From n to continuous: how to automate your load tests

1 DEFINE TEST CONFIGURATION AS CODE

Before automating execution, make sure your tests are fully reproducible and version-controlled.

- [ ] **Store simulation files in your source repository alongside application code**
- [ ] **Define all test configuration (environment, load profile, packages) in code or HOCON**
- [ ] **Use a build tool (Maven, Gradle, npm, sbt) so tests can be built and run without manual setup**
- [ ] **Pin dependency versions to avoid unexpected behavior across environments**
- [ ] **Validate that any team member can clone the repo and run a simulation with a single command**

Done when: Your entire test setup is reproducible from source control with no manual steps.

2 INTEGRATE LOAD TESTS INTO YOUR CI/CD PIPELINE

Triggering tests automatically on code changes catches performance regressions before they reach production.

- [ ] **Add a load test step to your pipeline (GitHub Actions, GitLab CI, Jenkins, Azure DevOps, etc.)**
- [ ] **Start with a smoke-level simulation on every pull request — fast, low cost, validates the script runs**
- [ ] **Run average-load simulations on merge to main or on deploy to a pre-production environment**
- [ ] **Keep CI load test duration under 10–15 minutes to avoid becoming a bottleneck in the release cycle**
- [ ] **Set pass/fail thresholds derived from your SLOs (e.g., if your SLO is p95 < 500ms and error rate < 1%, your CI gate should enforce those same boundaries)**

Done when: Every significant code change triggers at least a lightweight automated performance check.

3 AUTOMATE INFRASTRUCTURE PROVISIONING

Manually setting up load generators before each run doesn't scale. Infrastructure should be provisioned on demand, and that includes the data state those load generators depend on.

- [ ] **Use Infrastructure as Code to automatically setup and tear down load generators after each run**
- [ ] **Store IaC configuration in version control alongside your simulation code**
- [ ] **Identify all data dependencies for each simulation (user accounts, records, API tokens, seed datasets)**
- [ ] **Distinguish between data that should be stable and data that must be reset**
- [ ] **Treat an environment with stale or unknown data state the same way you'd treat a flaky test: do not run against it until state is verified**

Done when: Load generators and their data dependencies are provisioned, verified, and torn down automatically. No manual intervention, no unknown state between runs.

From n to continuous: how to automate your load tests

4 TUNE AND VALIDATE YOUR QUALITY GATES

Setting a threshold is easy. Making it trustworthy takes iteration; a poorly calibrated gate either blocks valid releases or gives false confidence.

- [ ] **Run quality gates in warning mode before enforcing them as release blockers**
- [ ] **Define SLOs in Gatling: response time percentile, error ratio, set your target, and compliance threshold**
- [ ] **Use SLO results as compliance gauges and as your ship/hold signal, not averages**
- [ ] **Review and adjust thresholds whenever the system changes significantly**
- [ ] **Document the rationale behind each SLO so maintainers understand what they're protecting against**

Done when: Quality gates are defined as SLOs with explicit compliance targets, evaluated automatically after every run, and trusted as a reliable ship/hold signal.

5 SCHEDULE TESTS ACROSS ENVIRONMENTS

CI/CD covers code-triggered runs and also extends to production, but some test types need to run on a fixed schedule regardless of deployments.

- [ ] **Map each test type to the environment and frequency that suits its purpose**
- [ ] **Use a dedicated scheduler (cron, cloud-native, or platform-native), don't rely only on CI/CD triggers**
- [ ] **Define an execution window for every scheduled test and treat a missing result as a failed one**
- [ ] **Document your production testing policy; tag synthetic requests to filter from real traffic**
- [ ] **Avoid scheduling heavy tests during peak production traffic hours**

Done when: Tests run consistently on schedule across all relevant environments, silence is treated as a failure signal, and your team has an explicit documented policy on production testing.

6 TRACK PERFORMANCE TRENDS OVER TIME

A single result is a snapshot. Regressions only become visible when you can compare runs across builds, branches, and time, and that comparison needs to be built into your workflow, not assembled manually after the fact.

- [ ] **Use Gatling's Run Trends to get a high-level view of your last 10 runs for any simulation:**
- [ ] **Use Run Comparison to investigate: select up to 5 runs simultaneously and compare across 11 metrics**
- [ ] **Set a reference run to anchor your comparison, all the other runs show percentage deviation from it**
- [ ] **Tag runs with environment, branch, and build ID so comparisons are always equivalent**
- [ ] **Establish a named baseline run for each simulation as your regression reference point for CI/CD gates**

Done when: Every simulation has a visible trend in Gatling and regressions are detectable from run comparison without manual cross-referencing.

From n to continuous: how to automate your load tests

7 DETECT AND ADDRESS TEST INCONSISTENCY

Automated tests that produce unreliable results are worse than no automation because they erode trust and cause alert fatigue.

- [ ] **Flag any test where two back-to-back runs with identical configuration produce different results**
- [ ] **Never promote an inconsistent test to a CI gate or a scheduled alert; fix the root cause first**
- [ ] **Isolate test environments as much as possible: shared staging environments will introduce noise**
- [ ] **Use test data that resets between runs to avoid state accumulation affecting results**
- [ ] **Investigate variance sources: load generator saturation, environment isolation shared infrastructure contention, or data state**

Done when: Your automated tests produce consistent, reproducible results that can be trusted as a signal.

8 TREAT COVERAGE GAPS AS ENGINEERING DEBT

Uncovered scenarios are risk you've accepted without deciding to. Gaps in automation coverage accumulate quietly, the same way technical debt does.

- [ ] **After every production incident, check whether an automated test would have caught the regression**
- [ ] **Maintain a record of tests suspended from automation and the reason they're not used anymore**
- [ ] **After each production incident, check whether an automated test would have caught it**
- [ ] **Set a coverage review cadence tied to your release cycle, not a calendar quarter.**
- [ ] **All untested critical paths need an owner, a ticket, and a target date**

Done when: Every critical simulation has a defined trigger and a justified status, and gaps are tracked as actionable engineering work, not accepted as background noise.

9 INTEGRATE WITH OBSERVABILITY

Measuring response time tells you that something is slow. Correlating load test runs with system telemetry tells you why. Without this integration, performance investigations require manual cross-referencing of test output against separately collected metrics

- [ ] **Annotate your observability dashboards with load test run markers so runs are visible as markers**
- [ ] **Store links between test result records and the observability window captured during that run**
- [ ] **Correlate before concluding: Treat slow p99 latency without an infra signal as incomplete evidence**
- [ ] **When a quality gate fails, require at least one infrastructure or application metric in the investigation**
- [ ] **During runs, collect at minimum: CPU and memory on application nodes, thread and connection pool saturation, GC pause times where applicable, and downstream dependency latency**

Done when: Every load test run produces both a Gatling report and a correlated window of system telemetry, and the two are navigable together.