Annales Univ. Sci. Budapest., Sect. Comp. 54 (2023) 211–232

INTEGRATING PERFORMANCE TESTING INTO CONTINUOUS INTEGRATION LOOPS

Attila Kovács, Gábor Árpád Németh and Péter Sótér

(Budapest, Hungary)

Communicated by Péter Burcsi

(Received April 13, 2023; accepted May 18, 2023)

Abstract. Performance testing is a practice, a technique, a process used for testing the speed, stability, scalability and responsiveness of applications. In contrast to the usual approaches – that apply performance testing after functional testing and concentrate only on system-level tests – the authors discuss from a new perspective why and how performance testing can be integrated with other testing challenges (like functionality, security and usability), why and how it can be applied at all test levels in the life-cycle. The aim of the authors is twofold: (1) to revisit the evolution of performance testing and to clarify the sometimes contradictory glossaries, (2) to add a guideline on how performance testing can be embedded into the software testing life-cycle in all test levels and how they interact with various testing goals in order to provide faster feedback, which is crucial in methodologies that apply frequent deliveries (like Agile, Continuous Delivery, DevOps). The findings are illustrated with practical examples from telecommunications, web-related applications and other problem domains.

1. Introduction

Performance testing plays a vital role in software quality assurance. Large-scale software systems should serve thousands or millions of parallel requests and performance testing is used to determine how a system performs in terms of responsiveness and stability under a given workload.

Key words and phrases: Performance test, load test, load generation, performance measurement, continuous integration gating, test level.

2010 Mathematics Subject Classification: 68M15

The first and second authors were supported by the project "Software and Data-Intensive Services" Nr. 2019-1.3.1-KK-2019-00011 financed by the Hungarian National Institute of Science and Innovation.

In the last few decades, the increasing demand for performance forced engineers to develop new methodologies, processes and tools. However, most of the approaches handle the testing of performance separately from other testing goals and advise scheduling performance tests at the end, only at the system level, after other tests succeeded; see for example the highly cited load test survey article [24] that overviews around 200 papers of this topic. This approach results in two important drawbacks: (1) one may get late feedback on system performance, (2) the resources allocated for testing may be used inefficiently. As current software development trends (like Agile, Continuous Delivery, DevOps) indicate more and more frequent deliveries, these are serious issues. To cope with these problems, in our viewpoint performance testing is not a separate testing entity: it has strong connections to other testing challenges, such as functional, security and usability testing. In this paper, we investigate how these different goals can be integrated, for example, which assumptions need to be fulfilled to create tests that are able to scale between functional and performance testing properties. We also think that performance tests should be started as early as possible in the STLC (Software Testing Life Cycle). We think that performance tests are applicable not only at the system level, but one should also consider introducing smaller performance tests at lower levels as well. Although these tests may not discover as many performance issues as larger performance tests, they can be executed more frequently providing faster feedback loops, which is crucial in the case of frequent deliveries.

The body of the paper is organized as follows. Section 2 gives an overview of performance testing by summarising consistent terminologies for different performance testing types, collecting relevant performance metrics and discussing the different approaches used in load generation, performance monitoring, reporting and log analyses. Section 3 investigates how performance testing should take part in the STLC: we discuss the connection with other testing challenges, overview the scheduling of different performance test types, add a guideline about at which testing levels performance tests should be applied and investigate performance tests at given levels with different metrics. Section 4 gives a list of possible future trends in performance testing extrapolated from our findings and most recent state-of-the-art research papers. The main messages of the paper are summarized in Section 5.

2. An overview

In this subsection, we overview the main ingredients of performance testing. Before we start, for the sake of completeness, we list some widely accepted notions. A *use case* is associated with one or more services; it defines interaction models between one or more actors (users, other machines) and the SUT

(System Under Test). Multiple *test scenarios* can be associated with each use case. *Workload models* can be considered as a matrix of scenarios (transactions) versus frequency of execution spread across the number of concurrent users accessing the SUT simultaneously. The workload is sometimes called *operational profile*.

The process of performance testing should be applied iteratively using reviews at each well-defined engineering step. The main steps are the following: (1) *high-level analysis* – setting up the objectives, the main characteristics and the architecture of the SUT, eliciting the technical, business and operational risks; (2) *technical analysis and planning* by unfolding the key scenarios, environments, tools, configurations, the scheduling and monitoring methods with relevant metrics, determining the exit criteria; (3) *design* the variability and distribution of the load based on workload analysis, design and collect the test data and the performance metrics; (4) *implementation and configuration* of the load generator, the monitoring and analysis tools, the environment; (5) *running and monitoring* the tests, validating the results; (6) *analysing* the logs and results, *reporting* the recommendations, risks, costs and limitations.

In the following, we shortly summarize the types, metrics, and characteristics of the performance testing development frameworks.

2.1. Types

Performance testing is an umbrella term that consists of different testing types. In general, it focuses on the responsiveness, efficiency and stability of the SUT, but it also deals with resource consumption and hardware sizing problems (memory, CPU and disk usage, network bandwidth, etc.). The basic types of performance testing are (see Figure 1):

- *Load testing*: Used to understand the *behaviour* of the SUT under a specific expected load. It is usually applied in a controlled (laboratory) environment and its main role is to test the sustainability of the SUT.
- *Endurance testing*: It focuses on the *stability* of the system over a large predefined time frame. Endurance testing is used to verify whether the system is capable of handling the extended load or results in memory leaks, thread problems, database problems, etc. of the SUT. Other terms for endurance testing are *soak* or *stability testing*[1].
- *Stress testing*: Used to understand the upper limits of the capacity of the SUT. The tests are performed altering around the maximum designed capacity to investigate how the system works near this maximum (i.e. it checks if a relatively small overload on the short scale can be handled by scheduling, buffering, etc.).

[1]Note that it is sometimes also referred to as "reliability testing", but it is not identical to the reliability term of ISO 25000.

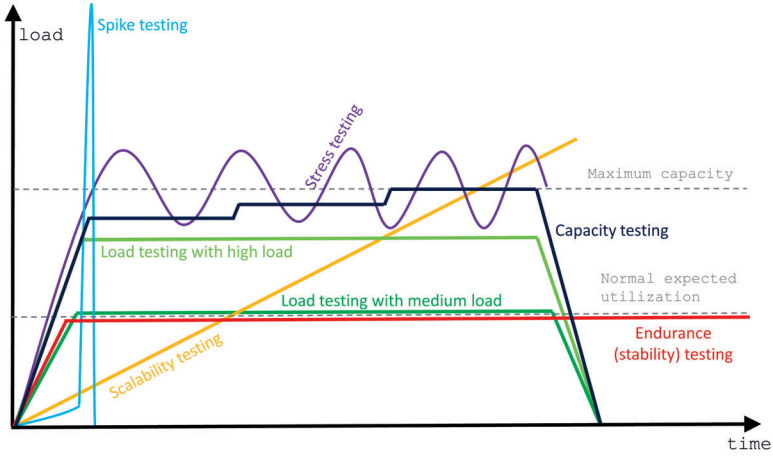

Figure 1. Different types of performance testing

- *Capacity testing*: Its role is to determine whether the SUT can manage the amount of workload it was designed for. In many cases it supports measuring the boundaries when they are not known in advance, it *benchmarks* the number of users or transactions that the system is able to handle in case of a given set of preconditions hold. The number of handled users or transitions found by capacity testing can be used as a *baseline* for later testing, i.e. to investigate how a given change affects the capacity of the SUT. Note that this baseline should be reconsidered periodically due to changes in the environments and preconditions. Capacity testing is also referred to as *volume* or *flood* testing as the volume of data is increased step-by-step (flooding) in order to analyse the actual capacity.
- *Spike testing*: Used to understand the functioning of the system if the load significantly exceeds the maximum designed capacity for a short time period. It investigates whether the SUT survives the sudden bursts of the requests, and if yes, then how it returns to its normal state (i.e. checks whether the system crashes, terminates gracefully or dismisses/delays the processing).
- *Scalability testing*: It shows how the SUT is capable of scaling up/out/ /down considering resources, like CPU, GPU, memory or network usage. Two different approaches are present in this investigation:

  (1) Gradually increase the load over a period of time to monitor the number of different types of resources used in the SUT,
  (2) Scaling up/out the resources of the SUT with the same level of load.

  Both techniques can be used to find possible bottlenecks in the system.

2.2. Metrics

In the following we briefly discuss the most important general performance-related metrics that can be applied virtually for all problem domains (and can be over-defined or refined if necessary):

1. Related to hardware utilization:
    - *CPU utilization*
    - *Memory utilization*
    - *Disk utilization*
    - *Network utilization*
2. Related to the characteristics of the tested system:
    - *Response time:* It can be investigated at different levels of the protocol stack. For example, in a telecommunication network, both neighbour participants and end-to-end participants can be considered; in a web application one may focus on the load of the requested page or if a requested transaction is done. It can further be subdivided into the worst, the best, the average, or the 90% percentile response time.
    - *Throughput rate*: It defines the number of requests processed per time unit. *Network throughput* is the rate of successful data delivery over a communication channel while *system throughput* is the rate of data delivery to all terminal nodes.
    - *Rate of successfully handled requests*: It is important to define what we consider under "successfully handled". This can be a given request handled successfully at first or x^{th} trying attempt or within a predefined time.
    - *Number of active sessions*: It defines the number of requests that the system can handle simultaneously.

2.3. Development frameworks

The functionality of the tools used in performance testing can be categorized into three main groups:

1. *Load generation*: Generate a given workload to the SUT using some preset or customized conditions or models.

2. *Performance monitoring and reporting*: During test execution, investigate some performance-related aspects of the SUT and may give an alert in case of suspicious or lower performance scenarios. In the end, a report is created based on monitored performance metrics.

3. *Log analysis*: After test execution, analyse and convert the existing logs into the desired format and may add an additional level of warnings and alerts to log data.

The tools used in practice may include the above-mentioned functionalities to different extents. Sometimes a single tool includes all the functionalities above, sometimes distinct tools are used. In the following, we mention some freely available tools from each category. We note that there exist numerous proprietary tools for each.

2.3.1. Load generation

Load generators create workloads for the SUT under various input conditions (such as the number of concurrent users, the frequency and distribution of requests, the type of requests, the different behaviours of users, etc.). The simulation or emulation of the load is achieved by creating *virtual users* (simulating or emulating the behaviour of the actors) that are distributed into load generators (see Figure 2). Note that besides virtual users, in some areas "real users" are used to generate a given load to the SUT. Also, note that the answers of the SUT may be looped back to the load generators themselves.

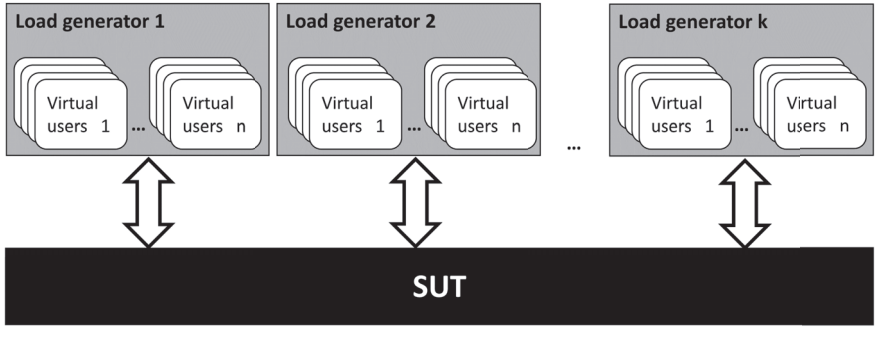

Figure 2. The general structure of load generation

Different solutions exist to describe the behaviour of these virtual users scaling from more simple solutions to more complex approaches:

(1) *Simple packet generators:* These tools create a discrete chunk of communication in a predefined format. Some data fields in the generated packages can be changed, but the same setting is used for the entire load. Due to their simplicity, they are easy to learn for the test engineer and they also provide the highest possible performance. The drawback of this approach is its inflexibility; only a few parameters can be fine-tuned and it is unable to handle alternative behaviours. For example, the Netstress [8] and MikroTik [7] tools are packet generators.

(2) *Traffic playback tools:* They are capable to play a given call flow back many times to generate the desired load. The call flow can either be edited manually or can be recorded. The drawback of this approach is the disability to handle alternative behaviours. Apache JMeter [1] is a traffic playback tool, but a small code that plays back a previously recorded Wireshark [9] trace with the given number of parallel threads can be also a suitable option.

(3) *Model-based generators:* They use formal models to describe the possible behaviour of the virtual users. Besides the normal call flow, alternate flows and exception flows are also considered. Various models can be used to *emulate* virtual users: EFSM (Extended Finite State Machines) [30, 35], Markov chains [15], Petri nets [16, 17, 42] PTAs (Probabilistic Timed Automata) [10, 32] and ETAs (Extended Timed Automata) [29] can be also considered. In [18] a TCFMM (Timed Communicating Finite Multistate Machine) model is proposed that is an extension of the EFSM model with tokens. Due to the complexity of the models used the generated load may be less than in the case of the previous approaches.

(4) *Generators with low level descriptions:* In this case, the given load generator is implemented directly. This approach can also handle alternative behaviours, but it requires programming skills both to develop and read tests. Maintainability could also be a problem, thus, this approach may be suitable only for the short term in the life-cycle. The lack of an abstract, high-level view can be a problem in the test design phase as well.

2.3.2. Performance monitoring and reporting

Performance monitoring and reporting functionalities show the test engineer the performance metrics that (s)he is interested in. Monitoring tools may also give an alert on lower performance conditions or suspicious behaviour. Please note that the results of performance testing usually can not be categorized simply as pass or fail. Multiple criteria need to be thoroughly investigated and evaluated. It is also important to emphasize that one measurement is not enough. Multiple performance tests are required to ensure consistent findings

due to problems related to hardware and frameworks used below the measured SUT. A *baseline* is also required for which the measured parameters can be compared. The reason for this is that the SUT is not completely independent from its environment (for example it can be connected to a real network, etc.).

2.3.3. Log analyses

One has to pay attention to collecting the measured data (e.g. various levels of logs) for further processing. As the number of tested transactions is high and the on-the-fly conversion of statistics would require too many resources that would decrease load generation performance, this conversion should be either done on a dedicated server or after the execution of tests. Note that depending on the testing purpose, the test engineer should set the appropriate level of logging. For example, in a capacity test of a real environment, the logging level should be turned into a minimum level to minimize its overhead, but if one investigates a root cause of a functional problem then the logging level should be raised. It is also important to emphasize that the detailed logging should be able to switch on or off for each system component independently providing the testability of the SUT without a significant drop in performance. Without this, one would not be able to reproduce problems that occur only with high load.

For the analyses of logs, different approaches can be used:

- *Search for regular expressions and create statistics based on them*: This approach can easily be adapted to the existing testing infrastructure and it requires low learning effort from the testing team. For example, one can parse the entire console log and collect the filtered parts by using a log parser plugin [6] in Jenkins [5] after job execution.

- *Create a custom solution for log conversions or data highlighting*: With proper classification, the afterlife of unsuccessful request attempts can also be tracked. However, this approach may be hard to adapt to the existing environment and it requires high learning effort from the testing team. The log collection can be done either during or after the test execution, the latter is proposed if no dedicated server exists for conversion. An example of this approach is Ericsson-JCAT [3] which contains logging-related extensions and several internal protocols used in telecommunication.

- *Using log manipulator applications*: The adaptation and learning costs are medium. The report is created on a dedicated server, thus the log collection can be done at any time. Kibana [4] is an example of a log manipulator application.

- *Mixed solutions*: Use some of the approaches above and add an extension for special needs. For example, a log manipulator application can be used when it is suitable for general log analyses, but one would also like to use some domain-specific analyses that require specific implementation.

3. Performance testing embedded into the STLC

3.1. Connection with other testing attributes

The performance of the system may have effects on some non-functional requirements, and thus, should be integrated organically into these other testing attributes:

Connection with functionality: Functional testing and performance testing can be applied together to show how the tested product handles different functionalities related to performance. In many cases, it makes sense to talk about the functionality of a product without investigating its performance. For example, if a webshop is unable to handle the given number of users simultaneously or a given node of a telecommunication network is unable to handle the given number of calls then they can be considered practically unusable. The integration of functional and performance tests is also important if one would like to see the root causes behind a given performance bottleneck. For example, if a given message was not handled successfully, it was lost, rejected for some reason (if yes, what was the root cause?) or just delayed? The key point in these cases is the capability of the system for parallel execution. If the functional tests are well-written then one can run several functional tests parallel with different data. Well-written means here that the parameters can be set independently, i.e. it is able to handle multiple transactions (the generated id should be present for each transaction). Note that this approach does not only have a benefit in performance testing, but it also makes functional testing more effective. A viable solution for harmonizing functional and performance tests was discussed in Section 2.3.1/(3), where various *models* were suggested for load generation. In this case, besides testing the performance of the SUT, the test engineer is also able to investigate its functionality with normal, alternate and exception flows. Note that the test engineer can balance and fine-tune between functional and performance testing: with more complex models (s)he can test more functionalities with less performance, while simpler models are able to test fewer functionalities with more generated load. Another advantage of this approach is that for a given model, systematic automatic test generation and test case selection algorithms can be used to find a proper balance between the tested functionalities and the complexity of the workload. Consider for example paper [28] where string edit distance-based test selection is used to reduce the size of the test set if the test cases of a telecommunication

software are given as different traversals of FSM (finite state machine) models. Later, this approach is extended for test case generation as well [27]. Note that this balance should also be set properly when logging is used: more detailed logging may help to identify the root cause of a functional problem but may cause a performance drop due to its overhead.

Connection with reliability: In safety-critical systems, the appropriate response time should be crucial. For example, if a given safety-critical message (related to brakes, steering, etc.) in the vehicle bus was not received in time, it may result in a serious accident. Hence, soft real-time and hard real-time systems may differ in how their performance is tested.

Connection with usability, availability and security: System performance has also effects on usability, availability and security which are key points in many application domains. For example, in a net-bank application, availability, usability and security are important conditions for the user, however, possibly with different risk levels. Network performance criteria can be also set in case of telecommunication traffic driven by QoE (Quality of Experience) [37]. All these aspects must have a clear focus on testing. Patch management and performance-related cyber threats, such as DoS (Denial of Service) attack, ransomware and crypto mining are parts of the IT operations that must be managed. For example, DoS attacks try to flood the target system with high traffic to make it inaccessible to its intended users, or even worse, to achieve a state of the system where the attacker can bypass the authentication method or can do transactions that would otherwise be prohibited. DoS attacks can also flood the server with valid service requests (SYN flood [36] for example), that may not be handled efficiently at the hardware level. In order to identify vulnerable points for DoS attacks one can apply e.g. spike tests. From an operation point of view, there is no distinction whether performance or security-related problem has occurred, both must be solved as soon as possible. This requirement projects the convergence of security and performance monitoring (and testing) in the near future. Performance metric parameters like response time may also have an effect on usability.

The integration of performance test with functional and with other non-functional tests raise some questions regarding process improvement methodologies as well. For example, in TMMi[2] [40] applying performance testing is only a supporting test practice within the process area "Non-functional testing". However, performance testing increasingly moves out of that approach.

3.2. Scheduling of performance tests

When the test engineer would like to determine how performance testing should be part of the STLC, (s)he must consider applying gating conditions in CI (Continuous Integration). The role of CI gating is to execute corresponding

[2]Test Maturity Model integration

test(s) after a code change, but before the code commit and to use the verdict of the test(s) to decide if the given code change can be allowed for commit or not. A precondition for gating tests is that these tests should be able to be executed within a reasonable time in order to give feedback to the developer about his/her code commit. For this reason, tests at lower levels are usually used as gating tests. In the case of performance testing lower level means mainly integration testing – for further details see Section 3.3.

Another important aspect is the scheduling of different types of performance tests. Load tests should be performed regularly, while the actual capacity with capacity test should only be measured once. Then, this capacity should be retested occasionally and re-calibrated if necessary. As endurance tests take significant time to be executed, they should be applied only at major milestones or they should be executed continuously in a dedicated server, where the execution of these tests is only stopped at major milestones for updating. The frequency of spike tests depends on how critical the actual system is[3]. For example, a telecommunication network, a net-bank application or even a webshop should take spike testing more seriously than a simple game application. Scalability tests should be applied at once and should be applied again if we would like to increase the performance of the SUT or if the environment has been changed.

Note that theoretically, all types of performance tests are applicable at all testing levels, but they may not have practical sense. For example, a load test is applicable at all levels, but there is no point in applying endurance tests at unit or at integration levels. The reason is that in the case of an endurance test, the test engineer would like to apply transaction numbers that are close to a real application, but in the case of unit or integration level tests, one could apply a load that has a multiple order of magnitude compared to the real case. Note that even the terminology of different types of performance tests is used only at the system or at user acceptance levels. At lower levels, we simply use the terms performance tests or load tests.

In many problem domains, there is a tendency to use methods to deliver more frequently (such as Agile, Continuous Delivery, DevOps), but the usual assumption about performance testing is that it takes a long time to be executed. But if one would apply shorter performance tests at lower levels, then they can be used more frequently. For example in a microservice architecture, performance tests can be adopted at lower levels. Note that many functional tests prepare the environment for test execution and these parts may be used for performance tests as well. If functional and performance tests would be integrated together, one could save a significant amount of time, that would be applicable for test execution.

[3]Note that what we consider critical does not necessarily mean that it is life-critical, it can be even "only" business critical.

3.3. Performance testing at different test levels

In many cases, it is assumed that performance testing can only take place after functional testing, at the end, for example in [22, 23] the authors propose to apply performance tests only at the latest quarter, Q4 stage, after the user acceptance testing. However, we think performance testing can take part of the testing process at any time, at any test level – the level of integration may depend on the application domain:

1) *Unit tests:* Corresponds to component tests in the ISTQB[4] terminology [21]. Performance tests at this level are not common; they are applied for performance-critical common components only (if it is known already at the design phase that these components are performance-critical components). Typically they are RestAPI calls that are used by other components, such as RestAPI in microservices, API calls used by GUI or fundamental queries of a database. The load generation at this level can be achieved either with simple packet generators or with traffic playback tools. The advantage of performance tests at this level is that one can focus on performance-critical components and can get feedback about performance in a reasonable time. The log analysis at the unit level is also simple. On the other hand, these performance tests can be too sensitive, i.e., they can show bottlenecks that are immeasurable and unimportant on higher levels[5]. For this reason, they can not be used for gating in CI. Note that it may not be easier to introduce a performance test at this level than at the integration or system level (but the implementation of a given test itself can be easier). The resource requirement of performance tests at the unit level is low, only one working component is required (for example a Docker [2] image). Performance testing at the unit level can be done simultaneously with security investigations, for example, to discover undesired effects such as memory allocation and deallocation anomalies, buffer overflow problems, etc. Some functional tests for microservices can be also executed together with performance investigations.

2) *Integration tests*[6]*:* Performance tests can be applied at this level if performance-critical sub-systems exist. Examples of the applicability of these tests can be multiple database queries due to the cooperation of different components (such as the connection of GUI and database) or the case when the OS (operating system) or the compiler of the SUT has been changed. For load generation traffic playback tools are advised. The ad-

[4]International Software Testing Qualifications Board

[5]The reason for this is that the given service can be tested with a higher load compared to the one that is possible if the real application uses this service.

[6]Note that in ISTQB foundation level syllabus [21] integration test is discussed as component integration testing and system integration testing.

vantage of performance tests at the integration level is that one can focus on performance-critical components, i.e., to identify bottlenecks. They are also good gating tests in CI and the log analysis is also relatively simple at this level. However, the tester needs to know before a higher level (system) test has been executed which parts (s)he needs to concentrate on, and at which parts bottlenecks may occur. Thus, one needs to measure at a higher level first, but after that performance tests at the integration level can be applied to reproduce the bottlenecks that were previously found. The resource requirement is higher than on the unit level because multiple working components are required (for example at least two Docker [2] images). Sometimes, at this level security aspects can be also investigated together with performance measurements such as checking the security of communication and the integration of third-party tools.

3) *System tests:* Performance tests at this level are the most common case; they are only known as "performance tests" for most of the testers. System performance tests test the entire application in an environment that is close to the real environment. The simulation of the load is achieved by creating virtual users that are distributed into load generators (see Section 2.3.1), based on the type of performance test (see Section 2.1). For load generation model-based generators or traffic playback tools are advised depending on the complexity of testing goals. Tests at the system level are used to investigate the capabilities and bottlenecks of the SUT. System performance tests have a possible connection with functional testing if they investigate the functionalities of the system with a given load. Examples are performance tests in a net-bank application where one can try different transaction types or performance tests in a smaller, sample telecommunication network with a preset traffic mix or traffic mix from "real" users. As the tester sees the entire system, tests can be designed based on functionalities known by developers and testers. On the downside, the focus can be shifted from the real usage to a theoretical usage of the system (as the tester/developer may not know the real importance of different features compared to each other); weights of use cases may not be the one required by the customer. To use appropriate weights, a "user acceptance test" level is required. Also, note that system performance tests typically require a long execution time, thus, they need to be shortened to use as gating conditions in CI. The resource requirement is high, as all working components are required. In addition to performance measurements, the entire product can undergo functional testing and usability investigations before being delivered to the customer. The testing process can also include reliability, availability, and security testing, which can be conducted simultaneously with performance testing.

4) *User acceptance tests:* The performance tests take place in the real environment. The tests are derived from a specific list of requirements given by the customer which describes the required levels of performance for given functionality mixes. If the product does not meet these requirements it can not be delivered. Model-based generators are advised to be used for load generation. In some cases, this level is used when the performance (virtualization, clusters with thousands of machines) of the customer's hardware environment is higher than the developer's or the customer can execute these enormous tests more frequently (that may be a real usage for him, but a performance test for the developers/testers of the product). Examples can be performance tests in a real telecommunication network with real users, or a distributed application used in a real environment. On the positive side, one can test what (s)he really needs with acceptance level performance tests, i.e. to test the performance of given functionalities with proper weights...etc. As the customer takes part in the testing process, the test engineer may get proper feedback. On the negative side, performance tests may be applied too late. The scheduling of testing may longer the delivery process, thus it may not be acceptable in methodologies that apply frequent deliveries. Applying performance testing at this level may be too expensive as special hardware may be required with enormous performance (a real environment or a system that can simulate or emulate the real environment). As the customer takes part in the testing process, NDA (non-disclosure agreement) may be required and if tests fail frequently it may be embarrassing in front of the customer. At the user acceptance level, performance testing is closely related to various other testing aspects. Usability testing takes precedence at this stage. The user environment is utilized to conduct reliability, availability, and security testing, with a focus on examining the product's installability and upgradeability. While functionality checks are carried out from the user's perspective, typically only a limited number of use cases are tested, and functional testing does not hold as much prominence as it does at the system level.

The main points are highlighted in Table 1.

Test level	Applicability, focus	CI gating	Resource requirements	Connection with other testing attributes
Unit	Focus on performance-critical components	Not applicable	Low, one working component	Security, functionality (microservices)
Integration	Focus on performance-critical sub-systems	Good gating test	Medium, multiple working components	Security
System	Investigate capabilities and botlenecks	Long execution time need to be shortened	High, all the working components	Functionality, usability, reliability, availability, security
Acceptance	Check customer's requirements in a real environment	Not applicable	Enormous, special hardware (real environment or emulation)	Usability, reliability, availability, security, functionality

Table 1. Performance testing at given test levels

To summarize, we would advise measuring performance as early as possible as performance testing is not only applicable to finished products. Note that there is a bigger risk of dependence on the surrounding environment at higher levels. For example in the automotive area, the measurement can be weather dependent due to the signal strength of the GPS unit, the quality (sharpness, brightness, contrast, saturation...etc.) of the video received from the camera unit ...etc. Note that nowadays demodularization trends – where big monolithic systems are split into smaller modules that work together – can also help to introduce performance testing at lower levels. The current software development trends (Agile, Continous Delivery, Devops...etc) clearly highly benefits from the shorter feedback loops introduced due to lower level performance tests.

3.4. The connection of test levels with different metrics

It is important to emphasize that the desired proportions of tests at different levels depend highly on the metrics the test engineer is interested in. Besides the number of tests, many other metrics are present. Figure 3 illustrates how a single performance test at a given level can be measured by different metrics. A possible metric is the number of tested lines of codes, where a unit performance test only covers a few lines, while a performance test at the system or acceptance level may cover a significant percentage of the code base. If one measures the number of covered functions then a simple unit and integration performance test cover essentially none of them. The reason for this is that a function is defined over a higher level than an API call (i.e. a function that

the customer can interact with), thus, function testing at these levels is not a goal. A system performance test covers one, while an acceptance performance test covers multiple functions, i.e. functional testing can be integrated with performance testing at these levels.

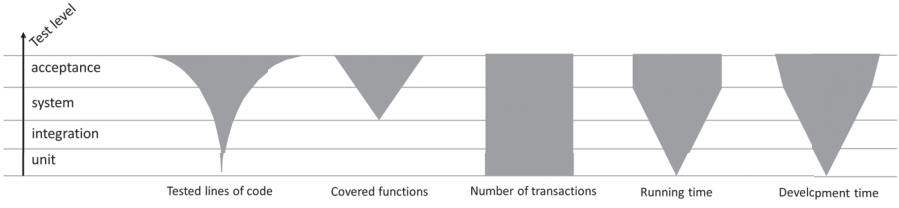

Figure 3. The effect of a simple performance test at various levels measured by different metrics

The number of transactions (i.e., atomic messages) may be the same at all levels, but in different distribution: on the unit level they are concentrated on the investigated component, on the integration level they are concentrated on a component and its surroundings, while at system or acceptance level they are distributed into the entire system. Thus, when one would like to select the right ratios of performance tests at different levels, (s)he must know how critical the entire system and an actual component are to select the desired level for performance testing. The running time of the unit level is very small, greater at the integration level and roughly the same at the system or at acceptance levels.

One can also consider the time that is required to develop tests at given levels: this is small at the unit level, higher at the integration level, and significant at the system level. At acceptance level it may be a little higher due to the environment, but not necessarily. Note that these are pure development times to write a given test, it does not include the time that is required to introduce performance tests at a given level. The different metrics listed above help the test engineer to focus on different aspects and select the right amount of tests at different levels considering costs and benefits that highly depend on the given domain and on the given tasks.

4. Possible future directions

We believe that in the near future, more and more solutions will exist that integrate performance testing with other testing goals, thus, more complex solutions – like model-based load generators – will be more dominant. We think that load generators that provide a dynamic workload to the SUT using some type of feedback from it will be more and more prominent as this approach may speed up the performance testing process, which allows us to

deliver the product more frequently. For example the method presented in [14, 20] first identifies workload sensitive input transitions, then it set load values for these identified transitions incrementally with constant monitoring of key performance indicators in order to find an appropriate load to the SUT (ie. to exercise as much as possible, but to avoid saturation). Another approach for speeding up the performance testing process is to simply terminate the given tests when they provide no further relevant information. The approach presented in [12] is based on periodically monitoring the combinations of given performance metrics, and if no new combination of these metrics is observed within a predefined time, then it stops execution. However, the number of monitored metrics and their applied combinations may cause a state explosion problem in the case of bigger systems and the fine-tuning of parameters triggering stopping conditions should be further investigated to find the appropriate trade-off between the length and coverage of test cases. We think that artificial intelligence will be also applicable in the near future in many problem domains to achieve at least some well-defined, simple goals connecting to load generation. For example, the methods presented in [19] and [11, 26] uses machine learning and reinforcement learning, respectively, to create the appropriate input conditions for the stress tests of different types of SUTs in order to find their performance breaking points and bottlenecks. Note that for similar problems, genetic algorithms can be also applicable [38].

In a large-scale system, even thousands of monitored performance parameters from different subsystems may exist that require significant efforts from the test engineer to handle. We think that in the future, approaches that automatically select an appropriate subset from these monitored performance parameters in order to give alerts about possible performance deviations in advance [33, 34] will be more and more prevalent. Note that similar solutions can be applied to logging as well; logs can be mined automatically to identify dominant behaviours and to flag deviations from these dominant behaviours [25, 39]. The paper presented in [31] extracts the logs from various workloads, and builds black-box performance models both to an earlier and to the current system versions detecting performance regressions automatically. Another interesting area is to determine the appropriate level of logging (i.e. to find the right balance between high load generation performance and the ability to find the root cause of the problems). In [41] a method is presented to suggest places where logging functions should be included. First, the method automatically inserts logging statements into various places to the source code and conducts performance tests. Then, using the results of the performance tests, statistical (linear regression) performance model is built. Finally, the method identifies the statistically significant performance-influencing logging statements in order to provide suggestions. We believe that in the near future dynamic logging will be a promising direction in which machine learning approaches change the

level of logs automatically during execution. In case of suspicious behaviour, or if a pattern is found that has resulted in problems in past executions, the level of logging can be raised automatically for the corresponding parts of the communication.

We also think that embedding performance testing into the cloud and other virtual testing environments will be more and more prevalent; this approach is especially suitable for microservice testing. Simple adding additional resources to bottleneck components is not enough, as the increased workload of these elements may result in performance degradation of its surroundings and thus in the entire system [13, 43]. To cope with this problem, the paper presented in [13] uses a quantitative approach. Based on operational profile data analysis, performance tests are generated. Then, the baseline requirements are calculated for each microservices to define pass-fail criteria based on the applied metrics and the generated performance tests are executed in each deployment alternative. In the end, the pass-fail verdict for each of the alternatives can be used to provide appropriate scaling between different microservices.

In virtualization the hardware equipment below the virtual environment are different; typically more devices – each of them with fewer resources – are used; thus interoperability between different resources will be even more pronounced: the entire system should be capable to scale up with smaller equipment. Note that after virtualization, different results will appear with the already applied metrics and one should focus on somehow converting between the old and the new results or one should concentrate to redesign the whole measurement; i.e. to introduce a new baseline, that should be accepted from the customer side as well.

5. Conclusion

In this paper, a consistent overview of terms and processes related to performance testing was presented. Different approaches and parameters of the tools used in performance testing were discussed including approaches that are able to emulate more complex behaviours by applying different models in order to integrate functional and performance testing. Guidelines have been given that describe how performance testing can be part of the software design life-cycle.

In our viewpoint performance testing is not a separate testing entity; it was investigated how performance testing can be connected to other testing challenges like functionality and security. The scheduling of different types of performance tests and the appropriate level of performance testing were also discussed.

The overview of different metrics for different performance test levels has highlighted the fact that there is no golden rule, just different aspects that should be considered in different domains with appropriate weights.

It was advised to run performance tests as early as possible. Although early testing is important, lower-level tests could be applied based on the results found at the earlier high-level testing. This feedback can be extremely useful for fine-tuning. In contrast to the usual assumptions of the industry claiming that performance testing takes a long time to be executed, it was advised to apply shorter performance tests at lower levels. The benefit of this approach is that these tests then can be executed more frequently, which makes them applicable to current software developing methodologies (such as Agile, Continuous Delivery, DevOps). As ShiftLeft continues (in which testing is performed earlier in the lifecycle) more and more development activities will be performed on performance testing. Performance tests at lower levels have also a benefit in testing microservice architectures. Of course, shorter performance tests have a benefit at other levels as well, they are more applicable in continuous integration gating. The authors believe that in the near future application monitoring and performance monitoring will converge: standard application monitoring tools will integrate more and more of the various metrics already present in performance monitoring tools.

References

[1] Apache JMeter. https://jmeter.apache.org/, Accessed: 2023-02-20
[2] Docker. https://docker.com Accessed: 2023-02-20
[3] Ericsson-JCAT https://github.com/Ericsson-JCAT Accessed: 2023-02-20
[4] Kibana. https://www.elastic.co/kibana Accessed: 2023-02-20
[5] Jenkins. https://plugins.jenkins.io/log-parser// Accessed: 2023-02-20
[6] Jenkins Log Parser plug-in. https://www.jenkins.io/ Accessed: 2023-02-20
[7] Mikrotik. https://wiki.mikrotik.com/ Accessed: 2023-02-20
[8] Netstress. http://nutsaboutnets.com/archives/netstress/ Accessed: 2023-02-20
[9] Wireshark. Network protocol analyzer. https://www.wireshark.org/ Accessed: 2023-02-20
[10] **Abbors, F., T. Ahmad, D. Truscan and I. Porres,** Model-based performance testing in the cloud using the mbpet tool, in: *Proceedings Of The 4th ACM/SPEC International Conference On Performance Engineering*, (2013), pp. 423–424.
[11] **Ahmad, T., A. Ashraf, D. Truscan and I. Porres,** Exploratory performance testing using reinforcement learning, in: *2019 45th Euromicro Conference On Software Engineering And Advanced Applications (SEAA)*, (2019), pp. 156–163.

[12] **AlGhamdi, H., C. Bezemer, W. Shang, A. Hassan and P. Flora,** Towards reducing the time needed for load testing, in: *Journal Of Software: Evolution And Process*, (2020).

[13] **Avritzer, A., V. Ferme, A. Janes, B. Russo, H. Schulz and A. Hoorn,** A quantitative approach for the assessment of microservice architecture deployment alternatives by automated performance testing, *Software Architecture*, (2018), 159–174.

[14] **Ayala-Rivera, V., M. Kaczmarski, J. Murphy, A. Darisa and A. Portillo-Dominguez,** One size does not fit all: In-test workload adaptation for performance testing of enterprise applications, in: *Proceedings Of The 2018 ACM/SPEC International Conference On Performance Engineering, ICPE 2018*, Berlin, Germany, April 09-13, 2018. pp. 211–222. https://doi.org/10.1145/3184407.3184418

[15] **Barros, M., J. Shiau, C. Shang, K. Gidewall, H. Shi, J. Forsmann,** Web services wind tunnel: On performance testing large-scale stateful web services, in: *37th Annual IEEE/IFIP International Conference On Dependable Systems And Networks (DSN'07)*, (2007), pp. 612–617.

[16] **Bause, F., H. Kabutz, P. Kemper and P. Kritzinger,** SDL and Petri net performance analysis of communicating systems, (1995).

[17] **El-Karaksy, M., A. Nouh and A. Al-Obaidan,** Performance analysis of timed Petri net models for communication protocols: A methodology and package, *Comput. Commun.*, **13(3)** (1990), 73–82.

[18] **Erős, L. and T. Csöndes,** An automatic performance testing method based on a formal model for communicating systems, in: *2010 IEEE 18th International Workshop On Quality Of Service (IWQoS)*, (2010), pp. 1–5.

[19] **Helali Moghadam, M., M. Saadatmand, M. Borg, M. Bohlin and B. Lisper,** Machine learning to guide performance testing: An autonomous test framework, in: *2019 IEEE International Conference On Software Testing, Verification And Validation Workshops (ICSTW)*, (2019), pp. 164–167.

[20] **Huerta-Guevara, O., V. Ayala-Rivera, L. Murphy, and A. Portillo-Dominguez,** Towards an efficient performance testing through dynamic workload adaptation, in: *Testing Software And Systems - 31st IFIP WG 6.1 International Conference, ICTSS 2019*, Paris, France, October 15-17, 2019, Proceedings, **11812** pp. 215-233 (2019). https://doi.org/10.1007/978-3-030-31280-0

[21] International Software Testing Qualifications Board (ISTQB) Certified Tester, Foundation Level Syllabus V4.0. (2023), https://istqb-main-web-prod.s3.amazonaws.com/media/documents/ISTQB_CTFL_Syllabus-v4.0.pdf

[22] **Janet Gregory, L.,** *Agile Testing: A Practical Guide for Testers and Agile Teams*, Addison-Wesley Educational Publishers Inc., (2009).

[23] **Janet Gregory, L.,** *Agile Testing Condensed: A Brief Introduction*, (2019).

[24] **Jiang, Z. and A. Hassan,** A survey on load testing of large-scale software systems, *IEEE Transactions On Software Engineering.* **41** (2015), 1091–1118.

[25] **Jiang, Z., A. Hassan, G. Hamann and P. Flora,** Automatic identification of load testing problems, in: *2008 IEEE International Conference On Software Maintenance*, 2008, pp. 307–316.

[26] **Koo, J., C. Saumya, M. Kulkarni and S. Bagchi,** PySE: Automatic worst-case test generation by reinforcement learning, in: *2019 12th IEEE Conference On Software Testing, Validation And Verification (ICST)*, 2019, pp. 136–147.

[27] **Kovács, G., G. Németh, M. Subramaniam and Z. Pap,** Optimal string edit distance based test suite reduction for SDL specifications, in: *Proceedings Of The 14th International SDL Conference On Design For Motes And Mobiles*, 2009, pp. 82–97.

[28] **Kovács, G., G. Németh, Z. Pap and M. Subramaniam,** Deriving compact test suites for telecommunication software using distance metrics, *Journal Of Communications Software And Systems (JCOMSS)*, **5** (2009), 57–61.

[29] **Krichen, M., A. Maãlej and M. Lahami,** A model-based approach to combine conformance and load tests: An EHealth case study, *Int. J. Crit. Comput.-Based Syst.*, **8** (2018), 282–310.

[30] **Krishnamurthy, D., M. Shams and B. Far,** A model-based performance testing toolset for web applications, *Engineering Letters*, **18** (2010), 92–106.

[31] **Liao, L., J. Chen, H. Li, Y. Zeng, W. Shang, J. Guo, C. Sporea, A. Toma and S. Sajedi,** Using black-box performance models to detect performance regressions under varying workloads: an empirical study, *Empir. Softw. Eng.*, **25** (2020), 4130–4160, https://doi.org/10.1007/s10664-020-09866-z

[32] **Maãlej, A., M. Hamza, M. Krichen and M. Jmaïel,** Automated significant load testing for WS-BPEL compositions, in: *2013 IEEE Sixth International Conference On Software Testing, Verification And Validation Workshops*, 2013, pp. 144–153.

[33] **Malik, H., H. Hemmati and A. Hassan,** Automatic detection of performance deviations in the load testing of large scale systems, in: *2013 35th International Conference On Software Engineering (ICSE)*, 2013, pp. 1012–1021.

[34] **Malik, H. and E. Shakshuki,** Performance evaluation of counter selection techniques to detect discontinuity in large-scale-systems, *J. Ambient Intell. Humaniz. Comput.*, **9** (2018), 43–59.

[35] **Németh, G.,** A finite state machine-based description in performance testing, *4th User Conference On Advanced Automated Testing (UCAAT)*, 2016.

[36] **Raed Bani-Hani, Z.,** SYN flooding attacks and countermeasures: A survey, *15th International Conference On Information And Communications Security)*, 2013.

[37] **Ramadža, I., V. Pekić and J. Ožegović,**, Network performance criteria for telecommunication traffic types driven by quality of experience, *Journal Of Communications Software And Systems (JCOMSS)*, **15** (2015), 233-244.

[38] **Shen, D., Q. Luo, D. Poshyvanyk and M. Grechanik,** Automating performance bottleneck detection using search-based application profiling, in: *Proceedings Of The 2015 International Symposium On Software Testing And Analysis*, 2015, pp. 270–281.

[39] **Syer, M., Z. Jiang, M. Nagappan, A. Hassan, M. Nasser and P. Flora,** Leveraging performance counters and execution logs to diagnose memory-related performance issues, in: *2013 IEEE International Conference On Software Maintenance*, 2013, pp. 110–119.

[40] Test Maturity Model integration (TMMi) Guidelines for Test Process Improvement, 2018, https://www.tmmi.org/tmmi-model/

[41] **Yao, K., G. Pádua, W. Shang, S. Sporea, A. Toma S. Sajedi,** Log4Perf: Suggesting logging locations for web-based systems' performance monitoring, in: *Proceedings Of The 2018 ACM/SPEC International Conference On Performance Engineering*, 2018, pp. 127–138, https://doi.org/10.1145/3184407.3184416

[42] **Youness, O., W. El-Kilani and W. El-Wahed,** A behavior and delay equivalent petri net model for performance evaluation of communication protocols, *Computer Communications*, **31** (2008), 2210–2230.

[43] **Arvidsson, Å. and L. Westberg,** Transport bottlenecks of eedge computing in 5G networks, *Journal Of Communications Software And Systems (JCOMSS)*, **15** (2019), 59–65.

A. Kovács and G. Á. Németh
Department of Computer Algebra
Eötvös Loránd University
H-1117 Budapest
Hungary
attila.kovacs@inf.elte.hu
nga@inf.elte.hu

P. Sótér
sotya21@gmail.com