job unexpectedly didn't run:** no rule matched (implicit `when: never`), or the ref doesn't match. Add a trailing `- when: never` / `- when: on_success` so intent is explicit.
- **`needs:` errors:** a job can only `need` a job in an **earlier or same** stage (with `needs`, it starts as soon as those finish — DAG). Needing a later-stage job is invalid. A typo'd needed job name fails config validation.
- `needs` **+ artifacts:** `needs: [{ job: build, artifacts: true }]` pulls that job's artifacts specifically; omit it and you get the DAG ordering without the files.
- `only/except` still works but is legacy — `only: [main]` vs `except: [tags]`. New pipelines should use `rules:` + `$CI_COMMIT_BRANCH` / `$CI_PIPELINE_SOURCE`.

---

## 10. Job Timeout, `script` exit 1, and Deployment/Environment Failures

The last mile: the script ran, or ran too long, or the deploy target rejected it.

```
deploy_prod:
  stage: deploy
  environment:
    name: production
    url: https://app.example.com
  rules:
    - if: '$CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH'
      when: manual                        # gate prod behind a manual click
  timeout: 30m
  script:
    - kubectl config use-context group/project:prod-agent
    - kubectl -n prod set image deploy/app app=registry.example.com/group/project:$CI_COMMIT_SHORT_SHA
    - kubectl -n prod rollout status deploy/app --timeout=120s
```

- **`ERROR: Job failed: execution took longer than … seconds`:** hit the job `timeout:` or the project/runner timeout (runner timeout caps the

project's). Raise the right one, or make the script faster — a 1h build is usually the bug.

- `exit code 1` **from your script:** GitLab fails the job on the first non-zero command (it runs `set -e` semantics). Scroll up from the failure line to the *first* red command, not the last. Use `command || true` only where failure is genuinely acceptable.
- **Deployment failed, environment stuck:** `rollout status` timed out means the new pods didn't become Ready — debug the *cluster* (§6 style: `kubectl describe` / `logs`), not the CI YAML. The pipeline is just the messenger.
- `allow_failure: true` turns a red job amber (pipeline stays green) — useful for flaky/optional checks, dangerous if you hide a real deploy failure behind it.

## Decision Tree

```
Pipeline is RED — which class of failure?
 ├─ "yaml invalid" / expected jobs missing? ── yes → §2 CI Lint: stages, rules vs only/except, includes
 └─ a specific JOB failed
     ├─ log ends "Job failed (system failure)"? ── environment, not your code
     │    ├─ "stuck… no active runners"      → §3 tags / online / protected / untagged
     │    ├─ image pull / daemon / dind      → §4 Docker executor
     │    ├─ command not found / permission  → §5 Shell executor (host state)
     │    ├─ ImagePullBackOff / Pod Pending  → §6 Kubernetes executor
     │    └─ "execution took longer than…"   → §10 timeout (job vs runner cap)
     └─ log ends "Job failed: exit code 1" ── your script ran and failed
          ├─ var empty on MR but set on main?  → §7 protected/masked variable scope
          ├─ downstream "no matching files"?   → §8 artifacts vs cache, needs/dependencies
          ├─ job didn't run at all?            → §9 no rule matched / needs ordering
          └─ deploy rollout timed out?         → §10 debug the target cluster, not the YAML
```

## Copy/Paste One-Shot Triage

```
PID=<project-id>; TOKEN=<read-token>; API=https://gitlab.example.com/api/v4
echo "== last pipeline ==";  curl -s -H "PRIVATE-TOKEN: $TOKEN" "$API/projects/$PID/pipelines?per_page=1" | jq '.[0] | {id,status,ref,sha}'
echo "== failed jobs ==";    PLID=$(curl -s -H "PRIVATE-TOKEN: $TOKEN" "$API/projects/$PID/pipelines?per_page=1" | jq '.[0].id'); \
  curl -s -H "PRIVATE-TOKEN: $TOKEN" "$API/projects/$PID/pipelines/$PLID/jobs?scope=failed" | jq '.[] |
{name,stage,failure_reason,runner:.runner.description}'
echo "== runners ==";        curl -s -H "PRIVATE-TOKEN: $TOKEN" "$API/projects/$PID/runners" | jq '.[] | {description,online,status,tag_list}'
echo "== lint config ==";    curl -s -H "PRIVATE-TOKEN: $TOKEN" "$API/projects/$PID/ci/lint" --data-urlencode "content@.gitlab-ci.yml" | jq
'{valid,errors}'
```

## Incident Notes Template

```
INCIDENT: GitLab CI Pipeline Failure
Started (UTC):          ____   Detected by: ____   Sev: ____
Project / pipeline:     group/project  #____   ref: ____   sha: ____
Failed job(s):          name ____   stage ____
Failure class:          [ ] config (yaml/lint)  [ ] system failure  [ ] script exit 1
Job log ends with:      ____ (exit code / system-failure reason)
Runner:                 desc ____  online Y/N  tags ____  protected Y/N
Executor:               [ ] docker  [ ] shell  [ ] kubernetes   evidence: ____
Variable/secret:        [ ] protected-scope miss  [ ] masked/empty  var: ____
Artifacts/cache:        [ ] needs/dependencies  [ ] no matching files  [ ] too large
Rules/needs:            [ ] no rule matched  [ ] needs ordering  [ ] only/except mix
Action taken:           (config fix / runner tag / var unprotect / retry + who + when)
Result:                 [ ] passed @ ____   [ ] escalated to ____
Root cause:             ____
Follow-ups:             (lint in MR, runner capacity, timeouts, secret scope, alerts) ____
```