Datenbank Spektrum (2022) 22:195–215
https://doi.org/10.1007/s13222-022-00426-x

SCHWERPUNKTBEITRAG

Testing Very Large Database Management Systems: The Case of SAP HANA

Thomas Bach[1,2] · Artur Andrzejak[2] · Changyun Seo[1] · Christian Bierstedt[1] · Christian Lemke[1] · Daniel Ritter[1] · Dong Won Hwang[1] · Erda Sheshi[1] · Felix Schabernack[1] · Frank Renkes[1] · Gordon Gaumnitz[1] · Jakob Martens[1] · Lars Hoemke[1] · Michael Felderer[3] · Michael Rudolf[1] · Neetha Jambigi[3] · Norman May[1] · Robin Joy[1] · Ruben Scheja[1] · Sascha Schwedes[1] · Sebastian Seibel[1] · Sebastian Seifert[1] · Stefan Haas[1] · Stephan Kraft[1] · Thomas Kroll[1] · Tobias Scheuer[1] · Wolfgang Lehner[4]

Received: 2 June 2022 / Accepted: 8 October 2022 / Published online: 24 November 2022

Abstract
Software Testing is an established activity in the software development process to ensure and improve the quality of a software. Consequently, there exists a wide range of literature, popular information, and even multiple ISO standards covering this topic. However, we found that testing very large database management systems (DBMS) requires special adaptations of the generally available guidance for software testing and requires to solve specific challenges that may not be relevant for other areas or smaller software projects. We therefore discuss the testing of SAP HANA, a very large software project with millions of lines of code, to share insights about our approach, best practices, and unsolved challenges that are open for further research.

Keywords DBMS · Testing · Software Quality · SAP HANA

1 Introduction

Today, software testing is an integral part of the software development process [43, 44, 81] to gain information about the quality of the software under test and to subsequently improve it if needed. Software testing is motivated by the experience that software programs can contain defects. These defects, if they are encountered during software execution (*failure*), can lead to undesirable results ranging from unimportant details to hundreds of million dollars in costs or even loss of life [40]. Loss of trust from users also translates to financial costs. Therefore, avoiding such costs is a strong motivation for software testing.

Generally speaking, the costs of defects may increase if costs of software testing decrease and vice-versa. Therefore, there is a break-even point between both costs. The existence of such a break-even point motivates that the testing of very large database systems (DBMS) such as SAP HANA is of special interest. The high costs of defects imply high effort for testing. The high costs of defects are caused by the purpose of the system and the size of the software project. First, a DBMS is typically the core platform for data handling and must store and retrieve data correctly. Data might be the most valuable asset of a user and defects in a DBMS can cause severe costs. Second, the size of a very large software project typically corresponds to its requirements. A very large project translates to a large number of users that use it in complex scenarios. This again makes defects costly as they might affect a large number of users. In summary, for a very large DBMS, the costs of defects are high and it follows, as reasoned by the concept of a break-even point with an economic approach to maximize the overall gains, that the expected efforts for software testing are also high.

In addition to the cost argument, DBMS, even more DBMS for enterprise scenarios, are typically complex software projects that need to fulfill a wide range of requirements. DBMS may even provide core functionality of an operating system [13] and therefore reach a similar com-

All authors contributed to the work in specific subsections and are ordered by first name. Except for the first author who is the corresponding author and contributed the general sections, combined and edited all parts, and filled missing gaps.

✉ Thomas Bach
thomas.bach03@sap.com

Extended author information available on the last page of the article

plexity as an operating system. In addition, as performance is important, the design of a DBMS may trade performance over simplicity. In the case of our study subject SAP HANA, the complexity is already represented by the source code size of about 40 million lines of code and the history that partly spans over several decades of development.

The high effort for software testing and the high complexity of a DBMS motivate that studying the software testing of a very large DBMS can provide additional information compared to other software types. In our work, we investigate the testing of SAP HANA, a DBMS targeted for enterprise scenarios by SAP. Similar work is available for other DBMS such as SQLite [39, 86], Oracle database [2, 16], MongoDB [18, 19, 41], PostgreSQL [68], or MySQL [58] and for other large projects from Google [36, 57, 87] or Meta [54].

The contributions of our work are:

1. A description of how SAP HANA is tested.
2. Solutions to several specific challenges in the context of testing a very large DBMS.
3. A set of open challenges with practical relevance for testing very large projects.

The remainder of this work starts with a description of SAP HANA and general concepts in Sect. 2 and, as testing is a complex topic, is structured according to multiple dimensions. Sect. 3 contains the people perspective and Sect. 4 presents the process dimension from the time a change is created and contributed to the source code repository (*pre-submit*) until it is released (*post-submit*). Sect. 5 describes continuous test activities that extend the development process. Sect. 6 focuses on challenges of testing SAP HANA Cloud. Sect. 7 lists open challenges, and we conclude with Sect. 8.

2 Overview

SAP HANA is an in-memory database management system targeted for enterprise scenarios [28, 29, 56]. It is available as an on-premises version and as *SAP HANA Cloud*, a database-as-a-service (DBaaS) offer comprising managed hardware environments and a wide range of other services. Depending on the context, the term SAP HANA Cloud might only refer to the database part.

We first provide numbers to reason why SAP HANA is a very large and complex project. Then, we describe the historical development of testing and quality for SAP HANA and finally, we describe testing and quality for SAP HANA as of today.

2.1 The Size of a Very Large Project

The number of lines for the source code provide a very general metric to represent the size of a software project. Measured with cloc [20], the source code repository for SAP HANA consists of 36 million code lines, see Table 1 for comparison to other projects. The source code is distributed over about 200 components where each component represents a separate part of the software project. The source code consists mainly of C++ and C code, but also includes Assembly. The test code mainly contains C++ and Python code, but also other programming languages, XML and JSON configuration files.

The source code is versioned via git [32]. The repository size is about 30 GiB and contains the history of several sub-projects up to the year 2000. All older history is contained in a different version control system. Per day, the repository is modified by over 800 source code changes (*commits*) to multiple development branches and about 80 commits are integrated in the main code lines. The number of developers contributing to SAP HANA depends on how each committer is accounted for, but there are over 100 people contributing regularly to SAP HANA. Since the year 2000, there are about 3000 different authors that contributed 1.4 million commits in about 1000 branches to the source code repository of SAP HANA and predecessors.

The development of SAP HANA creates per year more than 10 TiB of metadata such as build information, test results or development progress information. The data is partly stored in and analyzed by an internal SAP HANA database instance (Sect. 5.6). Still, maintaining accurate and consistent metadata is challenging for a large project.

For the context of this work, additional numbers about testing are of interest. Counting the total number of tests is a complex task. It is debatable how tests with multiple parameters should be counted or how often the same tests for multiple product versions should be counted. In addition,

Table 1 Sizes of multiple large projects as of 2022-05. Lines of code (LoC) measured with cloc [20]. Size, commits and authors based on github git repositories

Name	LoC 10^6	GiB	Commits 10^3	Authors
SAP HANA	36.1	29.6	1368	3082
MySQL	3.8	4.7	172	1685
Postgres	1.6	0.8	82	55
SQLite	0.3	0.4	87	38
FreeBSD	17.1	3.8	897	3358
Linux	23.4	5.2	1090	32486
Chromium	32.2	33.4	1302	12432
CERN ROOT	5.2	1.4	87	488
Firefox	29.3	7.1	893	10437
LibreOffice	9.3	5.6	559	2505

techniques such as fuzz testing create a variable number of tests depending on settings like available time or stopping criteria. Counting all tests from pre-submit testing to the final product release for a specific change for a specific product variant results in about 900 000 tests overall. Some of these tests are executed multiple times per day, yielding 3 million test executions per day [4, 5, 7]. The tests are executed on a dedicated infrastructure that consists of about 1000 servers with (on average) 40 CPU cores (3 GHz frequency), and 256 GB of RAM. Executed sequentially, the total execution of all automated tests would require up to 4 weeks. Executed in parallel, the execution time is typically between 30 min and 10 h. The variance can be attributed to different test types, required execution resources, requirement and availability of special hardware, cluster load factor, or test configuration. In addition, effects such as flakiness (Sect. 7) that result in restarts may increase the test execution time. For comparison, Google and Facebook report 150 [57] and 10 million [54] tests executed per day, respectively.

2.2 Historical Development

Common quality standards: The first release of SAP HANA in 2010 included several technologies that already existed before. These previous projects had their own working style, source code organization, and also their own approach to testing. Integrating all of these projects was the main challenge for the quality team in 2010. A common quality standard had to be established for the combined project SAP HANA that still preserves the individual best practices. While several previous projects had a strong focus on a specific test type, the combination formed over time a test pyramid that is discussed in Sect. 4 and shown by Fig. 2.

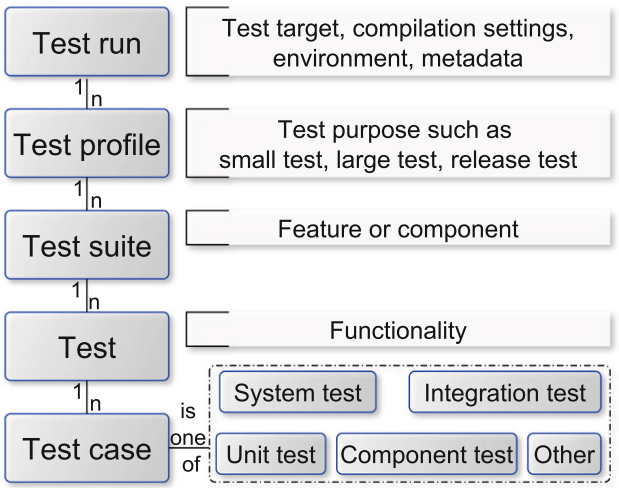

Fig. 1 Test deployment hierarchy

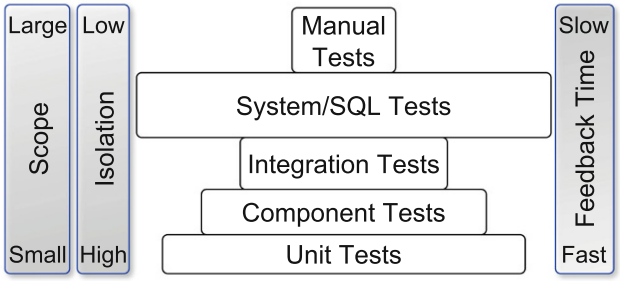

Fig. 2 DBMS variant of the test pyramid [15]. The block size of each layer represents focus during development

Distributed teams: The integration started by combining existing tests into a common test driver. However, executing the common test set then required loading several gigabytes of data from network stores. This was in the year 2010 a challenge for the distributed team setup across several locations in Germany (Walldorf, Berlin) and Korea (Seoul). The test execution time increased due to the amount of data transferred over the network. Consequently, test data sizes were reduced by replication and deduplication. These approaches are still important today with central test executions to reduce overall test execution times.

Central pre-submit testing: Once all previous projects were unified in a single source code repository, the increased frequency of distributed changes showed structural problems for the development process. Developers were used to execute the full test suite on their local development machines and then submit their code. This resulted in complex setup instructions and long waiting times for test executions. This consequently negatively affected the frequency of how often developers executed tests and by that also negatively affected the quality of most recent source code state. Therefore, central pre-submit testing was introduced. Changes had to pass a set of tests before integration into the central code repository. Over time, this resulted in the testing stages as shown in Fig. 3

Test execution framework: Over time, scalability issues affected testing. Test executions slowed down due to limited hardware resources for test executions, but also due to limitations in bandwidth and available disk space to transfer and store increasing sizes of project artifacts. The underlying framework for test execution was based on Jenkins [79], which, at least at that time, did not scale with the requirements of such a large project. Therefore, SAP developed a framework to define, schedule, distribute, and execute tests with fine-grained resource control adapted to the environment at SAP.

Customer first: In addition to project-internal testing, it was also important from the very beginning that SAP HANA correctly works with applications that utilize SAP HANA. For example, the development and testing of SAP

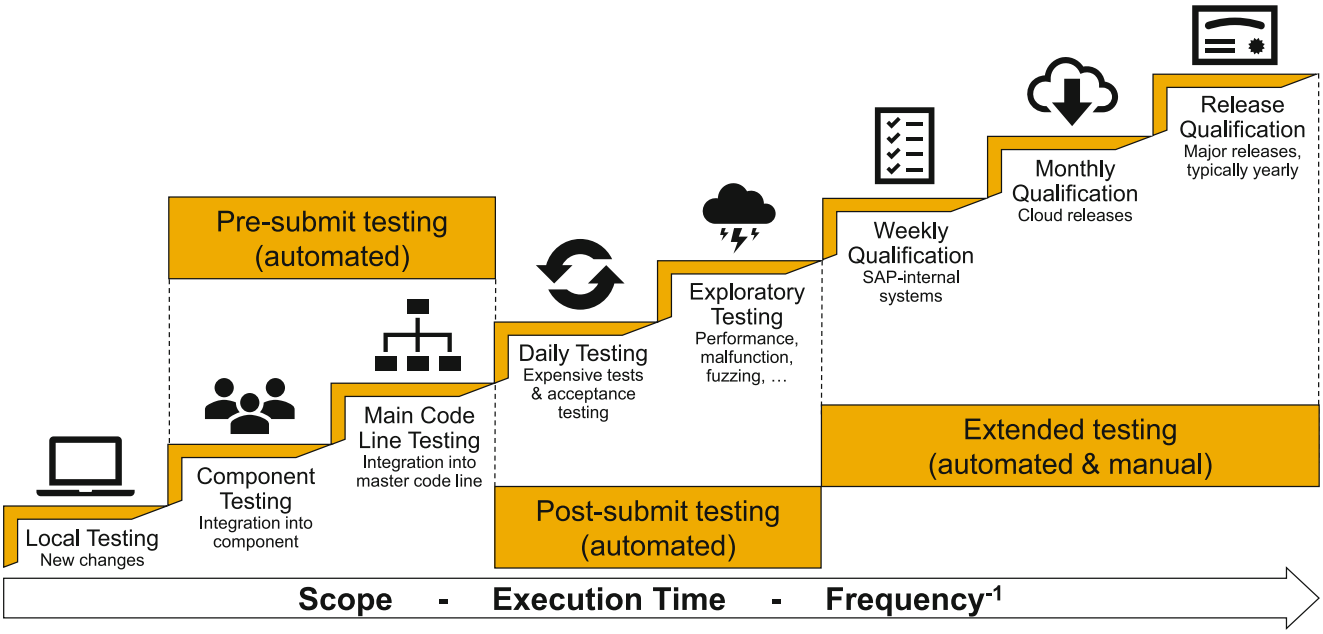

Fig. 3 Testing stages hierarchy of SAP HANA. From left to right, scope and execution time increase, frequency decreases. Sects. 3.2 and 4 cover pre-/post-submit testing. Sect. 5 covers continuous test activities in post-submit/extended testing

Business Warehouse [67] was interleaved with SAP HANA development that supported the interaction between both software projects if they were deployed together.

Superlinear increase in test costs: Over time, test costs became an important issue as they increased superlinearly [7]. SAP applied two fundamental strategies to reduce test costs but to keep a high degree of quality and development velocity. First, a layered test strategy reduces the average test execution frequency and therefore reduces the costs. Second, cross-function teams take care of general test result assessment and optimizing the general test infrastructure.

Outlook: Some challenges continue to exist. The ever-increasing complexity of SAP HANA itself and the (cloud-) environment it runs on, decreasing the time between an issue is reported and fixed, cases where testing obstructs development instead of supporting it, avoiding test flakiness, or test cost reduction remain to be challenges to this day.

2.3 Quality

Quality refers to "the degree to which a set of inherent characteristics fulfills requirements" [45]. Consequently, software quality refers to "(the) degree to which a software product satisfies stated and implied needs when used under specified conditions" [43]. In our experience, most requirements and needs are implicit. Two typical cases are stability and performance. For stability, the user expects to use a database instance for a long term even if a continuously increasing amount of data is stored. Obviously, this is not compatible with hardware limitations that the software cannot change. For performance, users expect answers from the database in a certain, often undefined amount of time, even if the data or retrieval queries change. Managing such implicit requirements with respect to quality is challenging.

Therefore, over decades, SAP has established a practical and customer-first handling of quality and encoded it in internal standards and guidelines. First, SAP identified the most important requirements for users of SAP HANA and invests into reaching a high quality there, i.e., a high degree of satisfaction. Second, the database is highly configurable to allow users mitigation of a subset of potential issues without waiting for changes to the software product itself. Lastly, SAP offers a multi-level support to provide tailored and effective solution to potential issues. These aspects are then supported by extensive SAP-internal testing as described by this work.

The most important requirement for users is stability. A DBMS manages data which is a core value for many businesses today. It may have direct negative consequences on the business if the DBMS does not provide the expected services. The most significant failure causes are crashes, out-of-memory situations, regressions, security issues, data loss, data corruption, wrong results, or severe performance degradation. A crash does not only abort all current operations, it can also result in considerable waiting times for restarts and restore operations. Out of memory situations can trigger crashes but can also result in cases where a DBMS refuses additional requests. A regression in a version v_{n+1} exists if v_{n+1} does not support functionality F

supported by v_n and v_{n+1+m} supports F again. If all versions $\geq v_{n+1}$ do not support F, then it is an intended incompatible change. SAP aims to avoid regressions, as they may negatively impact the stability from the view of a user. The effect of security issues depends on the threat vector (see Sect. 5.10). Finally, data loss, data corruption, wrong results, and severe performance degradation can have a negative impact, but depending on the use case of the user, they may not be noticed or may even be ignored.

SAP implements testing, prevention and mitigation strategies for all these failure causes (Sects. 4 and 5). Thus, testing is part of a plan-do-check-act (PDCA) cycle [22] for continuous improvements.

2.4 Test Dimensions

The test organization of SAP HANA may not follow typical terms of other projects due to historical reasons and the size of the project. In related work to software testing, a test suite is often the top-level category that contains all tests. SAP HANA as a large and highly complex project consists of several smaller projects (*components*). Therefore, terms such as test suite are then also used on a sub-level and are therefore not suitable to use anymore on an overall level. Fig. 1 shows how different terms connect to each other for SAP HANA as they have been established over time.

Providing an overview over all test activities is a challenging task because of the large set of testing activities for the development of SAP HANA. Therefore, we provide a 3-dimensional view on testing. First, a people dimension that shows the perspective of people working on SAP HANA (Sect. 3). Second, a temporal or process dimension from creation until delivery of a change (Sect. 4). Third, a continuous dimension with activities that do not fully match the other dimensions (Sect. 5).

Within the people dimension, from the perspective of a developer, a test pyramid represents a guidance for how tests should be implemented [15]. A special property of the test pyramid for a DBMS is that the pyramid is not strictly a pyramid as shown by Fig. 2 for SAP HANA. The system test layer contains a comparatively large amount of tests based on SQL-statements as SQL is the major communication language for DBMS.

Regarding the temporal dimension of a change, the overview is more aligned to a process and contains more steps compared to what one could conclude from the testing pyramid. Fig. 3 shows the testing stages hierarchy of SAP HANA that can be roughly separated to pre-submit and post-submit testing that represent activities before and after a change is included in the central code repository. Finally, the extended testing represents activities towards a release. Although continuous deployment is a desirable goal, it is in practice currently not fully feasible due to high costs and duration of the activities required for a release.

2.5 Bug Handling

We utilize for SAP HANA a bug handling system to manage the lifecycle of bugs where a bug has a $1 : n$ relationship to product versions. Each bug-fixing change must reference the corresponding bug. Automated tools check the existence of such a reference. Thus, the mapping between bugs and bug-fixing commits is accurate. SAP then utilizes an internal instance of SAP HANA (see Sect. 5.6) and the graph based data retrieval functionality of SAP HANA to implement extended tooling about the git repository of SAP HANA, the state of bugs and the corresponding fixes. A side effect of this mapping (and other factors such as the motivation to avoid unnecessary complexity or more work) is that bug fixes typically only contain bug fixing code and no other unrelated changes (tangling is a problem reported for open source projects [37]).

The available tools then also allow us to efficiently handle the maintenance of multiple product versions without introducing regressions. A user should never "loose" a bug fix or feature after an upgrade from version v_i to a later version v_{i+n}. This requirement is simple to fulfill if development is linear: The set of all changes in v_{i+n} (ignoring reverts) is a super set of the set of changes in v_i if development is linear. However, the linearity does not hold if multiple versions are maintained in parallel. We then must ensure that a new patch release for an older product version contains only changes that are also included in all product versions that can be a target version for an upgrade. To accomplish this, each (major) product version has its own git branch. We then define for each such branch a set of required branches where a change must be included first. This leads to a directed, acyclic graph of requirements with the most current state as the root node. Our tools then check for each change whether the required branch is fulfilled. With that, our tooling prevents regressions caused by missing changes.

The bug handling system also provides a common and central tool to collect and track all issues indicated by any other tool. As shown in Sect. 5, there are several dozens of tools that report issues where SAP HANA developers must act. Working with so many tools does not scale if every developer must access and understand each tool as this requires access handling, regular training and will still lead to confusion if developers only infrequently access some tools. Tracking the progress in all these tools for all developers is a complex task, too. To avoid such an $n : m$ relationship, we utilize the bug handling tool in a similar way as in a star topology. Developers primarily interact with the bug handling tool and all additional tools report

into the bug management tool. With that, developers do not have to interact with different tools anymore and we have an $n : 1 : m$ relationship that scales practically. If possible, we provide information to developers in their development environment and as comments for changes towards the central code repository to further simplify interactions.

A final observation over time is that bugs should have a single attribute that precisely defines the urgency or importance. There are recurring discussions to provide more dimensions such as value, cost, impact, effort. However, a developer can only work at one item and hence needs a one-dimensional list of sorted items to work on. Therefore, even if there were multiple attributes, they would have to be mapped again to a single dimension. Conclusively, these attributes are redundant and useless over time and maintaining them would be a waste of time and resources.

2.6 Risk-Based Quality Assurance

Risk-based quality assurance uses risk assessment to steer the application of quality assurance (QA) techniques [30]. We apply it within SAP HANA to enable both efficient QA to control and distribute resources to QA activities as well as effective QA to avoid, manage or remove critical issues as early as possible in the development lifecycle. For instance, we use risk-based QA to select and prioritize test activities for each change to the software. With that, we can reduce test costs without negatively impacting the quality of the software.

Exploiting the potential of risk-based quality assurance requires a proper risk assessment procedure via identification of risk items and the analysis of risks. For SAP HANA, an example are software defects. They are important risk items that also steer regression testing activities. Assessing the risk of a software defect manually via its report is an expensive task and the assessment results are unreliable. Therefore, we apply, among others, machine learning (ML) techniques (support vector machines, decision trees) to automate the risk assessment based on available data about bugs [51]. Similarly, we use ML-based approaches to classify issues for root cause analysis while replaying recorded workload data [47].

3 People Perspective

We show the perspective on testing from different groups of persons based on interviews and experiences of the authors. We start with a general description of the quality culture. Then, we show the view of developers, architects and management.

3.1 Quality Culture

Quality in software products often focuses on technical aspects, but the software product is created by persons. Overall, quality is not "tested into a product", but must be integrated into the whole development process that starts with the people. Leaders need to understand and communicate the value of quality and testing. Engineers must buy into the quality strategy. Therefore, we first describe how the quality culture develops and we then describe a key activity that gives engineers time to improve the quality.

Quality culture development: This requires active participation in shaping the quality strategy, exchange on challenges, and collaboration across team boundaries. A key component is to feature a quality leader. This is not the person who is responsible for testing or quality, but someone who advocates for quality by sharing information about quality, best practices, and development and testing experience. To foster a quality culture and a common understanding of what quality is and how it is achieved, there is a wide range of activities for exchange about software quality. Examples start from mandatory code reviews, static code analysis, presentations on detection of memory leaks, deadlocks, or optimization issues, over process simplifications, "How we test!" meeting series, post-mortem analysis, to developer efficiency, tools, and training courses. Such knowledge sharing is also frequently repeated or, if possible, the findings are encoded via automatic tooling, as any large project has a considerable churn of new employees.

Root cause analysis: One important process to reserve time for developers towards improvements are root cause analysis (RCA). An RCA is initiated by a set of pre-defined triggers such as outages, crashes, and regressions. These triggers are aligned with the main quality goals. Therefore, the additional time that developers spend on RCA is well-invested. In addition, an RCA can be requested on a per need basis for larger issues. The goal of an RCA is first to identify root causes of an issue and share knowledge about them. Second, an RCA should always define improvement actions (IA). The IA then aim to either prevent the issue from happening in the future or reduce the impact. Three key observations for the RCA process are that they must be conducted without blaming, in a timely fashion, and IA must be decided and planned timely. First, conducting RCA must be an open discussion process for finding technical limitations, it is not about blaming specific developers, and individuals are typically not the only root cause of an issue. If people are blamed, they will apply defensive behavior that is not compatible with a root cause analysis. Second, RCA must be conducted close to the incident. The larger the time delay between an incident and an RCA is, the less useful an RCA is as memory fades away, investigations must be replicated, logs are not available anymore

and finally the chance that an issue occurs again increases over time. Third, and maybe most important, the IA derived from an RCA must be utilized. Otherwise, the whole RCA process has only very limited benefits and may not be worth the effort. As the effort to implement an IA differs widely, IA must be categorized with respect to complexity and expected gains, and decisions about their state must be facilitated in a timely manner. Decisions whether an IA will be rejected, implemented now, or added to the backlog of the next planning cycle are complex decisions and involve all complexity of product management for large projects. However, such decisions are required to improve the quality.

Overall, the RCA process allows developers to invest time into understanding issues, identifying improvements, and finally implement improvements. Conclusively, this facilitates a quality culture towards continuous improvements.

3.2 Developer

As shown in Sect. 2.1, the source code of SAP HANA is large and heterogeneous. A single person may not be able to comprehend all the code at once. While the overarching structure follows a layered architecture, the size of the components and their level of interaction varies greatly. Similarly diverse is the development organization with team sizes reaching from a few experts to more than a dozen engineers distributed across multiple locations, using different development tools as well as personalized workflows. While some developers focus on specific topics, others are involved in and contribute to several components. Despite these differences all of them interact with central tooling (Sect. 5.1).

Figs. 2 and 3 show the relationship of different test scopes from a development perspective. Local testing is important for obtaining immediate feedback about modifications to the code base. The CMake-based [55] build system allows developers to build and execute unit and component tests locally, but this can only cover a limited number of scenarios. Offloading testing tasks to the central CI (Sect. 5.1) not only allows validating changes in more complex or distributed settings in parallel, but it also enables developers to switch contexts and continue working on other tasks, such as feature design, code reviews, and root cause analysis of failures.

The build and test infrastructure (Sect. 5.1) is not only a pool of hardware resources managed by a job scheduler, but it also integrates and interfaces with various tools for defect reporting and project management, among others. An intuitive web-based user interface presents summaries of static code analysis, builds, and test strategy runs, and allows exploring the details to analyze test failures. Known defects are detected automatically by comparing the test failure output and are marked as such in the user interface to be identified quickly, and new defects can be reported with the test log files attached. Test sets for pre-submit testing can be configured on a branch level or a developer can configure a specific set of tests for a specific code change. Bisecting test failures to identify a culprit commit is centrally provided to simplify and speed-up the usage for developers.

The test infrastructure of SAP HANA keeps evolving, both in terms of functionality and capacity. For example, SAP HANA Cloud requires new runtime environments for hardware architectures or container-based execution. Additionally, tools for static code analysis and security screening must be integrated into the development process (Sect. 2.5). The less time developers must invest in orchestrating tools, gathering or evaluating results, the more time they have for software development tasks. Thus, a stable infrastructure with fast reaction times is one of the most important requirements from a developer view.

However, even with all these achievements, improving test costs and quality is a permanent activity. Long running integration tests should be replaced with short unit tests, the structure of the source code must be adapted to changes in architecture, and legacy code should be separated into modules with interfaces to decrease coupling.

3.3 Architect

Architects are experts who design the principal architecture of SAP HANA from a high-level view. As software testing aims to gain information about the quality of the software under test, architects can utilize software testing to make better informed and evidence-based decisions.

A typical example is code coverage, i.e., information about what parts of the software are executed by tests (see Sect. 5.2). While there are comprehensive discussions in the literature about the usefulness of coverage ratios (like 95% of the code is covered), an architect utilizes coverage information to learn more about architectural and technical challenges. For example, a high code coverage from tests can indicate an architecture is testable without much effort and therefore has often well-structured interfaces. This is then helpful to evaluate effort and risks for refactoring activities which can be lower for areas with high code coverage. In addition, a low number of bugs together with a high code coverage can indicate a high code quality, although this still depends on whether the functionality represented by the corresponding code is frequently used by a large number of users or not. There is also a trade-off between high code coverage from tests and effort of test implementation and maintenance. A balance must be found that translates into a long-term high velocity of the development teams.

Velocity and developer efficiency is a second important example where in this case not the software testing results

are important, but the software testing process. From an architectural point of view, the software, the architecture, and the testing process must allow quick and reliable test activities. The time it takes to create and execute tests directly translates to longer waiting times or context switches for developers, both reducing efficiency. Unreliable test results create unnecessary work, frustration, and loss of trust. Again, these aspects reduce the development efficiency.

Considering the given examples, software testing can provide information for architects to initiate activities for improvements. A high number of bugs can trigger a further investigation of the types of the bugs and based on the types of bugs can trigger a re-implementation to avoid such bugs in the future [78]. A high number of bugs that cannot be reproduced can indicate complexity that is not well-understood. A high number of bugs that are in fact not bugs but misunderstanding can indicate a lack in supportability and usability of the system [65, 74]. Low code coverage can result in activities to separate concerns, create smaller concerns and introduce well-structured interfaces. This can even improve compile and test times.

3.4 Manager

From a management point of view, testing serves two main purposes. First, as management is "the process of dealing with or controlling things or people" (Oxford dictionary), testing provides information to enable effective management. Second, testing is part of the quality assurance process to ensure requirements of the users are fulfilled. For SAP HANA, the main requirements are stability and data integrity (Sect. 2.3).

With the help of testing, management gains information about required efforts and can plan budgets for developing new functionality and maintaining the software (see also Sect. 3.3). Therefore, there is a need for testing activities itself, but also to reduce costs for testing. For example, bug handling (Sect. 2.5) costs resources. Typically, the resource costs are higher, the later a bug is found. Therefore, it is important for management to facilitate development and test processes that prevent issues and find them early if they exist ("shift left" [80]). However, conducting test activities early typically means a higher frequency and higher costs. Consequently, analysis and decision about trade-offs is a permanent task for management. Similarly, if new features are planned or developed, testing can provide information about how much effort is required to implement or release the functionality with a sufficient degree of quality ("gap analysis").

Fundamentally, testing only provides information. However, this information is then used to trigger improvements if required (see Sect. 2.3). With that, testing is important for management to satisfy the requirements stated by users. For example, when issues with the stability of SAP HANA may negatively affect the business of a user, this user then may expect immediate solutions from management. Therefore, even if "testing can (only) be used to show the presence of bugs, but never to show their absence" (Edsger W. Dijkstra [23]), it is important to utilize it for reducing the cases where bugs lead to issues. Testing should not only prevent negative changes, but it is also required to document positive changes. For example, testing the performance or memory usage may document improvements that can be used by management to advertise the product.

The switch from classical on-premise software to SAP HANA Cloud requires adaptations in the quality concept that we further explain in Sect. 6. It is also the responsibility of the management to shift resources and culture towards such adaptations.

Finally, from a management perspective, it is a constant challenge to determine the best distribution of resources. The proportion of resources spent for implementation, improvements, or support (handling and fixing issues) is a topic of continuous debate. We experienced that concepts of control theory may provide good results. That means, a fixed distribution of resources has severe limitations in practice and applying corrective behavior leads to a better control of a dynamic system. For example, we control the resources for feature implementation of a component by the number of issues that are found in the respective component. Resource distribution in large software projects is still a topic where more best practices and research is welcome.

With respect to managing human resources, there are several hundred persons involved in the development of SAP HANA. There is a wide range of experiences and knowledge in addition to a constant natural churn of employees. This leads to the challenge of how software development can be fundamentally improved to avoid categories of issues and how such an improvement can be designed in a scalable way. It seems to be a poor approach that software engineering still mostly adopts a learning by mistake methodology.

4 Pre-/Post-Submit Testing

The lifecycle of a new change can be differentiated into *pre-submit* and *post-submit* states that differentiate whether the change is not yet included in the central code repository or it is.

4.1 Pre-Submit Testing

For pre-submit testing, we assume that a developer creates a new change locally. During the development process, the

developer will utilize and implement testing activities based on a test plan, given project guidance and experience. In addition, the developer utilizes a subset of the existing tests to test for regressions. Then, the first formalized step is doing component tests for which a developer might push a change to a development branch for the corresponding component. Each component, represented by a team of developers, defines a set of regression tests that are executed for all changes within their component. After the change is finished from a functionality point of view, the developer will push it to the main code branch to integrate it in the main code line. Due to scaling reasons, such a push might include multiple changes or multiple new features. The main purpose of the main code line testing is then to a) ensure that the main code line is in a good state, i.e., compiles and passes a fundamental set of tests, and b) that changes for one component do not introduce regressions in other components. The overall goal is that the main code line can be used at any state in production. A change can only be included in the main codeline if the pre-submit testing results have been positively assessed. Surprisingly, this is a rather complex task due to flaky tests (Sect. 7.3). For SAP HANA, a dedicated team supports development with test assessment, and we utilize automated approaches that classify for a test result whether any further action is required or not.

A challenge for a large project like SAP HANA is, as detailed in Sect. 2.1, that we must test about 80 changes per day in pre-submit tests. Considering that it can take several hours until we know the test results for a given change it is clearly not possible to run all these changes in a sequence and integrate them if the tests run successfully. Now, one could argue that we can execute pre-submit testing for multiple changes in parallel. However, this reduces the information we gain from testing. If we execute the tests for change c_1 and change c_2 in parallel, then even if all tests pass, we only know information about the state S_1 after c_1 and the state S_2 after c_2. We know nothing about the state S_{12} after applying c_1 and c_2. S_{12} could contain so-called higher order merge conflicts [12, 88] where, for example, c_1 starts to call a function, but c_2 changes the results of this function. As we need to integrate the given number of changes per day into our code repository, we accept the risk of such conflicts and execute pre-submit tests in parallel. We reduce the probability of issues by executing a rather fast syntax test for compilation before integrating the new state, S_{12} in our example. On average, we still find about 1 higher order merge conflict for every 800 changes.

4.2 Post-Submit Testing

For post-submit testing, we assume that a change is already integrated into the main code line for a new version of the software v_n. The state of v_n is then tested by additional regular testing activities. These are typically tests that have long execution times, high resource usage, or have shown in the past a low number of detected issues (for example tests for established components without further modifications). In addition to these post-submit testing activities, there are, depending on release cycles, extended test activities for a release of v_n. These activities then include for example manual testing where new functionalities will be tested from a user perspective based on requirements and documentation, upgrade paths from previous releases to v_n are verified, or user-specific workloads are extensively used to test against regressions. All the described activities are embedded in well-defined development, test, and release processes. The test results are documented, and their conformance to expectations is checked for releases. Together with other activities, these aspects of the software development process for SAP HANA then align with SAP global frameworks for development, quality management, and releases.

Challenges in pre-submit testing are test execution times and the mapping between issues found and responsible persons. As post-submit testing activities may execute over days and weeks, they must be carefully scheduled along release cycles. In contrast to pre-submit testing, where the author of a change and therefore the responsible person for an issue exists, we need to identify a responsible person for issues found in post-submit testing. We utilize a mix of git bisecting, data retrieval and human knowledge to solve this.

5 Continuous Test Activities

In contrast to pre- and post-submit testing, there are also test activities that happen at any time, independent of a change or a release. Theoretically, all the continuous activities could be conducted for each change, but the overhead in time and costs does not correspond to the expected benefits. Therefore, the scope of such continuous testing activities is typically the source code of the whole project and they can be understood as a continuous effort towards testing and improving the quality of SAP HANA.

5.1 CI Tooling

The staged testing concept of SAP HANA (Sect. 2.4) and the size of the project (Sect. 2.1) requires an automated, resilient, scalable, reliable, and performant Continuous Integration (CI) infrastructure. The main components of the build and test infrastructure are a CI tool, a cloud-native execution platform, and domain-specific services for data analysis and interpretation.

The central CI tool configures tests activities for each code branch. A central repository versions each change. The configuration is flexible and supports all the quality measures that are described in this work (and more). The user interface of the application shows for each CI run the progress, aggregated results, and mostly automated result assessment based on an underlying rules engine.

The execution platform translates the workload of a CI run into a domain-specific graph language. Each node in the graph represents an automated job and has defined resource requirements. For example, an integration test may consume multiple terabytes of RAM and more than 100 CPU cores, while a static code analysis job may require less resources. The platform governs a hardware cluster hosted both on-premises and on public clouds. A custom scheduler deduplicates the execution graph and delegates the workload to cluster nodes offering free resource capacity. Containerization ensures performance isolation at the cluster nodes. The execution platform also comprises a data management system, which transparently transports input and output data between graph nodes and takes care of data lifecycle management.

Internal SAP HANA instances (Sect. 5.6) persist the results of CI jobs together with the corresponding metadata. The information can be retrieved via SQL and APIs. We apply statistical analysis and machine learning techniques to derive insights and correlate information across CI runs, code lines, and branches [1, 6].

The CI stack is complemented by tools supporting the development process, such as a customized extensions for C++ IDEs (Qt Creator[1], Visual Studio Code[2]), bug tracker, and tools showing quality metrics per component such as code coverage, compiler warnings, or open bugs.

5.2 Code Coverage

Code coverage provides information about which part of the source code is tested. Obtaining this information has an overhead in terms of times and resources as we have to identify a mapping between execution of a test back to the underlying source code for the executed part of a binary [4]. For a very large software project as SAP HANA, the overhead translates to considerable costs. In addition to these expected costs, we experienced that most tools to collect code coverage are not compatible with the size and complexity of SAP HANA and therefore additional technical challenges must be solved, which further increase costs. Therefore, we proceed similar to other large projects [46] and collect code coverage not for every change but only 2 to 3 times per week.

While there are discussions in the literature about the usefulness of code coverage as an indicator for the quality of a test suite [42], we see code coverage as a source of information, but not as a specific goal for quality. Components must achieve a specific threshold for the coverage ratio. However, the underlying motivation is not that this threshold defines a specific degree of quality, instead a threshold regulates the amount of time that is spent for creating tests. In fact, the value of the threshold is not relevant as it just represents a relationship between coverage ratio and time invested into creating tests. Sect. 3.3 describes concrete examples how information gained by code coverage steers further activities.

In addition to the general aspect, code coverage also allows us on a technical level to apply techniques from research work to reduce test code or to improve quality [1, 4, 8]. However, we found that we must carefully check the usefulness of such techniques. First, the coverage of SAP HANA has the possible unexpected attribute that there is a large part of so-called common coverage, i.e., some parts of the software are executed by all tests. As a large part of our tests are based on SQL statements, each such test executes the whole database stack. With that, every test that executes any SQL statement may execute the same millions of source code lines as any other test and might only deviate by some hundred executed lines from another test. That is for example a challenge for spectrum-based techniques [1] or any technique that assumes every line included in coverage data is equally important. In addition to this technical limitation, we also found that some techniques are not relevant for large projects. There is for example a wide range of work for test case prioritization [62], which has no benefits for a large project due to flakiness and long test execution times where developers will not frequently check intermediate results but switch to other tasks and come back to checking the test results later. Previous work provides more information about the practical usefulness of code coverage for SAP HANA [4].

5.3 Customer Scenario Testing

As detailed in Sect. 2.1, SAP maintains an extensive set of regression tests for SAP HANA. However, even with such an extensive test suite, users may utilize the software in a way that is not explicitly tested. For example, users may utilize a different table and data distribution, workload management, system utilization, size or configuration. It is therefore important to test a new software version with workloads that replicate real usage scenarios.

One approach at SAP to achieve this is recording all workload from a user system over a certain time [9] in a GDPR-compliant way. This workload can then be replayed at any time for any new version. On a high level,

[1] https://www.qt.io/product/development-tools.

[2] https://code.visualstudio.com/.

the implicit test oracle, i.e., the verification whether the test has passed or not, is whether the replay executes successfully or if there are any errors encountered during the execution [3, 31, 61, 82, 85, 90]. In addition, the recording also allows a fine-grained analysis of individual queries and performance.

The recorded replays then represent a typical workload from a specific customer. Depending on the customer, such a workload can be dominated by updates, by analytical queries, or by a specific special functionality such as graph-related queries or machine learning.

Customer scenario testing aims to replay these workloads as part of the CI process. While the idea of replaying workloads sounds simple in theory, it poses severe challenges in practice while implementing a fully automated process. They span from identifying the most helpful workload patterns to record, risk of operational mistakes during the replay, false positives in test results, distinguishing different root causes, deduplicating redundant findings, to reliably reproducing new errors with a minimal reproduction setup.

Several challenges require to map a large amount of data to correct actions, i.e., typical classification tasks. Hence, we utilize machine learning to, e.g., analyze false positives in test results and identify root causes of failures [47].

Overall, customer scenario testing then ensures that users can upgrade to newer versions without regressions and that the required testing is automated and meaningful although the input of user workloads is diverse and only structured to some extent. Avoiding regressions for user workloads is especially important for SAP HANA Cloud where SAP controls software upgrades and their effects (see Sect. 6).

5.4 Logs, Traces, and Debugging

For large projects such as SAP HANA, it is important to provide extensive information for debugging. It must be available on multiple levels of detail for different target groups from users to developers. As stability is important (Sect. 2.3), SAP HANA collects a wide range of information in the case of a crash (unexpected end of a process) such as stack traces, CPU register and memory content, or state of program variables and threads. In addition, SAP utilizes reverse debugging to deterministically reproduce crashes and backwards in time execution to find the causes for issues [26].

In addition to support debugging, we automatically collect logs and traces from SAP HANA Cloud instances to either predict issues and mitigate them before they occur or to report and map them to known bugs or create new bug reports (Sect. 2.5). Therefore, logs and traces can be considered as a test tool in the sense that they provide information about the state of the software.

5.5 Malfunction Testing

Even with the availability of multiple terabytes of memory for single systems, out of memory (OOM) issues can still be relevant for a running SAP HANA instance. Fundamentally, either data is larger than anticipated, or the statement is running into a statement memory limit and the memory manager throws an exception. Such exceptions can occur at any place that allocates memory, i.e., almost anywhere in the code. As out of memory situations can impact the availability and due to the large number of places where issues can occur, it is important to have a specialized concept for testing such situations properly.

As SAP HANA utilizes a custom-tailored memory management framework [63], we can set the memory management into a special “malfunction” mode for testing purposes where the memory manager decides voluntarily and randomly to not provide memory but throws an exception instead. A heuristic further decreases the locality of such exceptions to avoid throwing exceptions repeatedly at the same code location. The malfunction mode can be used for C++ based tests or for a whole SAP HANA instance. Overall, the malfunction tests contributed and contribute to a large extent to the stability of SAP HANA even in OOM situations.

In addition to expected OOM situations, there can also be incorrect memory handling that results in memory leaks, use-after-free situations, or other memory safety violations. A memory leak occurs if memory is allocated but not released although it is not needed anymore. This does not only negatively affect the availability of SAP HANA as even small memory leaks might accumulate over the long periods that DBMS are executed, but also has a negative impact on the total cost of ownership that partially depends on required memory. In addition, issues with memory safety can result in crashes or security vulnerabilities.

Therefore, testing for and the detection of incorrect memory handling are important tasks for SAP HANA. SAP achieves this with multiple approaches. First, due to the custom-tailored memory management framework, we typically know at runtime where memory was allocated, where it is released (or not), and can attribute all allocated memory on a rather fine-grained level to the allocating places and components. In addition, we check for memory safety violations with tools such as AddressSanitizer [77], ThreadSanitizer [76], MemorySanitizer [83], Valgrind [59], fuzzing [91], or debug it via reverse debugging [26, 84]. Finally, we utilize our own and third-party static code checkers to detect issues with memory management as early as possible and have internal recommendations to avoid specific programming patterns that are error prone.

5.6 Qualification of Internal Systems

The qualification of SAP-internal SAP HANA systems is one step of the quality process. We use multiple SAP HANA systems to store multiple terabytes of information about development, testing, delivery, crash call stack information, bugs, or infrastructure observability events. This approach offers the possibility of testing development versions and new features of SAP HANA at an early stage with large systems and practical workloads. In addition, developers also experience stability issues by themselves. This provides a helpful regulation cycle.

5.7 Performance Testing

The existence of prominent database benchmarks such as defined by TPC [11], indicates the importance of performance for databases. Performance is also important for users if they need to achieve certain response times or throughput for their target workload. Therefore, performance testing is part of pre- and post-submit testing for SAP HANA [71]. Developers create performance tests to measure the performance of their code, but also to protect against regressions in future updates. In our experience, the latter becomes increasingly important for functionalities that have many dependencies as the complexity of the product increases. The pre-submit tests must execute quickly and typically focus only on specific features or components. The post-submit tests include both extensions of pre-submit tests, more complex tests that cover multiple features or components, and long running performance tests. The tests are typically written in C++ or Python and interact with SAP HANA via SQL statements. On a unit level, performance tests for core algorithms and data structures may utilize Google benchmark [33, 64].

In the context of performance tests, we speak of *measurements* as the smallest (executable) entity. In each measurement, we may measure multiple *metrics*. While the most common metrics are elapsed time, CPU time, and memory consumption, there can also be others such as throughput metrics for I/O or queries. We store all measurement results in a database (Sect. 5.6) which serves several purposes: The data is used to define and track the baselines that we compare against, to track performance changes over time, to detect "broken" hardware that produces inconsistent performance results, and to assess quality and stability of tests. For performance improvements, we automatically adapt the baselines that we compare against.

Special properties of performance tests: In comparison to functional tests, performance regression testing differs both in the way a regression is detected and in the requirements towards the test environment. In functional regression testing, we test for equality against an expected result and the test environment must provide enough resources to execute the test. That implies multiple tests can run in parallel on shared hardware, and tests can run on different types of hardware if the executed code is the same. In performance regression testing we in fact do statistical tests against a threshold and we must reduce the number of variables that can affect the result of the statistical test to a minimum. Therefore, the environment in which a test is executed must be exactly the same for all measurements. This implies exclusive use of identical hardware for each measurement to ensure reproducible performance measurements.

In an ideal world, a series of identical measurements for the same code is just a series of identical results (*flat line*). However, in practice, there are undesired derivations that we call *noise*. The three main factors for noise are hardware, runtime behavior, and the build.

To exclude noise from hardware effects, we run tests on pools of identical machines, i.e., same vendor, board, CPU, memory, discs, firmware, OS, and system settings. A landscape check verifies the execution environment before each test execution.

In case of noise from runtime fluctuations there are various strategies to stabilize the measurement. First, we must avoid noise from asynchronous background processes that are triggered on a schedule, by the setup before the measurement, or by the previous test. We deactivate such processes to a minimum and wait until they are finished. Second, we increase the number of repetitions and utilize the available additional data to remove noise via statistical methods.

We use the category of build noise for a complex but important aspect. We sometimes see regressions that seem unrelated to code changes. They even might disappear again after more unrelated code changes. The reasons for such regressions are manifold. For example, adding code in a different submodule can influence the inlining decisions of the compiler. Removing or adding a member variable can shift the cache alignment and cause false sharing. Changes to the binary layout can influence CPU behavior like instruction cache usage, TLB misses, or branch prediction. These types of regressions are challenging to locate and fix. Still, from a user perspective, it is a regression and must be treated as such.

Practical tooling: The evaluation of performance results for the pre-submit barrier is automated. We do automatic restarts if one of the measurements shows a regression to rule out fluctuations before the final evaluation. About 10% of all changes are rejected and blocked by this automated test. Such blocked changes can be re-evaluated manually to detect false positives. Typical reasons for false positives are accumulations of small regressions from previous changes, unstable measurements, or too aggressive adaptations of the baseline in case of improvements. For the post-submit

tests we generate reports with regression candidates that are reviewed by a team of experts. In case of regressions, our CI tooling (see Sect. 5.1) utilizes an automated git bisect to find the culprit, i.e., the change that caused the regression. The team then opens bugs for those performance regressions and assigns them to the responsible developers.

The performance test framework also integrates profilers to further analyze performance behavior and the causes of regressions. This includes the SAP HANA Kernel Profiler, JobExecutionLog [74], (allocator) traces, SQL plan analyzer, and other SAP HANA built-in profiling mechanisms. External tools like vTune [72] or perf [50] can also be integrated. While most performance regressions can be reproduced on developer machines, we also implemented a central hardware rental service to analyze specialized hardware or distributed scenarios.

For SAP HANA Cloud, performance testing is still a challenge. SAP HANA runs in Kubernetes pods, which raises concerns with noisy neighbor effects, CPU overprovisioning, network issues, or hardware capabilities [70]. The complexity of the cloud infrastructure raises challenges regarding the robustness of performance test execution [41] and requires further elaboration and tooling.

5.8 Multi-Modal Testing

Most of the testing in SAP HANA is done on relational data. However, SAP HANA, as a multi-model [53] database also supports graph, spatial, and (JSON) document models. Creating input data, workloads, and queries for correctness and performance for these models is a challenge [64].

For graph engines, we use data generators like R-MAT [14] and publicly available datasets [27, 52]. In comparison to graph models, there is less previous work on spatial and JSON document store benchmarks [21]. Existing benchmarks from other domains [17] have limitations regarding external validity [41, 48]. Thus, we created DeepBench [10] to generate JSON-specific data and queries for technical and domain-specific workloads. For spatial models, we generate data by ourselves and from OpenStreetMap [60]. We also employ regression tests for multiple clients, e.g., QGIS[3], GDAL[4], or ESRI[5].

5.9 Randomized Testing

Automatically generating input that was not expected by developers can provide new insights about the behavior of the software. In our experience, such an approach is useful to find concurrency issues such as race conditions, testing the interaction of multiple components and verifying correct input parsing of, e.g., SQL statements. Therefore, we utilize existing tools such as SQLsmith [75] but also developed our own tools for SQL [34] and unit test level [91]. Developing our own tools allows us to support a wider range of data types, more statement variants, parallel execution, and integration with internal debugging and analysis functionality.

The main challenges for tests based on automatically generated input is the test oracle. If we enter random input, we get random output of which we may not know in general whether it is correct or not. Therefore, our randomized testing approaches utilize implicit test oracles such as crashes, memory leak detection (Sect. 5.5), consistency checks, and internal assert statements. For generating input, our tools can randomize a wide set of entities, such as write, read and defining SQL statements, hints, monitoring views, configuration parameters, or commits and rollbacks for transactions.

Each month, we spent several years of CPU time to conduct randomized testing. These tests found a wide range of errors, such as crashes, deadlocks, wrong results, memory leaks and corruption, recovery issues, or regressions.

Integrating the results of the randomized testing into the development process is another challenge. For that purpose, we automated the failure assessment, bug creation (Sect. 2.5), debugging data collection (Sect. 5.4), the generation of a step-by-step manual to reproduce the issues, and the integration of fixes. We also utilize a sophisticated labeling and mapping system, which allows us to ignore known issues to avoid duplicated work.

5.10 Security Testing

The stability of a SAP HANA instance should not be negatively influenced by persons that use the system in a way that was not intended (Sect. 2.3). Furthermore, awareness and requirements for data protection increased over time. With SAP HANA Cloud, cloud-based threat models with severe impacts require appropriate measures. SAP has therefore a secure software development process that builds upon best practices of developing secure software within SAP and on industry standards.

As part of the development process, security experts conduct threat modeling to derive the requirements and security measures for the product or for features. These requirements are then part of the overall design that is implemented in the development phase. Beside those specific threats that need to be addressed, there are product security standards that developers follow. They contain information about topics such as cryptography standards, data protection management, or processes to handle security vulnerabilities.

[3] https://www.qgis.org/de/site/.

[4] https://gdal.org/.

[5] https://www.esri.com/.

Automated security checks included into central CI tooling (Sect. 5.1) support the developers. For example, static analysis tools identify potential security issues in the C++ code of SAP HANA. The usage of third party libraries is automatically detected and security information is provided for those. As an example for dynamic testing, fuzzing tests the robustness of the SQL parser (Sect. 5.9). The results of these activities are integrated in the central bug management tool (Sect. 2.5).

In addition to internal code checks, SAP also works together with external vendors that conduct security analysis and penetration tests for SAP HANA. SAP HANA might also be investigated by independent security researchers. For this case, SAP has publicly defined processes for reporting and disclosing security vulnerabilities. SAP provides regular information and recommended actions about security issues that are found in released versions of SAP HANA. In the case of SAP HANA Cloud, SAP mitigates security issues based on risk and threat analysis, see Sect. 2.6.

6 SAP HANA Cloud

As presented in Sect. 2, SAP offers SAP HANA Cloud, a database-as-a-service where the instances of SAP HANA are managed by SAP via containers. Managing SAP HANA in a cloud environment provides specific challenges in comparison to delivering software that is installed by users.

Testing HANA Cloud requires two major adaptations. First, SAP shares a greater part of the responsibility that users can use new versions without issues. In the past, it was expected that users test new versions on their own depending on a risk assessment. With SAP HANA Cloud, users expect that SAP manages the system and guarantees service level agreements (SLA). As a second major difference, there is a strong pressure on a more rationalized test strategy. Before SAP HANA Cloud, major releases had a rather long test and improvement period (“stabilization”) as also depicted by Fig. 3. With SAP HANA Cloud however, the release frequency increases which requires to reduce the time for testing. Still, users might even have higher expectations on quality. Therefore, from a management point of view, testing must be rationalized to be a best fit for the purpose.

These adaptations towards a cloud product have several practical consequences. For example, SAP may not know how users utilize and integrate SAP HANA into other systems. Therefore, SAP must focus more on providing (and testing) interfaces (see also Sect. 3.3) and must learn from usage data. Another example is a situation where one user may urgently require a fix for an issue. SAP must carefully assess the impact such a change has on other users, because the change might be immediately available to all instances of all users. A third example is the need to rationalize the testing concept towards a continuous testing approach to reduce test periods and increase release frequency.

The following sections provide more details for the general testing concept and the given examples.

6.1 Infrastructure Test Strategy

The SAP HANA database containers are hosted in a micro service environment based on Kubernetes [38]. These micro services follow their own release cycles. This infrastructure comprises more than 100 micro services that provide various features including an observability stack, an integration into SAP Business Technology Platform, and it allows to dynamically provision and host thousands of database instances in a stable and scalable setup. Identical to SAP HANA, users expect stability and data availability from SAP HANA Cloud. Therefore, the most important goals for the test strategy are preventing downtimes and data loss while maintaining a high release frequency.

SAP HANA Cloud follows a micro delivery paradigm where each change on any micro service leads to a new release of the respective micro service. A change must pass unit tests, integration tests, and compliance scans before it is delivered to users. Every new micro service version must show in scenario testing that it can be updated without issues and that other available features still work. Each new version is tested in a copy of the target environment comprising different Kubernetes clusters where the new version interacts in a production-like setting with all other services. A new version must pass all tests without downtimes or data loss before it can be deployed. The test process is fully automated, and we create and delete thousands of Kubernetes clusters per day for testing purposes.

After validating a new service version as described before, we apply the new version on existing long-lived environments. As an additional quality measure and risk mitigation we define subsets of environments through which new micro service versions are progressively deployed over time. We call that *rollout*. The first subset of environments includes internal development and validation clusters where a new service version is tested with diverse interactions and more realistic workload. The new service version can only reach internal production, which is the next subset of environments, if there are no open concerns or test findings. At that point the quality of a new version is sufficiently ensured to operate on the internal production environments, which are permanently monitored for any issues just like user-facing production environments.

After a validation period where the length depends on the service (average: 7 days), the new service version can enter production landscapes if no blocking concerns exist. The length depends on a risk analysis that includes the impact

of potential issues and the complexity of the new version. Based on the risk analysis, the length is between an hour and a month. Depending on the risk, additional continuous and manual testing activities are conducted during the validation period.

Depending on the risk and the benefits (Sect. 2.6), a new version can be made available more quickly if an urgent issue must be solved. In that case, testing activities will be focused and prioritized on this new version to shorten the time from creation to deployment to a few hours.

6.2 Feature Toggles

In our development environment with highly frequent deployments, ongoing feature development is not aligned with any planned deployment pipelines. Thus, new micro service versions might include code as part of a new feature that is not yet ready for release to users.

Feature toggles can hide ongoing development behind a configuration setting. That setting can be changed to be, e.g., active for development and testing, but inactive for other users. With that, developers can regularly submit smaller changes that are easier to review and have a smaller chance for introducing regressions as the integration of a new feature is not a binary once and all decision.

However, the higher number of configuration options draws the challenge to test suitable combinations of toggles. Testing the complete matrix of possible options results in a potential exponential number of combinations, which would dramatically increase the required testing time. This would then negatively impact the frequency of deployments and is not compatible with the desired development and testing process. Hence, every micro service must test meaningful combinations of configurations of their service. This usually includes the configuration as deployed in production environments (all new features toggled off), the configuration for release ready toggles where development is considered complete but not yet finally decided, and experimental setups to test ongoing development of incomplete features.

Feature toggles then also allow a progressive roll-out based on user groups or number of systems. This reduces the blast radius, i.e., the number of affected systems if there is an issue. Finally, feature toggles can also be deactivated at runtime which allows for faster responses to mitigate an issue compared to reverting code for a functionality and releasing a new version. An open challenge is the tracking of feature toggles and handling of obsolete feature toggles, i.e., the question of when the code for the toggling and the previous code path should be removed if a new feature is released and stable [69].

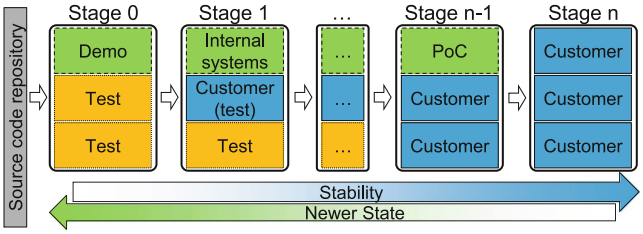

Fig. 4 Stages for SAP HANA Cloud releases

6.3 SAP HANA Stages

A new version of SAP HANA is not directly deployed into a SAP HANA Cloud production environment after it is built. Instead, there are multiple so-called *stages* where a new version is made available for testing before being deployed in the general production environment. As shown in Fig. 4, each stage represents a different state of stability and distance from the newest development state and is therefore suitable for a different purpose. These stages allow to balance the requirement to access and maybe influence new versions early and the requirement for stability that assumes less frequent changes. From a quality perspective, the stages allow to identify and mitigate issues early before reaching later stages. Therefore, we say that new SAP HANA versions progress through the SAP HANA quality pipeline.

Consequently, each stage has a position in the pipeline and consists of a set of instances with the same SAP HANA version. Furthermore, each stage defines the amount of time a version is expected to stay in that stage and a set of quality metrics. The quality metrics utilize feedback from testing and stability of the instances within the stage. They provide an indicator whether a version can progress to a new stage in the pipeline.

Instances are assigned to stages depending on a trade-off between having new functionality to test it and a requirement for stability. Instances which are located in earlier stages accept the higher risk of instabilities but can test and influence new functionality. It is expected that the earliest stages contain development and internal test instances. Stages in the middle of the pipeline contain instances from other SAP products that utilize SAP HANA and want to test new features or compatibility towards new versions. The latest stages contain productive instances.

A version can transition to the next stage via *promotion*. A promotion validates the quality of a version and concludes whether the quality is sufficient for the next stage. The decision is based on the aforementioned quality metrics and experienced-based input from so-called stage owners. Once a promotion is confirmed, a version v_i at stage s_n is assigned to the next stage so that v_{i+1} is at s_n and v_i at s_{n+1}. Alternatively, if a promotion is finally rejected, v_i drops out

of the pipeline as v_{i+1} will be promoted from s_{n-1} to s_n. s_{n+1} continues to stay on v_{i-1} as v_i did not advance. v_{i-1} is then available in two stages: s_{n+1} and s_{n+2}. Orthogonal to promotions, an instance can either switch to a lower stage by upgrading to a newer version, stay on the same stage by linking upgrades to promotions so that both happen in parallel, or progress to a higher stage by staying on the same version if a promotion occurs.

Overall, the stages concept allows a flexible quality management with multiple stages of different purposes where instances can trade stability for availability of new changes.

7 Challenges

Several challenges are already presented in previous subsections. Therefore, we will only briefly reference them. Due to the importance of test costs and flakiness, we will discuss them in detail.

7.1 Overview

Sect. 4 shows that a day does not have enough hours to conduct testing for all changes of a very large software project. In general, it is a constant open challenge how to improve turnaround times for pre-submit testing, i.e., the time from a change submission to the central CI system until test results are known, or the time-to-merge [49].

The question of test adequacy, i.e., when did we test enough, remains unanswered in general despite decades of investigations. We believe that in practice, the answer is if test costs are higher than the expected gain, but this only shifts the problem. For example, it is also unclear how tests should be distributed over different scopes such as the test pyramid, which is even not a pyramid in practice for DBMS – see Sect. 2.4. We welcome empirical studies and recommendation from research towards practical test adequacy. Similar, we welcome any recommendation how we can estimate the usefulness of a test. From a cost perspective, a test that never indicates an issue may not be worth to get executed over and over again, is it? On the other hand, are tests that indicate a lot of issues "better"? Or maybe only more flaky?

Maintaining metadata about testing is challenging as described in Sect. 2.1. We constantly learn from research and knowledge sharing about new data that we might have needed to collect in the past, although there are already some established data attributes [89].

Interacting with a large amount of tools related to testing is a constant challenge for developers where we aim to reduce $n : m$ relationships to $n : 1 : m$ (see Sect. 2.5). While we understand the simplicity of creating yet another tool for a new testing technique, it would be appreciated if there was a common understanding how tools can be embedded in workflows.

For performance testing, we see automated performance regression detection and performance testing in a cloud environment as open challenges. See Sect. 5.7 for further details and our open data set about performance regressions measurements to foster new research in this area[6].

Based on popular literature, the typical expectations towards cloud systems are that they are stateless and updates happen immediately. As discussed in Sect. 6, this is not true for SAP HANA Cloud, which creates considerable challenges where users expect that SAP maintains and updates a DBMS instance, but also expect availability of 99.99% per year or more. In addition, due to a combination of the large systems and growth in users of SAP HANA Cloud, we face continuous scaling challenges where trade-offs between time and costs must be adapted regularly.

In the context of SAP HANA Cloud, SAP has now access to a wider range of data and metrics about how DBMS are used. We believe that this data can help to improve the quality of SAP HANA and the offered service for users. It is an open question how the availability of a large amount of diverse data can shape future developments. For example, the availability of github has also contributed to the development of empirical research in computer science.

Finally, in the context of SAP HANA Cloud where we use feature toggles, there are open questions on how to test combinations of feature toggles and how to maintain them long-term from introduction to removal, see Sect. 6.2.

From a people perspective, it remains unclear how resources are best distributed and how software development could be improved to make better software with improvements that scale across the number of persons involved, see Sect. 3.4.

7.2 Test Costs

From a quality perspective, it may be desirable to test each change to the software by executing all available tests against the change before the change is included in the central code repository. However, as described in Sect. 2.1, the amount of tests and the number of commits is rather large. As the costs are then the number of tests multiplied with the number of changes, we can conclude without further calculations that the test costs will also be large. One could argue that a large project may just have large test costs and the costs scale with the project. However, in practice, the test costs scale super-linearly over time [7] and the available time per day is limited by a constant (typically 24 h). It is practically and economically not feasible to satisfy a super-linear increase in test costs. Also, if each change requires

[6] https://github.com/SAP/performance-regression-data-set.

2 h to test and there are 80 changes per day, then a day does not have enough hours to sequentially test all changes.

Therefore, as already shown in Sect. 2.4, SAP does not execute all tests for each change but a specific subset of tests for each stage. The core principle for this approach is risk-based quality management as discussed in Sect. 2.6 and a layered test strategy as shown in Fig. 3. For example, pre-submit tests are executed for each change, but post-submit tests only once per day (see Sect. 4). With that, the super-linear increase in test costs over time is reduced to a linear increase [7]

Still, decreasing tests costs while maintaining the same level of quality or accepting only small decreases on quality is a continuous challenge. Even small improvements in tests cost can be worth the effort for very large projects.

7.3 Flaky Tests

A test is considered as *flaky* if it can pass or fail without changes of the code under test [66]. Such behavior undermines the central idea of tests as guards signaling software defects or undesired outputs. The negative effects of flakiness include wasted human effort for identifying possibly spurious code defects, additional resource utilization for re-running such inconsistent tests, and loss of trust in the outcomes of tests, potentially leading to lower software quality standards. Even more, flakiness has detrimental effects on techniques assuming the test outcome depends only on the code under test. This includes fault localization and automatic program repair, test suite generation, and mutation testing methods [66].

Test flakiness is a well-known problem in the software industry for large companies like Google, Microsoft, Facebook/Meta or SAP and to open source developers [25, 66]. A typical finding is that 10% to 20% of all tests show flakiness [35, 57, 66], which is similar to numbers for SAP HANA [24]. A widely used method to mitigate flakiness is repeating test execution multiple times to accept the test as passing if at least one repetition passes [66], which is also used for testing SAP HANA.

Despite the intense academic research in this field and process improvements driven by practitioners [66], test flakiness remains a challenging and still unsolved problem. In context of large-scale projects such as SAP HANA, the reasons for this include the inherent complexity of the code under test, massively parallel test execution, rapidly evolving test code, heterogeneous testing platforms, and a large number of involved software and hardware components. These factors lead to a high diversity of causes for test flakiness and make it hard to adapt academic solutions whose assumptions might not be realistic or affordable in the industrial practice.

More specifically, for SAP HANA, the following aspects are unsolved: What would be a sound definition of flakiness? A question also raised by related work [66]. The common description of a test that passes and fails for the same code does not respect for example tests that fail due to infrastructure or hardware issues although this situation is common in large projects. Another aspect is the question of how to differentiate between flaky test results and concurrency issues? This is an important question for a DBMS, yet remains unsolved. Then, do we have to accept flakiness as a consequence of reality? Computer science builds upon well-defined mathematical concepts and a bit operates on 2 states. However, the reality does only approximate the theory to some probabilistic degree. And finally, although there is a wide range of other small open questions, how can we minimize the costs introduced by flakiness on a technical, human, and process level?

8 Conclusion

We described how testing is done for SAP HANA. Our descriptions contain solutions to several specific challenges in the context of testing a very large DBMS. Furthermore, we provide a set of open challenges with practical relevance for testing large projects and testing DBMS. Although we kept most of the descriptions brief, the overall sum of activities follows the very large size of the project. Our descriptions may provide a general understanding but solving specific challenges might require further efforts. Therefore, we continue to collaborate with the research community [73], and we aim to make more open data available to foster independent research.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. An G, Yoon J, Sohn J, Hong J, Hwang D, Yoo S (2022) Automatically identifying shared root causes of test breakages in SAP

HANA. In: Proceedings of the 44th International Conference on Software Engineering, ICSE-SEIP 2022
2. Anonymous oraguy (2018) Experience report of how the Oracle database is tested internally. archived by Internet Archive at https://web.archive.org/web/20220514124832/, https://news.ycombinator.com/item?id=18442941 from https://news.ycombinator.com/item?id=18442941. Accessed: 14.05.2022
3. Arruda F, Sampaio A, Barros FA (2016) Capture & replay with text-based reuse and framework agnosticism. In: SEKE, pp 420–425
4. Bach T (2020) Testing in very large software projects. PhD thesis, Heidelberg University, Germany, https://archiv.ub.uni-heidelberg.de/volltextserver/31757/
5. Bach T, Andrzejak A, Pannemans R (2017a) Coverage-based reduction of test execution time: Lessons from a very large industrial project. In: 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE, pp 3–12
6. Bach T, Andrzejak A, Pannemans R, Lo D (2017b) The impact of coverage on bug density in a large industrial software project. In: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp 307–313 https://doi.org/10.1109/ESEM.2017.44
7. Bach T, Pannemans R, Schwedes S (2018) Effects of an economic approach for test case selection and reduction for a large industrial project. In: 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE Computer Society, Washington, DC, USA, pp 374–379 https://doi.org/10.1109/ICSTW.2018.00076
8. Bach T, Pannemans R, Haeussler J, Andrzejak A (2019) Dynamic unit test extraction via time travel debugging for test cost reduction. In: Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, IEEE Press, ICSE '19, pp 238–239 https://doi.org/10.1109/ICSE-Companion.2019.00093
9. Baek S, Song J, Seo C (2020) RSX: Reproduction scenario extraction technique for business application workloads in DBMS. In: 2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), IEEE, pp 91–96
10. Belloni S, Ritter D, Schröder M, Rörup N (2022) DeepBench – benchmarking JSON document stores. In: Proceedings of the 9th International Workshop on Testing Database Systems, DBTestSIGMOD 2022, Philadelphia, Pennsylvania, July 17, 2022, ACM
11. Boncz P, Neumann T, Erling O (2013) Tpc-h analyzed: Hidden messages and lessons learned from an influential benchmark. In: Revised Selected Papers of the 5th TPC Technology Conference on Performance Characterization and Benchmarking, vol 8391. Springer, Berlin, Heidelberg, pp 61–76 https://doi.org/10.1007/978-3-319-04936-6_5
12. Brindescu C, Ahmed I, Jensen C, Sarma A (2020) An empirical investigation into merge conflicts and their effect on software quality. Empir Softw Eng 25(1):562–590
13. Cafarella MJ, DeWitt DJ, Gadepally V, Kepner J, Kozyrakis C, Kraska T, Stonebraker M, Zaharia M (2020) DBOS: A proposal for a data-centric operating system. CoRR abs/2007.11112. https://doi.org/10.48550/arXiv.2007.11112
14. Chakrabarti D, Zhan Y, Faloutsos C (2004) R-MAT: a recursive model for graph mining. In: Berry MW, Dayal U, Kamath C, Skillicorn DB (eds) Proceedings of the Fourth SIAM International Conference on Data Mining, Lake Buena Vista, Florida, USA, April 22-24, 2004, SIAM, pp 442–446 https://doi.org/10.1137/1.9781611972740.43
15. Cohn M (2010) Succeeding with agile: Software development using Scrum. Pearson Education
16. Colle R, Galanis L, Buranawatanachoke S, Papadomanolakis S, Wang Y (2009) Oracle database replay. Proc VLDB Endow 2(2):1542–1545
17. Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R (2010) Benchmarking cloud serving systems with YCSB. In: SoCC. ACM, pp 143–154
18. Daly D (2021a) Creating a virtuous cycle in performance testing at MongoDB. In: Proceedings of the ACM/SPEC International Conference on Performance Engineering, Association for Computing Machinery, New York, NY, USA, ICPE '21, pp 33–41 https://doi.org/10.1145/3427921.3450234
19. Daly D (2021b) Performance engineering and database development at MongoDB. In: Companion of the ACM/SPEC International Conference on Performance Engineering, Association for Computing Machinery, New York, NY, USA, ICPE '21, p 129 https://doi.org/10.1145/3447545.3451199
20. Danial A (2022) cloc: v1.93 https://doi.org/10.5281/zenodo.5760077
21. Dann J, Ritter D, Fröning H (2020) Non-relational databases on FPGAs: Survey, design decisions, challenges. CoRR abs/2007.07595. https://doi.org/10.48550/arXiv.2007.07595
22. Deming WE (2000) Out of the crisis. MIT Press
23. Dijkstra EW (1972) Notes on structured programming. Academic Press, New York, Boston, London, Oxford, pp 1–82
24. Heckmann D (2019) An analysis of flaky tests in SAP HANA (Master's Thesis). Master's thesis, Heidelberg University, Heidelberg
25. Eck M, Palomba F, Castelluccio M, Bacchelli A (2019) Understanding flaky tests: The developers perspective. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2019, pp 830–840 https://doi.org/10.1145/3338906.3338945
26. Engblom J (2012) A review of reverse debugging. In: Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference, IEEE, IEEE Computer Society, Washington, DC, USA, pp 1–6
27. Erling O, Averbuch A, Larriba-Pey JL, Chafi H, Gubichev A, Prat-Pérez A, Pham M, Boncz PA (2015) The LDBC social network benchmark: Interactive workload. In: Sellis TK, Davidson SB, Ives ZG (eds) Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015, ACM, pp 619–630 https://doi.org/10.1145/2723372.2742786
28. Färber F, Cha SK, Primsch J, Bornhövd C, Sigg S, Lehner W (2012a) SAP HANA database: Data management for modern business applications. SIGMOD Rec 40(4):45–51. https://doi.org/10.1145/2094114.2094126
29. Färber F, May N, Lehner W, Große P, Müller I, Rauhe H, Dees J (2012b) The SAP HANA database – An architecture overview. Bull Tech Comm Data Eng 35(1):28–33
30. Felderer M, Schieferdecker I (2014) A taxonomy of risk-based testing. Int J Softw Tools Technol Transf 16(5):559–568
31. Galanis L, Buranawatanachoke S, Colle R, Dageville B, Dias K, Klein J, Papadomanolakis S, Tan LL, Venkataramani V, Wang Y et al (2008) Oracle database replay. In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pp 1159–1170
32. Git contributors (2022) Git distributed version control system. Archived by Internet Archive at https://web.archive.org/web/20220520221539/, https://git-scm.com/ from https://git-scm.com/. Accessed 21.05.2022
33. Google Benchmark contributors (2022) Google benchmark – a microbenchmark support library. Archived by Internet Archive at https://web.archive.org/web/20220509170729/, https://github.com/google/benchmark from https://github.com/google/benchmark. Accessed: 09.05.2022
34. Greg Law SB (2018) Capturing bugs in extreme stress testing: Improving software quality in SAP HANA with Undo. In: Proceedings

of the 7th International Workshop on Testing Database Systems, DBTestSIGMOD 2018, ACM
35. Gruber M, Fraser G (2022) A survey on how test flakiness affects developers and what support they need to address it https://doi.org/10.48550/arXiv.2203.00483 (Tech. Rep. arXiv:2203.00483)
36. Henderson F (2017) Software engineering at Google. CoRR abs/1702.01715. https://doi.org/10.48550/arXiv.1702.01715
37. Herbold S, Trautsch A, Ledel B, Aghamohammadi A, Ghaleb TA, Chahal KK, Bossenmaier T, Nagaria B, Makedonski P, Ahmadabadi MN, Szabados K, Spieker H, Madeja M, Hoy N, Lenarduzzi V, Wang S, Rodríguez-Pérez G, Colomo-Palacios R, Verdecchia R, Singh P, Qin Y, Chakroborti D, Davis W, Walunj V, Wu H, Marcilio D, Alam O, Aldaeej A, Amit I, Turhan B, Eismann S, Wickert AK, Malavolta I, Sulir M, Fard F, Henley AZ, Kourtzanidis S, Tuzun E, Treude C, Shamasbi SM, Pashchenko I, Wyrich M, Davis J, Serebrenik A, Albrecht E, Aktas EU, Strüber D, Erbel J (2021) A fine-grained data set and analysis of tangling in bug fixing commits. https://doi.org/10.48550/arXiv.2011.06244
38. Hightower K, Burns B, Beda J (2017) Kubernetes: Up and running dive into the future of infrastructure, 1st edn. O'Reilly Media
39. Hipp R et al (2022) SQLite. Archived by Internet Archive at https://web.archive.org/web/20220517192940/, https://www.sqlite.org. Accessed: 18.05.2022
40. Huckle T, Neckel T (2019) Bits and bugs: A scientific and historical review of software failures in computational science. Society for Industrial and Applied Mathematics, Philadelphia https://doi.org/10.1137/1.9781611975567
41. Ingo H, Daly D (2020) Automated system performance testing at MongoDB. In: DBTestSIGMOD, pp 3:1–3:6
42. Inozemtseva L, Holmes R (2014) Coverage is not strongly correlated with test suite effectiveness. In: Proceedings of the 36th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE 2014, pp 435–445 https://doi.org/10.1145/2568225.2568271
43. ISO (2011a) Systems and software engineering – systems and software quality requirements and evaluation (SQuaRE) – system and software quality models. Tech. Rep. ISO/IEC 25010:2011. International Organization for Standardization, Geneva
44. ISO (2011b) Systems and software engineering – vocabulary. Tech. Rep. ISO/IEC/IEEE 24765:2017. International Organization for Standardization, Geneva
45. ISO (2015) Quality management systems – fundamentals and vocabulary. Tech. Rep. ISO 9000:2015. International Organization for Standardization, Geneva
46. Ivanković M, Petrović G, Just R, Fraser G (2019) Code coverage at Google. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2019, pp 955–963 https://doi.org/10.1145/3338906.3340459
47. Jambigi N, Bach T, Schabernack F, Felderer M (2022) Automatic error classification and root cause determination while replaying recorded workload data at SAP HANA. In: Proceedings of the 14th IEEE Conference on Software Testing, Verification and Validation (ICST 2022) https://doi.org/10.48550/ARXIV.2205.08029
48. Kamsky A (2019) Adapting TPC-C benchmark to measure performance of multi-document transactions in MongoDB. Proc VLDB Endow 12(12):2254–2262
49. Kudrjavets G, Nagappan N, Rastogi A (2022) Do small code changes merge faster? A multi-language empirical investigation https://doi.org/10.48550/ARXIV.2203.05045
50. Kukunas J (2015) Power and performance: Software analysis and optimization. Morgan Kaufmann
51. Lenz L, Felderer M, Schwedes S, Müller K (2020) Explainable priority assessment of software-defects using categorical features at SAP HANA. In: Proceedings of the Evaluation and Assessment in Software Engineering, Association for Computing Machinery, New York, NY, USA, EASE '20, pp 366–367 https://doi.org/10.1145/3383219.3383268
52. Leskovec J, Sosic R (2016) SNAP: A general-purpose network analysis and graph-mining library. ACM Trans Intell Syst Technol 8(1):1:1–1:20. https://doi.org/10.1145/2898361
53. Liu ZH, Lu J, Gawlick D, Helskyaho H, Pogossiants G, Wu Z (2019) Multi-model database management systems - a look forward. In: Gadepally V, Mattson T, Stonebraker M, Wang F, Luo G, Teodoro G (eds) Heterogeneous data management, polystores, and analytics for healthcare. Springer, Cham, pp 16–29
54. Machalica M, Samylkin A, Porth M, Chandra S (2019) Predictive test selection. In: Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, IEEE Press, Piscataway, NJ, USA, ICSE-SEIP 2019, pp 91–100 https://doi.org/10.1109/ICSE-SEIP.2019.00018
55. Marson RL, Jankowski E (2016) Build management with CMake. In: Introduction to scientific and technical computing, pp 119–132
56. May N, Böhm A, Lehner W (2017) SAP HANA – the evolution of an in-memory DBMS from pure OLAP processing towards mixed workloads. In: Datenbanksysteme für Business, Technologie und Web, Gesellschaft für Informatik, Bonn, Germany, BTW 2017, pp 545–563
57. Memon A, Nguyen B, Nickell E, Micco J, Dhanda S, Siemborski R, Gao Z (2017) Taming Google-scale continuous testing. In: Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track, IEEE Press, Washington, DC, USA, ICSE-SEIP 2017, pp 233–242 https://doi.org/10.1109/ICSE-SEIP.2017.16
58. MySQL contributors (2022) The MySQL test framework. Archived by Internet Archive at https://web.archive.org/web/20220521164527/, https://dev.mysql.com/doc/dev/mysql-server/latest/PAGE_MYSQL_TEST_RUN.html from https://dev.mysql.com/doc/dev/mysql-server/latest/PAGE_MYSQL_TEST_RUN.html. Accessed: 21.05.2022
59. Nethercote N, Seward J (2007) Valgrind: A framework for heavy-weight dynamic binary instrumentation. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, Association for Computing Machinery, New York, NY, USA, PLDI '07, pp 89–100 https://doi.org/10.1145/1250734.1250746
60. OpenStreetMap contributors (2017) Planet dump retrieved from https://planet.osm.org. https://www.openstreetmap.org. Accessed: 2017 and 2022
61. Orso A, Kennedy B (2005) Selective capture and replay of program executions. Acm SIGSOFT Softw Eng Notes 30(4):1–7
62. Orso A, Rothermel G (2014) Software testing: A research travelogue (2000–2014). In: Future of Software Engineering Proceedings, Association for Computing Machinery, New York, NY, USA, FOSE 2014, pp 117–132 https://doi.org/10.1145/2593882.2593885
63. Oukid I, Booss D, Lespinasse A, Lehner W, Willhalm T, Gomes G (2017) Memory management techniques for large-scale persistent-main-memory systems. Proc VLDB Endow 10(11):1166–1177
64. Paradies M, Kinder C, Bross J, Fischer T, Kasperovics R, Gildhoff H (2017) Graphscript: Implementing complex graph algorithms in SAP HANA. In: Rompf T, Alexandrov A (eds) Proceedings of The 16th International Symposium on Database Programming Languages, DBPL 2017, Munich, Germany, September 1, 2017, ACM, pp 13:1–13:4 https://doi.org/10.1145/3122831.3122841
65. Park K, Jeong T, Jeong C, Lee J, Lee DH, Lee YK (2020) ProcAnalyzer: Effective code analyzer for tuning imperative programs in SAP HANA. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, Association for Computing Machinery, New York, NY, USA, SIGMOD '20, pp 2709–2712 https://doi.org/10.1145/3318464.3384686

66. Parry O, Kapfhammer GM, Hilton M, McMinn P (2021) A survey of flaky tests. ACM Trans Softw Eng Methodol 31(1):17:1–17:74. https://doi.org/10.1145/3476105
67. Patel B, Palekar A, Shiralkar S (2010) A practical guide to SAP netweaver business warehouse (BW) 7.0. Galileo
68. PostgreSQL 14 contributors (2022) PostgreSQL 14 regression tests. Archived by Internet Archive at https://web.archive.org/web/20220521163923/, https://www.postgresql.org/docs/current/regress.html from https://www.postgresql.org/docs/current/regress.html. Accessed: 21.05.2022
69. Ramanathan MK, Clapp L, Barik R, Sridharan M (2020) Piranha: Reducing feature flag debt at uber. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp 221–230
70. Rehmann KT, Folkerts E (2018) Performance of containerized database management systems. In: Proceedings of the Workshop on Testing Database Systems, Association for Computing Machinery, New York, NY, USA, DBTest'18 https://doi.org/10.1145/3209950.3209953
71. Rehmann KT, Seo C, Hwang D, Truong BT, Boehm A, Lee DH (2016) Performance monitoring in SAP HANA's continuous integration process. SIGMETRICS Perform Eval Rev 43(4):43–52
72. Reinders J (2005) VTune performance analyzer essentials vol 9. Intel, Santa Clara
73. SAP employees and others (2022) Scientific publications and activities of the SAP HANA database. Archived by Internet Archive at https://web.archive.org/web/20220521194026/, https://wiki.scn.sap.com/wiki/pages/viewpage.action?pageId=448477861 from https://wiki.scn.sap.com/wiki/pages/viewpage.action?pageId=448477861. Accessed: 21.05.2022
74. Scheuer T, May N, Böhm A, Scheibli D (2016) JexLog: A sonar for the abyss. Proc VLDB Endow 9(13):1493–1496. https://doi.org/10.14778/3007263.3007292
75. Seltenreich A, Bo T, Mullender S, SQLsmith contributors (2022) SQLsmith. Archived by Internet Archive at https://web.archive.org/web/20220522152342/, https://github.com/anse1/sqlsmith from https://github.com/anse1/sqlsmith. Accessed: 22.05.2022
76. Serebryany K, Iskhodzhanov T (2009) ThreadSanitizer: Data race detection in practice. In: Proceedings of the Workshop on Binary Instrumentation and Applications, Association for Computing Machinery, New York, NY, USA, WBIA '09, pp 62–71 https://doi.org/10.1145/1791194.1791203
77. Serebryany K, Bruening D, Potapenko A, Vyukov D (2012) AddressSanitizer: A fast address sanity checker. In: Proceedings of the 2012 USENIX Conference on Annual Technical Conference, USENIX Association, USA, USENIX ATC'12, p 28
78. Sherkat R, Florendo C, Andrei M, Blanco R, Dragusanu A, Pathak A, Khadilkar P, Kulkarni N, Lemke C, Seifert S, Iyer S, Gottapu S, Schulze R, Gottipati C, Basak N, Wang Y, Kandiyanallur V, Pendap S, Gala D, Almeida R, Ghosh P (2019) Native store extension for SAP HANA. Proc VLDB Endow 12(12):2047–2058
79. Smart JF (2011) Jenkins: the definitive guide. O'Reilly Media
80. Smith L (2001) Shift-left testing. Dr Dobb's J 26(9):56
81. Society IC, Bourque P, Fairley RE (2014) Guide to the software engineering body of knowledge (SWEBOK), 3rd edn. IEEE Computer Society, Los Alamitos
82. Sprenkle S, Gibson E, Sampath S, Pollock L (2005) Automated replay and failure detection for web applications. In: Proceedings of the 20th IEEE/ACM international conference on automated software engineering, pp 253–262
83. Stepanov E, Serebryany K (2015) MemorySanitizer: Fast detector of uninitialized memory use in C++. In: 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp 46–55 https://doi.org/10.1109/CGO.2015.7054186
84. Undo (2020) UndoDB – an interactive reverse debugger for C/C++. Archived by Internet Archive at https://web.archive.org/web/20200110153106/, https://undo.io/solutions/products/live-recorder/ from https://undo.io/products/undodb/. Accessed: 10.01.2021
85. Wang Y, Buranawatanachoke S, Colle R, Dias K, Galanis L, Papadomanolakis S, Shaft U (2009) Real application testing with database replay. In: Proceedings of the Second International Workshop on Testing Database Systems, pp 1–6
86. Winslett M, Braganholo V (2019) Richard Hipp speaks out on SQLite. SIGMOD Rec 48(2):39–46. https://doi.org/10.1145/3377330.3377338
87. Winters T, Manshreck T, Wright H (2020) Software engineering at Google: Lessons learned from programming over time. O'Reilly media
88. Wuensche T, Andrzejak A, Schwedes S (2020) Detecting higher-order merge conflicts in large software projects. In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), IEEE, pp 353–363
89. xUnit XML Format contributors (2022) xUnit.net v2 XML format. Archived by Internet Archive at https://web.archive.org/web/20220522165035/, https://xunit.net/docs/format-xml-v2 from https://xunit.net/docs/format-xml-v2. Accessed: 22.05.2022
90. Yan J, Jin Q, Jain S, Viglas SD, Lee A (2018) Snowtrail: Testing with production queries on a cloud database. In: Proceedings of the Workshop on Testing Database Systems, pp 1–6
91. Yoo H, Hong J, Bader L, Hwang DW, Hong S (2021) Improving configurability of unit-level continuous fuzzing: An industrial case study with SAP HANA. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 1101–1105 https://doi.org/10.1109/ASE51524.2021.9678685

Affiliations

Thomas Bach[1,2] · Artur Andrzejak[2] · Changyun Seo[1] · Christian Bierstedt[1] · Christian Lemke[1] · Daniel Ritter[1] · Dong Won Hwang[1] · Erda Sheshi[1] · Felix Schabernack[1] · Frank Renkes[1] · Gordon Gaumnitz[1] · Jakob Martens[1] · Lars Hoemke[1] · Michael Felderer[3] · Michael Rudolf[1] · Neetha Jambigi[3] · Norman May[1] · Robin Joy[1] · Ruben Scheja[1] · Sascha Schwedes[1] · Sebastian Seibel[1] · Sebastian Seifert[1] · Stefan Haas[1] · Stephan Kraft[1] · Thomas Kroll[1] · Tobias Scheuer[1] · Wolfgang Lehner[4]

Artur Andrzejak
artur.andrzejak@uni-heidelberg.de

Changyun Seo
changyun.seo@sap.com

Christian Bierstedt
christian.bierstedt@sap.com

Christian Lemke
christian.lemke@sap.com

Daniel Ritter
daniel.ritter@sap.com

Dong Won Hwang
dong.won.hwang@sap.com

Erda Sheshi
erda.sheshi@sap.com

Felix Schabernack
felix.schabernack@sap.com

Frank Renkes
frank.renkes@sap.com

Gordon Gaumnitz
gordon.gaumnitz@sap.com

Jakob Martens
jakob.martens@sap.com

Lars Hoemke
lars.hoemke@sap.com

Michael Felderer
michael.felderer@uibk.ac.at

Michael Rudolf
michael.rudolf01@sap.com

Neetha Jambigi
neetha.jambigi@student.uibk.ac.at

Norman May
norman.may@sap.com

Robin Joy
robin.joy@sap.com

Ruben Scheja
ruben.scheja@sap.com

Sascha Schwedes
sascha.schwedes@sap.com

Sebastian Seibel
sebastian.seibel@sap.com

Sebastian Seifert
sebastian.seifert@sap.com

Stefan Haas
stefan.haas@sap.com

Stephan Kraft
stephan.kraft@sap.com

Thomas Kroll
thomas.kroll@sap.com

Tobias Scheuer
tobias.scheuer@sap.com

Wolfgang Lehner
wolfgang.lehner@tu-dresden.de

[1] SAP, Walldorf, Germany

[2] Heidelberg University, Heidelberg, Germany

[3] University of Innsbruck, Innsbruck, Austria

[4] TU Dresden, Dresden, Germany