Devops Shack

Common Mistakes with GitLAB CI

(How to Avoid Them)

BY DEVOPS SHACK

[Click here for DevSecOps & Cloud DevOps Course]

DevOps Shack

Common Mistakes with GitLab CI

(How to Avoid Them)

Table of Content

- Allows unreviewed changes to go live, increasing risk.

8. Missing before_script and after_script Standardization

- Causes duplicated code and inconsistent environment setup.

9. Ignoring Job and Pipeline Failures

- Creates false positives and hides broken stages.

10. No Monitoring or Notifications on CI/CD Events

- Delays incident response and blinds teams to pipeline health.

☑ 1. Triggering Pipelines on Every Branch Push

⚠ The Problem

By default, GitLab CI/CD will trigger pipelines on every push to any branch unless explicitly configured otherwise. In many projects, this leads to **CI jobs being executed on every commit**, including:

- Minor documentation changes
- Feature branches under active development
- Work-in-progress (WIP) commits
- Quick fixes that don't need full pipeline runs

This overloads GitLab runners, slows down the pipeline queue, burns unnecessary compute minutes (especially on shared runners), and — in the worst cases — **triggers deployments from untested or incomplete code**.

🔬 Real-World Impact

- **Noise:** Dozens of pipelines running with no intention to deploy
- **Cost:** Runner time consumed needlessly
- **Risk:** A branch push triggered a deployment job and auto-deployed an incomplete feature
- **Delays:** Important pipelines queued behind unimportant ones

🛠 The Right Way to Handle This

☑ Use rules: to Define Exactly When a Job Should Run

GitLab CI offers a rules: block to control **when** and **under what conditions** jobs run.

☑ Example 1: Only Run Job on Merge Requests to main

```
deploy:
  stage: deploy
```

```
  script:
    - ./deploy.sh
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
      when: manual
    - when: never
```

☑ Example 2: Run Tests Only on Feature Branches or MRs

```
test:
  stage: test
  script:
    - npm test
  rules:
    - if: '$CI_PIPELINE_SOURCE == "merge_request_event"'
    - if: '$CI_COMMIT_BRANCH =~ /^feature\/.+/'
```

🧠 Key Concepts to Know

- CI_COMMIT_BRANCH → the branch name triggering the pipeline
- CI_PIPELINE_SOURCE → identifies what caused the pipeline (push, merge_request_event, schedule, etc.)
- rules: → supersedes only/except and provides more flexibility
- when: manual → adds a manual approval step for sensitive jobs (like deploy)

💡 Tips to Optimize Triggering Behavior

- Use rules: to **differentiate pipelines** between:
  - Merge requests (for tests & reviews)
  - Pushes to main/release (for deploy)

  - Schedules (for backups or cleanup jobs)

- Introduce **manual deployments** that require approval:

```
deploy_production:
  stage: deploy
  script: ./deploy-prod.sh
  environment: production
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
      when: manual
```

- Add **commit message conditions** to skip pipelines if needed:

```
rules:
  - if: '$CI_COMMIT_MESSAGE =~ /\[skip ci\]/'
    when: never
```

☑ What We Learned

1. **Not all pushes need a pipeline** — filter them precisely using rules:
2. Uncontrolled triggers waste resources and may lead to **accidental deployments**
3. Use **branch names**, **pipeline sources**, and **commit messages** to trigger pipelines only when appropriate
4. Manual approval gates and conditional logic improve both **speed** and **safety**

☑ 2. Misusing only and except Instead of rules

⚠ The Problem

GitLab CI/CD historically used only and except to control when jobs run. Many teams still rely on these, but they are now considered **less flexible and harder to maintain** compared to the modern rules: syntax.

Here's an example of older usage:

```
deploy:
  script: ./deploy.sh
  only:
    - main
```

At first glance, this seems fine. But only/except:

- **Don't allow conditional logic** based on pipeline source
- Can't evaluate **commit messages, file changes, variables**, or **merge request context**
- Lead to **unexpected job runs** in complex pipelines (especially when multiple triggers exist: push, tag, MR, etc.)

🔬 Real-World Impact

- Jobs triggered during merge_request_event even when they shouldn't
- CI pipelines ran on tags even when deployment wasn't required
- Developers confused why a job triggered (no context visibility in only)
- Multiple jobs skipped unintentionally because except masked other branches

🛠 The Right Way to Handle This

☑ Use rules: Instead of only/except

rules: is **declarative**, **powerful**, and **context-aware**. You can use it to:

- Run jobs only on specific branches

- Include/exclude pipelines based on trigger source
- Check for commit message content
- Use custom variables to trigger conditional logic

🔄 **Migration Example: From only/except to rules:**

❌ **Before (old style):**

```
deploy:
  script: ./deploy.sh
  only:
    - main
  except:
    - tags
```

☑️ **After (modern style):**

```
deploy:
  script: ./deploy.sh
  rules:
    - if: '$CI_COMMIT_BRANCH == "main" && $CI_PIPELINE_SOURCE == "push"'
      when: always
    - when: never
```

▧ **Useful Conditions for rules:**

- CI_COMMIT_BRANCH == "main" → for main branch only
- CI_COMMIT_TAG =~ /^v\d+\.\d+\.\d+$/ → for versioned tags
- CI_PIPELINE_SOURCE == "merge_request_event" → for MRs
- CI_COMMIT_MESSAGE =~ /\[skip-deploy\]/ → to skip on demand
- $MY_ENVIRONMENT == "prod" → using custom variables

🔒 Safer and Smarter Pipelines

rules: allows for **granular, layered control**, such as:

```
deploy:
 script: ./deploy.sh
 rules:
  - if: '$CI_COMMIT_BRANCH == "main"'
    when: manual
  - if: '$CI_COMMIT_BRANCH == "release/*"'
    when: always
  - when: never
```

This logic handles:

- **Auto deploy on release branches**
- **Manual approval on main**
- **Prevents job from running elsewhere**

☑ What We Learned

1. only/except are deprecated for complex pipelines
2. rules: provide **greater clarity**, **context awareness**, and **reliability**
3. You can use conditions on **branches, triggers, variables, messages**, and more
4. Migrating to rules: simplifies troubleshooting and improves pipeline maintainability
5. Every job should be **intentionally triggered**, and rules: make that easy

☑ 3. Hardcoding Secrets in .gitlab-ci.yml

⚠ The Problem

A common and dangerous mistake in GitLab CI/CD is **placing sensitive credentials, tokens, passwords, or keys directly inside the .gitlab-ci.yml file** or hardcoded scripts. For example:

```
script:
  - curl -u "admin:SuperSecret123" https://api.example.com/deploy
```

Since .gitlab-ci.yml is typically version-controlled and visible to all project members, this exposes secrets to:

- Internal developers who shouldn't have access
- Accidental commits and Git history
- CI logs and job artifacts
- Potential exfiltration if the repo is public or cloned

This violates **security best practices**, breaks **compliance rules**, and opens the door to **severe breaches**, such as unauthorized access to databases, APIs, cloud resources, or production systems.

🔬 Real-World Impact

- Secrets were accidentally committed and **pushed to Git history**, visible to contributors
- An API token printed during a curl command ended up in CI logs and leaked via job artifacts
- A staging token reused in production was discovered by attackers scanning public GitLab instances
- Teams scrambled to **rotate tokens**, revoke credentials, and audit logs after exposure

🛠 The Right Way to Handle This

✅ 1. Use GitLab CI/CD Variables for All Secrets

GitLab provides a secure way to manage secrets via **CI/CD Variables**:

- Go to **Settings → CI/CD → Variables**
- Add secrets as **Masked** and **Protected** (so they're hidden in logs and restricted to protected branches/tags)

Then use them safely in .gitlab-ci.yml:

```
script:
  - curl -u "admin:$API_TOKEN" https://api.example.com/deploy
variables:
  API_TOKEN: ${{ CI_JOB_TOKEN }}
```

☑ **2. Never Echo or Print Secrets in Logs**

Even if using variables, avoid printing them:

✗ Wrong:

```
script:
  - echo $DB_PASSWORD
```

☑ Right:

```
script:
  - psql -U user -d db -W <<< "$DB_PASSWORD"
```

Use set +x or disable command echoing in shell scripts to avoid accidental leak.

☑ **3. Use External Secret Managers When Needed**

For sensitive or dynamic secrets, integrate with tools like:

- **HashiCorp Vault**
- **AWS Secrets Manager**
- **Azure Key Vault**

You can fetch secrets during the pipeline run:

```
script:
  - DB_PASSWORD=$(vault kv get -field=password secret/db)
```

☑ 4. Rotate and Revoke Leaked Credentials Immediately

If a secret was ever committed:

1. **Revoke it**
2. **Rotate it**
3. Use tools like git-filter-repo or BFG to purge it from Git history:

```
bfg --delete-files secrets.env
```

☑ 5. Scan Your Git Repo for Secrets

Use tools in your pipeline:

- Gitleaks
- TruffleHog
- GitLab's own Secret Detection

Example with Gitleaks in .gitlab-ci.yml:

```
gitleaks:
  image: zricethezav/gitleaks
  script:
    - gitleaks detect --source=. --exit-code 1
```

☑ What We Learned

1. Secrets should **never be stored** in .gitlab-ci.yml or committed to Git
2. Use **GitLab CI/CD variables**, marked as **masked and protected**
3. Prevent secrets from leaking into **logs, artifacts, or echo commands**
4. Use external secret managers for better control and rotation
5. Always **scan your repositories** and pipelines for exposed secrets

☑ 4. Not Caching Dependencies Properly

⚠ The Problem

In many GitLab CI/CD pipelines, developers forget to use **caching** for package dependencies like Node modules, Python wheels, Maven artifacts, Docker layers, etc.

Without caching:

- Dependencies are downloaded and re-installed from scratch on **every pipeline run**
- CI jobs take longer to execute
- Bandwidth is wasted
- Build speed varies inconsistently

This becomes a major bottleneck in medium-to-large projects where **the same dependencies are reused across jobs**.

🔬 Real-World Impact

- Pipelines slowed down from **2–3 minutes to 10–15 minutes**
- Redundant downloads increased cost and build time
- Developers lost productivity due to longer feedback loops
- Teams avoided running CI on small changes because it felt "too slow"

🛠 The Right Way to Handle This

☑ 1. Use the cache: Keyword Correctly

GitLab's cache is used to **persist files** (like downloaded dependencies) **between jobs and pipelines**.

Example: Node.js Project

```
cache:
  key: ${CI_COMMIT_REF_SLUG}
  paths:
    - node_modules/

install_dependencies:
```

```
  script:
    - npm ci
```

✅ key: defines when to invalidate/reuse the cache. You can use:

- default → shared cache
- ${CI_COMMIT_REF_SLUG} → per-branch cache
- package-lock.json checksum → changes only when dependencies do

✅ **2. Differentiate Between cache: and artifacts:**

- cache: → speeds up **future jobs/pipelines** (usually dependencies)
- artifacts: → used to **pass build outputs** (like compiled files) between jobs **in the same pipeline**

Example with Artifacts:

```
build:
  stage: build
  script: npm run build
  artifacts:
    paths:
      - dist/
```

Example with Cache:

```
cache:
  paths:
    - node_modules/
```

✅ **3. Use Language-Specific Cache Directories**

Language	**Cache Directory**
Node.js	node_modules/

Language	Cache Directory
Python	.venv/, pip cache
Java	~/.m2/repository/
Go	GOPATH/pkg/mod/
Rust	~/.cargo/registry/, target/

Python Example:

```
cache:
  paths:
    - .venv
```

☑ 4. Use key: to Scope the Cache Properly

Per-branch cache:

```
cache:
  key: ${CI_COMMIT_REF_SLUG}
```

Global (shared) cache:

```
cache:
  key: default
```

Smart cache invalidation:

```
cache:
  key:
    files:
      - package-lock.json
```

☑ 5. Avoid Caching Mistakes

🚫 Don’t:

- Cache dist/ unless it's reused

- Cache .git directory
- Cache secrets or environment files

☑ Do:

- Limit cache size and expiry if using self-hosted runners
- Regularly review cache hit/miss stats in GitLab pipeline logs

☑ What We Learned

1. Not caching dependencies increases **build time and cost**
2. Use GitLab's cache: keyword to persist packages across jobs
3. Scope your cache with key: to avoid unnecessary rebuilds
4. Know when to use cache: vs artifacts: — they serve different purposes
5. Caching dependencies can **cut pipeline time in half**, improving DevOps velocity

☑ 5. Using latest Tags for Docker Images

⚠ The Problem

Using the latest tag in Docker builds and deployments is a common habit—but in CI/CD pipelines, it becomes a **major liability**.

In GitLab CI, developers often write:

```
docker build -t myapp:latest .
```

```
docker push myapp:latest
```

Then reference it in Kubernetes manifests or Docker Compose files:

```
image: registry.example.com/myapp:latest
```

This seems convenient—but the latest tag is **mutable**. Every time it's rebuilt, it **overwrites the previous version**. There's no guarantee of what version is actually running unless it's explicitly tracked.

🔬 Real-World Impact

- Deployments pulled the wrong image due to Docker layer cache or pull policy
- CI/CD pipelines broke because latest was updated by another branch
- Rollbacks became impossible — previous versions weren't tagged or stored
- Bugs reappeared due to redeploying an unexpected image under the same tag

🛠 The Right Way to Handle This

☑ 1. Use Git SHA or Version Tags Instead of latest

Generate versioned image tags automatically:

```
variables:
  GIT_SHA: $CI_COMMIT_SHORT_SHA


build:
  stage: build
  script:
    - docker build -t registry.example.com/myapp:$GIT_SHA .
    - docker push registry.example.com/myapp:$GIT_SHA
```

Update your deployment manifests accordingly:

```
image: registry.example.com/myapp:$CI_COMMIT_SHORT_SHA
```

☑ 2. Use tags in Git to Mark Releases

For production-grade releases, tag your Git commits semantically (v1.3.5) and use that as your Docker image tag:

```
git tag v1.3.5
git push origin v1.3.5
```

In your GitLab CI:

```
build:
  script:
    - docker build -t myapp:$CI_COMMIT_TAG .
```

☑ 3. Add Image Metadata for Traceability

Add Git metadata during your Docker build:

```
docker build \
  --label "org.opencontainers.image.revision=$CI_COMMIT_SHA" \
  --label "org.opencontainers.image.source=$CI_PROJECT_URL" \
  -t myapp:$CI_COMMIT_SHORT_SHA .
```

☑ 4. Disable Auto Pull of latest in Production

If you still reference latest, be sure to **pin the image digest**:

```
image: myapp@sha256:abc123def456...
```

Or explicitly set imagePullPolicy: IfNotPresent in Kubernetes:

```
imagePullPolicy: IfNotPresent
```

☑ 5. Keep Older Images for Rollback

Avoid deleting old images prematurely. Set retention policies that store at least 5–10 tagged versions, so you can easily roll back:

```
helm rollback myapp 3
```

Or:

```
kubectl set image deployment/myapp myapp=myapp:<previous-tag>
```

☑ What We Learned

1. The latest tag is **convenient but dangerous** in automated pipelines
2. Always tag images using Git SHA, tags, or semantic versions
3. Use GitLab CI variables like $CI_COMMIT_SHA, $CI_COMMIT_TAG to generate consistent tags
4. Avoid tag collisions that can **hide bugs**, **block rollbacks**, or **break reproducibility**
5. Traceability and rollback capability start with **proper image versioning**

☑ 6. No Separate Stages for Build, Test, and Deploy

⚠ The Problem

In many GitLab CI/CD pipelines, especially small or rushed projects, developers combine all tasks (build, test, deploy) into a **single job** or single script: block like this:

```
job:
  script:
```

```
  - npm install
  - npm test
  - npm run build
  - ./deploy.sh
```

This might work initially, but it's an anti-pattern that leads to:

- Poor visibility into which phase failed
- No separation of concerns
- Impossible rollback from failed deploys
- Test failures **after** the build already passed
- Deploys that trigger even if tests break
- Harder reuse of build artifacts or test results

🔬 Real-World Impact

- A test failure broke production deployment because the deployment wasn't gated
- Developers wasted time debugging why a job failed — was it the install, test, or deploy?
- Build artifacts weren't reusable across environments
- No way to run only tests or only deployments when needed

🛠 The Right Way to Handle This

☑ 1. Use GitLab CI Stages: build, test, deploy

Break your pipeline into **clear, sequential stages**:

```
stages:
 - build
 - test
 - deploy
```

☑ 2. Define Jobs per Stage

Each job should handle only **one responsibility**:

```yaml
build:
  stage: build
  script:
    - npm install
    - npm run build
  artifacts:
    paths:
      - dist/

test:
  stage: test
  script:
    - npm test

deploy:
  stage: deploy
  script:
    - ./deploy.sh
  dependencies:
    - build
```

☑ 3. Use Artifacts Between Stages

Artifacts let you **pass built files** from one stage to another:

```
artifacts:
  paths:
    - dist/
  expire_in: 1 hour
```

This ensures your deploy stage doesn't rebuild code unnecessarily.

☑ 4. Gate Deployments Using Rules or Manual Triggers

Prevent auto-deploys on test failure:

```
deploy:
  stage: deploy
  script:
    - ./deploy.sh
  when: manual
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
```

This ensures deploys only happen **after successful build + test**, and optionally with human approval.

☑ 5. Use Stage-Level Timing and Visibility

In the GitLab UI, each stage has its own timeline, logs, and status:

- You can immediately see **what broke and where**
- Retrigger only the **failed stage**
- Cancel pipeline after failed tests (to save time)

☑ What We Learned

1. Mixing build, test, and deploy in one job **hurts clarity, control, and safety**
2. GitLab CI supports clean separation using stages: and artifacts:

3. Each job should be **single-purpose** and **modular**
4. Use rules and manual approvals to **gate deploys**
5. A well-structured pipeline is easier to debug, audit, and extend

☑ 7. Deploying to Production Without Approval Gates

⚠ The Problem

In many GitLab CI/CD setups, the deploy job runs automatically once the pipeline reaches the deploy stage — **even for critical environments like production**:

```
deploy_prod:
  stage: deploy
  script: ./deploy-prod.sh
```

This means:

- Any push to main or release can **immediately deploy to production**

- Bugs, failed tests, or misconfigurations may **slip through and go live**
- There's no **manual intervention** or approval process
- Teams accidentally **break production with a single commit**

This violates **Change Management** and **Deployment Control** practices — especially in regulated or high-availability environments.

🔬 Real-World Impact

- A junior developer pushed code and unintentionally triggered a full production deploy
- An incomplete feature merged into main and deployed during off-hours
- Incident recovery took hours because there was no rollback gate
- Organizations failed audits due to lack of deployment approval workflows

🛠 The Right Way to Handle This

☑ 1. Use when: manual for Production Deployments

Gate production jobs with manual approvals:

```
deploy_prod:
 stage: deploy
 script: ./deploy-prod.sh
 environment:
   name: production
   url: https://myapp.com
 when: manual
 only:
   - main
```

This adds a **“Play” button** in the GitLab pipeline UI, so deployment won’t proceed without human confirmation.

☑ 2. Restrict Deployment Jobs to Protected Branches or Tags

Ensure only approved branches can trigger production deployment:

```
deploy_prod:
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
      when: manual
  only:
    - tags
```

Then protect the branch/tag via **GitLab → Settings → Repository → Protected Branches**
Only specific users or maintainers can push or merge.

☑ 3. Use environments: for Better Visibility & Control

Define environments to track deployments in GitLab:

```
deploy_prod:
  environment:
    name: production
    url: https://app.example.com
```

Benefits:

- Track deployments by environment
- Rollback from GitLab UI
- Audit logs and change history per environment

☑ 4. Integrate with Slack or Teams for Approval Notifications

Use GitLab's webhook or integrations to notify teams when a production deployment is **waiting for approval**.

☑ 5. Use GitLab's Manual Approval + Merge Request Checks

GitLab Premium supports **MR approval rules** where specific people must approve changes before merge (and deploy).

☑ What We Learned

1. Auto-deploying to production without human approval is **risky and unsafe**
2. Use when: manual + rules: to require human review
3. Limit production deploys to **protected branches**
4. GitLab environments give visibility and rollback capabilities
5. Production pipelines should have **change gates**, **approvals**, and **alerting**

☑ 8. Missing before_script and after_script Standardization

⚠ The Problem

Many GitLab CI/CD pipelines copy the same setup or teardown commands repeatedly in every job. This leads to:

- **Redundant code**
- **Inconsistent behavior** across jobs
- **Hard-to-maintain pipelines**, especially when setup steps change (e.g., auth, env vars, tools)

For example, jobs without before_script repeat:

```
job1:
```

```
  script:
    - export ENV=prod
    - npm install
    - npm run build

job2:
  script:
    - export ENV=prod
    - npm install
    - npm test
```

Or worse, they don't run a proper cleanup — temporary files, dangling containers, and logs are left behind.

🔬 Real-World Impact

- Inconsistent environments across jobs (some install tools, others don't)
- Security tools (e.g., Trivy, Gitleaks) not run consistently
- Teams forget to apply updates to all jobs, causing version drift
- Developers waste time debugging jobs with missing dependencies or mismatched config

🛠 The Right Way to Handle This

☑ 1. Use Global before_script: and after_script:

Define shared scripts at the **top level** of .gitlab-ci.yml:

```
before_script:
  - echo "Setting up CI environment"
  - export ENV=prod
  - apt-get update && apt-get install -y curl
```

```
after_script:
  - echo "Cleaning up"
  - rm -rf temp/
  - docker system prune -f
```

Now all jobs **inherit** these steps unless overridden.

☑ 2. Override Locally When Needed

Jobs can override global before_script or after_script if they have special requirements:

```
test:
  before_script:
    - echo "Custom setup for tests"
    - npm ci
  script:
    - npm test
```

This **adds flexibility** while keeping core steps centralized.

☑ 3. Use .default-template Jobs for Common Behaviors

For even more structure, create reusable templates using YAML anchors or extends:

```
.default-job:
  before_script:
    - export ENV=staging
    - npm ci
  after_script:
    - echo "Job completed."

build:
```

```
  extends: .default-job
  script:
    - npm run build
```

This promotes DRY (Don’t Repeat Yourself) pipeline code.

☑ 4. Use after_script for Clean-Up and Diagnostics

Typical after_script: use cases:

- Cleanup temp files, containers, or cache
- Post-job reporting (e.g., log upload, Slack notification)
- Metrics collection (e.g., upload JUnit or test logs)

```
after_script:
  - curl -X POST -d "Build complete: $CI_JOB_NAME" $SLACK_WEBHOOK
```

⬤ Note: after_script runs **even if the job fails** — great for cleanup and alerts.

☑ 5. Enforce Common Steps Across Jobs

Examples:

- Ensure security tools like Trivy/Gitleaks are always present
- Standardize docker login steps
- Prepare directories like /tmp/build, etc.
- Set env vars like NODE_ENV, JAVA_HOME, PYTHONPATH

☑ What We Learned

1. Without standardized setup/teardown steps, pipelines become **messy and unreliable**
2. before_script ensures all jobs start with **clean, consistent environments**
3. after_script handles **post-processing**, cleanup, and notification
4. Use .default-job or extends: to eliminate repetition

5. Clean and predictable pipelines reduce maintenance overhead and debugging time

☑ 9. Ignoring Job and Pipeline Failures

⚠ The Problem

Many teams fall into the trap of treating CI/CD pipelines as "passive observers." Jobs fail, but no one investigates. This happens due to:

- Job failures being silently ignored in script blocks
- Overuse of allow_failure: true
- Lack of alerts or failure notifications
- No owner assigned to broken pipelines
- No policy for failing test cases (pipelines pass even when unit tests fail)

This leads to a dangerous DevOps culture of **"green-looking red pipelines"**, where the pipeline looks okay on the surface, but is functionally broken inside.

🔬 Real-World Impact

- A pipeline failed during artifact upload, but deployment still happened
- Broken test stage was marked allow_failure: true and never fixed
- Staging environment had missing assets because the build job failed silently
- Teams became numb to red pipelines — no one felt responsible to fix them
- A production deploy was based on a failed build, causing an outage

🛠 The Right Way to Handle This

☑ 1. Fail Fast and Loud — Don't Suppress Errors

Every job should **fail clearly** if anything goes wrong.

Avoid this:

```
script:
  - ./build.sh || true
```

Instead, do this:

```yaml
script:
  - ./build.sh
```

Let the job **fail naturally** and halt the pipeline.

☑ 2. Use allow_failure: true Only When Absolutely Needed

Legitimate use cases:

- Experimental or optional jobs
- Security scanners with non-blocking warnings
- Nightly jobs where failure doesn't impact the main workflow

Example:

```
sast_scan:
  script: trivy fs .
  allow_failure: true
```

But NEVER do this on core jobs:

```
build:
  allow_failure: true  # ❌ Don’t do this!
```

☑ 3. Assign Job Owners or Set Code Owners

Use GitLab’s CODEOWNERS file or team conventions to assign responsibility:

```
# CODEOWNERS
.gitlab-ci.yml @devops-team
/test/ @qa-team
```

If a job breaks, **someone gets notified** and is responsible to fix it.

☑ 4. Fail the Pipeline When Tests Fail

Avoid constructs that let tests fail silently:

```
script:
  - npm test || true  # ❌
```

Use:

```
script:
  - npm test  # ☑ fails on test error
```

If using test runners, output JUnit reports and make sure they’re evaluated.

☑ 5. Enable Notifications for Failures

Set up:

- GitLab email notifications
- Slack/Discord webhooks
- PagerDuty/Alertmanager integration for critical failures

Example:

```
notify_failure:
  stage: notify
  script:
    - curl -X POST -H "Content-Type: application/json" \
      -d '{"text": "Pipeline failed on $CI_COMMIT_REF_NAME"}' \
      $SLACK_WEBHOOK
  when: on_failure
```

☑ 6. Monitor Pipeline Health

Use GitLab's **Pipeline Analytics** to track:

- Success/failure trends
- Longest-running or most unstable jobs
- Broken pipelines over time

This helps detect **chronic failures** that are silently ignored.

☑ What We Learned

1. Every job must **fail clearly and stop the pipeline** when something goes wrong
2. Misusing allow_failure or suppressing errors hides critical problems
3. Assign job ownership so failures are addressed quickly
4. Never accept test failures — CI must enforce quality
5. Build a culture of **pipeline hygiene** to maintain long-term reliability

✅ 10. No Monitoring or Notifications on CI/CD Events

⚠ The Problem

Many teams set up complex pipelines but forget one critical piece — **monitoring and alerting**. When builds fail, deployments break, or test coverage drops, there's **no one watching**.

Common issues include:

- No Slack, email, or webhook notifications on pipeline status
- Teams not knowing when a critical stage fails
- No observability into CI/CD trends (e.g., pipeline success rate, test flakiness)
- Delayed reaction to incidents due to lack of visibility
- Developers unaware of broken builds for hours or days

Without feedback loops, CI/CD becomes a **black box** — issues compound silently until they explode.

🔬 Real-World Impact

- A staging deploy failed over the weekend, but no one noticed until Monday
- Test coverage dropped by 15%, but no alerts were configured
- A job failed on 7 consecutive MRs, but the team assumed everything was fine
- A production deployment failed silently due to a script typo — no one was notified
- Missed deadlines and SLAs due to untracked pipeline regressions

🛠 The Right Way to Handle This

☑ 1. Set Up Notifications for Pipeline Events

Use GitLab's built-in notification features:

- **Email Notifications**: Users can configure notifications under their GitLab profile → Preferences → Notifications
- **Slack Integration**:
  - Go to **Settings → Integrations**
  - Enable Slack notifications for push, merge, pipeline events
  - Or use custom webhooks in .gitlab-ci.yml:

```
notify_slack:
 stage: notify
 script:
  - curl -X POST -H 'Content-type: application/json' \
   --data '{"text":"🚨 Pipeline *$CI_PIPELINE_ID* failed on *$CI_COMMIT_BRANCH*"}' \
   $SLACK_WEBHOOK
 when: on_failure
```

✅ 2. Monitor Pipeline Health with GitLab Analytics

Use GitLab features (Premium/Ultimate tiers) or integrate with external tools like:

- Grafana dashboards via GitLab Exporter
- Prometheus metrics for runner usage and job durations
- GitLab CI/CD Reports for pipeline duration, test trends, code quality

Track:

- Pipeline success rate
- Median build time
- Longest-running jobs
- Jobs that fail most frequently

✅ 3. Use when: + Conditional Alerts on Failures

You can run notification or rollback jobs **only on failure**:

```
notify_failure:
  stage: notify
  script:
    - ./notify.sh "Build failed for $CI_COMMIT_REF_NAME"
  when: on_failure
```

And on success:

```
notify_success:
  stage: notify
  script:
    - ./notify.sh "✅ All jobs passed!"
  when: on_success
```

✅ 4. Create Fallback or Auto-Rollback Strategies

Set up conditional jobs that react to failures:

```yaml
auto_rollback:
  stage: deploy
  script:
    - ./rollback-to-previous.sh
  when: on_failure
```

This allows fast recovery when something breaks.

☑ 5. Make Alerts Actionable and Contextual

Good alerts should answer:

- What failed?
- When did it happen?
- Who committed the change?
- What was the impact?
- What should be done next?

Include metadata in Slack/webhook messages:

```json
{
  "pipeline_id": "12345",
  "branch": "main",
  "status": "failed",
  "commit": "a1b2c3d",
  "author": "dev@example.com"
}
```

☑ What We Learned

1. A pipeline without monitoring is a **silent failure waiting to happen**

2. Use GitLab's built-in notification system + Slack/Email integrations
3. Add fallback and notification stages to handle pipeline success/failure
4. Monitor CI/CD health using GitLab analytics or external observability tools
5. The best CI/CD systems **talk back** — they tell you when something needs attention

🧩 Conclusion: From Fragile Pipelines to Reliable Automation

GitLab CI/CD is a powerful tool that can accelerate software delivery when used wisely — but like any automation system, it's only as strong as the discipline and structure behind it. The mistakes we've covered — from misconfigured triggers and dependency caching to unsafe deploys and invisible failures — are not just technical oversights. They are process gaps that slowly erode trust in the pipeline and disrupt developer velocity.

The good news? Each of these pitfalls is fixable.

By:

- Breaking down pipelines into logical stages,
- Using semantic versioning over latest,
- Guarding production with manual approvals,
- And monitoring failures with proper notifications,

...you transform CI/CD from a fragile chain of scripts into a **reliable backbone of your DevOps culture**.

Remember, pipelines are not “set it and forget it.” They require **iteration, observation, and ownership**. Treat them as living systems — regularly audited, tested, and improved.

When teams take the time to get CI/CD right, they unlock more than just faster deployments — they build confidence, stability, and resilience into every release.