Self-Service Performance Testing Platform for Autonomous Development Teams

Aleksei Vasilevskii
Performance & Observability Team
Wolt
Munich, Germany
aleksei.vasilevskii@wolt.com

Oleksandr Kachur
Performance & Observability Team
Wolt
Helsinki, Finland
alexander.kachur@gmail.com

ABSTRACT

In the modern fast paced and highly autonomous software development teams, it's crucial to maintain a sustainable approach to all performance engineering activites, including performance testing. The high degree of autonomy often results in teams building their own frameworks that are not used consistently and may be abandoned due to lack of support or integration with existing infrastructure, processes and tools.

To address these challenges, we present a self-service performance testing platform based on open-source software, that supports distributed load generation, historical results storage and a notification system to trigger alerts in Slack messenger. In addition, it integrates with GitHub Actions to enable developers running load tests as part of their CI/CD pipelines.

We'd like to share some of the technical solutions and the details of the decision-making process behind the performance testing platform in a scale-up environment, our experience in building this platform and, most importantly, rolling it out to autonomous development teams and onboarding them into the continuous performance improvement process.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**.

KEYWORDS

performance testing, continuous integration, microservices, autonomous teams

ACM Reference Format:
Aleksei Vasilevskii and Oleksandr Kachur. 2024. Self-Service Performance Testing Platform for Autonomous Development Teams. In *Companion of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24 Companion), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3629527.3652268

1 INTRODUCTION

Like many other software organizations developing cloud-native applications, we at Wolt have chosen microservices architecture because of the high degree of autonomy in software engineering teams, that enables feature development and short time to production [14]. The downside is that as the size of the organization grows, it becomes exponentially more difficult to maintain operational visibility over a system consisting of hundreds of microservices [5, 19]. This applies not only to functional aspects, but also to non-functional requirements such as performance and scalability. In a traditional organization, having a centralized performance team responsible for these properties may still work, but in a dynamic scale-up setup, where workloads are constantly increasing and team staffing is lagging behind, the only viable solution is to distribute the responsibility to the development teams [5]. However, this decision can lead to divergent approaches and tools for performance testing, with each team implementing its own framework that works best for its particular use case, but may be far from the "global organizational optimum" [20]. In addition, the groundwork associated with performance testing can be seen as "less important" by teams under pressure to develop features, resulting in poor integration with existing infrastructure and processes [5, 19]. The potential for such artifacts to be reused by other teams, or to be collaborated on and pushed toward a standardized platform, remains relatively low, exacerbating silo problems caused by team communication overhead [5, 6, 21].

Given these constraints, the scope of a dedicated performance team is shifting more and more toward standardized tools, development experience, knowledge sharing, and the definition of best practices. Clearly, there are challenges to building a standardized performance testing platform, but there are also benefits: reducing engineering effort by streamlined teams, eliminating code duplication, empowering teams with the necessary tools and knowledge, and accelerating time to market [21]. Continuous performance testing integrated into CI/CD processes enables shorter feedback loops for software changes. Ongoing platform support, common tooling, and new feature development all increase team buy-in, making performance testing a standard practice rather than an obscure one-off activity.

2 REQUIREMENTS

The primary purpose of the performance testing platform is to provide a unified way to conduct repeatable end-to-end system performance tests on existing cloud-native infrastructure. The need to implement an in-house platform stems from the fact that there is no tool-agnostic open-source implementation that meets our requirements. Existing platforms are often product- or company-specific [12], making it impossible to reuse them in a different environment. Other solutions, however, may be too generic [11] and require the implementation of missing components and integrations, which can be as time-consuming as building a solution from scratch. More specifically, the high-level requirements in our case were:

1. Cloud-native, cost-effective and scalable solution.
2. HTTP, WebSocket and gRPC traffic generation.
3. Easily extensible and configurable.
4. Support both manual tests and integrate into existing CI pipelines.
5. Automation for test execution and results evaluation.
6. Long-term results storage with possibility to visualize trends.
7. Results repeatability.

When it came to a build vs. buy decision, it was clear that no existing SaaS solution could meet our needs in a cost-effective way. On the other hand, building a tool from scratch wasn't a viable option: different teams were already using different tools, and the longer it would take to release the first version of the framework, the harder it would be to migrate existing tools [5].

There are many decent open-source performance testing tools to integrate with, and this presents the next challenge: choosing the right tool that meets our needs, but is also suitable for most engineering teams [3]. Wolt is a multi-language environment with the main programming languages being Python, Scala and Kotlin, which makes it impossible to force a tool with a specific language (e.g. Locust), while tools that lack one (e.g. ab) or are mostly GUI based (e.g. JMeter) and do not provide the required level of flexibility. Another hard requirement is load generation efficiency at high throughput levels, reaching up to 10k RPS. In addition, the tool of choice should provide decent real-time reporting that is easy to understand and use for comparison, integration capabilities for CI/CD pipelines, observability tools and Kubernetes, and extensibility.

Considering all of the above requirements, Gatling seemed like the optimal choice, see Table 1. Most of the engineers were already familiar with one of the JVM languages, and the learning curve for mastering a framework with expressive DSL built on top of Scala was relatively flat. On the technical side, Gatling convinced us with reasonable load generation efficiency, measurement precision, extensibility, and a wide range of supported protocols [8].

During the early development stages of our performance testing platform, we noticed that test results were showing significant variations that can be attributed to the cloud environment itself [9, 10]. Since we wanted to measure the actual performance characteristics of our services in shared Kubernetes clusters, the only way to improve the repeatability of the measurements was to increase the minimum test duration to collect enough data for a statistically meaningful measurement during each execution [18].

3 ARCHITECTURE

We now present the high-level architecture of our solution. As shown in Figure 1, it consists of a Gatling-based load generation application (wolt-load-test), a results storage and analysis module (Witness), custom GitHub Actions to trigger tests from CI/CD pipelines and Argo Workflows to schedule tests on Kubernetes.

The load test can be triggered either automatically by calling the corresponding GitHub Action in a CI/CD pipeline or manually by a user via the Argo WebUI or CLI. In both cases, an Argo Workflow is triggered to start a Kubernetes job with the wolt-load-test workload

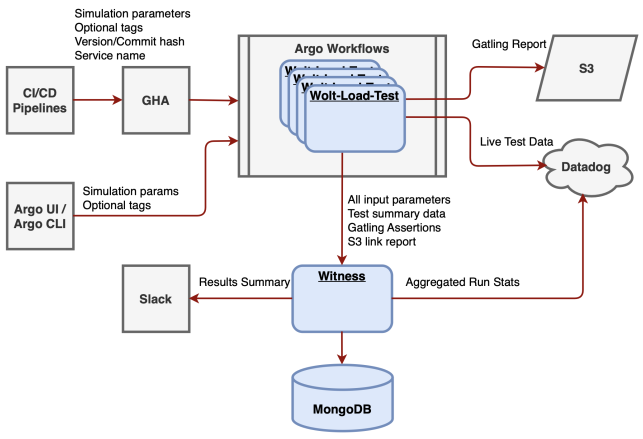


Figure 1: High-level architecture.

in the target cluster [13]. The wolt-load-test uses a custom reporter module to send live data to Datadog via the Statsd protocol [7]. Once a load test is complete, the framework generates a report, persists it to an AWS S3 bucket, and sends a summary of the test run to our custom results management module, Witness.

Witness stores the summary of the test run in MongoDB, sends a notification to a predefined Slack channel, and provides a REST API that can be consumed for further automated processing.

3.1 Load Generation

We use a custom wrapper around the open-source Gatling tool for load generation, which reduces the learning curve for developers and provides features for a smoother development experience such as standard service-to-service authentication, integration with observability tools (e.g. logs, metrics, traces), and test data management. At its core, the framework provides load test definition templates with multiple levels of hierarchy that can be easily combined. At the top of this hierarchy is a Simulation that defines load levels and durations across one or more Scenarios. Each Scenario combines Requests from one or more Services into a sequential execution chain. A Service contains basic configuration elements such as target host, name, and description, and bundles Requests for ease of navigation. Finally, a Request combines request templates with test data providers - Feeders [8].

In addition to standard Gatling feeders, we also implemented custom Redis-based feeders that support more advanced data structures such as hashmaps and that can be used to store shared or TTLed data. This effectively reduced the amount of memory required for wolt-load-tests with extremely large datasets, and enabled the handling of auto-expiring JWT tokens used for authentication.

The wolt-load-test has its own repository and a CI/CD pipeline that packages all load simulations in an executable JAR, creates a Docker image around this JAR and publishes it to an AWS ECR repository for deployment to respective environments. The load test dispatch is handled by Argo Workflows, an open source container-native workflow engine for orchestrating parallel jobs on Kubernetes implemented as a Kubernetes CRD [1]. This engine was already in place for scheduling other types of jobs, so it was natural to reuse the existing infrastructure.

Our primary observability platform is Datadog, but Gatling does not integrate with it out of the box. To get around this limitation and avoid the overhead of maintaining a fork of Gatling, we created custom runtime monitoring for load tests by implementing a Statsd protocol reporter using the Byte Buddy framework. Byte Buddy is a code generation and manipulation library for creating and modifying Java classes during the runtime of a Java application and without the help of a compiler, which is widely used in Java Agents of APM and observability tools [23]. To instrument the Gatling code and make the instrumentation as lightweight as possible, we created our own Java Agent. The agent was included in the manifest of the executable JAR in the Launcher-Agent-Class to be launched before the main method of the application is invoked, see Listing 1.

Listing 1: Maven configuration for loading Agent.

```
<transformers>
   <transformer implementation="ManifestResourceTransformer">
       <manifestEntries>
           <Main-Class>io.gatling.app.Gatling</Main-Class>
           <Premain-Class>com.wolt.Agent</Premain-Class>
           <Can-Retransform-Classes>true</Can-Retransform-Classes>
           <Launcher-Agent-Class>com.wolt.Agent</Launcher-Agent-Class>
       </manifestEntries>
   </transformer>
</transformers>
```

In this configuration, the JVM first attempts to invoke the agentmain method on the agent class, as shown in Listing 2.

Listing 2: Agentmain implementation.

```
public static void agentmain(String args, Instrumentation inst) {
    ByteBuddyAgent.install();
    new AgentBuilder.Default()
            .with(AgentBuilder.Listener.StreamWriting.toSystemOut()
            .withTransformationsOnly())
            .type(named(Transformers.STATS_ENGINE.label))
            .transform(Transformers.STATS_ENGINE.transformer)
            .type(named(Transformers.RESULT_PROCESSOR.label))
            .transform(Transformers.RESULT_PROCESSOR.transformer)
            .installOn(inst);
```

Our agent uses Byte Buddy Advice to transform the Gatling classes and inject the custom code for Statsd monitoring. The high-level workflow of the agent is shown as a sequence diagram in Figure 2.

With this agent in place, we were able to send real-time test data to Datadog and provide an integrated developer experience for both load test results and system-under-test metrics, traces, and profiling data in one place. A dashboard with live test data is shown in Figure 3.

The results we got with the custom agent were quite promising, and we found a few more uses for it. In our reporting pipeline, we relied on Gatling reports, which contain statistics calculated over the entire duration of a load test, including a ramp-up and a steady load period. In some cases, e.g. when running tests in CI/CD pipelines, it's extremely beneficial to calculate statistics over measurements taken only in the second phase, during the steady

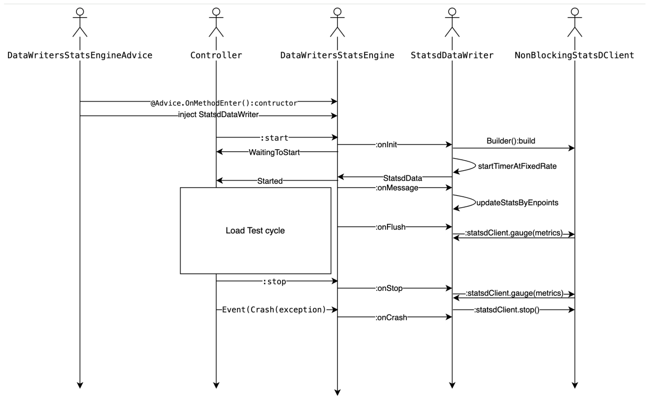


Figure 2: Sequence diagram for custom runtime monitoring.

load period. This filters out the cold-start effects that occur in recently deployed applications, and the measurement results are more accurate, reliable, and repeatable. We initially considered developing a custom simulation log processor, but this would have meant rewriting the logic already present in Gatling and introducing a potentially recurring maintenance effort. Instead, we decided to write another ByteBuddy Advice that would skip all data before a specified timestamp during the native Gatling report processing step, see Listing 3.

Listing 3: Advice for skipping ramp-up period.

```
public class ResultHolderAdvice {
  @Advice.OnMethodEnter(skipOn = Advice.OnNonDefaultValue.class)
  public static boolean skipRecordProcessing(
    @Advice.FieldValue("minTimestamp") Long startTime,
    @Advice.Argument(value = 0, readOnly = false) Object o) {

    if (o instanceof RequestRecord r) return r.start() < startTime;
    if (o instanceof GroupRecord r) return r.start() < startTime;
    if (o instanceof ErrorRecord r) return r.timestamp() < startTime;
    if (o instanceof UserRecord r) return r.timestamp() < startTime;
    return true;
  }
}
```

We found another useful application for the Byte Buddy Agent. At the end of each test, wolt-load-test reports a summary of the results to our storage service - Witness. Instead of developing and maintaining separate scripts that are not part of the main codebase, we decided to implement yet another Advice attached to io.gatling.charts.report.ReportsGenerator that makes an appropriate HTTP call to Witness once the report data is available.

3.2 Test Execution

In our setup, load tests are run as pods in a shared Kubernetes cluster. This allows us to simulate service-to-service traffic in the exact same environment used by our applications, including concurrency of resource allocation, possible network shenanigans, etc. [19]. At the same time, we benefit from using existing infrastructure for configuration and secret management, resources provisioning, and generate no additional maintenance overhead. The downside of this approach is the potential for load generation to interfere with other workloads - or vice versa - the common "noisy neighbor problem"

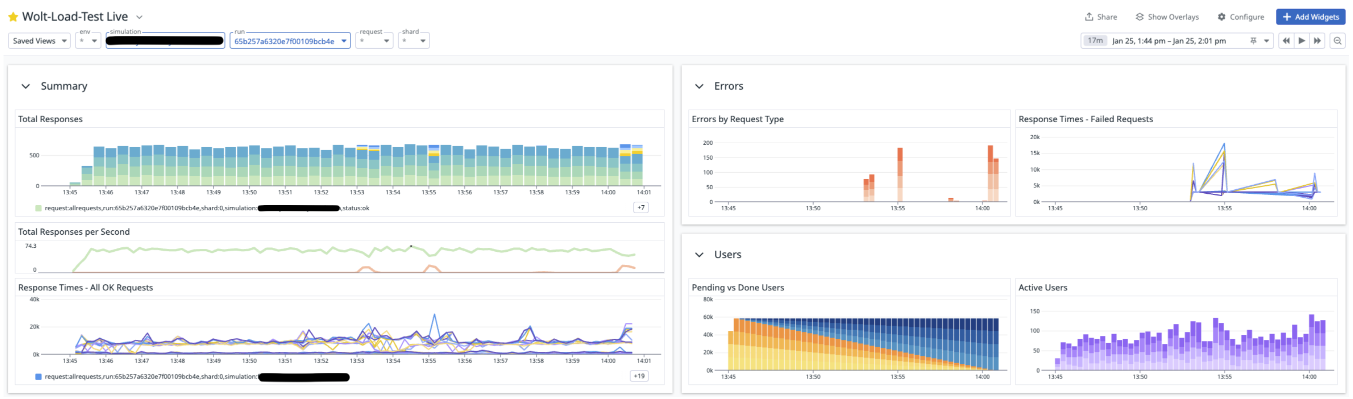

Wolt-Load-Test Live
Saved Views
env
simulation
[redacted]
run
65b257a6320e7f00109bcb4e
request
shard
Share
Show Overlays
Configure
Add Widgets
17m Jan 25, 1:44 pm – Jan 25, 2:01 pm
Summary
Total Responses
request:allrequests,run:65b257a6320e7f00109bcb4e,shard:0,simulation:[redacted],status:ok
Total Responses per Second
Response Times - All OK Requests
request:allrequests,run:65b257a6320e7f00109bcb4e,shard:0,simulation:[redacted]
Errors
Errors by Request Type
Response Times - Failed Requests
Users
Pending vs Done Users
Active Users

Figure 3: Wolt-load-test Live Dashboard.

[24]. This is mitigated by applying anti-affinity rules to specific pods, and by exposing the Kubernetes resource configuration for the wolt-load-test pod as part of the test run configuration. The latter enables a more granular resource management: regular CI load tests get the bare minimum to run, while larger production-sized workloads can allocate much more resources. Actual usage can be monitored using the standard Kubernetes dashboards in Datadog.

One of the alternatives we considered was an external load generation point, such as a separate Kubernetes cluster dedicated to running load test workloads. This would allow us to exercise all components along the external path, including firewalls, external load balancers, etc. However, this would come at a significant cost, both in terms of running an additional Kubernetes cluster and the traffic between the clusters. Given the rate limiters and firewall rules set on the external route, this initiative would require even more effort to invest in workarounds, so it was left out of the scope for now.

Starting a test execution with our solution is possible in several ways. The simplest and most straightforward is to trigger an Argo Workflow via WebUI or CLI by specifying several run parameters, such as simulation class name, maximum target load level, ramp-up and test durations, and some optional custom parameters. It's also possible to specify how many pods should be scheduled for a given workload - we call them shards - and how much resource should be allocated to each shard.

In fact, the original implementation had no sharding support and was quite simple: it just started a single pod with the desired simulation and post-processing step.

After running this setup for some time, we realized that conducting load tests from a single pod can produce results of insufficient quality in terms of connection patterns, load distribution, and resource allocation. Since Argo Workflows also supports more sophisticated DAGs, we split the simple two-step "run test - collect results" job into a multi-step workflow. This new workflow included a preparation step to transform input data and generate a common test run ID, a fan-out step to launch multiple wolt-load-test pods in parallel, and a post-processing step to collect and combine all results into an aggregated test report, as shown in Figure 4. This change not only allowed for more accurate load generation, but also greatly improved the scalability characteristics of our load testing platform, making it suitable for conducting large-scale performance tests.

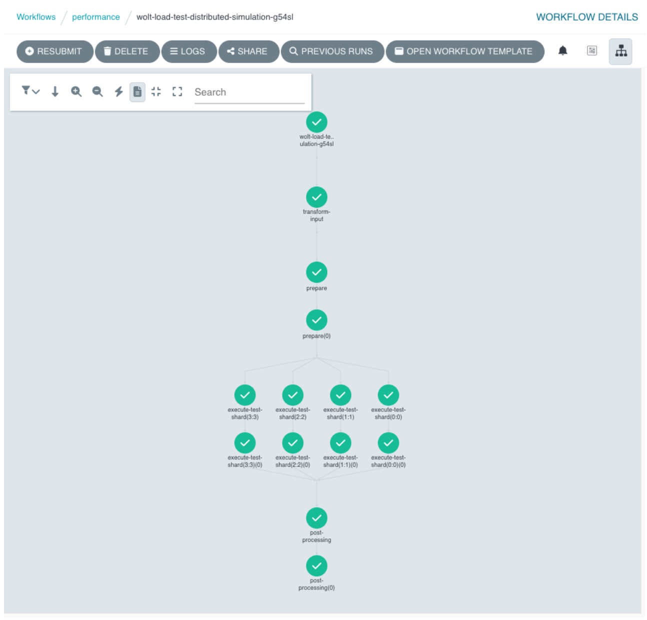


Figure 4: Argo Workflow for wolt-load-test with multiple shards.

There is a caveat to the fan-out approach: the start times of the pods may not be perfectly synchronized due to possible scheduling delays in Kubernetes. However, in practice, these delays are usually insignificant and can be ignored for most tests. To further mitigate this, we are considering having a dedicated pool of nodes for running load tests and introducing a synchronization checkpoint during test startup.

Integrating performance testing into services' CI/CD pipelines was as simple as writing a reusable GitHub Action that triggers the appropriate Argo Workflows. This allowed engineering teams to store load test configurations in their respective service-under-test

GitHub repositories and have full control over how and when they run load tests in CI.

A special case is the production environment: in some cases, it's the only environment where meaningful performance measurements can be made due to size, cost, or data volume constraints [19]. In order to support the performance testing in production, we had to implement additional safety measures to avoid any negative impact on the live system, to remain compliant, and to provide auditability of production changes. To this end, the production Argo Workflow instance was hardened to prevent users from running workflows via the WebUI or CLI. Instead, a pull request to a central GitHub repository and a config deployment is required to start any type of load test in production. The downside of this approach, of course, is the reduced user experience compared to the interactive WebUI, but it was accepted by the engineering community as a reasonable compromise.

3.3 Data Store and Feedback Loop

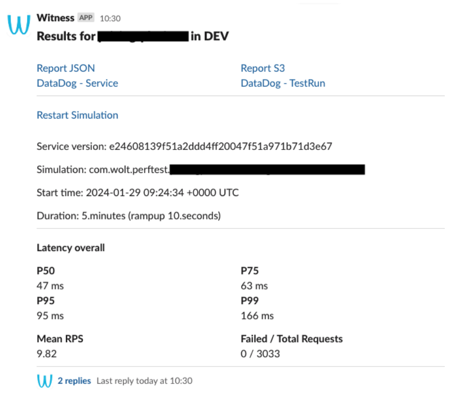
Upon completion of the test, a Gatling report is generated by wolt-load-test and stored in an S3 bucket with a unique run ID. Searching and exploring reports directly in the S3 bucket was acceptable in the early stages of adoption with a relatively small number of reports, but proved to be quite cumbersome in the long run. Unfortunately, there are no established open-source results tracking tools that would meet our needs and provide all the integrations we were looking for. So we decided to develop our own service that would simplify the management of test results. To reduce the resource footprint and keep the service minimalistic, we chose Golang as the programming language. Since the concept of test results and the expected access patterns fit well into the document-oriented paradigm, we chose MongoDB for the storage layer.

Listing 4: Data structure representing a test report.

```
type GatlingReport struct {
    ID primitive.ObjectID `bson:"_id" json:"id,omitempty"`
    SimulationName string `json:"simulation_name,omitempty"`
    ServiceName string `json:"service_name,omitempty"`
    StartedAt int64 `json:"started_at,omitempty"`
    Version string `json:"version,omitempty"`
    Tag string `json:"tag,omitempty"`
    Params SimulationParams `json:"params,omitempty"`
    Stats GatlingStats `json:"stats,omitempty"`
    Assertions AssertionsResult `json:"assertions,omitempty"`
}
```

The initial Witness implementation included an HTTP REST interface to accept test result summaries with all aggregated per-request and global statistics: response time percentiles, assertions, request and failure rates, along with test metadata that allowed each test run to be uniquely identified [18]. The latter included run ID, simulation class name, load profile details such as max RPS and duration, test start timestamp, service name, version, and tags - see Listing 4. This information is used to group test runs by type for historical comparisons, for example, to distinguish between CI-triggered runs, low-, medium-, and high-load tests, and so on.

Just keeping this information in Witness and providing an HTTP REST interface to read it didn't add much value by itself. Not many people outside of the performance team were using it. To get more traction, we introduced some integrations, the most important of which is a Slack bot: Witness sends a notification to a predefined Slack channel when a test starts, and another message when a test finishes. The latter provides a brief summary and links to relevant information, as shown in Figure 5, which can be used as an initial entry point for detailed analysis based on the metrics, tracing, logs, and profiling information available in Datadog.


Witness APP 10:30
Results for [redacted] in DEV

Report JSON
DataDog - Service
Report S3
DataDog - TestRun

Restart Simulation

Service version: e24608139f51a2ddd4ff20047f51a971b71d3e67

Simulation: com.wolt.perftest.[redacted]

Start time: 2024-01-29 09:24:34 +0000 UTC

Duration: 5.minutes (rampup 10.seconds)

Latency overall

P50
47 ms
P75
63 ms
P95
95 ms
P99
166 ms

Mean RPS
9.82
Failed / Total Requests
0 / 3033

2 replies Last reply today at 10:30

Figure 5: Witness run summary in Slack.

This message also provides historical values for that specific combination of simulation class name, load profile, and tag, as well as assertion evaluation results - if any are defined for the simulation. If there are any assertion failures, an additional alert message is generated. This Slack integration has been extremely useful for engineering teams to get an early indication of performance degradation when running load tests as part of their CI/CD pipelines. As a side effect, the Witness Slack channel now acts as a global log of all performance tests, with quick links to all relevant details.

On the other hand, searching for messages in Slack and manually comparing them was not ideal for evaluating long-term trends for specific simulations. To address this, Witness was enhanced with a Statsd integration with Datadog to plot measurement trends.

4 ORGANIZATIONAL CHALLENGES

One of the biggest challenges with the self-service performance testing platform was not on the technical side of things, but rather on the organizational side. With multiple development teams having a different vision [5, 6, 19, 21] of how performance testing should look like and what tools should be used, and given the high degree of autonomy in these teams, it was not possible to force any one solution. A much better approach was to first build a foundation by making it as easy and straightforward to use as possible, integrating it with existing processes and infrastructure, and providing the core functionality common to all teams and departments.

In our case, the first step was to make the framework cloud-native, running as a container in Kubernetes with standard infrastructure tools, as opposed to the more common practice of spinning up dedicated EC2 instances for load testing. The next step was to integrate authentication and user management, which had been an integral part of the first release of the platform. With this feature set as a solid foundation, we began to seek out teams that were maintaining their own load tests and interviewed them about their pain points and desired improvements. By implementing these in our platform, we were able to gradually migrate their homegrown solutions to the common platform. Using this gradual adoption approach and promoting the platform in internal meetings and workshops, wolt-load-test/Witness became the de facto standard for running load tests across the company.

A special case were teams using non-JVM based languages for development, such as Python or Javascript. These teams were understandably reluctant to introduce an entirely new language just for the sake of writing tests. To make it a success story, we provided even more support and guidance to get started, more concrete reference test implementations, and finally, more thorough PR reviews. In the end, the benefits of having great developer experience, tooling support, and a fully integrated solution outweighed the burden of setting up the project and learning the few bits and pieces of Gatling DSL and Scala.

The best part that both product engineering and the performance teams like about the established collaboration is that there's no gatekeeping or heavy reliance on the framework maintainers, since all engineering teams own their load testing code and can run any experiments they want. CI checks on pull requests ensure that the platform is always in a runnable state, and for complex changes or larger feature requests, engineering teams can always get support from the performance team [21].

The rollout of the testing platform was accompanied by a detailed set of guidelines and best practices for systematically addressing performance activities. These guidelines were marketed internally as a "Performance Engineering Framework" with actionable items and tracking for individual teams.

The overall methodology of platform adoption at Wolt can be described in the following steps:

(1) The performance team reaches out to development teams that are not yet onboarded to discuss their needs. This activity can be scheduled on a regular basis for specific teams to increase their engagement. The ultimate goal is to make development teams proactive and ensure that they go through this process on their own, without external requests. Alternatively, development teams can contact the performance team with support requests.
(2) The two teams work through the action items in the "Performance Engineering Framework" and determine if the service should be enrolled for the platform. This depends primarily on the service's APIs, expected load and criticality, as well as any third-party dependencies, data volumes, and overall complexity.
(3) If a service is deemed eligible for enrollment, the two teams evaluate how well it fits into the existing feature set of the platform. If any deficiencies are identified, the performance team proceeds with the implementation of new features.
(4) Once all requirements are met, the development team implements load tests and bindings for integration into their CI/CD process. The entire process is well documented with many examples and detailed step-by-step instructions. If the development team has little or no prior experience with the platform, the performance team can support them with thorough reviews of the changes or draft PRs.
(5) Once the initial implementation is complete, the development team gains full ownership of their load tests and can evolve them as needed. If questions arise during the evaluation of the results, the performance team can assist with root cause analysis.

In rare cases, some services may be excluded in step 2. This usually happens for applications that do not fit well into the current platform paradigm, such as stream processing applications [11]. For these applications, alternative solutions can be considered, such as canary releases or traffic mirroring in production to evaluate performance without impacting real users [4, 22].

Ultimately, addressing these organizational challenges influenced the way engineers approached load testing and helped foster a healthy performance engineering culture, making performance testing a standard part of the development process rather than an obscure one-off exercise. Other notable impacts include freeing up engineering resources by retiring existing scattered testing solutions, improving the efficiency of resource allocation in the cloud and reducing associated costs based on performance testing results, and increasing reliability KPIs in various parts of the Wolt system.

5 RELATED WORK

At the time we started developing the platform, there were a few similar end-to-end performance testing tools and publications. However, most of them either target a specific technology or provide a different subset of features.

One example is the MongoDB's Distributed Systems Infrastructure [12]. This framework is used to conduct fully automated performance testing in a CI environment for MongoDB clusters, as well as to provision and deploy the clusters. While it has some similarities to our solution, such as the cloud-native approach and support for both manual and automated CI benchmarks, it can only be used to generate loads for MongoDB and integrates only with the Evergreen CI system [16].

Dell Technology uses JaaS - JMeter as a Service - a performance testing solution built on top of JMeter, Docker, Elastic and Axon - to validate Dell servers before shipping them to customers [17]. It provides distributed load generation for multiple workloads, including HTTP and database traffic, live dashboards and results storage. JaaS does not support running tests natively in Kubernetes and does not provide automated results analysis in the feedback loop.

Theodolite is a framework for benchmarking the scalability of cloud-native applications, running on Kubernetes. It automates the benchmarking process by deploying the system under test (SUT) on a Kubernetes cluster, generating load on the SUT, and collecting performance metrics during load generation [11]. This advanced framework is similar to our solution in many ways, and it provides a set of out-of-the-box benchmarks for streaming processing applications such as Apache Kafka Streams and Apache Flink. For all other types of traffic generation - including HTTP and gRPC - it requires a custom implementation to be provided externally. Test execution is triggered by deploying a custom resource definition (CRD) to a Kubernetes cluster, which is similar to our approach of using Argo Workflows to schedule load generating pods. The distinguishing feature of Theodolite is how it runs isolated experiments for different load intensities and provisioned resources for the SUT. It provides a set of search strategies to evaluate possible combinations of resources and loads based on configurable service level objectives (SLOs). To store and analyze results, it utilizes a persistent volume, a Grafana server, and a set of Jupyter notebooks.

6 FUTURE WORK AND CONCLUSION

The platform we have presented in this submission, is still a work in progress and there are several features and improvements in the works.

One of these improvements touches on the currently used threshold based alerts via assertions: these tend to be flaky, and we have considered using one of the change-point detection algorithms to introduce outlier detection and reduce false positives, e.g. by implementing E-Divisive with Means or similar approaches [15].

In addition, we plan to collect aggregated statistics on application-side metrics such as CPU and memory utilization, profiling and tracing data, etc. and bundle them with the results summary to provide resource utilization comparisons and regression analysis.

Developer experience with the platform can be further enhanced by introducing an interactive integration with Slack, e.g. by providing an easy way to re-run a failed load test directly from the Slack message (party implemented), or by introducing a chat bot functionality to manage load tests without the need for the Argo WebUI or CLI.

Another improvement to the developer experience is planned integration with the internal development portal based on Backstage [2]. This would allow performance measurement data to be embedded into a service health scorecard, to track the progress on "Performance Engineering Framework" action items and provide a single point of entry for all interactions with the platform directly from Backstage.

To address the scheduling delays, completely separate load generation from the system under test, and have a way to stress all components via external endpoints, we had considered setting up an additional Kubernetes cluster dedicated to load testing. However, this would add significant fixed costs and maintenance overhead. At this point, we don't have a specific use case that would justify this effort, but we may revisit this idea in the future. A more efficient solution would be to have a dedicated pool of nodes for running load tests and introduce a synchronization checkpoint.

In this submission, we have presented our solution for a self-service performance testing platform for microservices. This is a scalable, fully integrated performance testing framework built from open-source components by a platform performance engineering team that has been widely adopted in a large engineering organization.

REFERENCES

[1] Argo. 2018. *Argo Workflows.* https://argo-workflows.readthedocs.io

[2] Backstage. 2020. *What is Backstage?* https://backstage.io/docs/overview/what-is-backstage

[3] Chris Baeckstrom. 2021. *Comparing K6, Gatling and JMeter.* https://www.redline13.com/blog/2022/09/comparing-k6-gatling-and-jmeter/

[4] Jeremy J. Carroll, Pankaj Anand, and David Guo. 2021. Preproduction Deploys: Cloud-Native Integration Testing. arXiv:2110.08588 [cs.NI]

[5] Adrian Cockcroft. 2016. *Microservices workshop.* https://www.slideshare.net/adriancockcroft/microservices-workshop-craft-conference

[6] Melvin E. Conway. 1968. How Do Committees Invent? *Datamation* (April 1968). http://www.melconway.com/research/committees.html

[7] Datadog. 2015. *DogStatsD.* https://docs.datadoghq.com/developers/dogstatsd/

[8] Gatling. 2013. *Gatling.* https://gatling.io/docs/gatling/

[9] Mohammad Hajjat, Ruiqi Liu, Yiyang Chang, T. S. Eugene Ng, and Sanjay Rao. 2015. Application-specific configuration selection in the cloud: Impact of provider policy and potential of systematic testing. In *2015 IEEE Conference on Computer Communications (INFOCOM).* 873–881. https://doi.org/10.1109/INFOCOM.2015.7218458

[10] Sen He, Glenna Manns, John Saunders, Wei Wang, Lori Pollock, and Mary Lou Soffa. 2019. A statistics-based performance testing methodology for cloud applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019).* Association for Computing Machinery, New York, NY, USA, 188–199. https://doi.org/10.1145/3338906.3338912

[11] Sören Henning and Wilhelm Hasselbring. 2021. Theodolite: Scalability Benchmarking of Distributed Stream Processing Engines in Microservice Architectures. *Big Data Research* 25 (July 2021), 100209. https://doi.org/10.1016/j.bdr.2021.100209

[12] Henrik Ingo and David Daly. 2020. Automated system performance testing at MongoDB. In *Proceedings of the workshop on Testing Database Systems (SIGMOD-/PODS '20).* ACM. https://doi.org/10.1145/3395032.3395323

[13] Kubernetes. 2020. *Kubernetes Documentation Concepts Workloads Workload Management Jobs.* https://kubernetes.io/docs/concepts/workloads/controllers/job/

[14] James Lewis and Martin Fowler. 2014. *Microservices.* https://martinfowler.com/articles/microservices.html

[15] Mark Leznik, Md Shahriar Iqbal, Igor Trubin, Arne Lochner, Pooyan Jamshidi, and André Bauer. 2022. Change Point Detection for MongoDB Time Series Performance Regression. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering* (Bejing, China) *(ICPE '22).* Association for Computing Machinery, New York, NY, USA, 45–48. https://doi.org/10.1145/3491204.3527488

[16] MongoDB. 2017. *Evergreen.* https://www.mongodb.com/blog/post/testing-linearizability-jepsen-evergreen-call-me-continuously

[17] Vishnu Murty. 2023. Distributed WorkLoad Generator for Performance & Load Testing Using Opensource Technologies. https://raw.githubusercontent.com/ltb2023/ltb2023.github.io/master/slides/LTB23_VMurty.pdf

[18] Alessandro Vittorio Papadopoulos, Laurens Versluis, André Bauer, Nikolas Herbst, Jóakim von Kistowski, Ahmed Ali-Eldin, Cristina L. Abad, José Nelson Amaral, Petr Tůma, and Alexandru Iosup. 2021. Methodological Principles for Reproducible Performance Evaluation in Cloud Computing. *IEEE Transactions on Software Engineering* 47, 8 (2021), 1528–1543. https://doi.org/10.1109/TSE.2019.2927908

[19] Matt Ranney. 2016. *What I Wish I Had Known Before Scaling Uber to 1000 Services.* https://www.youtube.com/watch?v=kb-m2fasdDY

[20] Tanya Reilly. 2022. *The Staff Engineer's Path: A Guide For Individual Contributors Navigating Growth and Change.* O'Reilly Media.

[21] M. Skelton, M. Pais, and R. Malan. 2019. *Team Topologies: Organizing Business and Technology Teams for Fast Flow.* IT Revolution. https://books.google.fi/books?id=oFdRuAEACAAJ

[22] Cindy Sridharan. 2018. *Testing in Production, the safe way.* https://copyconstruct.medium.com/testing-in-production-the-safe-way-18ca102d0ef1

[23] Rafael Winterhalter. 2014. *Why runtime code generation?* https://bytebuddy.net

[24] Ailin Yang. 2022. *"Noisy Neighbors" Problem in Kubernetes.* https://www.intel.com/content/www/us/en/developer/articles/technical/noisy-neighbors-problem-in-kubernetes.html