Ranorex | Ebook

UPDATING TRADITIONAL TEST PLANS FOR A DEVOPS ENVIRONMENT

Updating Traditional Test Plans for a DevOps Environment

Continuous testing requires a well-orchestrated pipeline with a variety of testing types, tools and frameworks. Teams can be ad hoc or take a laissez-faire attitude about testing; however, if they truly want to trust the value of the software they are delivering, they should actively develop and maintain a strategic test plan.

The journey to a team's successful agile test planning begins with understanding the test phases within a DevOps delivery pipeline. The test plan for a continuous testing environment must be lightweight, flexible and aligned to the boundaries of the phases in the pipeline.

The shift-left movement still requires critical thinking when it comes to test strategy and planning. Let's explore what a modern, lightweight approach to a test plan can look like in a DevOps environment.

TRADITIONAL TEST-PLANNING PRACTICES

First, though, we should understand the foundation established by traditional test-planning methods. In this context, traditional testing is generally classified as waterfall in nature, where quality takes precedence over speed to market. Traditional testing can also be viewed as agile, yet there is a defined phase or phases where testing is not fully automated. Often there is a reasonable dependency on manual testing and iteration.

A test plan is designed with the system under test in mind, and the rigor increases if the product is subject to compliance or regulation. Traditional test plan methodologies derived from the Rational Unified Process (RUP) or Capability Maturity Model Integration (CMMI) suggest deep layers of documentation, which is supported by the ISTQB definition of a test plan.

Teams adopting RUP or striving for CMMI certification navigate a complex world of expected artifacts to build a comprehensive test plan. Initially, RUP was designed to be a flexible framework, but somewhere along the software development journey it became prescriptive.

In the context of traditional approaches to software delivery, a test plan requires many supporting artifacts. The approach may start with a master test plan and then a series of incremental test plans for each software iteration. Teams are constantly updating and maintaining a high level of documentation. Within RUP, a test plan has foundations of the requirements for testing, a risk assessment and the actual test strategy.

The test strategy breaks down even further into the types of tests and their objectives. Functional testing covers unit, integration, system and acceptance testing. Nonfunctional testing covers performance, reliability and accessibility. The documentation includes the detailed techniques leveraged for each testing type, the mechanisms for measurement and any special conditions that may be required, and the stages where the testing takes place may live in yet another document. Requirements are matrixed to this documentation.

The complexity of traditional testing is illustrated by dissecting one artifact within a test plan: the test case. A test case is built using these attributes:

> Test case name
>
> Test case description
>
> Pre-conditions
>
> Test inputs
>
> Observation points
>
> Control points
>
> Expected results
>
> Post-conditions

The process for critically thinking about these eight attributes and writing them into documents can be arduous. There are many test-planning applications available with the intent of injecting speed into this documentation process.

But in an agile world, working software and the adaptation to change is highly valued over comprehensive documentation. An agile mindset based on lean principles inspires a more nimble test plan embedded in modern testing methodologies.

MODERN TEST PLANNING

In contrast, DevOps testing today moves at great speed, and there is little time to maintain detailed artifacts such as the test strategy and the test cases described by RUP. DevOps approaches to a test plan embrace the tenets of agile.

Modern testing is development and quality being craftily folded into a continuous testing and continuous delivery process. Traditional test planning can be leveraged for DevOps development lifecycles, but modern testing practices require a lighter touch.

A test plan in a continuous testing environment at a minimum contains the strategy, which is typically found in a collaborative platform such as a wiki, and the automation code, which defines the atomic test cases in the form of scripts.

Teams may leverage a test case name and a short description, leaving out all of the other detailed attributes described above in RUP. The test details are ideally written and discoverable within the application code, in a lightweight visual tool, or in an effective digital agile test-planning solution. Knowing the pre-conditions and being able to assert on expected results remains critical to automation, but code tells the story here, versus comprehensive documentation as within traditional test planning.

The environments are well established and orchestrated for a DevOps testing model. Continuous testing is expressed within the software delivery pipeline. The pipeline defines the phases of testing, and the test plan aligns to those defined phases. The testing gets executed during the building and delivery process.

When crafting an agile test strategy with rapid and effective delivery in mind, the team needs to consider a lean and lightweight test plan that balances all phases of testing. Balance is critical to maintaining the desired delivery velocity.

TESTING PHASES IN A DEVOPS PIPELINE

A standard pipeline for high-performing agile teams traverses through unit, API, security, performance, accessibility and UI testing methodologies.

In a continuous testing world, all phases of the pipeline are optimized for speed and efficiency. Pipeline efficiency is dependent on crafting a comprehensive test plan, which will create a shared understanding for the delivery teams of the testing that will take place as code is checked into version control. At a minimum, the wiki strategy document will define these phases and the technology to be used for testing each phase.

Unit

Unit testing applies focus to a specific unit or a component, such as a class. The objective is to validate that the function was correctly implemented. High-performing agile teams use discipline to test each unit of code they create as they code.

Teams can leverage test-driven development (TDD) to build unit tests. A strong delivery pipeline surfaces code coverage metrics to illustrate compliance to a standard as set by the team. The code coverage standards are defined in the test plan, but due to the nature of unit testing itself, it doesn't need a test strategy.

Integration

Integration or API testing, on the other hand, require a testing strategy. Integration testing evaluates one unit in isolation or multiple units working as a group. In theory, every API should be thoroughly tested, but some endpoints are more important than others. A common test plan methodology for the API layer is based on a risk analysis. The API portion of a test plan considers business impact to determine how comprehensive an API should be tested.

Many teams leverage behavior-driven development (BDD) to craft their API tests. Continuous testing is dependent on test automation, so the details of test cases live in code. The DevOps test plan connects the strategy to the code by links to the repositories, and many organizations today use BDD to put test cases into readable code. BDD involves acute test planning, as described in this template, that requires close collaboration between product and engineering.

Agile teams also consider contract testing (such as Pact.io) in order to confirm that the producers consistently meet the expectations of the consumers. Contract testing is a strategy where the team of the consuming service writes the tests for the team building the producers. Contract testing requires a test plan and potentially a working agreement between the teams to effectively deliver against a common test plan.

Other nonfunctional phases

Security, performance and accessibility typically require special tooling integrated into the delivery pipeline. Each tool will require efficient orchestration, so test planning is required. The test plan outlines what tooling will be used for these nonfunctional testing outcomes. The test plan defines the tools and sets the engineering standards for delivering code to production.

User interface

Teams testing a user interface (UI) implementation will require unit testing and functional browser testing. Unit testing the UI is expected, and there are several tools and frameworks (such as Jest, Mocha and Chai) to accomplish this. Functional tests or cross-browser testing can have too many permutations to test all things. Additionally, testing in the UI can be the most fragile part of the pipeline.

The team has the goal of efficient delivery of value to the customers, so it is important to balance the speed of the UI tests. The tests must be atomic so that, for efficiency, they can be executed in parallel and keep the DevOps pipeline timely. There are many frameworks to inject into the testing strategy for UI cross-browser verifications, including Selenium, Watir, Puppeteer and Cypress.

TEST PLANNING FOR MANUAL VS. AUTOMATION PROJECTS

Test plans and the methodology for generating them are the same for manual and automation projects. Teams executing manual tests are prioritizing their work using a test plan. Teams focused on automation also need an agile testing strategy in order to enable continuous testing. However, there is a stark difference between teams following traditional test methodology and teams embracing DevOps and continuous testing practices.

The test plan defines whether or not manual testing is part of the delivery lifecycle, and the test plan is explicit as to when in the delivery lifecycle the team will pause to execute the manual portion of the test plan. DevOps and continuous testing environments strive to minimize the amount of manual testing, but when needed for customer-centric quality, the manual testing is conducted by the entire delivery team.

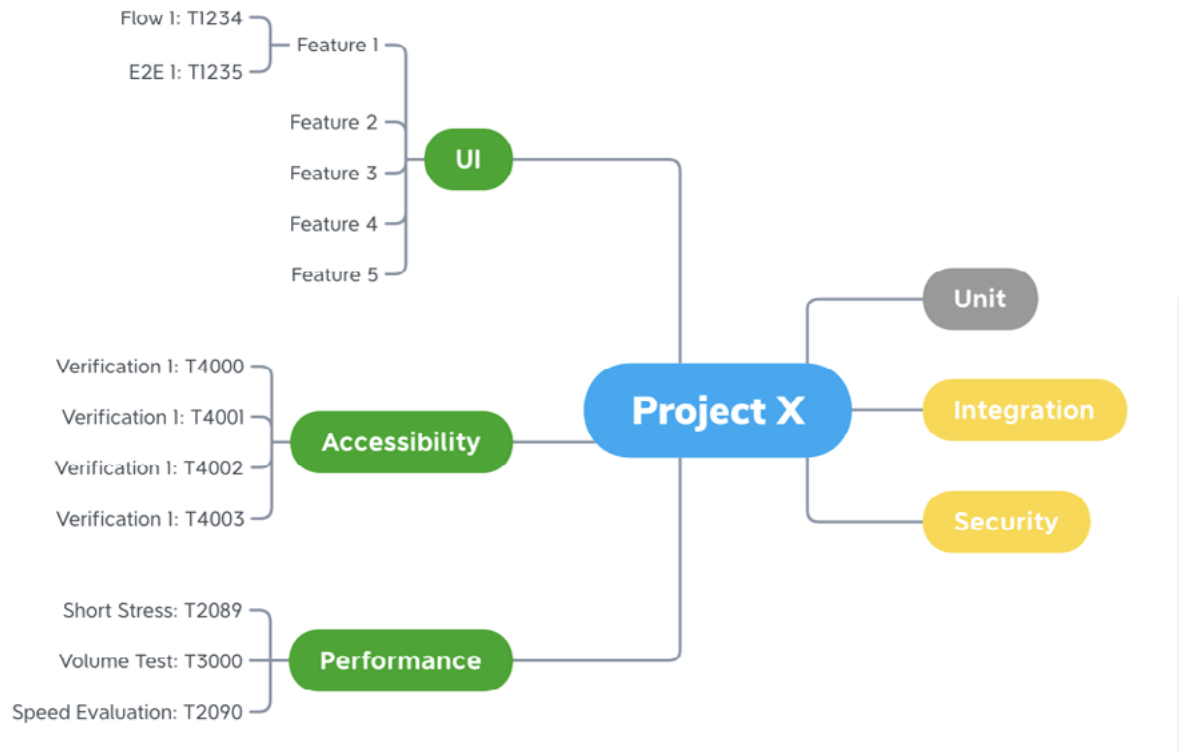
A conversation that constrains DevOps test planning is the ROI on automation versus manual testing. Test planning practices traditionally rely on units of testing captures in terms of test cases. Organizations that are constrained by this debate on the journey to continuous testing and that have well-documented test cases with a history of results can leverage this tool to illustrate the value for the investment in continuous testing. In an organization focused on continuous testing, ROI is far less important than having a balanced and well-thought-out testing plan. In this context, teams are all in on automation. However, ROI calculations might be valuable for organizations that desire to transition from a traditional testing practice to a DevOps model supported by continuous testing.

USING MIND MAPS FOR AGILE TEST PLANS

A lightweight agile testing strategy for DevOps may include a mind map to collate testing ideas and gaps. Mind maps are a visual tool that any extended member of a team can use to expand or reduce the scope of the flexible test plan, to move testing around efficiently based on risk or exposed testing gaps, and to get a shared understanding of how testing will be organized within the delivery pipeline. Professional testers who work with agile teams and rapid DevOps delivery processes also advocate the use and the flexibility of mind maps.

An agile mind map can serve as the team's comprehensive yet flexible test plan. Mind maps can include reference to a test planning solution or direct links to the associated testing code files. In a testing mind map, a title and the test plan tool's unique ID may link a simple design to more detailed test cases.

Mind maps are also great for agile work because they can be changed quickly. The ease of use and speed allow the team to keep pace with delivery pipelines. Teams can lose velocity maintaining the cross-referencing to a test planning tool, but in certain contexts, this traceability is important for compliance:



A test plan crafted in a mind map can continuously be adjusted. For example, the team may only want to do monthly performance testing in the pipeline instead of every build. The engineers may want to swap accessibility testing for performance testing every other build. These choices can be made by the team and coded as such. High-performing teams may deploy 15 builds per day, and the CI/CD pipeline could be configured to be more comprehensive every fifth build.

In a continuous testing world, the test plan is critical to the process, but it cannot become a burden. A mind map is a great way to quickly get a shared understanding of the desired testing needed by the team.

DOCUMENTATION OF TESTS IN CODE

BDD can also serve agile squads as a test plan for a given feature. Capturing test plans as code fits into the DevOps delivery model, and using BDD tools like Cucumber can articulate user interactions in a clear, ubiquitous language. BDD frameworks also take advantage of labels to organize the feature testing. With a little bit of code, the automation test runners can generate an artifact that resembles a test plan.

Defining acceptance testing as code with BDD is the primary use case. BDD provides a structure and pattern for building automation tests. Injecting acceptance tests into the automation suite at all layers of the pipeline is an effective way for agile teams to obtain and monitor test coverage.

BDD in the frame of integration testing serves to document the expectations of API end-points that connect a complex system. The integration testing layer in modern testing is expected to include the largest volume of testing within a CI/CD pipeline, and the verification of connectivity between microservices is key to maintaining velocity.

End-to-end, or E2E, testing can also be described using BDD. E2E testing is extremely hard and time-consuming, so they can easily become a constraint to delivery. Using BDD, the team can capture manual testing efforts as code. The CI/CD pipeline can be paused when the team executes the manual verification flows that are documented in the BDD file. Additionally, with a mature delivery pipeline, canary builds or blue-green deployment can signal to the team to execute the manual testing phase of the delivery lifecycle.

Teams using BDD get a descriptive, readable testing format that gives the entire organization visibility into tests. To get started, use a test plan template that establishes sound BDD test scenario formats.

CRAFTING A CONTINUOUS TESTING PLAN

A DevOps-oriented test plan is flexible, lightweight and framed in the boundaries of an efficient software delivery pipeline. A mind map is used to explore testing phases, the test ideas within each pipeline phase, and atomic test cases as details are required. The test plan is documented on a wiki page or in the code repository as an artifact. The strategy on the wiki contains test phase definitions, the tooling, and the standards expected for high quality software to be delivered at speed.

For organizations delivering customer-facing software, strategic test planning is a sound practice. Whether the teams use traditional approaches requiring multiple detailed documents or an agile workflow focused on rapid software delivery, a test plan guides a shared understanding of the delivery lifecycle. Documents, test management tools, visual tools such as mind maps, and code expressed as readable scenarios together create test plans that unite teams around desired objectives.

Teams must design a technique that best suits them and iterate as the organization changes, but in the fast-paced world of DevOps, a nimble and flexible approach to test planning is required for the desired continuous testing environment.

Ranorex®

US Headquarters

10801 N Mopac Expressway
Building 1, Suite 100
Austin, TX 78759
USA

+1 512 226 8080

sales@ranorex.com