From 1 to n: How to scale your load testing practice

1 SCALE THE NUMBER OF VIRTUAL USERS

Your first load test established a baseline at an initial target load. Now push beyond it to reach the traffic volumes your system will actually face in production.

- ☐ **Increase virtual user count beyond your initial baseline, in increments**
- ☐ **Move from a single load generator to distributed load generation when you approaches CPU limits**
- ☐ **Monitor load generator CPU, memory, and network and keep injector CPU load below 80%**
- ☐ **Calculate source IP requirements for high concurrency**
- ☐ **Set automated stop criteria to abort a run before injector saturation corrupts results**

Done when: You can simulate high traffic volumes without bottlenecks coming from your load generators.

2 TEST PEAK AND STRESS CONDITIONS

Your baseline test confirmed behavior at expected load. Now go further: stress tests and spike tests answer a different question, not "does it work?" but "where does it break, and how?"

- ☐ **Run stress tests beyond expected production traffic to find the breaking point**
- ☐ **Use a progressive injection profile to increase load in controlled steps**
- ☐ **Observe how the system degrades under pressure: which endpoints fail first, which errors appear**
- ☐ **Validate recovery behavior after load decreases, confirm the system returns to baseline**
- ☐ **Document the breaking point and degradation pattern as a reference for capacity planning**

Done when: You understand where the system breaks, how it degrades, and how it recovers, not just that it passes under expected load.

3 VALIDATE LOAD DISTRIBUTION REALISM

As you scale traffic, ensure it reflects real-world usage patterns.

- ☐ **Distribute load across multiple regions if your users come from different geographies**
- ☐ **Base injection profiles on real production traffic sourced from APM data or access logs**
- ☐ **Adjust request distribution across endpoints to match observed production ratios**
- ☐ **Add realistic think times between requests to mimic real user load**
- ☐ **Add a 2–5 minute warmup phase at the start of each test and exclude its metrics from analysis**

Done when: Your load profile accurately reflects how users interact with the system in production.

From 1 to n:
How to scale your load testing practice

4 EXPAND TO BUSINESS-CRITICAL BACKEND FLOWS

After validating individual endpoints, move to modeling complete backend interactions that reflect how users interact with your system.

- ☐ **Inventory all services, APIs, and dependencies in your architecture**
- ☐ **Prioritize systems based on criticality (e.g., revenue, user-facing, core infrastructure)**
- ☐ **Define a coverage target (e.g., all critical services have load tests)**
- ☐ **Test interactions between services, not just isolated components**
- ☐ **Ensure new tests reuse existing infrastructure and avoid duplication**

Done when: Multiple key user journeys are covered in your load testing suite.

5 EXPAND TEST COVERAGE ACROSS YOUR SYSTEM ARCHITECTURE

Move beyond single services to validate performance across all critical components and their interactions.

- ☐ **Inventory all services, APIs, and applications in your architecture and classify by business criticality**
- ☐ **Set a coverage target: e.g., all revenue-critical services have a load test within the next two quarters**
- ☐ **Track coverage gaps as engineering backlog items and assign an owner to each uncovered service**
- ☐ **Test interactions between services, not only individual endpoints in isolation**
- ☐ **Validate that adding a new service test does not require duplicating infrastructure or setup logic**

Done when: Your tests reflect the full system architecture, not just isolated endpoints.

6 SUPPORT ALL PRODUCTION PROTOCOLS

As your system evolves, ensure your load testing covers all communication layers used in production.

- ☐ **Identify all protocols in use: HTTP/REST, WebSocket, gRPC, messaging (Kafka, MQTT, JMS), etc.**
- ☐ **Add load test coverage for each protocol used in production**
- ☐ **Validate performance for each protocol independently before testing cross-protocol flows**
- ☐ **Ensure metric definitions, SLOs and SLA thresholds are consistent across protocols**
- ☐ **Adapt scenarios to protocol-specific behaviors**

Done when: All critical protocols used in your system are covered by load tests.

From 1 to n: how to scale your load testing practice

7 STRUCTURE TESTS BY PROJECTS AND CAMPAIGNS

As the number of simulations grows, organization becomes critical for discoverability, maintenance, and coordinated execution.

- ☐ **Group simulations by application, team, or product domain**
- ☐ **Define campaigns (named collections of simulations that run together)**
- ☐ **Standardize naming conventions: include team, service, environment, and test type**
- ☐ **Define which campaigns run in CI, which run on a schedule, and which are triggered before a release**
- ☐ **Assign a campaign owner responsible for keeping the simulation set current and reviewing results**

Done when: Tests are easy to find, run, and maintain across teams.

8 SCALE THE NUMBER OF USERS ON THE PLATFORM

As adoption increases, enable more teams to participate in load testing.

- ☐ **Onboard developers, QA, DevOps, and SRE teams to the load testing platform**
- ☐ **Create users through the platform UI or via API for automated provisioning**
- ☐ **Ensure each team can access relevant tests and results without seeing unrelated ones**
- ☐ **Identify performance champions in each team who can support onboarding and answer questions**
- ☐ **Create a starter template or internal guide so any developer can write and run a test without help**

Done when: Multiple teams actively use and contribute to load testing.

9 MANAGE ACCESS, ROLES, AND COLLABORATION

Ensure collaboration remains efficient and secure as usage scales.

- ☐ **Define roles and permissions: admin, contributor, viewer that match to each team's responsibilities**
- ☐ **Assign per-team quotas for load generator capacity and compute credits to prevent conflicts**
- ☐ **Automate user and permission management via API to eliminate manual provisioning overhead**
- ☐ **Share run results across teams using report links or direct integrations (Slack, Teams, Jira)**
- ☐ **Connect load test results to your observability stack for cross-team visibility**

Done when: Load tests run automatically as part of the delivery pipeline and help detect regressions before deployment.