Tricentis

A practical guide to continuous performance testing

Tricentis

Table of contents

INTRODUCTION

This guide presents pragmatic considerations and experienced-based best practices for performance engineers looking to automate performance testing in CI/CD pipelines.

The #1 challenge we hear from those responsible for performance testing is the difficulty in keeping up with the pace and scale of development teams so that you can deliver performance feedback early and often. They're caught in a crossfire between performance being more imperative than ever — slow or buggy systems and applications have a material impact on customers/users, brand reputation, and bottom-line revenue — and releases coming faster than ever. The clock speed of testing is mismatched to the speed of development. With today's automated builds taking minutes — not hours (or days) — integrating continuous performance testing into automated pipelines is the only answer.

That means breaking down performance verification into smaller meaningful feedback loops that fit into shorter cycles. "Right-fitting" performance tests into automated pipelines provides early signals that are actionable (i.e., feedback loops). As product teams develop new features and make changes to application code, they know right away what kind of impact each change has on application performance.

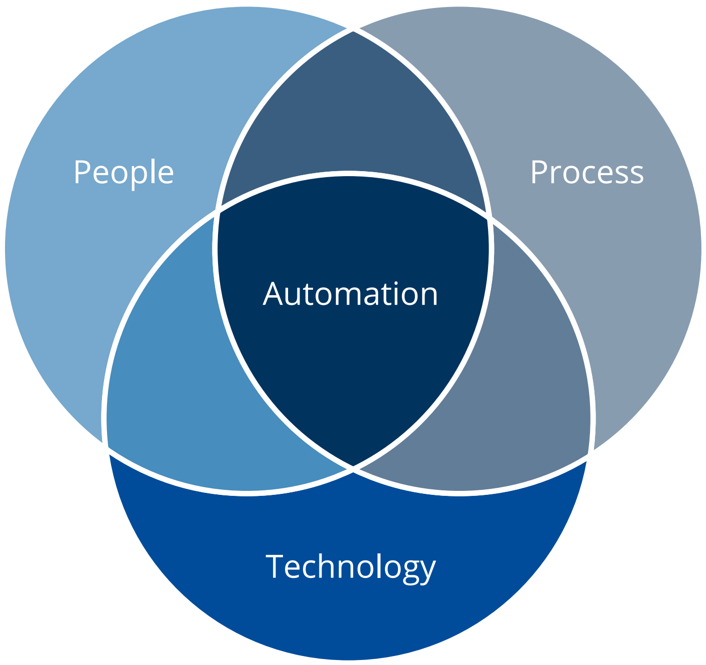
This avoids the pain and delay of discovering a major performance issue at the 11th hour, when it's more costly and difficult to isolate and fix the problem. Nobody wants to be the person who holds up a release at the last minute because they found a major performance issue that wasn't resolved earlier in the cycle.

We've seen that virtually every organization is looking to enable testing "early and often." However, many find it difficult to get from theory to practice. This white paper offers guidance on laying the foundation for a successful transition to an automated testing approach and discusses practical solutions to overcoming common automation "blockers."

GETTING STARTED

Assess your gaps

Before you can begin implementing continuous performance testing into automated pipelines, you must take stock of your readiness for automation. It's instinctive to say that the goal of automated testing is to go faster or accelerate delivery, but that applies only when it works reliably and produces actionable outcomes. Getting to "reliable" and "actionable" embraces people and process as well as technology. Automation forces you to articulate your goals, requirements, and activities in a way that machines can execute on your behalf. It shines a light on gaps in your processes, technologies, and skills — not in a negative sense of creating a laundry list of deficiencies, but rather to identify potential blockers to continuous performance testing. Subsequent sections of this white paper discuss processes, strategies, and approaches for overcoming common obstacles.



One way of looking at things and assessing gaps that may impede automation is to look where aspects of performance testing is easy (or not). Within the context of your current tech stack, processes, and staffing, is it easy to script tests, automate processes and infrastructure requirements, and consume the results of testing? The harder things are, the more you will remain dependent on a handful of experts to do them.

In evaluating your automation readiness, think about where you want to be in the future. Many organizations that are currently executing performance testing "as a service" are on the road to changing to a "self-service" model where product teams are able to seamlessly and autonomously integrate performance testing into their development sprints. This scales performance testing beyond a few scarce subject matter experts (SMEs) to an organization-wide approach. These SMEs then move beyond just running tests and analyzing results; they act in a consultative manner to apply their expertise to PI planning, providing product teams with ways to do more things themselves (easily) while implementing safety guardrails and best practices so that teams iteratively build up their own performance competency over time.

Automation allows organizations to formalize performance testing practices into work where machines — not people — do the laborious, time-consuming heavy lifting.

Prioritize, then systematize

Automated continuous performance testing isn't just about running a higher volume of test cases. While a greater number of tests can certainly improve coverage and mitigate more risks, the key is quality and timing rather than quantity. The whole point of continuous performance testing is to provide developers with early signals that are actionable (feedback loops) so that they know — immediately and precisely — what kind of impact their code changes have on application performance. If you run 1,000 tests continuously without uncovering defects in time to fix them, what's the point?

Prioritization is often the missing ingredient to the recipe for testing "early and often." Just as you can't test everything, you can't automate feedback loops for everything. That only slows things down. When determining which performance tests to "right-fit" into automated pipelines, find high-value tests that focus on business risk and the main paths that customers/users actually follow.

It's much easier to make a case for performance testing systems, applications, and components in the critical path — those that everyone agrees are important and have bottom-line impact — than services that are far removed from what others think is critical. Therefore, figuring out which systems and components need continuous feedback isn't on the shoulders of one individual; it calls for conversations among engineers, developers, and business stakeholders.

Prioritization methodologies

One method for prioritizing where performance feedback loops should be automated is the Utilization-Saturation-Errors (USE) model. As its name suggests, the model is based on three metric types and can be adapted to the context of performance testing:

- **Utilization:** Which core APIs or microservices are at the nexus of many other business processes, thereby ballooning their usage as a shared enterprise architecture component?
- **Saturation:** Which services have experienced "overflow" — the need for additional VMs/costs to horizontally scale — or are constantly causing SEV-n incidents that require "all hands on deck" to get back into reasonable operational status?
- **Errors:** From a perspective of "process error," how many times has a specific service update or patch been held up by long-cycle performance reviews? Which systems do you need fast time-to-deploy or where slow feedback cycles cause the product teams to slip their delivery deadlines (immediate or planned)?

Services that usually rank high on one or more of these vectors include user/customer authentication, cart checkout, and claims processing.

Other ways to evaluate system performance — such as rate-error-duration (RED) signals and business metrics from analytics and monitoring platforms — ensure that critical-path and revenue-generating user experiences are working as expected.

More about system performance analysis

- System Performance: Enterprise and the Cloud by Brendan Gregg
- Utilization-Saturation-Errors (USE)
- Rate-Error-Duration (RED)

LAY THE FOUNDATION FOR SUCCESS

Before embarking on continuous performance testing, it helps to have a few things fleshed out ahead of time. These are common roadblocks to automating performance tests, and it's the first two that are often overlooked but are crucial to delivering automated feedback loops. For automated performance tests to produce meaningful, actionable outcomes, people need to communicate about goals and outcomes.

- **Non-functional requirements/criteria.** There should be some kind of performance criteria intake process. It could be something as simple as a form or questionnaire about the systems and timeframes that performance testing should target.
- **Clear and communicated goals/outcomes.** Precise system performance and reliability expectations should be described as service level agreements (SLAs), objectives (SLOs), and indicators (SLIs). These monitoring metrics should be aligned across and mutually agreed-upon by production, development, and operations teams as well as business stakeholders.
- **Consistently available system/environment to test.** You can't test an application or service that you can't contact, so often this means having a "performance testing target environment" be part of the project plan.
- **Results analysis and response action plan that all teams — not just the performance engineers — agree to uphold.** If you're not fixing issues in a timely manner, why spend the energy and effort to build timely feedback loops?

START WITH THE END IN MIND

To automate performance tests that produce meaningful, actionable outcomes, the criteria for those outcomes must be communicated at the outset. If your "early and often" tests don't sufficiently warn you when performance is outside an acceptable range, they're not really telling you what you need to know when you need to know it. Discussing SLAs, SLOs, and SLIs up front is crucial to providing actionable feedback loops and realizing automated go/no-go decisions. Omitting performance criteria from the planning process is a common but fundamentally problematic occurrence. Often the people responsible for drafting SLAs are not performance experts and would benefit from such experts' insight into what and how to measure performance. Performance experts' having a "seat at the planning table" goes a long way to ensuring effective, efficient feedback loops.

Something as simple as digital intake forms that are pre-filled for product teams encourage these discussions to produce baseline automation artifacts like SLA definitions and API test details.

Distinguish between SLA, SLO, and SLI

When thinking about defining how your software or services need to work, you probably are talking in terms of a **Service Level Agreement (SLA)** that describes the commitment between teams about the expectations of a particular service, laying out measurable metrics (like uptime or responsiveness) and the responsibilities of each team. SLAs are essential, but they tell only part of the story. They are typically written by people who are not actually "in the trenches" and can be difficult to measure. More specificity is needed.

A **Service Level Objective (SLO)** is an agreement within an SLA about a standard, well-understood, mutually agreed-upon set of metrics. If the SLA is a commitment, SLOs are the individual promises. These are the goals that the different teams need to hit so that everyone is comparing apples to apples.

An **SLI (service level indicator)** measures compliance with an SLO (service level objective). SLIs provide the details regarding how operational performance is measured in terms of the SLA. The more exact, the better.

An SLA based on a well-defined SLO, then measured against a set of metrics that are the result of a detailed SLI, benefit general operations as well as the testing process. Tests that meet the SLA to the letter according to clearly defined parameters described in the SLI provide greater accuracy and more reliable analysis.

PICK THE RIGHT TARGETS TO AUTOMATE

Like any journey, the road to automated performance testing can be steep if you don't start in the right place. You need to pick the right target. You probably have dozens of teams that eventually need timely performance feedback loops in their delivery process, but it's best to start small with "easier" low-hanging fruit — specifically, APIs.

This approach provides threefold benefits. First, if it's done correctly, you'll realize "quick wins" demonstrating that baking performance tests into the development process early provides value without incurring delays. That makes it easier to get everyone on board; success breeds adoption. Second, starting small allows teams of all skill sets and levels of expertise to develop confidence and build automation competence, which can then be extended to other, more complex types of testing. Finally, breaking down the performance verification process into smaller feedback loops that match the faster clock speed and shorter work cycles of development provides teams with performance feedback earlier and more often so that by the time you test end to end, it's not a Dumpster fire. If an API doesn't perform properly, then there's no way end-to-end testing of all the integrated APIs will meet expectations.

Project / Team	Tests created?	Deploy-ready (able to QA)?	Relationship and value to team	Using SCM and pipelines
Mobile APIs	In Postman	Yes, in UAT	Very positive	Yes
Customer billing APIs	In Postman & testing tool + SLAs	Yes, in UAT; all shared services mocked out	Some work together in prior years, needs SLA revisions	Yes
Claims batch processing APIs	SoapUI and some testing tool	Yes, in ESB shared service	Unknown, not discussed w/ PO	Yes
Mobile app	Custom Espresso and XCTest scripts	Partial, relies on some shared production services	Initial intros, no requirements discovery yet	No, and app sandbox publishing process is slow

Sample Considerations Matrix for Picking the Right Early API Performance Testing Targets

▨ Easy ▨ More effort ▨ Difficult

APIs aren't the only places where performance feedback matters, but they're ubiquitously dispersed along the user experience and up and down distributed system call chains. The most consistent and predictable performance issues show up in API calls. Large payloads, hard-to-cache request patterns, and latency due to variances in payloads are issues that can be overcome early by writing a few low-complexity API load tests within development cycles.

Most organizations already have a huge backlog of APIs and microservices that aren't yet covered by performance testing. You probably already know which systems have experienced unexpected downtime or have identified them as high-risk. They'll have APIs for which you lack visibility into performance, scalability, or reliability. You almost certainly have a cross-section of APIs in your portfolio that are easy targets yet also on a critical business path in terms of revenue or risk. These services are the ones you should focus on as initial targets for building out continuous performance feedback loops.

Pick the right teams

Part of picking the right APIs to target is picking the right teams to work with. Teams that already have something deployed and ready to test — either in lower environments or in production — are a great place to begin. Consider starting with teams that performance engineers already have a relationship with. This makes obtaining architecture diagrams, getting access to monitoring tools, and initiating a proper performance criteria intake process early in sprints more straightforward.

Why APIs are easier

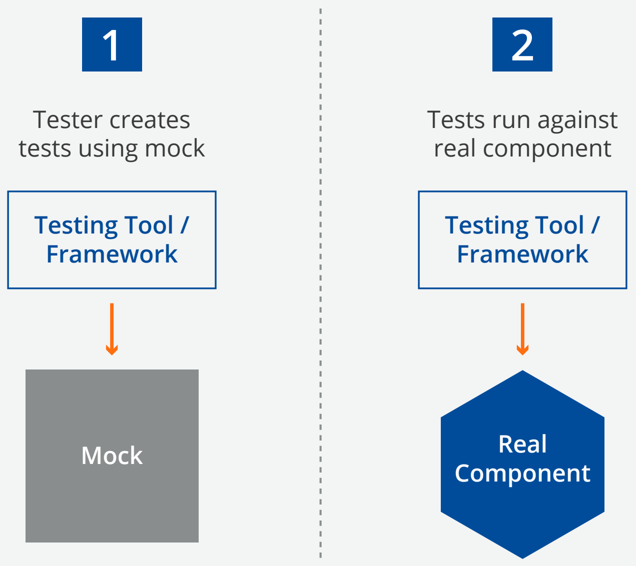
First and foremost, APIs have a much smaller "surface area" to test in terms of complexity and SLAs. Often API teams already have a manifest of the REST API as described by an API specification, such as OpenAPI, Swagger, or WSDL documents. Additionally, test data is often more straightforward to inject since the payloads are often self-descriptive (i.e., field names and data match formats in examples). Finally, organizations with APIs often have functional test assets, such as in Postman or REST Assured test suites, which provide a reference point for constructing performance test scripts.

Dealing with API endpoints and payloads is often far less complicated than dealing with complete traffic captures of end-to-end web applications, which often include dynamic scripts, static content, front-end API calls only, and other client-side tokenization semantics. API endpoints described in specification docs make scripting and playing tests back a far simpler proposition than ever before, rendering them as "easy" targets for early testing. Service descriptors also make it far easier to mock out APIs than entire web servers for end-to-end app tests.

How early is early enough to performance test APIs?

How can you test something that hasn't been built yet? That's often the case with non- functional API testing — you have to wait until it's in staging before you can script and run tests. But the unfortunate reality is that performance engineers no longer have the luxury to waterfall the process or defer questions of testability and proper requirements gathering to "later."

One common practice when left-shifting feedback loops is to "mock out" dependencies. In the case of APIs, you may want to write your tests before the actual API is deployed. Many organizations use service virtualization for fault isolation (removing specific third parties from integration testing) but fail to realize how useful API mocking is for the process of creating test suites before the actual APIs are deployed. This practice accelerates the process of including performance feedback loops simply by having automatable test assets ready to go before the actual automated testing process needs them. In fact, some performance teams have even created their own open-source tools, such as Mockiato, to make this a part of their daily Agile testing practices.



"It's rarely 'too early' to think about testing or expect performance requirements to be in order before moving on in development cycles. But you have to be 'in the room' at the early PI and architecture meetings, and don't expect that others are going to capture and document what is needed to know to test earlier."

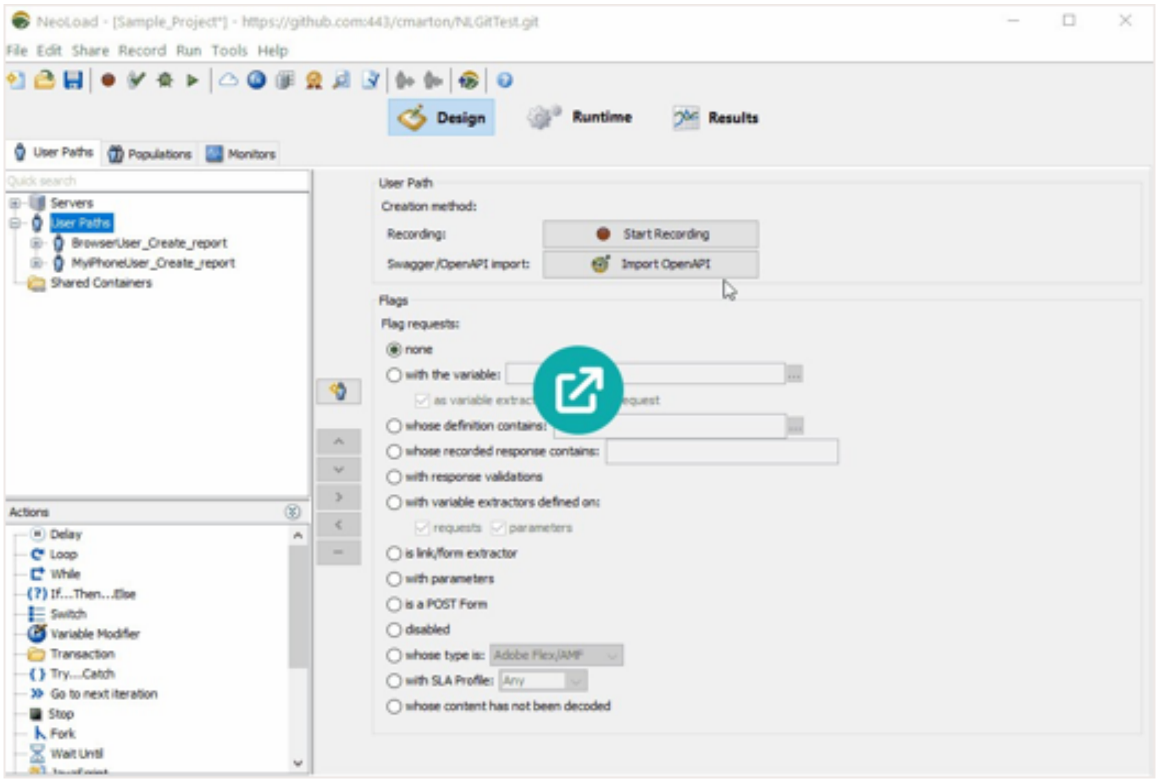
Paul Bruce, Director of Customer Engineering, Tricentis NeoLoad

MAKE SCRIPTING EASY FOR MULTIPLE TEAMS

It doesn't help to pick "easy" targets to test if designing, maintaining, and running your tests isn't also "easy." And that starts with creating test scripts. Scripting has to be easy and fast to enable the high degree of automation needed to make continuous performance testing a reality.

As part of your people/process/technology assessment, you will have identified what resources you have available — specifically, the technical experience and expertise of the people you expect to do the testing. Consider not just where you are today but where you want to be in the future. For example, you may have tagged a small group of performance experts to do API testing now, but down the line you'd like to move toward developers autonomously running performance tests themselves as part of their go/no-go decisions.

These considerations dictate how you ensure "easy scripting" and, to a large extent, which performance testing tools you adopt. Look for a tool that the people doing the testing will actually use. Just about every script-based performance testing solution (commercial or open source) claims that you don't need any scripting experience to create tests, but the reality is that most of them do in fact require specialized expertise. This essentially asks people who are already too busy to become performance experts overnight, or expects them to devote time they don't have to learn some new tool. That's a leading reason why legacy tools have never gained traction with DevOps teams.



The method for scripting tests should adapt to the way testers work, not the other way around. If your testers are more comfortable writing scripts in a GUI, the tool should accommodate that. But many developers prefer to express tests as code that can be integrated into automated CI pipelines, so the tool should allow that approach as well. To realize continuous performance testing, you should avoid tools that make you choose one way or the other. Find a single solution that enables both.

Also be aware that with most script-based testing tools, you'll probably spend an inordinate amount of time maintaining scripts. That might be okay if you're testing only a single API, but as the volume of APIs to test increases, so do the number of scripts to maintain. Beware of testing tools that break scripts every time code changes — instead, look for tools that can maintain scripts automatically, updating only the part of the test that's changed.

CO-LOCATE APP AND TEST AUTOMATION CODE

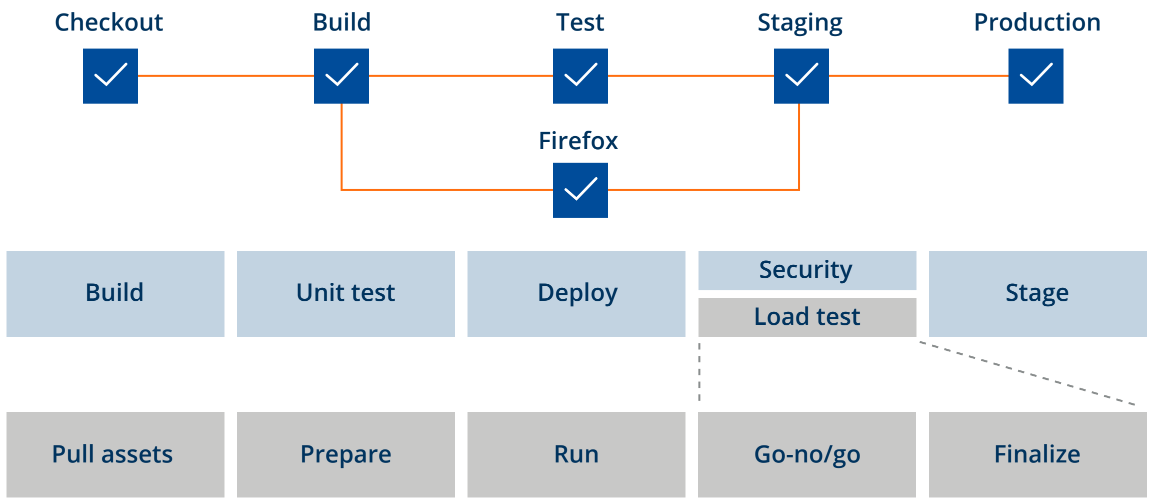
If you don't already, it's a good idea to keep your test scripts and other test artifacts in the same source repository as your production application code. Most organizations have settled on some form of Git-compliant version (GitLab, AWS CodeCommit, Azure Repos, Bitbucket) as their default version control repository. This enables you to maintain your test artifacts in a system of control that's well understood by many different teams across the organization.

Conversely, storing your performance tests in a monolithic silo (as some legacy performance testing solutions do) isolates performance testing activity from the rest of your automated CI tools (Jenkins, Bamboo, TeamCity, Digital.ai Release, née XebiaLabs XLRelease). That keeps performance testing in the hands of only the select few, not the many and undermines your ability to get earlier, faster feedback loops. When your test automation code lives in a different repository than application code, you typically check in and build the app code before executing the test automation. But when automation code is in the same repository, developers can execute automated scenarios at the same time they run their unit tests — before they check code in. This provides them with much earlier feedback on the code they wanted to submit.

DEVELOP PERFORMANCE PIPELINES

You may consider storing not only your test suites in Git but also store "performance pipeline" scripts in the same repos as load test projects. Performance testing comprises versionable artifacts that must match the app or service code you're testing, so app and test code assets should be in lockstep with each other. Changes to test scripts, testing semantics, and test data sources all usually predicate incorporation into the semantics of your pipeline scripts. But with multiple teams developing features that are flying out the door simultaneously, making sure that versions are the same across code and test assets can be a challenge.

One solution is to make "performance pipelines" triggerable processes separate from broader delivery pipelines. Rather than treating the delivery pipeline as one big monolith, having discrete jobs or pipelines for specific post-build, longer-running test processes allows teams to decide when to trigger them (e.g., not on every commit/push but upon pull requests and before significant merges). Short branches in Git that version your new code and tests together — and are a separate branch from the primary/master branch already approved and used by teams — can be run as the pipeline and proven to be working in order for promotion back to the master/trunk.



You may want to name the branches the same across app and test repositories, or possibly using Git tags that match between these assets. You might clone the current working test suite, modify it and run local load tests, then check it into a new branch and have your pipelines run that branch of test assets on the latest version of the services in a pre-release environment. You may even version your pipeline code right alongside all these assets so that once you prove that everything's working, the promotion (e.g., merge) of these branches in various repositories happens simultaneously.

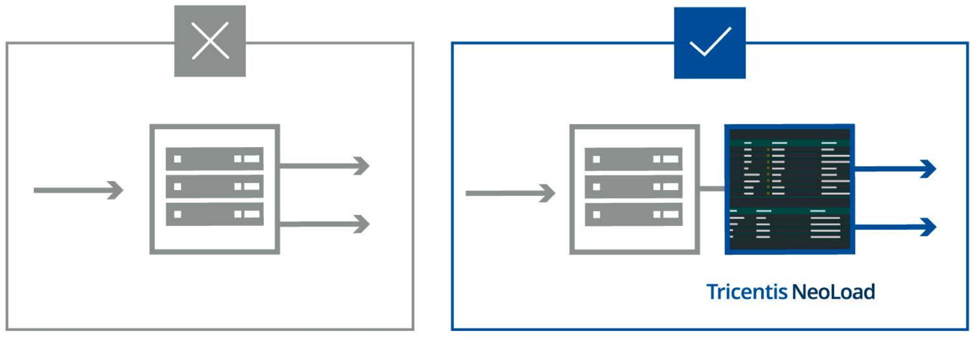
The end goal of performance pipelines is to make it easier to re-run just the performance test if it fails, not the whole delivery pipeline. Separating pipeline stages enables you to run and re-run the load testing process if and when everything compiles and deploys properly but performance SLAs fail due to misconfiguration or environment issues. You don't necessarily need to re-run all the predicating build, package, and deploy steps. Often just some operational configuration and deployment tweaks are sufficient to achieve optimal performance. Then, re-running the performance testing pipeline is a simple task.

Ignore the urge to fix test issues in the master branch. When performance tests fail, understanding why they failed is critical not only for traditional deep-dive analysis but also to inform automated go/no-go indicators in pipelines. Having separate versions of pipelines to diagnose issues helps engineers isolate faults, correct, verify, then integrate their changes back into the production version of the automated process. The number one reason continuous performance testing fails is not testing changes — neglecting to update performance pipelines inadvertently makes them seem flaky or broken.

USE DYNAMIC INFRASTRUCTURE FOR TEST ENVIRONMENTS

Test environment infrastructure is the "big rock" obstacle that prevents many from getting reliable, statistically significant feedback on application or component performance. Specifically, the infrastructure you're using for testing the system under test (SuT) needs to be separate from the infrastructure used to put pressure on that SuT.

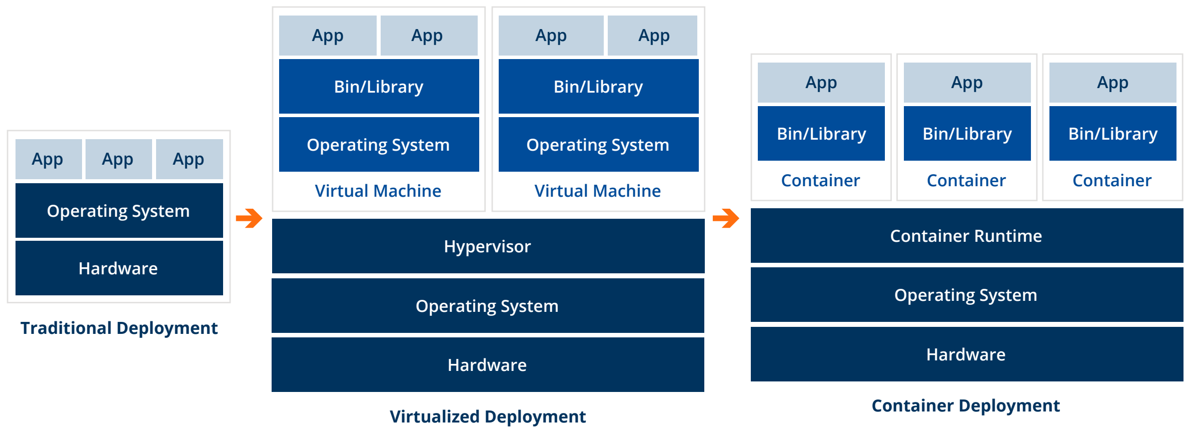
For short, small local performance checks, using a single compute resource is fine; but once you get to larger and longer tests, you have to break up the work across multiple compute resources (load generators) and orchestrate the tests with a controller. The load generators split up the responsibility of putting lots of pressure on the SuT, allowing the controller to consolidate real-time results during tests. It's the same idea as build nodes in CI pipelines, just for the purpose of getting performance objective data that is not biased by the SuT resources when under pressure.



In a pipeline it's problematic to conflate the role of the build node with the load-making software, particularly if that resource is stressed by generating the load — as build status then stops getting reliably reported to the CI master. No one wants flaky agents or failed builds, especially not at scale when running many tests independently of one another. A preferable approach is to leave the pipeline's worker node to just execution semantics (see "Develop performance pipelines" above) but requisition load infrastructure dynamically from a separate system.

The problem with traditional static "always-on" infrastructure in continuous delivery is that you eventually run out of resources when the demand gets too high. If you have too many concurrent build triggers due to the high velocity of feature teams' work, you may wind up failing the build if resources aren't available because someone else is using them. This problem mushrooms if you need multiple resources per build, such as multi-test environments or multiple load controllers/load generators just to run a test. Or conversely, you may find yourself oversizing (and overspending on) resources so that sufficient hardware for testing is available when you need it — but otherwise sits idle.

Reserving and queuing resources doesn't tackle the underlying condition and ultimately becomes an anti-pattern that gets in the way of delivery. A better option is dynamically provisioning using automated resource descriptions.



Many organizations make the switch from always-on static load generators to dynamic only-when-needed infrastructure with traditional virtual machines via VMware on-prem or cloud-based solutions like AWS EC2, Azure or Google Cloud. But this approach has two drawbacks:

- Takes too long to spin up (on the order of minutes)
- Often requires customized manual configuration and repackaging every time as small part of the software version changes

Alternatively, Docker-based solutions like OpenShift and Kubernetes (including Amazon EKS, Google GKE, Microsoft AKS) launch in seconds and are fast to update/publish new versions to a container registry. Auto-provisioning testing infrastructure as needed via these elastic container orchestrators enables you to abstract load infrastructure from performance pipeline semantics more efficiently and effectively. It makes your pipelines more concise, easy to understand, and quick to adjust as necessary. And it removes one of the biggest obstacles to autonomous self-service performance feedback loops across large organizations.

If you're not ready to go that route, most leading performance testing vendors offer some sort of dynamic infrastructure capability. Key things to evaluate are (obviously) the reliability of the system and ease of use: Do you need to write lengthy scripts to provision machines? Manually connect dynamic testing resources to CI pipelines? Manually change the number of load generators used for a test?

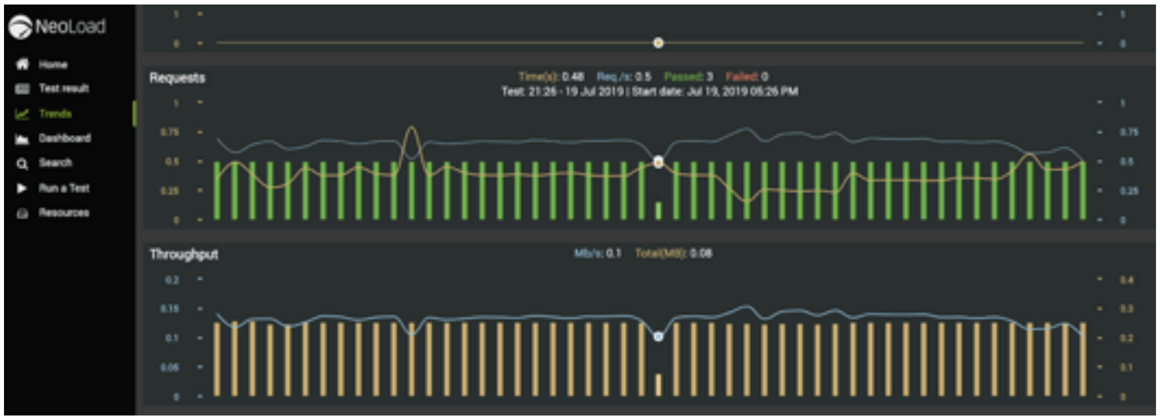
Whatever provisioning approach you take, it has to be easy for teams of non-experts to run a test autonomously.

ENSURE TRUSTWORTHY GO/NO-GO DECISIONS

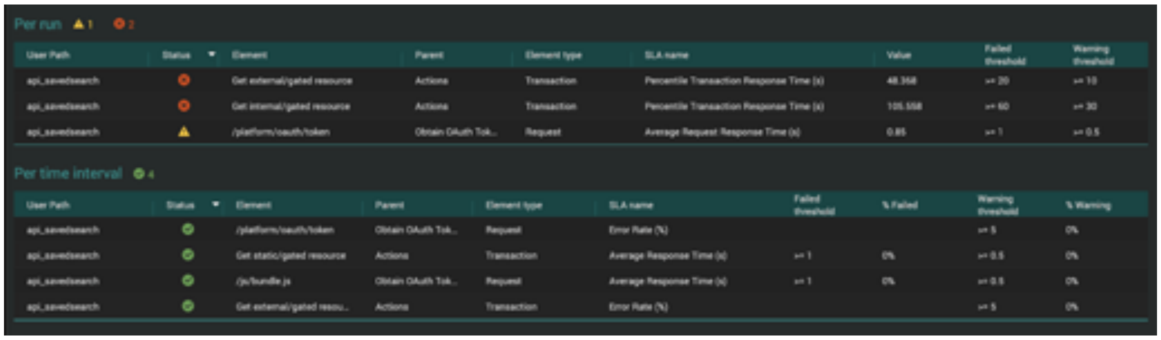
The whole point of implementing continuous performance testing is to provide the right people with useful, frequent feedback that can actually help them do something about the performance of the system, application, or component. In CI pipelines, these early feedback loops manifest themselves into actionable, trustworthy go/no-go decisions.

Test results must be easy to understand and actionable for product teams and developers, and you shouldn't have to wait a long time to know if during your performance pipeline your systems, application, or service is failing performance thresholds. Too often, gains realized from automating test execution are undermined by the need to manually swivel from screen to screen, cobbling together data to get a clear picture of what's going on — so-called cognitive thrash. A large part of automating go/no-go decisions is ensuring that the tests incorporate and reflect specific SLOs as measured by precise SLIs (see "Start with the end in mind" above).

The illustration below displays results of nightly load testing where critical API endpoints and workflows provide an easily accessible view of performance over time:


NeoLoad
Home
Test result
Trends
Dashboard
Search
Run a Test
Resources
Requests
Time(s): 0.48 Req./s: 0.5 Passed: 3 Failed: 0
Test: 21:26 - 19 Jul 2019 | Start date: Jul 19, 2019 05:26 PM
Throughput
Mb/s: 0.1 Total(MB): 0.08

Additionally, each of these test results has specific SLAs, further clarifying what meets vs. what falls short of expectations:



Per run ⚠ 1 ⊗ 2

User Path	Status	Element	Parent	Element type	SLA name	Value	Failed threshold	Warning threshold
api_savedsearch	⊗	Get external/gated resource	Actions	Transaction	Percentile Transaction Response Time (s)	48.368	>= 20	>= 10
api_savedsearch	⊗	Get internal/gated resource	Actions	Transaction	Percentile Transaction Response Time (s)	105.558	>= 60	>= 30
api_savedsearch	⚠	/platform/oauth/token	Obtain OAuth Tok...	Request	Average Request Response Time (s)	0.85	>= 1	>= 0.5

Per time interval ✓ 4

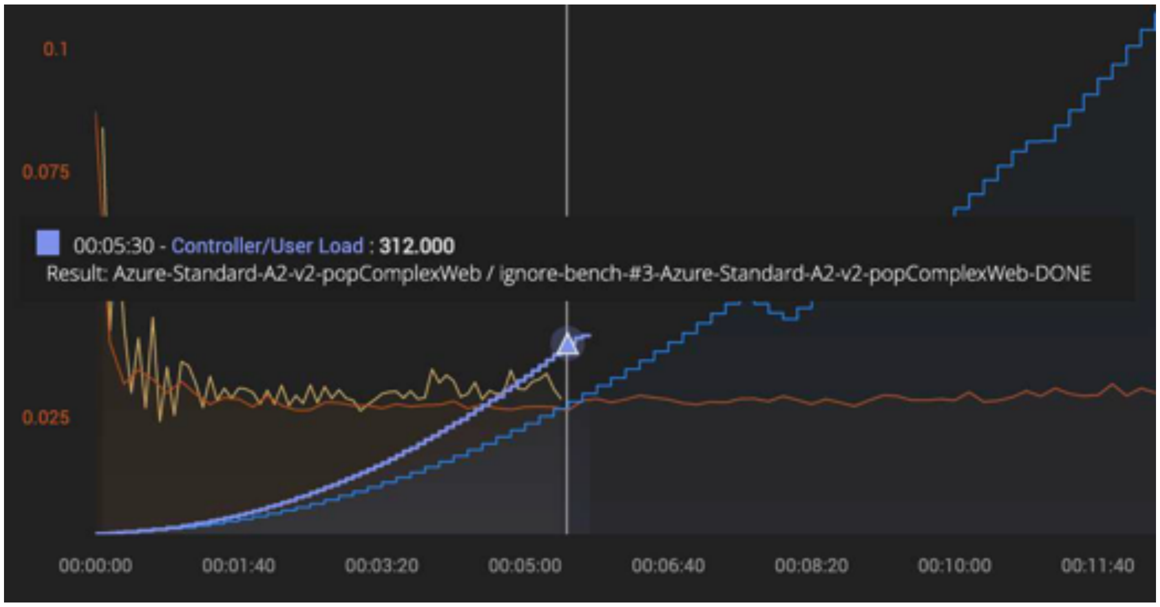
User Path	Status	Element	Parent	Element type	SLA name	Failed threshold	% Failed	Warning threshold	% Warning
api_savedsearch	✓	/platform/oauth/token	Obtain OAuth Tok...	Request	Error Rate (%)			>= 5	0%
api_savedsearch	✓	Get static/gated resource	Actions	Transaction	Average Response Time (s)	>= 1	0%	>= 0.5	0%
api_savedsearch	✓	/js/bundle.js	Obtain OAuth Tok...	Request	Average Response Time (s)	>= 1	0%	>= 0.5	0%
api_savedsearch	✓	Get external/gated resou...	Actions	Transaction	Error Rate (%)			>= 5	0%

Service “impact” metrics such as USE and RED should be a critical part of your go/no-go decision strategy. The combination of RED and USE measurements during a load test allows performance pipelines to “fail fast” across both the pressure/load and impact/service observations. Without both these types of metrics, you simply do not have a sufficient view to know if performance is acceptable or not.

Reduce cognitive thrash

Performance data must be easily consumed and domain-specific. The more (and more often) you have to look at data, graphs and charts, the more likely you will get desensitized to essential indicators. A big data dump of all testing results in a PDF doesn’t offer the kind of actionable, timely feedback that actually helps. Developers and product teams shouldn’t have to wade through a sea of statistics or manually cobble together information in order to get meaningful insights.

Automated continuous performance testing also enables you to frequently generate performance data to understand trends over time, not just “big bang” point-in-time samples. Comparing current tests to a baseline and accessing trend-based data should be easy within the context of pipelines. Trends provide context at the moment a particular performance issue presents itself. Even with phenomenal application performance monitoring (APM) and tracing tools in place, you won’t know if the issue is an outlier or a symptom of a bigger problem. Armed with historical data, you can focus on themes and distributed patterns rather than log entries. Keeping focus reduces the toil of getting paged again when the same problem pops up in a different place in your API ecosystem.



00:05:30 - Controller/User Load : 312.000
Result: Azure-Standard-A2-v2-popComplexWeb / ignore-bench-#3-Azure-Standard-A2-v2-popComplexWeb-DONE

Layer in guardrails

As more of the test-execution toil comes off the performance engineer's plate (and automated into distributed teams' workflows), they are freed up to focus more on providing safety guardrails and best practices around testing practices so that teams can advance their own performance abilities. Often these experts will look to layer in additional testing artifacts and monitoring processes over and above what the product teams use in their automated performance testing. Or they may further train and coach the product teams on what to look for, better analyze results, etc.

This gives rise to the notion of "guardrails" — best practices — that are distilled into testing artifacts and templates so that performance experts don't have to repeat the same labor-intensive tasks across hundreds of teams and performance pipelines. This is really what makes automation a force multiplier that benefits all teams: how to automate something that people keep asking to be done?

Additional homegrown tooling might be implemented before and after performance test pipelines run, such as automated environment configuration comparison utilities, chaos injection (such as with Gremlin) to prove that load balancing and pod scaling in Kubernetes works as defined, and custom baseline comparison and trends reporting that runs after every load test. These "guardrails" help product teams by not requiring them to be experts at performance testing process details, but still providing them with the deep insights they need to take early action and keep their product delivery cycles free from late-stage surprises.

CONCLUSION

The easiest place to get started with automating continuous performance feedback loops is with API testing. But enabling automated go/no-go decisions within API testing calls for more time spent on planning than is currently the norm. And processes and procedures need to be in place to open up timely, actionable feedback loops — especially around defining SLAs, SLOs, and SLIs; being able to express performance tests as code; branching pipelines; provisioning test infrastructure; and enabling easy test results analysis.

But once the foundations of automated continuous performance testing are in place, you are well on your way to going from testing-as-a-service performed by a handful of performance experts in silos to testing-as-self-service where performance testing becomes democratized so that teams across the entire organization have a way to test performance themselves, painlessly and autonomously. Performance experts are freed up to focus on more strategic, higher-value activities that have a greater bottom-line impact.

ABOUT TRICENTIS

Tricentis is the global leader in enterprise continuous testing, widely credited for reinventing software testing and delivery for DevOps and agile environments. The Tricentis AI-based, continuous testing platform provides automated testing and real-time business risk insight across your DevOps pipeline. This enables enterprises to accelerate their digital transformation by dramatically increasing software release speed, reducing costs, and improving software quality. Tricentis has been widely recognized as the leader by all major industry analysts, including being named the leader in Gartner's Magic Quadrant five years in a row. Tricentis has more than 1,800 customers, including the largest brands in the world, such as Accenture, Coca-Cola, Nationwide Insurance, Allianz, Telstra, Dolby, RBS, and Zappos.

To learn more, visit www.tricentis.com or follow us on LinkedIn, Twitter, and Facebook.

AMERICAS
2570 W El Camino Real,
Suite 540
Mountain View, CA 94040
Unites States of America
office@tricentis.com
+1-650-383-8329

EMEA
Leonard-Bernstein-Straße 10
1220 Vienna
Austria
office@tricentis.com
+43 1 263 24 09 – 0

APAC
2-12 Foveaux Street
Surry Hills NSW 2010,
Australia
frontdesk.apac@tricentis.com
+61 2 8458 0766