A Broadcom Company
technologies

White Paper

Continuous Testing for Continuous Delivery

Executive Summary

Challenge

User needs are changing fast, and user expectations are set higher than ever. More organizations are therefore moving toward continuous delivery, to deliver software that accurately reflects the desired functionality. However, enterprises often find that the bottlenecks of past delivery methods remain, slowing projects down and damaging quality. Nowhere is this more true than in testing. Starting with low-quality requirements, the test design and execution process, test data allocation and test environment setup are simply too slow and manual for continuous testing, and still allow an unacceptable number of defects through.

Opportunity

Testing becomes an end-to-end, cross-functional operation, collaboratively involving all teams throughout the product lifecycle. Continuous testing applies the methods and concepts of agile development to the testing and QA process, resulting in greater efficiency. Automation is key, requiring an automated, end-to-end testing solution that is integrated with existing continuous integration and continuous delivery processes. Only then can errors and overly manual processes be eliminated, facilitating true continuity throughout the development lifecycle.

By removing the constraints that force testing to the end of a development cycle, developers and QA teams can achieve accurate, automated testing from day one.They can create an integrated pipeline, capable of taking an idea from design to deployment, at speed, without compromising quality. Creating this pipeline starts with upfront knowledge of the changing user needs, which is then used to drive the continuous delivery ecosystem, from requirements and automated test execution, to standardized release deployments.

Benefits

Continuous testing helps to ensure that quality is built in from the requirements, while validating each component at the development level before it enters the system. Meanwhile, iterative QA starts from day one. Defects and miscommunication of changing user needs can be resolved as they emerge, avoiding late rework and damage to the ultimate user experience. Design, test, and development assets are all built on the users' desired functionality and are maintained consistently as this changes. Meanwhile, software is accurately deployed, resulting in better-quality software, delivered earlier, and at lower cost.

Barriers to Continuous Testing: A Choice Between Speed and Quality

Continuous delivery is only as fast as its slowest part. Just one constraint during one stage of the development lifecycle will result in mounting bottlenecks.

Testing is often blamed for delays in the continuous delivery pipeline, viewed as something that is necessary for quality but slows down projects. The reality is that it is rarely the test team's fault; the fault lies with traditional testing practices that are too slow, manual, and unsystematic to be continuous. They allow an unacceptable number of defects to slip through the net to production, creating costly rework and harming quality.

Poor Quality Requirements and User Stories

The majority of defects occur before a line of code has even been written. They are rooted in poor quality requirements and then perpetuated throughout the entire lifecycle. The longer these defects go undetected, the more damage to system quality and the more time-consuming rework they create.

The bulk of requirements is written in natural language and stored in disparate file formats, with little dependency mapping between them. This is true even with new agile project management and traditional application lifecycle management tools. Though these tools provide a centralized repository for requirements and user stories, they do not typically improve the way in which the requirements are gathered. Users still enter chunks of text into open fields; the difference is that these stories and change requests are not compiled into the monolithic documents that they would be at the start of waterfall projects.

Natural language is far removed from the logical steps of a system that need to be developed and tested. Natural language tends to lead to ambiguity and incompleteness, increasing the likelihood that the desired functionality will be miscommunicated and misunderstood. When this happens, defects enter the code; and the test cases might not be accurate or complete enough to detect them.

Much research has been published on requirements, both before and after the publication of The Agile Manifesto.[1][2][3] The picture that this research paints is of a consistently negative, year-on-year impact that requirements make on software quality. Nevertheless, in 2015, only 4% of the investment for the average project went into establishing clear business objectives, while 19% of projects failed, and 52% were deemed *challenged*.[4] Focusing on the quality of requirements appears, therefore, to be a necessary step for any organization aiming to drive up delivery speed and quality.

Test Design Time and Quality

Written requirements and flat diagrams are static, meaning that test cases have to be created manually from them. This is highly time-consuming and prevents testing from becoming continuous. And it rarely leads to tests of sufficient quality. A typical system today has several million or even billion paths through its logic, each of which is a potential test case. The time required to manually create tests in a linear manner is neither feasible nor scalable. CA Technologies, A Broadcom Company, has worked with testing teams that have taken six hours to create 11 test cases for one component alone, achieving only 16% coverage.[5]

1. Bender RBT, Inc., "Requirements Based Testing Process Overview," 2009
2. Soren Lauesen and Otto Vintro, IT University of Copenhagen, "Preventing Requirement Defects: An Experiment in Process Improvement," 2001
3. P Mohan, A Udaya Shankar and K JayaSriDevi, Hyderabad Business School, GITAM University, "Quality Flaws: Issues and Challenges in Software Development," 2012
4. The Standish Group, "CHAOS Report 2015," Oct 2015
5. Huw Price, CA Technologies, "Test Case Calamity," May 29, 2014

Moreover, test cases derived manually typically cover whatever tests that testers can come up with in their heads. Faced with incomplete documentation and fast-growing, complex applications, even the most talented testers will struggle to cover a fraction of the millions of possible paths. Tests are usually therefore *happy path* focused, repetitively testing the expected functionality while neglecting (or simply missing) scenarios that can cause system collapse. In our experience performing audits, we have found just 10% to 20% functional coverage and four times over-testing of certain functionality to be the norm. In one team in particular, we found that three test cases covered just 5% of the system under test,[6] which is clearly not enough to provide the assurance that testing is responsible for.

Similar to user story creation, this manual, linear effort remains whether using spreadsheets or inputting tests into fields in project management tools.

Test Data Allocation

Once test cases have been defined, testers need data to execute them. Again, the first challenge is quality. Test data for many organizations equates to an anonymized copy of production data, which contains only a fraction of the possible scenarios that need to be tested. This data is sampled from past scenarios that have occurred in production and is, therefore, always out of date and sanitized to exclude bad data or unexpected results. Even if testers had tests to cover all the functionality that needs testing, they are not likely to have the data needed to execute them.

There is also the data provisioning time. As Figure 1 shows, data provisioning is often performed in an unsystematic manner that creates cross-team dependencies and time-consuming manual effort. At the same time, there is little or no visibility, reporting or tracking.

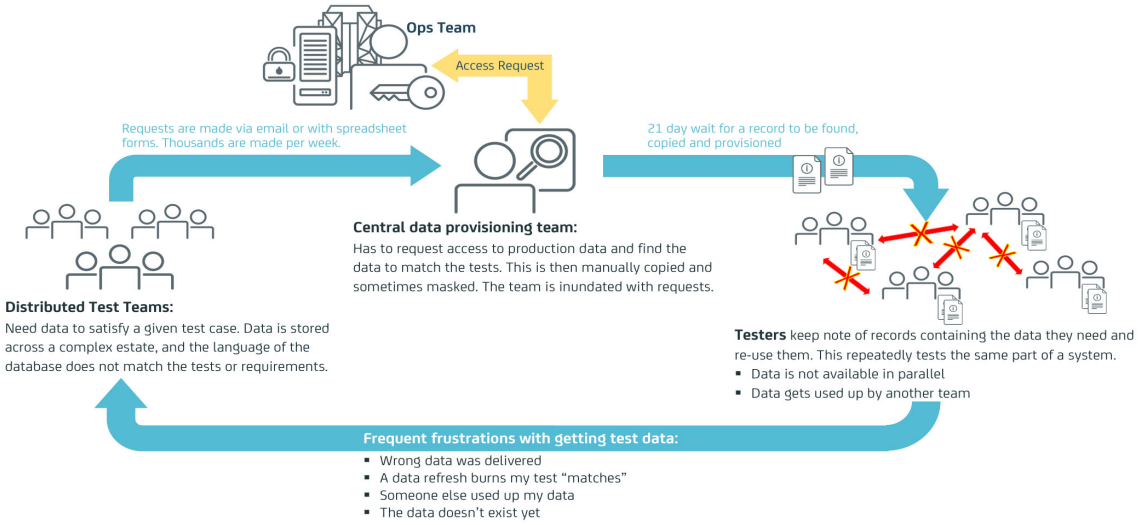


Figure 1: A typical data-provisioning process.

In this approach, testers request data in emails or by filling out rudimentary request forms in spreadsheets. A central team responsible for locating, copying, and provisioning the data into test environments then processes these requests. This team is often dependent on another team that controls access to the potentially sensitive production data, creating a further dependency and even greater room for delays.

6. Huw Price, CA Technologies, "Test Case Calamity," May 29, 2014

The central team must find the exact data set from among complex production sources to fulfill a given set of test cases. These production sources are typically poorly documented, and the test data engineers often lack even semi-automated discovery technologies. The test engineers are further inundated with requests, with some teams expected to deliver tens of thousands of records a week.[7] A lack of re-usability means that, once fulfilled, the same request will often be repeated again and again. Wait times of weeks or even months are not uncommon.[8] In some instances, the time spent waiting for a data refresh can be longer than the planned iteration.

Once the data has been provisioned, testers often rely on common data sources. They are, therefore, left frustrated when another team overwrites their data or when the data gets lost during a refresh. Often, testers wait idly for another team to finish using their preferred set of data; and meanwhile, a code change made by another team might cause their tests to fail when there is no genuine defect in the code. In practice, testers find or create data themselves, from whatever sources are available. This is not a viable strategy and detracts from the time that testers should spend assuring that quality software is delivered on time.

System Constraints

Assuming testers have their test cases and the data to execute them, they need production-like systems to execute them against. However, they frequently rely on a limited number of shared environments, which take weeks to configure and require costly preproduction infrastructure.

Moreover, testers often require access to the unfinished, back-end or third-party components in a composite application. These components might be in use by another team, again leaving testers waiting for them to become available. Many organizations use service virtualization to simulate unavailable or constrained components, but this requires realistic and consistently defined virtual data. Defining the complex request-response pairs accurately can create significant overheads. Record and playback might be used, but this is only possible when a service already exists, and recorded data might not cover the future scenarios needed for rigorous testing. Scripting offers an alternative to defining complex scenarios manually, but this is likewise slow and complex.

In spite of all this time and effort, there is often no guarantee that testers will have environments in which to execute every test needed for rigorous testing. The virtual data will typically only cover a fraction of possible scenarios. Defects are then detected late while the ultimate user experience suffers. This approach to provisioning environments stands in stark contrast to the speed and quality expected of continuous testing.

Test Automation

Executing every test manually is too slow and is simply not an option when regression testing and releasing software in short iterations. However, automating test execution alone is not a silver bullet for testing bottlenecks. Automation does not increase test coverage and usually introduces the effort of repetitious, complex script definition or keyword selection. Often, this time can outweigh the time saved on test execution, replacing one bottleneck with an even greater one.

Moreover, testing needs to move beyond the UI layer, including APIs and becoming more data driven. Automated testing must therefore start much earlier in the application lifecycle and cannot wait until the UI is in place before starting.

Access to Dependent Systems and Test Environments

A further challenge is executing the automated tests. This brings us back to the issue of having available systems to execute against and depends on the ability to quickly release a system to test environments.

7. Alfredo Glower, Orasi Software, “Test Data Management—The Manheim Case Study,” Feb 25, 2016
8. CA Technologies case study, “Manheim Accelerates Systems Testing and Business Transformation with CA Test Data Manager,” 2016

The problem is that multiple teams contribute to software delivery, each using different technologies. This includes file management systems, version control, continuous integration tools, databases and more. These all need to be aligned before code is moved along, while any additional infrastructure, test databases and virtual services then need to be deployed into the test environment. Often, complex manual scripting is used in an effort to connect the disparate tools, but this is highly time-consuming and requires constant maintenance.

A lack of standardization further risks errors in deployment and more defects making it to production. Sometimes, testers persevere with an out-of-date version or build because they cannot wait for a deployment.This is equivalent to not testing the latest functionality because any tests written since the last build was deployed will either not find defects or will throw up test failures where there might be no fault in the latest code.

"*GUI testing is slow and notoriously difficult to maintain due to changing user interfaces. Developers and testers lower reliance on GUI testing by focusing on a layered approach instead*."[9] —Gartner, Use Layered Testing to Enable Continuous Delivery of Digital Business Capabilities

Performance Testing Remains Piecemeal, In-House, and Incomplete

Today, slow-performing applications will simply not succeed. Fifty-three percent of sites will be abandoned if a mobile site takes more than three seconds to load[10]. Organizations need to design and build applications with a view to minimal or no degradation at peak load from day one.

Legacy tools and techniques mean that performance testing cannot be performed early enough, nor rigorously enough. These outmoded ways of testing cannot scale to the millions of users required to realistically test peak stress performance, nor are they sophisticated enough to reflect actual user behavior. They are not suited to modern applications, because performance and load tests rarely extend beyond the services and infrastructure within the organization. As a result, performance testing is neither rigorous nor realistic enough to guarantee that an app and its third-party integrations can perform quickly enough for experience-driven users.

Meanwhile, most larger organizations route all performance testing through a center of excellence (COE). The COE has a few specialized performance engineers who hold the keys and schedule for the testing infrastructure. As a precious resource, this team cannot help but function as a bottleneck, because performance testing is carefully doled out to only the most-critical projects. This creates a high degree of friction, because engineering teams have to transfer knowledge and wait for tests to be developed, while test coverage is typically low.

An Inability to React to Change

Continuous testing depends on the ability to build seamlessly on the effort of past iterations when something changes. Change requests today are rarely captured formally, but rather through a continuous barrage of email requests and fragmented user stories. When a change request comes in, developers and testers must first identify how it will affect the numerous, interdependent components of a system.

Static requirements and a lack of dependency mapping mean that this is usually a manual, error-prone task. Dependency mapping relies on the knowledge of individual teams, but no one developer or tester can have subject matter expertise of the relationships between every component in a heterogeneous system. A low-priority, innocuous-seeming change might have a huge impact system-wide, requiring weeks or months of a rework. Numerous high-profile outages point to the danger of unforeseen consequences following a change.

9. Sean Kenefick, Gartner, "Use Layered Testing to Enable Continuous Delivery of Digital Business Capabilities," Nov 14, 2016
10. Alex Shellhammer, DoubleClick by Google, "The need for mobile speed: How mobile latency impacts publisher revenue," Sep 2016

Then there is the time required to update existing test and development assets, before creating any new ones needed to implement the change. Because test assets have been derived manually from requirements and are stored in formats different to the requirements, they are not traceable back to the requirements. They must, therefore, be updated manually, which is highly time-consuming. Sometimes, every test is checked and updated by hand; at other times all the existing tests are burnt and created from scratch. In some instances, testers let all their existing tests pile up in an effort to retain coverage, but this leads to wasteful over-testing with invalid tests causing test failures.

"*It was taking two testers two days to check and update test cases after a single change was made*."—A large financial institution[11]

The data and virtual services must then be updated or created for the new test pack, before a new system is deployed to the test environment. This repeats much of the laborious effort set out above. If testing is going to keep up with the rate at which user needs change, existing assets need to become more reactive to change. The effort of previous iterations must be built upon, rather than repeated, allowing teams to focus on new development work and innovation.

The End-to-End Continuous Delivery Ecosystem

For software delivery to become truly continuous, a holistic approach is required to root out and eliminate constraints in testing. Disparate tools and processes need to be joined up, from design and planning, through to release and deployment. Only then can we avoid the wasted effort and the sacrificed quality caused by going from one format and tool to another, while the otherwise linear stages of development are collapsed into a truly parallel approach. The defects introduced by poor visibility and a lack of cross-functional collaboration across the lifecycle can be reduced, integrating continuous testing within a continuous delivery pipeline that is capable of taking an idea from design to production at speed, without compromising quality.

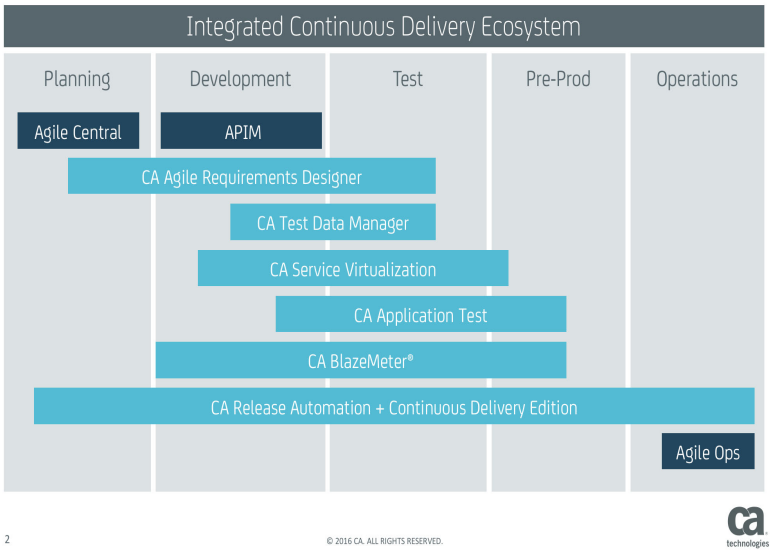


Figure 2: The parallel stages and technologies in an end-to-end delivery pipeline.

A continuous delivery ecosystem capable of continuous testing should include the following aspects:

- Active requirements gathering to capture the desired user functionality accurately and iteratively, before using the design to derive and maintain test and development assets.
- Optimized, automated test design to create the smallest set of test cases needed to validate that the desired user functionality (that was captured in the requirements and user stories) has been delivered.

11. Huw Price, CA Technologies, "Test Case Calamity," May 29, 2014

- Self-service data provisioning, to find, secure, or create the data needed to execute every test case and to allocate it in parallel and on demand systems.
- Service virtualization using comprehensive, accurate virtual data to create affordable, production-like systems in which to execute every possible test.
- Automated maintenance and traceability so that when the design changes, the user stories, test cases, data, and virtual endpoints can all be updated at the same time.
- Accurate, automated test execution that is easily managed across all systems, APIs, and services.
- Full orchestration and release automation to align all the moving parts and seamlessly deploy, test, and release code.
- A centralized repository of up-to-date assets and tasks to enable cross-functional collaboration and project-wide visibility.
- Performance tests which closely mimic actual user behavior that can be scaled beyond the level of stress possible in operations, and that can be executed by anyone, at any stage of the development.

Accurately Capturing Changing User Needs

Rally Software lets business analysts quickly record new user stories and changes, providing a central repository for the testing and development assets that are needed to deliver the functionality. The status of the interdependent stories and their associated tasks can then be tracked easily and visually, with central dashboards, roadmaps, and backlogs. Meanwhile, cross-functional teams can communicate and collaborate more closely, with project-wide visibility.

The value of the user story does not stop there; the steps of the user stories will provide the basis for the creation of subsequent test and development assets. These are dragged from Rally Software and dropped as blocks to a flowchart model in Agile Requirements Designer. Stakeholders with critical modeling skills and subject matter expertise can then connect up the user stories and build additional functional logic around them.

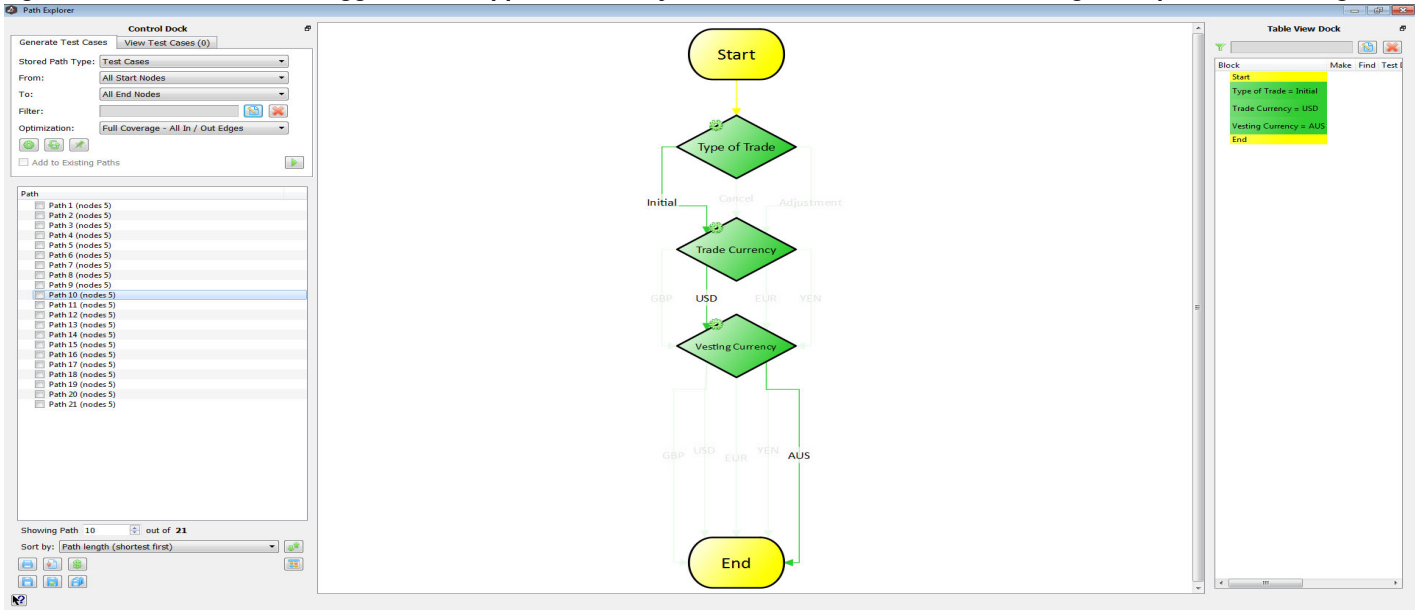
The goal is to create a system model that is as complete as possible, connecting the overlapping logic of the otherwise disparate user stories, and developing an up-to-date model of the system's components and their dependencies. To support this collaborative process, Agile Requirements Designer provides validation algorithms to identify missing or broken paths. In addition, the model can be visually verified with users for design-phase quality assurance.

This works towards a complete, unambiguous model, breaking the system down into its core cause and effect logic. Further, sub-flows can be used to allow subject matter experts to model individual components in detail. These sub-flows are then incorporated into a higher-level master flow, enabling greater collaboration, reducing silos, and allowing stakeholders to drill down into as much detail as required to fulfill their roles.

"*In Agile Requirements Designer, requirements are graphically depicted in a flowchart: a simple, easy way to read and understand a system's logic, with clear paths through it.*" —Test data engineer, a.s.r.[12]

12. CA Technologies case study, "CA Agile Requirements Designer at a.s.r.," 2016

Figure 3: User stories are dragged and dropped from Rally Software to a flowchart model in Agile Requirements Designer.



Creating and Maintaining Test and Development Assets Directly from the Design Requirements

A flowchart model provides a single point of reference for creating design, test, and development assets. These assets can be updated as the requirements change, thereby allowing testing to become continuous, and they can be exported out to the relevant tool or team.

User Story Creation and Allocation

Flowchart models created in Agile Requirements Designer can be exported to user stories in Rally Software, providing developers with a clear, complete specification of the desired user functionality that needs to be developed. For example, the product owner creates a user story in Rally Software. The person responsible for testing that user story (the Modeler) can bring it into Agile Requirements Designer in order to model it and add details (such as multiple flows, paths, alternate and negative scenarios, acceptance criteria, a definition of done, and so on). Lastly, the textual version of that completed user story is exported back into Rally Software and incorporated in the product backlog.

Maintenance of the user story is done in Agile Requirements Designer, although, if someone else on the team edits something directly in Rally Software, the connector between Rally Software and Agile Requirements Designer would flag the difference between the two products. The Modeler, the person responsible for testing the user story, would be able to synchronize both and also automatically identify the impact to all downstream artifacts. This way, the test model becomes the single source of truth, and from it, Agile Requirements Designer auto-generates all artifacts needed by all the teams (business, development, test, and ops). This is a fantastic representation of shift-left testing.

Optimizing Test Case Design

At the same time, test cases can be generated from the flowchart, by virtue of it being a formal representation of the system logic. Agile Requirements Designer provides automated algorithms to identify every path through the flowchart, which are equivalent to test cases. If a complete system model has been established based on all available information, these tests will cover every negative path, outlier, and unexpected result possible. This coverage eliminates the time wasted on manual test design, while ensuring the accuracy of testing.

"*In a little over a week, all the processes had been documented, with the accompanying test cases generated, and all with 100% test coverage. All test cases had been executed within just three days of further work*." —Test data engineer, a.s.r.[13]

Often, there is neither time nor resources to execute every possible test, even with automated execution. Optimization algorithms processes are, therefore, provided to reliably reduce the number of tests while still covering either every block, edge, pair of edges, or in/out edge. This approach consolidates the overlapping logic of tests, avoiding wasteful over-testing. In one instance, 326 tests were reduced down cases had been executed within to just 17; for more complex systems, tens of thousands of tests will typically be reduced to thousands or even hundreds.[14]

Risk-based approaches can also be used to prioritize and to execute the most critical tests based on the available time and resources. The probability that users will exercise certain paths can be reverse-engineered automatically from production data, defining risk thresholds empirically and reliably. Alternatively, risk values can be manually assigned to each block in a flowchart, before identifying every path above a certain probability.

The optimized or exhaustive test set can then be exported back out of Agile Requirements Designer and allocated to available teams in Rally Software. Test cases are attached to user stories, with all of the relevant test steps needed to execute them, while auto-synchronizing sprint boards can be used to update the status of tasks from within Agile Requirements Designer for better cross-team visibility.

Matching Test Data

Agile Requirements Designer and Rally Software integrate with Test Data Manager, introducing complete, end-to-end test data management to test creation and management. Test matching can be set up in Agile Requirements Designer to mine the data needed to execute the optimized test cases as they are created. If no data exists, synthetic data generation can be used to identify and create it. This can be performed on the basis of an accurate data profile created in Test Data Manager or by using data variables defined at the flowchart level in Agile Requirements Designer. If production data needs to be used, Test Data Manager offers several native masking engines to identify and remove sensitive information quickly.

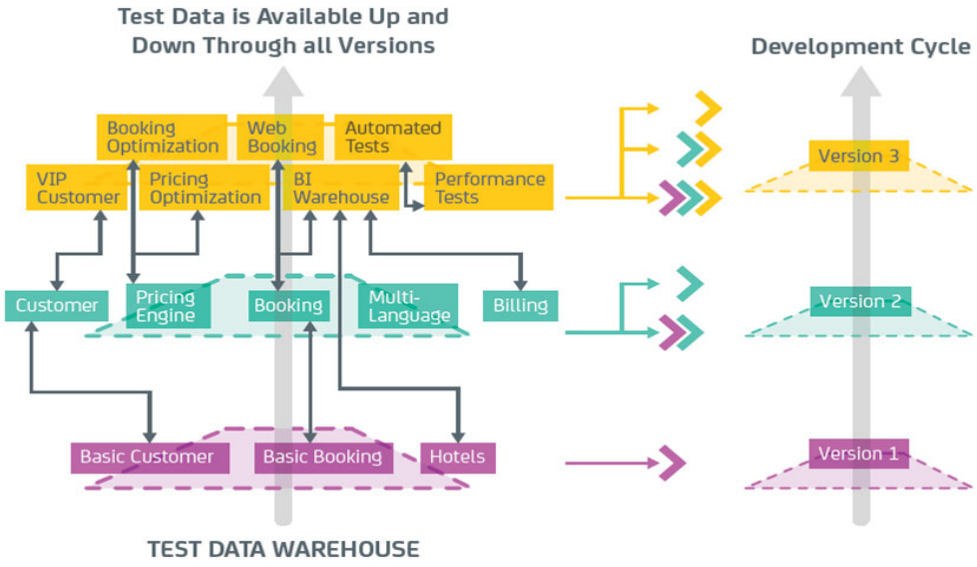
"*With CA's Test Data Manager solution, we reduced our time to get test data from 21 days to two days, making our entire software delivery cycle more efficient.*" —Director of quality assurance and strategy at a financial services company[15]

The data is stored as reusable assets in a test data warehouse, along with the queries needed to automatically find it. From there, it can be pushed out and attached to test cases in Rally Software, while testers can request data on demand using a self-service web portal.

13. CA Technologies case study, "CA Agile Requirements Designer at a.s.r.," 2016
14. Huw Price, CA Technologies, "Test Case Calamity," May 29, 2014
15. Anish Shah, Forrester Research, "The Total Economic Impact™ of the CA Technologies Test Data Manager Solution," December 2015

If data has been requested once, the request can be fulfilled automatically in the future. Data constraints between test teams are thereby eliminated, as are dependencies of a central provisioning team. Testers are provided with the data that they need, when they need it, as required for rigorous, continuous testing. Data is further cloned as it is delivered, so that it can be used in parallel. Meanwhile, powerful version control means that multiple releases and versions can be developed in parallel, using the most up-to-date data. Rare and interesting data can be reserved, to prevent it from being disrupted by another team.

Figure 4: Data is stored in a central test data warehouse and is made available across versions and releases.



Creating Production-Like Test Environments

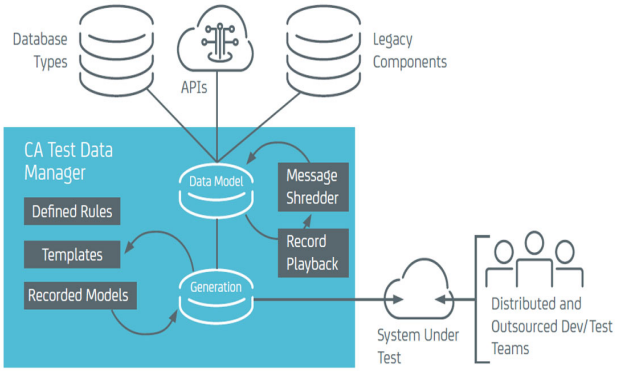
The growing complexity of application architectures, along with globally distributed organizations, means that development and testing teams face bottlenecks and constraints on the way to delivery. Given all the interdependencies involved, these teams have to contend with labor-intensive efforts. They either have to generate a copy of a complete environment in a development lab or test lab, or they need to build their own code in order to replicate the behavior of a dependent system or environment. The intensive costs and time constraints associated with getting access to the right environments has led to the rise of service virtualization.

Service virtualization is the practice of simulating the behavior, data, and performance characteristics of a dependent system that you can then use at all stages of development and testing. In contrast to stubs and mocks, virtual services are created by recording actual services by contract or by sample requests and responses. Virtual services are more dynamic and do not require a developer to write them. Using service virtualization, you can test for a virtually unlimited set of test cases and scenarios whenever you want. You can set up tests to verify all independent services and endpoints, not each in a vacuum, but all together. Service virtualization frees you from call limits so you can keep testing and testing and testing until your app is completely debugged.

If components of a system are unfinished or constantly unavailable, the virtual services required to execute the tests can be accurately defined and maintained. The request-response pairs needed to simulate the constrained components are modeled as a sub-flow, which is then embedded within the master model of the system under test. As optimized test cases are generated, a definition of the virtual service is included as steps of the test case, accelerating the creation of production-like systems in which to execute every test.

An accurate set of request-response pairs that covers every test case can then be created and injected into a deployed virtual service in CA Service Virtualization. To accelerate the otherwise complex definition of the data scenarios, JSON and XML files can be imported, creating an initial flowchart model of the service or message. A reusable configuration file is then applied to the flowchart, mapping snippets of virtual data to blocks. This data is compiled as the complete test cases are generated.

Figure 5: Accurate virtual data is generated on the basis of a message definition and is injected into a deployed service.



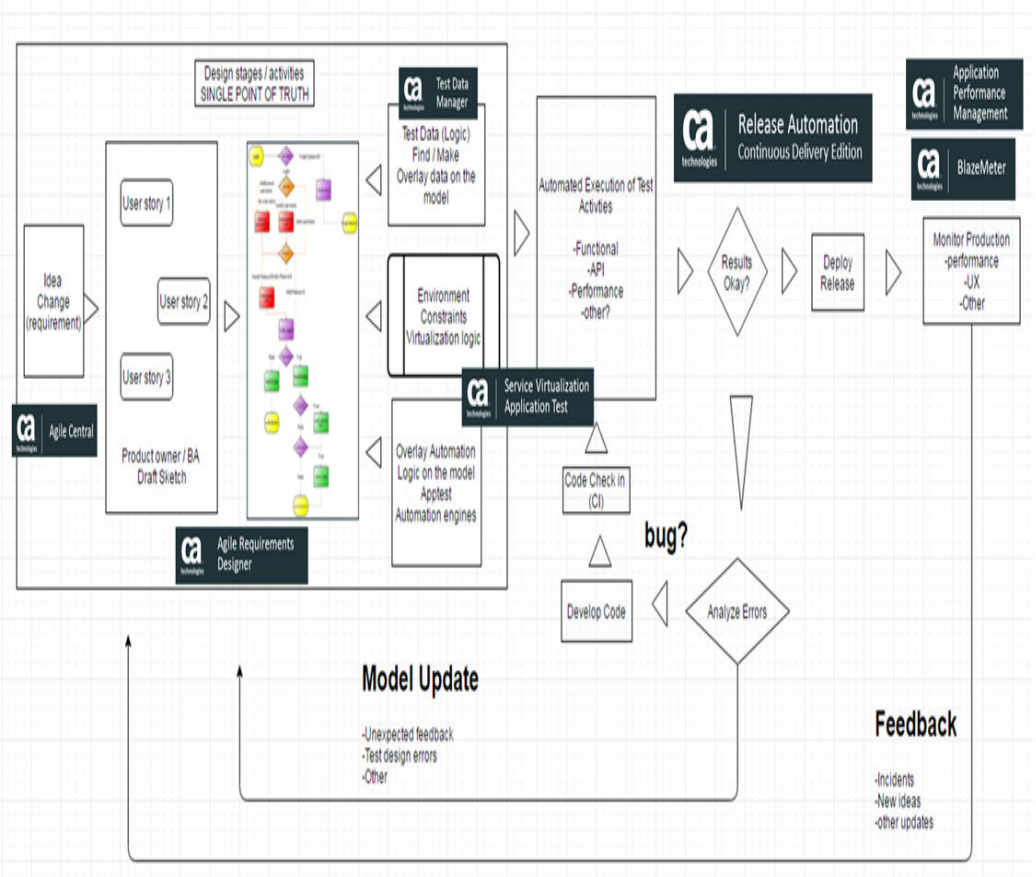
Alternatively, Test Data Manager can be used to generate a set of request-response pairs that cover every possible test scenario, injecting them into a deployed virtual service within CA Service Virtualization. This provides a rich seedbed of virtual data, which can then be exposed to teams on demand from the Test Data On Demand portal within Test Data Manager. Testing one component does not need to wait while the whole system is built; instead, you can iteratively test and improve a system from day one.

Setting Up and Executing Tests

At the point that you set up and execute tests, the desired user functionality has been used to define the test and development assets. Optimized test cases have been created at the same time as the test data and virtual services needed to execute them, eliminating the bottlenecks that prevent truly continuous testing.

Furthermore, the assets are stored in one place, the flowchart, matching the tests to the relevant data and virtual service.This process is collaborative, beginning with the user and involving requirements gatherers, QA, development, test teams and even the ops teams, all whom are responsible for the quality of their work. Equipping these teams with the tools needed to work effectively from the same page is one thing. Their collaborative effort must then be aligned and executed. This is where standardized orchestration and cross-layer test automation execution comes in.

Figure 6: A collaborative process, utilizing an integrated, continuous-delivery ecosystem.



Automating Test Execution

The optimized tests stored in Rally Software can be turned into an automated test suite in Application Test, using a visual workflow and easy-to-use web editor to accelerate the conversion process. As the tests have been optimized down to the smallest number possible, the manual effort required of automated engineers has also been significantly lowered. They know exactly what tests need to be automated to test the latest user needs, avoiding the time wasted on repetitious, linear script definition or keyword selection.

"*We've already created and switched over 10,000 test cases from manual to automated execution. Our QA team can now focus more on complex manual test cases.*"—IT Central Station, Real User Review[16]

With Application Test, the automated tests are executed in a framework designed for modern, heterogeneous applications. Application Test provides a single platform to execute the tests across systems, APIs and services. Test execution can be performed cross-browser and on mobile, and can use real devices, simulators, labs, or cloud platforms. Tests can further be strung together; and the results of one test can be used to feed the next test. This approach helps overcome the complexity of modern applications and infrastructure, making it possible to execute tests that cover all known application logic within a short iteration.

SunTrust bank achieved a 22% cost savings of its managed services. This involved $1.97 million savings due to automation and $900,000 direct savings.[17]

16. IT Central Station, "I Like The Support To Build Modular Test Cases Using Reusable Blocks," February 2, 2016
17. CA Technologies and UBM, "Enterprise IT leaders share their personal continuous delivery experiences," Oct 27, 2016

Meeting User Performance Demands

Continuous testing in the deployed QA environment does not stop with functionality. It must also validate that an application is of high-enough performance to satisfy experience-driven users. This is where CA BlazeMeter® comes in, allowing users to create and attach performance tests to user stories or test cases in Rally Software. The models created in Agile Requirements Designer, which describe the behavior of the application, can be used to derive performance test scenarios/ business functions and automated scripts (such as JMeter, Gatling, Taurus, and so on) according to the performance testing scope. CA BlazeMeter tests can be triggered as part of a deployment using CA Release Automation.

CA BlazeMeter provides a self-service SaaS tool, enabling users to run performance tests at any stage of the development lifecycle. No installation and minimum setup is required to test both mobile and web applications, while performance and load tests can be executed using virtual users on the cloud and global data centers. This avoids prohibitive infrastructure requirements, and applications can be stressed far beyond what is possible in operations.

Accurate stress tests can easily be scaled to the millions, with real mobile and web traffic loaded in to ensure that they mimic user behavior. If user interactions have already been defined in Selenium tests, these can be leveraged in combination with open-source tests like Apache JMeter, working to ensure that performance tests mirror user behavior accurately. Meanwhile, dashboards and integrations into application performance management tools allow users to closely monitor, maintain, and improve performance, using comprehensive analytics to deliver the functionality that users expect, with none of the frustrating wait times.

It is critical to run a CA BlazeMeter performance test with an APM solution running in the background. Otherwise, developers have to do manual performance tuning, by going through thousands of lines of text in logs scattered across countless servers to determine the root cause of performance problems, before they can actually fix it. Integrations with APM tools, such as CA Cross-Enterprise Application Performance Management (formerly CA Application Performance Management), can pinpoint the culprit precisely at the method level, reducing mean time to repair (MTTR), which is the whole point of shifting performance testing to the left.

Deploying the Latest Build When Testers Need It

When you use the Nolio Release Automation (formerly CA Release Automation) deploying engine, testing heterogeneous applications is as easy as lining up the up-to-date assets and hitting play. The collaborative effort invested in Application Test, Test Data Manager, CA BlazeMeter, and CA Service Virtualization can be completely leveraged. The virtual services can be spun up and the rich test data plugged in before the automated functional and performance tests are exercised on a readily deployed system.

The exact release process required can be defined once, before being reused by the core automation engine. The CA Continuous Delivery Director is tailored for multi-team, multi-application releases, providing the tools needed to identify and resolve dependencies. This means that the latest build can be deployed quickly and consistently, no matter how diverse the servers, data centers, and environments. Meanwhile, test and development teams can focus their effort back on delivering innovative software that reflects the desired user functionality, integrating continuous testing within a continuous delivery pipeline.

Reacting to Changing User Needs

Organizations are waking up and realizing that their business requirements must align exactly with user needs. However, users' desired functionality changes quickly. These changing needs must be captured iteratively and accurately, leveraging insights from the design phase right through to operations.

In the approach set out in this paper, the flowchart serves as the requirements and has been used to define the user stories, test cases, test data and virtual endpoints. A change made to the requirements can, therefore, be reflected quickly and accurately in the subsequent test assets, significantly reducing manual maintenance, and allowing each iteration to build on the effort that has come before it.

The flowchart model provides a full dependency map of the system's moving parts, allowing the impact of a change to be identified across the system. When a new piece of logic is added to the flowchart, Agile Requirements Designer will identify which dependent master flows and sub-flows have been impacted. This allows business analysts and developers to evaluate how much work a change will generate, weighing up the relative value of user stories accurately. Further, following a change, developers know exactly which parts of a system need to be updated, avoiding the potentially catastrophic damage caused to software by unforeseen consequences.

The impact of a change on existing paths is also identified, so that the test cases, data, and virtual end-points can all be maintained in tandem. The Path Impact Analyzer feature will remove or repair any broken or invalid paths following a change, while any new tests needed to validate the latest version of the requirements will be created.

A large financial institution reduced the time taken to update test cases from two days to five minutes. A large financial institution[18]

The new test cases can then be synchronized with the existing Rally Software user stories, communicating to automation engineers exactly which tests need to be updated or created. Testers no longer have to check and update every existing test, and they can automate only the number of tests that are needed to validate the new functionality. With the data and virtual data updated at the same time as test cases, the functional tests are ready to be kicked off again using CA Release Automation. The lightweight, SaaS platform offered by CA BlazeMeter further enables performance testing to be executed at every stage of the development process, even as software evolves. Rigorous testing is made possible within a short iteration, building off of the effort of past iterations.

Conclusions

User needs are changing fast, but the expectations of these experience-driven consumers are set higher than ever. Organizations need to be able to continuously deliver software that accurately reflects constantly changing user needs. Business and development teams must be able to iteratively and accurately capture the desired user functionality, before reflecting this in software and test assets. All of these components must further be kept aligned, from user stories to test cases, test data, virtual services and the deployed system.The best way to achieve this is to drive the assets directly from the requirements themselves. This is possible with continuous testing, integrated into an end-to-end continuous delivery ecosystem that's capable of taking an idea from design to operations at speed, without compromising quality.

Next Steps

For more information, please visit blazemeter.com.

18. Huw Price, CA Technologies, "Test Case Calamity," May 29, 2014