TechnologyMatch.com

CI/CD Pipeline Health Checklist

Done (Green) Not Done, but Not Critical (Yellow) Not Done (Red)

Pipeline Health Checklist

1. Is your build environment fully isolated (e.g., clean containers/VMs per build) to prevent cross-build contamination and supply chain attacks?
Why it matters: Shared runners or persistent build agents can leak artifacts, secrets, or environment variables across builds.
Green Yellow Red

2. Are all third-party dependencies (including submodules, libraries, and build tools) pinned to specific, trusted versions and scanned for vulnerabilities on every pipeline run?
Why it matters: Unpinned dependencies and outdated modules are a top cause of supply chain exploits and unpredictable builds.
Green Yellow Red

3. Are container images and deployment artifacts signed (e.g., with Cosign, Notary, or Sigstore) and verified before promotion to production?
Why it matters: Artifact provenance is critical for preventing tampering in multi-stage, multi-team pipelines.
Green Yellow Red

4. Are all secrets and credentials injected into the pipeline at runtime from a secure secrets manager (not stored in CI/CD tool variables, .env files, or as plaintext in build logs)?
Why it matters: Secrets leaked in logs, configs, or pipeline metadata are a top cause of breaches.
Green Yellow Red

5. Are you using ephemeral, short-lived credentials and tokens for all pipeline actions (including cloud deploys, artifact pushes, and test runs)?
Why it matters: Long-lived or static credentials are a major risk if the CI worker or pipeline is ever compromised.
Green Yellow Red

6. Do you enforce least privilege for pipeline service accounts (e.g., separate deploy, test, and read-only accounts; no broad admin rights)?
Why it matters: Overprivileged CI/CD service accounts can be exploited to pivot throughout your infrastructure.
Green Yellow Red

7. Are all pipeline stages event-driven and stateless, supporting parallelization and horizontal scaling (e.g., fan-out for test jobs, parallel environment spin-up)?
Why it matters: Non-parallel or stateful stages become bottlenecks and single points of failure at scale.
Green Yellow Red

8. Are environment configurations, secrets, and infrastructure definitions managed as code and versioned with strict change controls (e.g., Terraform, Ansible, Helm, Kustomize)?
Why it matters: Manual configuration drift is a top source of “works in dev, fails in prod” issues.
Green Yellow Red

9. Do all pipeline stages emit structured, queryable logs and metrics (e.g., JSON logs, OpenTelemetry traces, Prometheus metrics) to a central observability platform?
Why it matters: Unstructured or siloed logs make debugging, audit, and compliance nearly impossible.
Green Yellow Red

10. Do you run security scanning (SAST, DAST, container scanning, IaC scanning) as first-class pipeline stages, and enforce blocking policies on high/critical issues?
Why it matters: Security gates that are non-blocking or “report only” quickly become ignored.
Green Yellow Red

11. Are all test failures, deploy errors, and pipeline anomalies automatically ticketed or alerted to responsible engineers (with context) and tracked to resolution?
Why it matters: Silent failures and unactioned errors undermine pipeline trust and production safety.
Green Yellow Red

12. Are blue/green or canary deployments supported natively, with automated health checks and instant rollback/roll-forward on failure?
Why it matters: All-or-nothing deploys risk mass outages; safe rollout patterns are essential for reliability.
Green Yellow Red

13. Is every pipeline and deploy action fully auditable (who, what, when, where) and protected by strong authentication (SSO, MFA) and audit logging?
Why it matters: Weak auth and missing audit trails are a compliance and security disaster.
Green Yellow Red

14. Can you rebuild any environment (including production) from code and pipeline artifacts alone, with no manual steps required?
Why it matters: True reproducibility is the gold standard for disaster recovery and compliance.
Green Yellow Red

15. Is pipeline performance (queue time, stage duration, flakiness rates) monitored continuously, with regular retrospective reviews and performance tuning?
Why it matters: Slow, flaky, or unreliable pipelines are ignored by engineers and become bottlenecks.
Green Yellow Red

Top 7 Deep Pipeline Mistakes & Quick Fixes

1. Untrusted or Outdated Third-Party Actions/Plugins
   Fix: Audit pipeline plugins/actions monthly. Use only maintained and signed actions. Pin all versions.

2. Secrets Leaked via Logs or Artifacts
   Fix: Mask secrets in logs; use tools like **git-secrets** and **truffleHog**; regularly rotate all tokens/keys.

3. Non-Deterministic Builds (e.g., “latest” tags, mutable images)
   Fix: Use SHA digests or immutable tags for all images and artifacts; fail pipelines on version drift.

4. Pipeline Drift Between Environments
   Fix: Enforce single source of truth for IaC and config; use GitOps patterns for all environments.

5. CI/CD as a Single Point of Failure
   Fix: Run redundant runners, backup CI/CD config/code, and test disaster recovery for your pipeline itself.

6. No Security Review of Pipeline Definitions
   Fix: Review pipeline YAML/scripts as part of code review; use tools like **Checkov** or **TFSec** for IaC.

7. Overly Permissive Network/Cloud Access
   Fix: Lock down runner/workload network egress; use private subnets and VPC endpoints for artifact and secrets access.