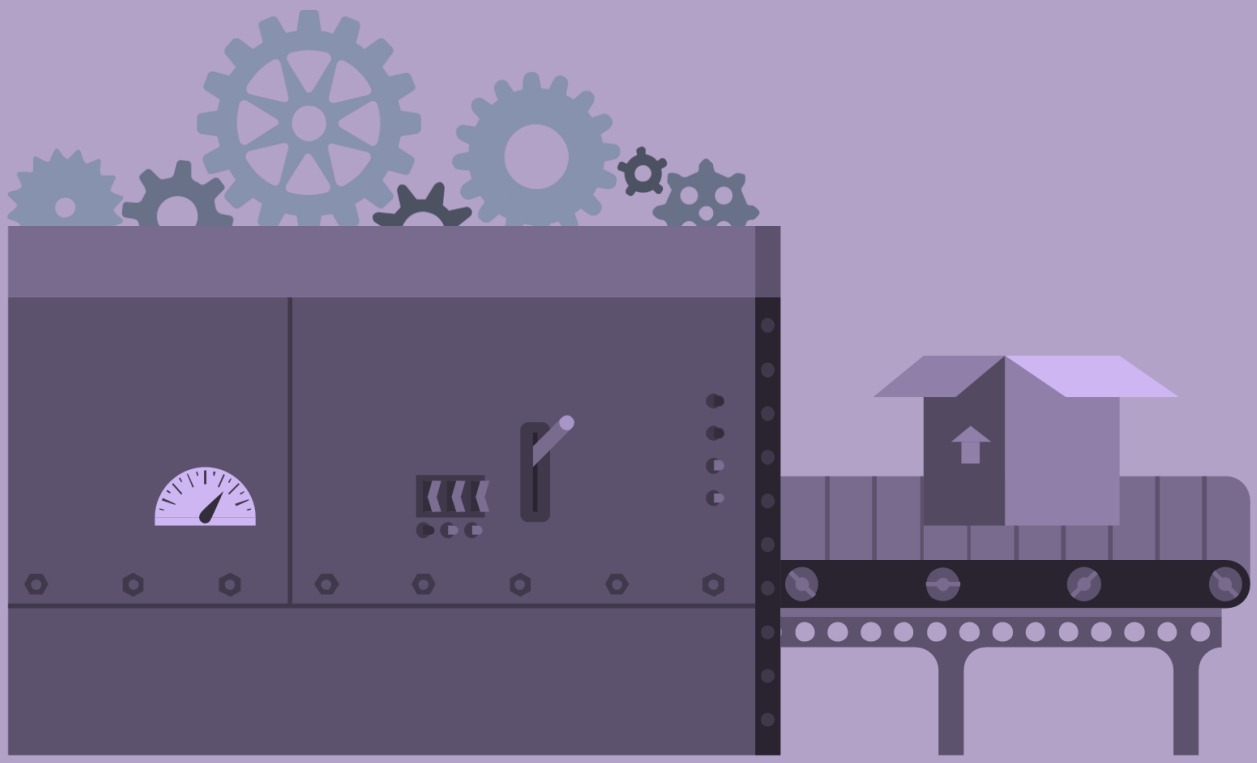
V1.0

Continuous Testing

A practical guide with concepts and approaches



Visual Studio ALM Rangers

Microsoft Visual Studio

The MIT License (MIT)

Table of Contents

Foreword

In this competitive world, everyone wants to react to customer feedback quickly and ship a high quality product. But how do you make continuous changes to your software without comprising on quality? The answer is test automation. Automated tests can measure quality on a continuous basis and help you decide if your product is ready to release—or not. When you know the quality of your product all times, you can confidently make a go, no-go decision during a release cycle. In the DevOps world, test automation is considered as one of the top five DevOps success factors that were presented at the DevOps Enterprise Summit 2015.

Some key questions come to mind for every manager before committing resources for test automation. Why should I automate? Isn't manual testing good enough? What is the ROI of automation? What should I automate? When should I automate? Finally, what are the best practices for automation?

As a Program Manager in the Visual Studio Test product group, I get these questions all the time from customers who are new to DevOps or thinking of investing in automation. Sometimes I send few bullet points to them, sometimes I understand a customer's specific areas and answer these questions. I found a great resource and wealth of knowledge in ALM Rangers and requested them to deliver a product that can answer these questions for all customers. They have great depth of knowledge in this area and work with real customers in solving some of these exact problems. This is a fantastic guide, built by ALM Rangers, with great details on automation concepts, practices, examples, and case studies. I strongly recommend developers, testers and leads/managers read this guide to deepen their understanding of automation practices and methodologies.

THANK YOU RANGERS for a comprehensive document on Test Automation.

Gopinath Chigakkagari – Principal Program Manager, Visual Studio Test

Introduction

In the new DevOps culture, it's becoming increasingly important to react to customer feedback quickly and release a quality product – quality is the key. One of the best practices to achieve or maintain quality is through test automation. Test automation has many benefits, including measuring quality, avoiding regressions, and, ultimately, shipping a high quality product that can keep customers happy. Automation has to be done in the right way and at right time or it can negatively impact productivity or quality. This document goes in details about some of these impacts, using examples and case studies.

Audience

This guide addresses primarily one persona, Christine the **Test Lead**. See ALM Rangers Personas and Customer Profiles[1] for more information.

Supporting testimonials

These testimonials demonstrate the use of the DevOps culture and automated testing:

- The "Circle of love" at Universidad Rafael Landívar[2]
- Automated testing implemented at Universidad Rafael Landívar[3]

Visual Studio ALM Rangers

The Visual Studio ALM Rangers includes members from the Visual Studio Product group, Microsoft Services, Microsoft Most Valuable Professionals (MVP) and Visual Studio Community Leads. Their mission is to provide out-of-band solutions to missing features and guidance. A growing Rangers Index is available online.

Home	aka.ms/vsarunderstand
Solutions	aka.ms/vsarsolutions
Membership	aka.ms/vsarindex

Contributing ALM Rangers

Bob Hardister, Charles Sterling, Hassan Fadili, Hosam Kamal, Oscar Garcia Colon, Patricia Wagner, Pierre Donyegro, Danny Crone, Darrin Rich, Baruch Frei, Ricardo Serradas, Rob Maher, Vladimir Gusarov, Ricardo Serradas, Case O'Mara, James Waletzky, Mike Douglas

[1] http://vsarguidance.codeplex.com/releases/view/88001
[2] http://blogs.msdn.com/b/visualstudioalmrangers/archive/2015/10/14/the-circle-of-love-at-universidad-rafael-land-237-var.aspx
[3] http://blogs.msdn.com/b/visualstudioalmrangers/archive/2015/10/14/automated-testing-implemented-at-universidad-rafael-land-237-var.aspx

Test Automation Concepts

Importance of automated testing

Automated testing is an important activity to ship high quality software!

Some of the advantages of implementing the right level of test automation are:

- **Early quality indicator**. Automated tests that run after every build of the source code provide early feedback about possible quality issues in the product. For example, if a test that has historically executed successfully suddenly fails, a code change might be the cause. Finding out about potential defects in existing functionality *quickly* is paramount to keep the quality of the product high and reduce the cost of fixing the problem.
- **Time to market**. By automating routine tasks such as manual regression tests, a team can ship software sooner and achieve or maintain competitive advantage through increased efficiencies. Tests that execute in minutes could take a human being an order of magnitude more to execute. Additionally, automated tests are not prone to human error and execute the same way every time they are run.
- **Write once, run anywhere**. Automated tests can run repeatedly with different configurations. For example, the same test can validate different combinations of web browser and operating system for greater coverage of your test matrix. Running automation many times in production (or equivalent) also implements a type of monitor for availability metrics.
- **Accurate validation**. A test might be easy to execute or automate, but validating the results of the test could be time consuming. Consider an accounting system that does complex calculations then generates a spreadsheet of results. A human might have difficulty validating a result in a sea of numbers but automated validation might take a fraction of a second.
- **Validate difficult scenarios**. Attempting to validate the scalability of a SaaS product without automation is extremely difficult. How can you simulate thousands of simultaneous accesses to the product to find the realistic point of failure? Test automation helps solve this by simulating many users accessing the software simultaneously, or over a lengthy period.
- **Clean architecture**. Building test automation (and thinking "test first") helps produce a clean software architecture. By building testability into the design so that automated tests can call the right APIs to execute positive and negative paths, the architecture inherently is more decoupled and has cleaner interfaces than if API level tests were never considered.
- **Confirm product coverage**. Running automated tests at a regular cadence helps indicate how much of the code/product is tested. Without automation in place, it's difficult to obtain this data. Regular test automation runs compute code coverage and provides feedback on where additional tests are needed. The coverage percentage should increase over time, providing incremental confidence that regressions are minimized.

> NOTE
>
> Automation of deployment and environment management via a DevOps function is another practice that leads the team to similar efficiency gains as test automation. DevOps best practices are beyond the scope of this material, but something worth looking into. See DevOps - Enabling DevOps on the Microsoft Stack[4], DevOps and Application Lifecycle Management[5] and Our DevOps Journey[6] for more information.

Here are some important points to consider when you start automated testing:

- **Return on investment.** For any suite of tests being considered for automation, determine how much effort it will take to develop and maintain the test vs. the amount of effort involved in executing the test case manually. If the test case is not executed frequently or requires some heavy configuration to automate, the effort expended in implementation might not be worthwhile. Additionally, if the automated tests touch an area of the product that is under heavy development, the amount of maintenance required to keep the test passing might not be warranted.

[4] https://msdn.microsoft.com/en-ca/magazine/mt422586
[5] https://www.visualstudio.com/features/alm-devops-vs
[6] http://stories.visualstudio.com/devops/

- **Stable legacy components**. Some teams initiate a project to reduce technical debt and add automation to older parts of the product. Even though this practice can be useful when that part of the product is under heavy churn, automating a stable component that does not undergo frequent change is likely not worthwhile.
- **Diagnosing test failures**. The pass/fail state of a test provides valuable data on quality. In the case of a failure, however, engineers need to identify root cause as quickly as possible, so keep automated tests as granular and cohesive as possible, and implement a solid product logging strategy.
- **Test automation requires technical skills**. Not everyone has the skill set to implement test automation. While UI automation is accessible to those without a development skill set, it is a brittle form of automation, particularly if the product is changing regularly. Use UI automation strategically. API level automation is more robust, but requires development skills to implement. Do not discount the training required, investigation into a suitable test framework, and the necessary infrastructure investments, such as a QA structure in version control.
- **Developer automation**. Generally, "test automation" refers to integration and functional tests. Unit tests, however, are also classified as automated tests. Unit tests should be viewed as a mandatory task for all layers beneath the UI.
- **Test automation does not supplant testers**. Although test automation has many benefits, it does not detect all problems. An automated test has no idea how usable a product is, for example. Exploratory testing is still a best practice.

Prerequisites for test automation

What should be in place before you get started?

Here are some tips, broken into three different levels of pre-requisites:

- **Must haves.** It is strongly recommended that Level 1 items be in place before you consider implementing test automation.
- **Should haves**. To make the most of automated tests, try to implement as many of the level 2 items as possible.
- **Nice to haves**. The items in Level 3 might make your life as a test automation engineer or leader easier, but are not required to get started with test automation. The most mature automation teams fulfill the requirements in Level 3.

Must have

- **Commitment and expectations are set.** Automation is not implemented in a day. There has to be a commitment in place from leadership and engineers that automated tests are a priority for the team and their implementation is part of the "done" definition of an iteration or milestone. It is expected that automation requires a time and cost investment and that there is a significant learning curve and barrier to becoming productive. The team understands the reasons for test automation and buys in to the return on investment (ROI), dedicating time and resources towards its success.
- **Automated build passes regularly.** A precursor to automated tests is an automated build. The automated build provides the first feedback to quality — the product must build successfully in a deployable state. To automate the entire build/deploy/test cycle, it starts with the build. After that, consider automated deployment and tests. Ideally, continuous integration is in place, but a nightly build is a reasonable place to begin.
- **Test strategy in place**. Create a general strategy for how to attack test automation. For example, what pieces of the product and architecture are highest priority and why? What is the automation target coverage and goals/timeline for additions? Will automation cover functional issues as well as customer experience and business requirements? What areas should be tested using exploratory testing? Who is responsible for automation? Is it shared between a developer (unit tests) and QA engineer (integration)? Will tests be automated using a UI-driven (record-and-click) approach or (ideally) API-level tests? Is the appropriate team and skillset in place to accomplish the testing? What tools are being used to help with test automation? What is the definition of success for automation? What is "just enough" test automation? Can we determine the hot areas of the product where problems occur more frequently? How?
- **Build testability into architecture and design**. It is difficult to write effective, quick, and reliable automated tests without a software architecture or design that is testable. This concept implies that the right hooks are exposed for automation and that components are small, communicate with a defined protocol, and do one thing (single responsibility). A tightly coupled architecture without clean layers (for example, business logic in the UI) is difficult to test using automation. Some refactoring of the code might be beneficial before creating automated tests.
- **Test framework vision and ownership**. Whether starting with an off-the-shelf test framework or the team is organically building one, the framework needs someone responsible and accountable for its creation and maintenance, such as a test architect. That

person sets the standards for the creation of tests, in the same way an architect or development leader establishes a coding standard for the development team. Shared functionality across tests (such as login routines) must be thought out.

- **Core infrastructure (e.g. logging and reporting) in place**. To get the most out of test automation, put a common logging and tracing system in place so that product failures can be diagnosed quickly, increasing the ROI of the automation. Additionally, automated tests driven by an orchestration engine (like TFS) have clear and rapid visibility into the existing progress or outcome of test runs, including pass/fail rates and drill-down on test failures.
- **Test cases exist to create a starting point**. Have an understanding of which automated tests will provide the largest return on investment. The steps required to execute those tests in a manual fashion should be documented as test cases in a test case management system. The engineer implementing the test can use the test case to achieve basic understanding of the scenario to be automated. In addition to having test cases as a baseline, test cases are classified in priority buckets (e.g. P0, P1, P2…) so that the highest ROI test cases are automated first.

> NOTE Clear traceability between test cases and product requirements (e.g. user stories) help provide additional context for automation implementation. Tools like TFS and Jira allow links between requirements, code, test cases, and other work items. Some specific guidance on test selection and prioritization is provided later in this guide.

- **Representative test data available**. Automated tests should operate as realistically as possible. Securing real customer data to execute against is recommended to validate real-world scenarios.

> NOTE Using customer data for testing can be tricky. Depending on what is stored in the database, the data may need sanitization before being imported into a QA environment. Additionally, a method of resetting the data such that every test starts in a known state is highly recommended.

Should have

- **Test environments in place**. Automation should execute in a known-good and maintained test environment. Many teams are moving their infrastructure and virtualized environments to the cloud (like Azure or Amazon Web Services). Have a plan in place for what environments are required, and managing those environments with respect to known-good snapshots and the automated tests.
- **Automated deployment scripts created**. The next logical step after automating the build is automating deployment of the application to the appropriate environments. A hands-off deployment process sets up the appropriate environment automatically for tests to execute.
- **Automated tests automatically invoked via orchestration**. Avoid relying on the team to start an automation run - the execution of automated tests should happen automatically as part of the build/deploy/test process.

> NOTE Deploy and run unit tests as well as other types of automated tests, such as integration or end-to-end tests.

- **Code coverage results visible**. As part of automated test execution, consider measuring the code coverage. The feedback that code coverage provides influences the decision on what further areas of the application require automation. Although code coverage percentages are less useful than the specific blocks that are touched, specifying a target number for teams to hit provides a natural goal to ensure wide automation coverage.

Nice to have

- **Consider testing in production.** "Testing in production" involves monitoring and injecting certain scenarios during periods of light usage against production systems and detecting, diagnosing and repairing issues as they arise. While not a requirement to get started in automation, the close monitoring of even test environments can make diagnosing failures much more efficient. Here,

monitoring using tools like New Relic is the key — determining when there is a problem is quick and production logs provide history on what happened.

Getting started with automation is challenging. Put an appropriate test strategy in place, ensure the team has the right capabilities to create the automation and commitment to do it right, and have an economical mindset to automate the right things and maximize your return on the automation investment.

What to automate

What are the different types of automated testing?

Automated Testing can be used to verify functionality of the system under test (SUT) at multiple levels to achieve different goals. Different types of testing can be used singularly or can be utilized together to provide the most effective use of test automation.

- Unit testing. Unit Tests provide validation at the unit or method level. These tests are designed to help ensure small amount and isolated code perform as expected.
- Functional testing. These tests are used to validate functional requirements that can be verified with UI tests, API tests, etc.
- Integration testing. These tests are used to validate that components function properly when integrated with other components. This can involve UI tests as well.
- End to end scenario testing. Scenario testing verifies the functionality through the system from the start to completion.
- Performance / load testing. These tests help ensure the SUT can perform within a timeframe and/or function when the system under heavier user loads to meet non-functional requirements (NFR) such as response time of a web page.

Does it make sense to automate all of your tests?

The answer is typically "no," particularly in scenarios where there is continual enhancement of a legacy application that has been tested manually throughout its history. These applications typically were not designed with testability in mind, and it would be difficult to retrofit any kind of automation on top of it, with the exception of perhaps UI automation. Additionally, the investment in automation for stable areas of the application might not be warranted. For Greenfield applications that can be designed for testability, on the other hand, teams should automate a higher percentage of tests, but there is still room for manual verification and mandatory exploratory testing.

Assuming the decision is made not to automate everything, what should be automated and what test cases should be automated first? A useful input for making these decisions is a set of existing test cases that have historically been run manually. This may be a large list, however, so how to prioritize what to automate?

A recommended prioritization scheme separates tests into four buckets:

- **P0 - smoke tests**. These tests run quickly and are executed as frequently as possible - ideally after every build of the software. Smoke tests validate the most basic and important functionality of the system. For example, does the application launch? Can a basic user login? Can the primary use case be initiated? The number of test cases marked as P0 are typically small relative to the other buckets, and smoke tests are almost always automated.

> NOTE
> The term "smoke test" comes from the electronics domain. When a circuit board is built, the most basic test is to plug it in and hope the circuitry doesn't get fried due to a flaw in the design or implementation. If the smoke test passes, other tests can be executed. If the smoke tests fails, well, no need to execute any further tests.

- **P1 - build verification tests (BVTs).** These tests go a level deeper than smoke tests and typically cover the most important use cases of the software. BVTs are executed daily after a nightly build is generated, ideally in an automated fashion.
- **P2 - functional tests.** These tests are executed whenever a regression test pass is required, such as at the end of an iteration or release. This bucket contains the most tests and exercise the majority of user interaction with the software. Not all tests in the P2 bucket are automated.
- **P3 - Low Severity Tests.** Tests in this bucket reflect lower severity user scenarios, such as boundary cases or areas of the application that are less frequently touched by either the end user or developer. These tests are generally not automated and executed less frequently than the other buckets.

Given the priority definitions above, automating P0 and P1 test cases is highly recommended. However, deciding how to assign test cases to the various buckets, as well as what to automate in the P2 and P3 buckets, requires a few more guidelines. The following table illustrates the types of tests that should and should not be automated.

Type of Test	Prioritization Bucket	Automate	Notes
Repetitive	P0, P1	Yes	Rote tests that are run very frequently for daily or continuous integration builds. Aim to automate these using testable APIs if possible.
Varying platforms and configurations	Any	Yes	Tests that are frequently executed the same way on different platforms (for example, operating systems, browsers) or configurations (for example, different editions or versions of SQL Server).
Multiple data sets	Any	Yes	Test that involve multiple data sets (for example, same scenario with different types of users).
APIs	P0-P2	Yes	APIs are inherently automatable. Tests such as integration tests, interface / contract tests, performance tests, and load tests should all be driven through testable APIs.
Human error-prone	Any	Yes	Tests that are difficult to execute due to frequent human error (for example, configuration not correct, lots of fields to select).
Manually difficult	Any	Maybe	Automate tests that are difficult to perform manually, such as stress and load testing. Automate esoteric scenarios that may be difficult to reproduce manually, such as a race condition, but if the scenarios are core. Otherwise, the time investment in automation might not be worthwhile.
Effort-intensive	P2, P3	Yes	Tests that require a lot of steps, setup time, or execution time. P0 and P1 tests should execute quickly by default, so consider breaking down tests with lots of steps into smaller units.
Infrequent scenarios	P2, P3	No	Focus automation efforts on more effective test cases.
User Experience	Any	Maybe	Automating core end-to-end user scenarios through the product of simple predictable workflows via record-and-click mechanisms may be worthwhile. However, minimize the number of these types of tests and prefer API-based automation.

Type of Test	Prioritization Bucket	Automate	Notes
Usability validation	Any	No	Automation is good at finding regressions in existing functionality using a prescriptive set of steps. Automation is not good at finding obvious user issues (for example, inverted colors) that might be functionally correct from the automation perspective.
Low-level white box tests	N/A	Yes	Unit tests are always automated - no exceptions!

Table 1 - Types of tests that should and should not be automated

In the end, will automating a test provide a return on the investment of creating it?

If the effort involved in creating *and* maintaining the test case is larger than the efficiency gains projected over the life of the test case, then do not automate it. Prioritize the test cases accordingly and start by automating the P0 and P1 buckets.

Measuring quality

How does your team know how well it is going with overall product quality, and specifically with quality assurance practices, since that is the topic at hand? Below is a sampling of some metrics that might assist you in determining the answer to this question. While not a complete list, it represents some basic measures to consider.

NOTE When considering metrics for your team, keep in mind that metrics can influence bad behavior. For example, nowhere in the table below is "defects found per tester" listed. This metric is poor for several reasons, including influencing testers to wait until later in the development cycle to find bugs instead of focusing on preventing them, and also slowing the team down since every bug is reported instead of working through a more efficient collaborative process to address issues. For similar reasons, never measure a developer on the number of lines of code produced.

NOTE There is no distinction below between a "metric" and a "key performance indicator" (KPI). KPIs are generally tied to a business goal/objective and track progress toward meeting that objective. Determine which measures below are most applicable for your organization and target a specific value at a moment in time to create a suitable KPI.

Build quality based on tests

Key Question(s)	Metric	Description	Typical Target
Are our development and QA practices addressing quality early enough in the development cycle? What is our confidence in shipping this release?	Number of regressions found during System Integration and/or User Acceptance Testing (UAT)	One indicator whether we have a problem with brittle code; could also indicate that there are issues with development practices causing bugs to be found too late in the cycle.	< 10
Has anything broken with the most recent code changes?	Unit test pass/fail rate (Automated)	Pass/fail rate of automated unit test execution that is part of an automated build (like CI or nightly).	100% pass

Key Question(s)	Metric	Description	Typical Target
Have we executed enough regression tests successfully to be ready for release?	Test run/pass rates	Number of tests passing / the number of tests executed. Applies to both automated and manual tests. If targets are not met per phase, than it is clear indication that the project needs attention and corrective actions need to be taken.	Varies by project phase. End of release 100% execution target 95% success rate
Is our first-level quality acceptable (i.e. are we suffering from daily quality issues)? Are recent changes conflicting with one another? Has a developer made a fundamental mistake with a recent change?	% of successful builds	A "successful" build is defined as one that compiles (all binaries generated), unit tests pass, free of high severity static analysis violations, build is deployed, and smoke tests/BVTs execute successfully.	target > 80%
Is our regression of key user scenarios as efficient as it can be?	% of high priority functionality automated to date	The higher % automated, the more relevant the delivery pipeline becomes and the ability to take on more features increases as regressions are caught earlier and faster.	Upward trend of percentage
What is our automated regression coverage for new code? Where must we dedicate humans to manual regression leading up to release?	Number of new integration test cases automated during iteration development	Ensure that our iteration deliverables include associated automated tests. Also allow us to plan for manual execution where automation is behind or not feasible.	100% of regression test cases flagged for automation complete during the iteration
What is our progress in automating our regression tests?	% of completed automated tests / the total flagged for automation	Indicates progress on automating tests that will ultimately increase regression efficiency. Note that the total flagged for automation should only represent high priority tests and not every test.	100% by System Integration Time
What is the progress of executing all our planned test cases?	% of test cases executed / total number of test cases to run	Basic percentage of test cases that have been executed relative to the number of planned test cases to execute. Low progress might be indicative of systemic problems (like product quality, or tester roles being insufficient).	100% at release-time

Table 2 – Build quality based on tests

Build quality based on bugs

Key Question(s)	Metric	Description	Typical Target
Are we ready to release the product?	Bug burn down chart	Ideal charts show lines representing number of:	During development - Saw

Key Question(s)	Metric	Description	Typical Target
Should we focus on improving quality vs. adding new functionality? Are we keeping bug debt under control?		• open bugs not yet being addressed • bugs actively being worked on • resolved bugs waiting for validation • closed bugs A high rate of open bugs may indicate the team is busy working on other things (features perhaps) and may not be prioritizing quality. Continuously increasing open bugs generally means not finishing what the team started.	tooth shape representing found bugs followed by immediately addressing Close to release - downward trend to 0
How much of our development is dedicated to customer-discovered issues? Are we resolving the defects that are in production?	Number of hot fixes per month	High trend indicates poor understanding of customer scenarios, poor testing, aggressive schedules and possible wrong elements in the continuous delivery pipeline - wrong environments, wrong test automation, no exploratory testing, and poor customer understanding.	0
How significant are the types of issues we are finding?	Bugs found per severity	Number of bugs found/open per severity. This assumes that each bug is categorized by severity, such as Sev1 meaning a crash or user-blocking scenario and Sev4 indicating a cosmetic issue.	0 Sev1 issues (numbers for other severities depend on definition and other factors) Downward trend to 0
What types of bugs are our greatest enemy? Where should we focus development practice improvements?	Bugs found per type	Number of bugs identified in each of several categories, such as security, UI, performance, internationalization, functionality, etc. Indicates new areas of re-enforcement for the development practices (as in not sufficient testing done upfront), holes in skill level and training need.	Downward trend to 0 (but no set recommendation)

Table 3 – Build quality based on bugs

Code coverage and metrics

Key Question(s)	Metric	Description	Typical Target
How complex is our code to maintain?	Cyclomatic complexity	# of linearly independent paths through program code; This metric indicates complexity of a method, block or program.	Average under 10 is preferred

Key Question(s)	Metric	Description	Typical Target
Where are the potential areas of bugs based on how complex the system is?		Higher numbers indicate that the code is hard to test and cost of fix is higher. Release plans must take this into consideration.	
How much dead code do we have in the system that impacts our development efficiency?	Dead Code Ratio	(Total lines of code - total lines of active code) / total lines of code Signals opportunities for refactoring, excessive complexities, wasteful effort for developers to learn, wasteful test cases to execute, etc.	< 5%
How thorough is our test coverage? Have we missed any key test cases?	Code Coverage	The number of lines, blocks, or paths that are executed by automated tests divided by the total number of items (expressed as a percentage). Note that the number is less important than identifying specific lines of code that are never touched by automation and ensuring that additional tests are created.	100% for API functions 80% for other functions (excluding UI code) As a general metric, 70%+ coverage for all new code.
What is our risk of regressions?	Rate of code churn	Churn measures total added, modified and deleted LOC over a period of time. It can indicate how large the recent changes were and what the most frequently changing components are, providing an indication of regression risk and where regression testing should be focused. A high code churn happening at the end of software cycle even though the change requests or bug fixes are simple can be an indication of poor design.	Downward trend to 0 approaching release. Churn not focused on one central component of the software.

Table 4 – Code coverage and metrics

Technical debt and backlog

Key Question(s)	Metric	Description	Typical Target
How many defects escape our product development cycle and are found by users?	Defect Escape Rate	Defects found in production / total defects found Typically done for a release.	< 5% Downward trend to 0
Are core product scenarios usable enough?	Fluency	Time that it takes a user (or set of users) to accomplish a particular scenario.	Goals depend on scenarios. As changes are made to the software and usability, fluency

Key Question(s)	Metric	Description	Typical Target
			should have a downward trend.
What is the general quality of our story development within iterations?	Iteration Defect Leakage	# of defects per story/work item completed If number is large, potentially indicates low quality focus during iteration development (e.g. lack of unit testing, high priority automation and exploratory testing); Definition of Done was likely not met.	< 5 (depending on complexity of story)
How maintainable is our code? What effect does our code maintainability have on our ability to deliver?	Maintainability Index SQALE	Calculates an index value between 0 and 100 that represents the relative ease of maintaining the code. A high value means better maintainability and (likely) greater efficiency to deliver. • 20 -100 => good • 10-19 => moderate • 0-9 => low Color coded ratings can be used to quickly identify trouble spots in your code. SQALE is measured by SonarQube and evaluates source code on several levels, such as testability, reusability, and changeability.	20-100
Are customers happy with our software, quality and/or brand?	Net Promoter Score	NPS = # of Promoters (9-10 answer) - # of Detractors (0-6 answer) Survey customers on how likely they are to recommend your software to a friend or colleague (0-10).	Upward trend

Table 5 – Technical debt and backlog

Test frameworks and when to use them

A “test automation framework” is scaffolding that is laid to provide an execution environment for the automation test scripts. The framework provides the user with various benefits that helps them to develop, execute and report the automation test scripts efficiently. It is more like a system that has created specifically to automate our tests.

In a very simple language, we can say that a framework is a constructive blend of various guidelines, coding standards, concepts, processes, practices, project hierarchies, modularity, reporting mechanism, test data injections etc. to pillar automation testing. Thus, user can follow these guidelines while automating application to take advantages of various productive results.

The advantages can be in different forms like ease of scripting, scalability, modularity, understandability, process definition, re-usability, cost, maintenance etc. Thus, to be able to grab these benefits, developers are advised to use one or more of the test automation frameworks.

Moreover, the need of a single and standard test automation framework arises when you have a bunch of developers working on the different modules of the same application and when we want to avoid situations where each of the developer implements his/her approach towards automation.

NOTE	A testing framework is always application independent, what means that it can be used with any application irrespective of the complications (like Technology stack, architecture etc.) of application under test.

The framework should be **scalable** and **maintainable.**

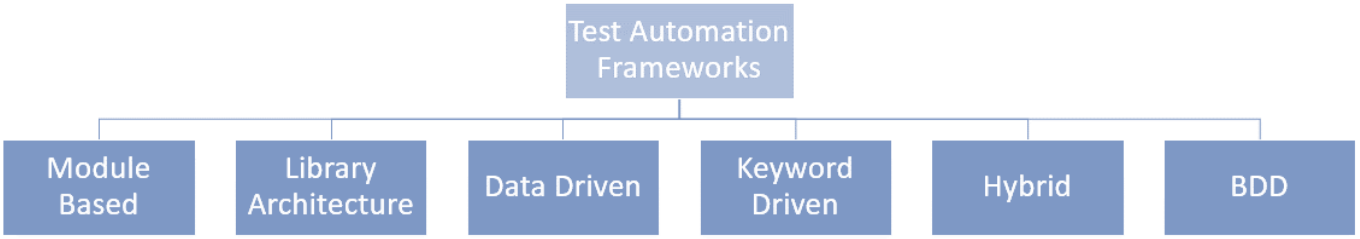


Figure 1 - Test Automation Frameworks

Advantage of Test Automation framework

- Reusability of code
- Maximum coverage
- Recovery scenario
- Low cost maintenance
- Minimal manual intervention
- Easy Reporting

Types of test automation framework

Now that we have a basic idea of what is an automation framework, in this section we would harbinger you with the various types of test automation frameworks those are available in the market place. We would also try shed lights over their “pros and cons” and usability recommendations.

There is a divergent range of automation frameworks available nowadays. These frameworks may differ from each other based on their support to different key factors to do automation like reusability, ease of maintenance etc.

- Module based testing framework
- Library architecture testing framework
- Data driven testing framework
- Keyword driven testing framework
- Behavior driven development framework

When to use each framework?

Test Framework Name	When to use?
Module based testing framework	The framework divides the entire "Application Under Test" into number of logical and isolated modules. For each module, we create a separate and independent test script. Therefore, when these test scripts are taken together, they build a larger test script that represents more than one module.
Library architecture testing framework	The fundamental principle behind the framework is to determine the common steps, group them into functions under a library, and call those functions in the test scripts whenever required.
Data driven testing framework	Data Driven Testing Framework helps you segregate the test script logic and the test data from each other. It lets the user store the test data into an external database. The external databases can be property files, XML files, Excel files, text files, CSV files, ODBC repositories, etc. The data is conventionally stored in "Key-Value" pairs. The key can be used to access and populate the data within the test scripts.
Keyword driven testing framework	The Keyword driven testing framework is an extension to Data Driven Testing Framework in a sense that it not only segregates the test data from the scripts, it also keeps the specific set of code belonging to the test script into an external data file. Keywords provide an abstraction layer from the test code to interact with the UI so that non-developers can write automated tests. Actions can be UI level such as click or enter text. They can also represent a group of interaction with the UI such as Login. The keywords and the test data are stored in a tabular like structure, so it's popularly regarded as Table driven Framework. Keywords and test data are entities independent of the automation tool being used.
Behavior driven development framework	Behavior Driven Development Framework allows automation of functional validations in a format that's easily readable and understandable to Business Analysts, Developers, Testers, etc. It can be useful when Domain Driven Development and Test Driven Development practices are used during development cycle. It's also important that both the development and the functional professionals who've committed to the project are knowledgeable about these practices and tools.

Table 6 – When to use each framework

Advantages / disadvantages of various test frameworks

Test Framework Name	Pros	Cons
Module based testing framework	• The framework introduces high level of modularization, which leads to easier and cost efficient maintenance. • The framework is pretty much scalable • If the changes are implemented in one part of the application, only the test script representing that part of the application	• While implementing test scripts for each module separately, we embed the test data (data with which we are supposed to perform testing) into the test scripts. Thus, whenever we are supposed to test with a different set of test data, it requires the

Test Framework Name	Pros	Cons
	needs to be fixed leaving all the other parts untouched.	manipulations to be made in the test scripts
Library architecture testing framework	• Like Module Based Framework, this framework also introduces a high level of Modularization, which leads to easier and more cost efficient maintenance and scalability, too. • Because we create common functions that can be efficiently used by the various test scripts across the Framework, the framework introduces a great degree of re-usability.	• Like Module Based Framework, the test data is lodged into the test scripts, so any change in the test data requires changes in the test script as well. • With the introduction of libraries, the framework becomes a little complicated.
Data driven testing framework	• The most important feature of this framework is that it considerably reduces the total number of scripts required to cover all the possible combinations of test scenarios. Thus, less code is required to test a complete set of scenarios. • Any change in the test data matrix would not hamper the test script code. • Increases flexibility and maintainability. • A single test scenario can be executed alter the test data values.	• The process is complex and requires an extra effort to determine the test data sources and reading mechanisms. • Requires proficiency in the programming language that is being used to develop test scripts.
Keyword driven testing framework	• In addition to advantages provided by Data Driven testing, Keyword Driven Framework doesn't require you to possess scripting knowledge, unlike Data Driven Testing. • A single keyword can be used across multiple test scripts.	• The user should be well versed with the Keyword creation mechanism to be able to efficiently leverage the benefits provided by the framework. • The framework becomes complicated gradually as it grows and new keywords are introduced.
Behavior driven development framework	• BDD allows automation of functional validations in a format that's easily readable and understandable to Business Analysts, Developers, Testers, etc.	• Some additional points that aren't in the illustration: o Object Repository: Object Repository acronym (OR) is comprised of the set of locators types associated with web elements. o Test Data: This is a set of inputs and expected results. The inputs are passed into the test and the expected results are compared against the actual results. o Configuration File/Constants/ Environment Settings: This file stores the information about the application URL, browser specific information, etc. It's generally the information that remains static throughout the framework.

Test Framework Name	Pros	Cons
		○ Generics/ Program logics/ Readers: These are the classes that store the functions, which can be commonly used across the entire framework. ○ Build tools and Continuous Integration: The tools to generate test reports, email notifications, and logging information.

Table 7 – Advantages / Disadvantages various test frameworks

Test data in automated testing

A successful test automation approach relies on repeatable test data. For the automation suite to run successfully, it requires assumptions to be made about the data available, for example does the data change, expire or get stale. There are a number of strategies that people use to manage automation test data, each with their own advantages and disadvantages. In this section, we'll highlight some of the popular approaches.

Automation data approaches

Production Copy – is a full copy of the production data used for automation purposes. This copy is refreshed periodically.

Advantages:

- The data represents the real "shape" of production.
- The data represents real life.
- The data is easy to set up. Simply restore a backup of production.

Disadvantages:

- The data may contain identifiable information that breaches privacy and data protection laws. As such, the data might need to be masked and cleansed in order to hide sensitive information.
- The data might be very large, taking up a lot of resources.
- Even if only a subset of the production data is taken, it can be difficult to build a subset that maintains referential integrity and still contains no sensitive data.

Tests populate required data – requires each test or suite to insert the data it needs before the test is run so that the data is available to the test. After the test or suite finishes, the data is removed.

Advantages:

- Tests are self-describing including all required data.
- The tests are idempotent.
- Data schema changes may reflect directly on the tests.

Disadvantages:

- It may be very complex to insert some types of data.
- Some types of data may need to be "aged" for a test and this might not be possible.

- Requires that test authors understand how to create the data they need.

Tests select required data as part of run – each test performs a search on the test dataset before the run to find the data that they need, for example, a customer that has a foreign currency account with a balance greater than $50,000. If the search fails, then the test is failed.

Advantages:

- Tests are self-describing and include all required data.
- Tests don't assume that specific data records are present.

Disadvantages:

- Some tests could be destructive to the data, such as closing an open account. This might quickly exhaust available records, causing failures
- Certain types of data could require very complex queries to locate.

Transaction Reset – A baseline dataset containing all required data is located. Each test is performed as a transaction that is rolled back upon completion, leaving the data unchanged

Advantages:

- A one-time setup of data that can be re-used many times.
- The test author does not need to understand data structures.

Disadvantages:

- Transaction limitations. There might be distributed data that cannot enlist in a single transaction.
- Schema changes need to be introduced into the baseline version as they are developed. This data also needs to be populated.

Choosing an approach

To decide which approach might work best for the team, take a single or small number of features and try to automate them using some of the prescribed approaches listed in this section. The team can then make a decision on which approach might work best for them. A blend of different approaches may be the most successful.

Automated team processes

We will show you how to successfully manage and analyze your test runs and automation in the most efficient manner, while gaining valuable insight on your product's quality and health.

Sample Infrastructure

For this guidance, let us assume you have an infrastructure like the one shown in Figure 3 – Visual Studio Team Services integration with Azure test machines, on page 37. We won't go into the pieces needed for the testing infrastructure, but once you have these pieces in place, you'll need to analyze the test results from your automated regression test runs.

Frequency

The frequency will have a lot to do with your DevOps strategy and how to analyze the results. Based on how often you are building, testing, and deploying, you'll need to find a way to quickly analyze and find failures. In a large infrastructure with multiple releases and, additionally, multiple places to go to get more information on your failures, this can pose a problem. Consider that you might also be supporting multiple products and groups with your infrastructure, your testing matrix could get quite large and unwieldy very quickly. This is where equivalence class testing and Test Mix will come into play. You'll also need to quickly determine if there is value in a complete run, or will work only on Severity 1 and 2 test cases.

The Objective

Your main objective is to get your regression testing as close to the failure as possible. In a perfect world, you would expect that every piece of code checked in that day would have a test case. Your test runs will run, at the very least, a full build and test run nightly. In most scenarios, they are done close—if not at the same time—with unit tests, before the code is checked in.

If the development team motto is "fail fast, fail often," then your testing motto should be "Analyze faster, and create bugs often." Every failure needs a bug, and every bug needs a test case. Your test cases need to have a priority around them. This will make sure that, based on time or availability, you're hitting the most relevant test cases. Here is an example of how you would cover it:

Figure 1 – Test coverage

How to rank and prioritize test cases will be greatly debated, but for this example, let's assume the following priority criteria:

- Priority 1 – Major Core Functionality (entire feature area would be broken if this didn't work)
- Priority 2 – Major Basic Functionality (some areas could be blocked if these didn't work)
- Priority 3 – Minor functionality based on areas above (These would be outside of the normal 80% of test cases your users would hit on a day to day basis)
- Priority 4 – Minor functionality / Edge/ Boundary / Negative tests (These would focus on the minor part and edges of your development code)

In this scenario, if a test case in Priority 1 failed, you wouldn't run test cases under priority 2, 3 or 4. You know that a major large area of basic functionality is broken. This not only saves you time, but also helps to fail fast. How good is a large number of failures if you know that the entire area is broken on the most important parts? Would it bring more value to get more edge tests cases to fail?

Analyze the Test Results

These test results will run on a nightly basis at a minimum for a daily build cycle. The results from the previous night will need to be reviewed and appropriate actions should be taken to determine if failures were bugs or were improperly coded test cases or environmental issues. This could also be the result of requirement changes where the test case needs updates to reflect them. You'll need to go through and determine which of these is the correct result.

To accomplish this, you'll need to assign one or more people to review test cases. Initially, these test cases should be marked with "needs investigation. This will allow you to iterate through these test cases that have failed and determine what happened or re-run and see what is going on. In most cases, you should log the results through the test case so you can quickly determine what failed. In some scenarios, where you have some lag between when you ran the test case and looking at the results, the environment could be in a different state. This is especially true when you have limited resources, and multiple releases that are waiting to be tested. So, capture as much about the failure at run time as possible. This will also help when filing the bug and you have not only the exact repro, but the data on where and why.

Three results

It's a failure

Once you've assessed that this is truly a bug, you will need to file it and assign an owner. Attach all your information about the build, area, and relevant log information from that test case run failure.

Owners of these areas should start investigating the bugs as close as possible to when the failure occurred. One common issue is that developers will delay working on bugs to focus on current development. This is usually a bad practice because after time has passed, they'll no longer be familiar with the code, they'll have to be reacquainted with what they were doing at the time, and remember why they did they did it Here's a list of the failure types and resolutions:

Failure Types

- Regression
- New Issue

- Known Issue
- Unknown

Resolutions

- Configuration Issue
- Needs investigation
- Product issue
- Test issue

You can also have failures around “unreliable tests.” Test cases that are unreliable should be pulled and examined quickly to reduce failure points, or your technical debt will increase substantially. To reduce these failure points, you can remove the test while you fix it, change the environment, or reduce the number of steps it takes to execute that test case. You might also have to change the environment if it’s related to instability, bad dependencies, etc. Good test logging will always help in this case. You should aim to record exactly what went wrong in your log so you can see what was happening at that time, because it might not reproduce. “No repro” makes it even more difficult to identify this unreliable test.

Test case needs to be modified

If you’ve looked at the test case and the code and determined that the developer did the check in based on changed requirements, you should start by updating the test case and then look for any others that will be changed. This is referred to as “Halo Testing.” Make sure you flag this test case to be re-run either in that current run or in a subsequent run.

The problem could also be with the data you’re using. Try to minimize your failure points by specifying how many things need to be working, updated, online, or have security/passwords to work. Any one of these failure points can add up when you’re trying to maintain your data or environment. Always try to bring your validation data with you and then reset criteria when possible so that the next run won’t be invalidated by something that can only run once to pass.

You’re not sure

Start by re-running the test case. If it passes, try to determine why the test case failed and what you didn’t capture in the logs. Maybe look at increasing your logging level to capture why it failed. This is a good opportunity to make sure you’re capturing everything you can.

This could be an infrastructure or environmental failure that occurred at that time. Take note of that the failure could have been caused by one of these elements, and maybe have a pre-check to see if those elements are working correctly. Another approach to analyzing failures is to try to put the validation in a post step to gather more information.

Practical guidance

Best practices of writing reliable tests

Reliable tests and their automation are important, if not critical, for the pursuit of quality, adopting a shorter application lifecycle cadence, embracing the nature of cloud development, and the ability to test in production.

NOTE

Practices mentioned here are based on:

- Learnings from in-house product tests.
- Eradicating Non-Determinism in Tests[7], by Martin Fowler.

Importance

Test failures are a combination of **legitimate issues** the tests are designed to identify and **noise** generated by unreliable tests. The noise can become overwhelming, impact the ability to make decisions based upon test results, and more seriously, undermine the **trust** we have on our solution. Test reliability is an important and visible measure. To analyze and drive test improvement, we need to review test reliability in every sprint review..

When **test reliability**, not to be mistaken with test success, is near 100%, we have a high degree of **confidence** that test failures are real failures.

NOTE

Recommended objectives to raise confidence and allow engineers to make changes with confidence:

- Near 100% test reliability.
- Friction-free, fast and reliable set of tests.
- Automated testing and continuous deployment.

Best practices

Goals

The following goals empower us to ship frequently and with confidence:

- **Friction free** system supports automated test and update within 15 minutes of a check-in.
- **Fast and reliable tests** allow us to test and react to test failures quickly.
- **Automated tests** save time, increase accuracy and coverage, and perform what manual testing cannot.

Principles

The following principles enable us to transform to a reliable and automated test taxonomy while remembering that in times of change, we have a business to run.

[7] http://martinfowler.com/articles/nonDeterminism.html

- **Design for testability from the start**
  - Testability is part of the design, not an afterthought.
  - Strive for explicit boundaries and clear contracts (interfaces) to define behavior.
  - Shift balance in favor of unit testing over functional testing.
- **Lowest level**
  - Favor tests with the fewest external dependencies over all other types of tests.
  - Strive for high cohesion and low coupling.
    - Limiting knowledge or reliance on implementation change ensures low coupling.
    - Loose coupling reduces the need for specialized knowledge.
  - Unit tests must be fast, reliable and not rely on deployment, external dependencies, file system, or DBMS.
  - Functional tests should be stubbed out or faked in some way to reduce external dependencies.
- **Deterministic tests**
  - Tests must have a predictable outcome and always pass or fail if there is no noticeable change in code, tests or environment.
  - Tests must not be affected by other tests.
  - Use callbacks or polling instead of sleeps to avoid timing issues or slowing down your tests.
  - Where possible, wrap the system clock to synchronize test data with data and time.
- **Regard test code as product code**
  - Apply the same quality bars and code reviews to test and product code.
  - Build, maintain, and store test and product code as one.
- **Test against production deployment**
  - Functional tests should only use the public API of the product.
  - Introduce the diversity and scale of production to the test environment.
- **Quarantine unreliable tests**
  - Unreliable tests are non-deterministic, expensive, and noisy, they work against test reliability.
  - Quarantine and exclude unreliable and non-deterministic tests from automated testing.
  - Refactor or retire quarantined tests.
- **KISS (keep it simplistic)**
  - Tests must be easy to author and run.
  - Tests should live alongside product code, be built with the product, and run as part of build.
  - Tests must be visible and run under Visual Studio Test Explorer, if Visual Studio is your default environment.

Introducing test automation for existing or new products

When introducing automation, the strategy for new products differs from existing or legacy products.

Greenfield products/applications

For new products, the strategy can focus on forward thinking questions like what type of tests need to be automated? What tools can we use? How can we design with automated testing in mind? How can we dovetail our automated tests into our build/deploy/test process? See “What to Automate,” on page 9, for more information.

Legacy products/applications

For existing products, some of the questions remain largely the same. However, the emphasis needs to on the biggest wins for the least amount of effort. The strategy should focus heavily on what to automate, and, as identified in “What to Automate”(page 9) unit tests are often difficult to retrofit onto legacy code. If the code is well structured and supports unit testing then this is where the focus should be. However, if it is not, then we must turn to functional tests at the UI or API layers for scenarios that are churning with particular attention to bangs for the buck.

> NOTE We've already chosen the tools and factored in their costs, including training. Now we should compare how long the test takes to execute manually and how many times will we execute it with how long will it take to automate it.

Although it is never safe to assume anything, we do assume that the legacy product will have an arsenal of manual regression tests; if not, then these tests need to be identified and prioritized first. These regression tests will inevitably be one of the major pain points for the team(s).

Here's a list that helps prioritize what to automate when introducing automation into legacy products:

- **Focus on the current sprint/release**. As with all good regression testing, identifying the minimum set of tests to be run is key, so tests that focus on critical or churning components should be prioritized.
- **Ease of Automation**. API Tests will be much easier to automate and therefore the strategy should focus on tackling these easier tests first.
- **Manual tests that often fail**. Manual tests that historically fail are a risk that needs to be mitigated because they can indicate an area of the code that regresses frequently. Although automating these tests does not address the root cause of the problem, they will provide a critical safety net for the team.
- **Tedious manual tests**. Sometimes these can be long running tests and often difficult to automate reliably; however, human beings often make mistakes when performing long and repetitive manual tests.

> NOTE Experience show us that the tedium of manually executing the same long running tests repeatedly can be a big factor in overlooking defects. It is important to note that humans become blind to defects because they focus too much on the task at hand, it would be impossible to have the robot look for all of these defects, so the automation of these tests will free the tester's vision and allow them to see the changes easier.

- **Tests with multiple configurations**. Multi browser tests or multi-platform mobile app tests can take a long time to execute and therefore represent a large saving to the regression testing effort of legacy products.

> NOTE Caution should be taken with multi-browser or multi-platform mobile tests. They can seem like ideal candidates for automation because they appear to be the same test and they take a long time to execute. However, they are notoriously difficult to standardize across the various platforms and will require a variety of tools and infrastructure or services to implement successfully.

- **Critical components**. Because our intent is to help reduce residual risk in the product or application, we should focus our attention on critical components or End to End scenarios that cover critical components

Automation team structure options

Deliver test automation using a centralized team or include it as a deliverable in a development iteration?

Some advantages and disadvantages of each model.

Centralized automation team

Advantages

- A team focused exclusively on test automation minimizes distractions that might otherwise hinder you from implementing more end-to-end scenarios, Feature team productivity increases.

- The creation and maintenance of automated tests is simplified and potentially streamlined because work can be distributed to more team members Knowledge sharing increases and impediments decrease.
- The Team may count on more technical people to avoid the default “click/record” UI automation mode. This generates more code-level automation, which might be more robust with changing product code.
- It’s easier to maintain consistency across automated tests because they are implemented by one centralized team.
- An organically-grown (or customized third-party) test automation framework could arise and make everyone's job of automation easier.
- The team can focus on different types of system-level automation, such as automated performance tests, load tests, and integration tests.
- Failing tests are more efficiently diagnosed and addressed because the tests are the core responsibility of the central team.

Disadvantages

- Team members feel less ownership of user scenarios and inherently have less understanding of the end user.
- Test automation is done by different people from those who built the scenario. This requires more communication and collaboration between the person implementing the automation and the person who implemented the feature to get a full understanding of the scenario being tested.
- There might be a delay between functionality in the build vs. automation being ready, increasing the risk of regression in new features.
- If automation is outsourced, quality might be lower due to an absence of ownership and (possibly) passion. = Issues found during test execution could take longer to resolve; the team needs to keep a watchful eye on deliverables.
- The team might automate *too much*, incorporating test cases without a significant return on investment (ROI).
- The team might not be able to automate enough, if the team is too small or is also automating the testing of legacy code (technical debt) in addition to newly developed code.
- Developers might be less accountable for quality because the development team does not own it end-to-end.
- Both development and test teams might not have commitment to the success of the project because they’re committed with different parts of a feature’s definition of done.

Embedded in Agile team

Advantages

- The team has a deep understanding of the overall system and can implement automation faster and quickly address test execution issues.
- The team that developed the functionality is also accountable for (and committed to) automation.
- Testability is inherently designed into the product because the person who developed the functionality also develops the automation.
- Automation can be implemented at the same time as the functionality without a context switch.
- Automation deliverables can be included as part of the iteration exit criteria and "definition of done". This minimizes the delay in execution.
- Automation test bed for code that will eventually become legacy is already in place when the code eventually does become legacy.
- The learning during test automation implementation might lead to product improvements (such as API usability).
- Help with and code review of automation is close by because it’s owned by the Agile *Team.*

Disadvantages

- No centralized team managing the automation infrastructure, which might cause test architecture and tools inconsistency across teams.
- Test automation might be deprioritized because other iteration deliverables take precedence.

- The perception might be that the team develops less functionality, but product quality is actually higher because automation is available quickly.
- Automation may focus on a particular feature or scenario, but there is a risk if the automation does not cover the system end-to-end.

While there are advantages and disadvantages to both models test automation implemented in a centralized team vs. done by an Agile team—it is recommended that the team creating the functionality also owns the automation. This approach raises quality early in the software development lifecycle, and the team can own and deliver test automation quickly and efficiently as part of an iteration's "definition of done". In this scenario, a best practice is to have a test architect who can manage the holistic test automation approach the same way a developer architect manages the overall software architecture.

NOTE

Alternately, consider a hybrid approach - a small dedicated team of test automation engineers who manage the infrastructure (test harness, test framework) and Agile teams that contribute with tests using that infrastructure. There is a risk of delay to manage, however, which is incurred when the centralized team is slow to meet the requirements of the Agile teams or address outstanding issues.

Using prescriptive approaches such as ATDD, BDD, or TDD

When choosing an approach for automated testing, it is useful to consider some popular alternatives to see if they will meet your automated testing goals.

Automation approaches

Test Driven Development (TDD)

A developer-focused activity, TDD involves writing a failing automated unit test case, writing the minimum amount of code to allow the test to pass, and then refactoring the code to an acceptable standard. This cycle is known as Red -> Green -> Refactor.

Advantages

- Cleaner code design, because the developer must isolate the code to be tested. This often results in explicit dependency management and improved code quality.
- High test coverage. Minimal code is written in order to pass a test resulting in only the necessary code.. As a result, all of the code tends to be exercised by the unit tests.
- Regression test suite. The test suite is usually executed after every change (and during every build as part of a Continuous Integration suite). A successful test run can ensure that regression defects were not introduced as the product changes.

Disadvantages

- Extensive use of Mocks and or interfaces can be hard to understand and maintain.
- Existing code architecture might not support a TDD approach.

Acceptance Test Driven Development (ATDD)

ATTD is a team-based approach to collaboratively discuss and distill requirements acceptance criteria into a set of explicit tests. These tests are defined before development starts and are usually automated using a framework such as SpecFlow[8], Concordion[9] and Cucumber[10].

Advantages

- The entire team understands a feature before development begins.
- Failing tests provide feedback on which requirements have not yet been satisfied. This allows the team's progress to be accurately measured, using completed requirements as the scale.
- The specifications provide living documentation of the system behavior. This documentation is guaranteed to be accurate.

Disadvantages

- The entire team is needed, including the Product Owner / client representative, in order to gain a consistent understanding of the feature before development. This intense collaboration can be difficult to achieve with remote team members or an offsite client or Product Owner.
- Setting up automation from the specification through to the system under test can require teams to learn new skills. If the testers in the team do not have a strong technical skill set, this can place a lot of pressure on the developers to wire up the specifications to the system under test for each feature.

Behavior Driven Development (BDD)

BDD is a combination of TDD and some concepts from Domain Driven Design[11]. It's a design activity that focuses on the language and interactions of the system and encourages different roles of a team to collaborate to the software project. Expected behavior is often expressed using the Gherkin Given-When-Then format. This format describes a pre-condition then some kind of action, which is followed by an observable system operation. The behavior is often specified "outside-in," focusing on the user actions. Because the idea of BDD is, behind the scenes, very close to ATDD, we can also use SpecFlow, Concordion and Cucumber as a framework to implement it.

Advantages

- The tests help people to understand the intended behavior of the code.
- The specifications provide living documentation of the system behavior. This documentation is guaranteed to be accurate.

Disadvantages

- This technique builds on TDD experience. If the team are not experienced in TDD then starting from a BDD approach will prove very challenging.
- BDD is ideally used in combination with Domain Driven Design. The ubiquitous language forms the basis for the specifications. If DDD is not being used, then BDD might not be a good fit for the team.

[8] http://www.specflow.org/
[9] http://concordion.org/
[10] https://cucumber.io/
[11] http://stories.visualstudio.com/devops/

Choosing an approach

To decide which approach might work best for the team, take a single or small number of features and try to automate them using some of the prescribed approaches listed in this section. The team can then make a decision on which approach might work best for them.

DevOps and how it impacts testing

Incorporating automated testing into the build

A good way to catch bugs more quickly and efficiently is to include automated tests in the build process[12]. Martin Fowler explains this topic well.

Ask yourself these questions:

- What are the required different types of testing? (Starting from Unit, and all the way up to UI, Nightly Functional Runs, batches, migrations, etc.)
- How does manual testing fit in this?
- What are the ways to show real-time assessment of state of the product-under-test in order to take meaningful decisions for next deployment stage?
- Which automated tests should be incorporated into the build?
- How are these tests to be managed?

Types of automated tests to include in the build

Incorporating tests into the build is typically the *pièce de résistance* of test automation. It provides continuous product verification and helps maintain a more consistent level of product quality. However, it is impossible to achieve until automated tests and proper tools are available. Here we assumed those prerequisites have been met.

With the proper tools for incorporating automated tests into the build and a pool of automated tests to choose from, the question becomes "what tests should be run in the build?"

The section on "What to Automate", page 9, identifies the following types of automated tests:

- P0 – Smoke Tests
- P1 – Build Verification Tests (BVTs)
- P2 – Functional Tests
- P3 – Low Severity Tests

Start with introducing a small sample of P0 tests into the build. This will allow you to focus on getting the process and the tools working correctly. Once you have a reliable and repeatable process, then introduce the remaining P0 tests into the build. Take the same incremental approach with P1 and P2.

Consider the following steps when implementing types of automated tests into the build:

- Configure a proof-of-concept (POC) build environment for developing builds that run test automation.
- Select a few P0 tests for the POC.
- Expand the scope of the POC to include P1 and P2 tests.
- Implement the test automation into the real build system.
- Before adding additional tests, ensure reliability and repeatability.

[12] http://www.martinfowler.com/articles/continuousIntegration.html

Managing automated tests in the build

CI (Continuous Integration) is now a necessity and not a luxury!

There are various practices that organizations and enterprises need to implement to enable Continuous Delivery. Automated testing is one of the important practices that needs to be set up correctly for Continuous Delivery to be successful.

Automated testing during continuous integration flow

Martin Fowler defines CI as follows: *"… a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily-leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible…***"**

With this concept in mind, we all strive to integrate our unit tests with our CI builds to quickly respond to failures in case of failures and be sure that all development parts are well integrated. This can be checked by frequent CI builds including tests.

The viability of build test automation requires a reliable and repeatable process. In other words, any problems detected should be in the failure of the application to meet the conditions of the test, not the failure of test automation to execute properly within the build. The value of build test automation is enhanced further by the visibility of results, the quality of reports available, the integration of results into bug tracking and reporting systems, and the ease in which automated tests are managed.

There are two common approaches for creating automated tests. The approach can depend on the structure of the team and your overall testing strategy and approach. One approach values the traceability from requirements to acceptance criteria to test case. In this case, the test case is the specification that automation needs to verify and the automated is associated or linked back to the test case. Alternatively, the approach is code centric where

test cases are generated from the automated tests. Even though the order of development proceeds from manual to automated tests, the nature and organization of automated tests are better served by code centric management. Managing test automation as code means that test cases are automatically generated, linked to the test code, and maintained dynamically in response to changes in the test code base. These dynamic test cases support visibility, reporting, and integration into bug tracking.

Build, deploy, test in build

Running tests as part of build can be enabled with the Build, Deploy, and Distributed Test build definition template. This provides an integrated experience with traceability to achieve actionable results.

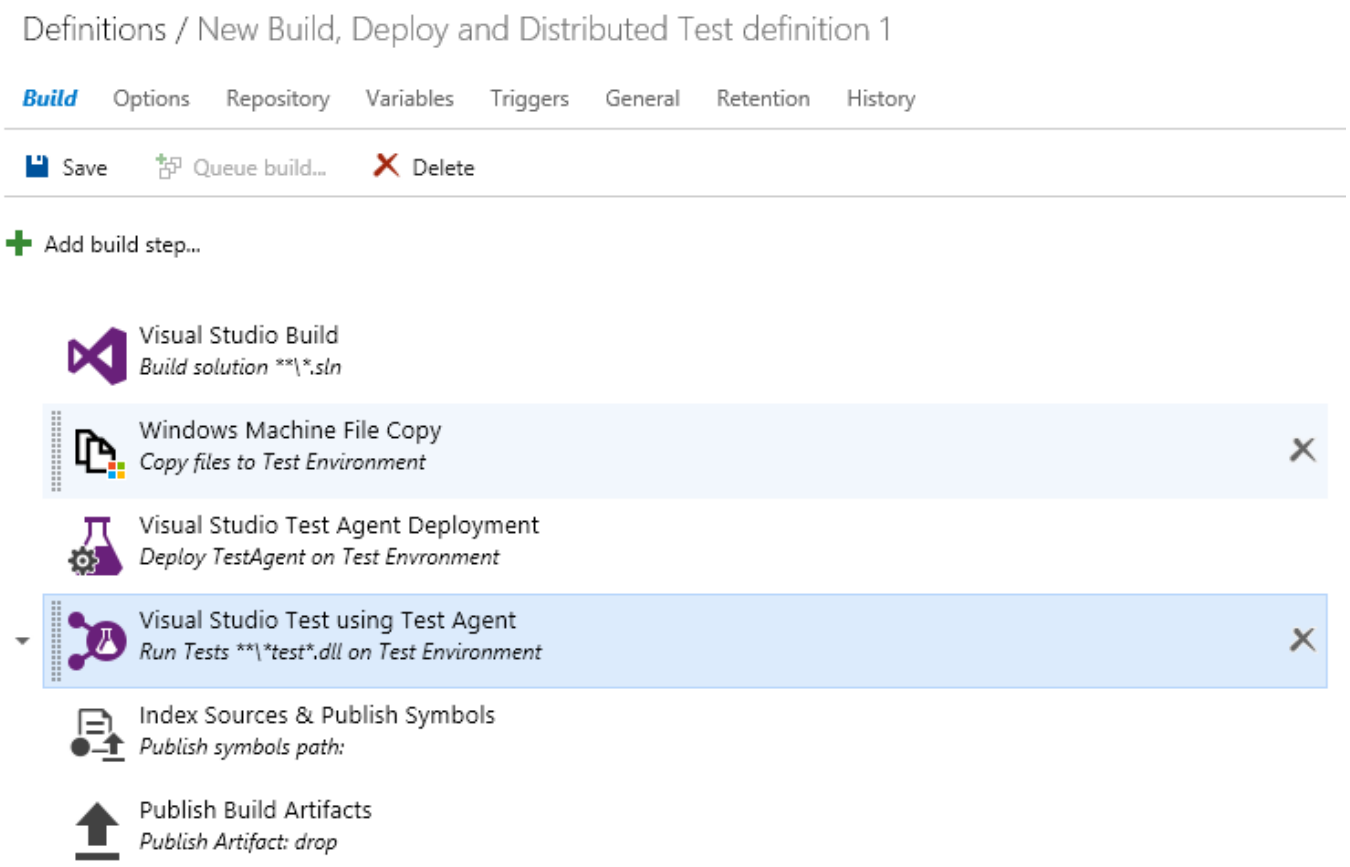


Figure 2 – Build, Deploy, and Distributed Test build definition

Incorporating tests into release pipeline (release quality)

The key objective of organizations is to deliver value from the products / services they offer. They need to deliver their offerings as quickly as possible and with a good quality!

Automated testing during continuous delivery

Martin Fowler also defines Continuous Delivery as follows: "*…the neutral extension of Continuous Integration: an approach in which teams ensure that every change to the system is releasable, and that we can release any version at the push of a button. Continuous Delivery aims to make releases boring, so we can deliver frequently and get fast feedback on what users care about.*"

Using Continuous Delivery pipeline, we can get the feedback faster on what the users care about so that we can act faster to help the release and deployment processes. Here are some takeaways to speed up the CD Pipeline process.

Considerations:

- Don't do UI end-to-end testing if your UI is not frozen.
- Don't start load testing if your basic scenario is not working.
- You don't have to execute Performance Testing if the non-functional requirements on performance aren't clear.
- You don't have to run all test cases every sprint to check the working of the system based on code changes.
- You can execute only test cases that are applicable for the changed code (Test Impact Analysis is good here).

Common test types on continuous delivery include:

- functional tests
- functional UI tests
- end-to-end scenario testing
- load testing
- compliance validations

Leveraging Azure for apps to deploy for testing

There are several steps that must happen to make this a successful endeavor. Some of these will be covered in more detail from the Continuous Deployment: Dev/Test in Azure and Deploy to Production On-premises[13] on this topic, but we'll break it down for you into some easy steps.

Azure subscription

You will need to have your Azure subscription set up and ready to go. You can get an account through a variety of methods, for example:

- Visual Studio Professional or Enterprise with MSDN.
- Microsoft programs such as Microsoft Partner Network or BizSpark.
- You can sign up directly for a free 90 day trial account.
- Pay-as-you-go or with a 6-month plan.

Link your Azure account to Visual Studio Team Services

Link your Visual Studio Team Services account to Azure. You can do this with the gear in the upper right of your VS TEAM SERVICES screen, and then you can click Services. You can add a new service connection and endpoint here. Keep in mind this will need to be something other than a Hotmail or a Microsoft account.

Build agent options

Deploying will require a build agent to build your code. The great thing with Azure is that you're only a few clicks away from creating a new machine on Azure resources.

Azure hosted pool agent

Because this resource is behind the Azure firewall and the fact that it communicates over HTTPS (443) to VS Team Services you can set this up with very little effort. VS Team Services also provides hosted build agents for you if

[13] http://blogs.msdn.com/b/visualstudioalm/archive/2015/07/31/dev-test-in-azure-and-deploy-to-production-on-premises.aspx

you want to use those to build code and deploy the application. The build agents are an on-demand resource. Follow the Hosted pool[14] directions if you would like to use the hosted agent pool.

NOTE You will need to be on Visual Studio 2013 or 2015 and will need Azure PowerShell installed on the agent if you deploy it in Azure by yourself.

On-premises build agent

You will need the proper permissions and the machine should have proper visibility to Azure and VS Team Services in order to you to configure it. See Deploy an agent to build Windows and Azure apps[15] for more information.

Special considerations (cross platform agents, commonly referred as Xplat)

Make sure youconsider what you're building. With Visual Studio 2015, you now have the ability to go into Xplat agents, and build XCode, Android, Java and other languages and technologies. We won't go into it in detail in this document, but keep in mind there are guidelines you will need to follow.See Xplat build agents[16] for more information.

Deploy your application

For this next part, we could write a short story. Instead, what we'll say is that you should have these requirements at a bare minimum or at least have thought about them in your deployment:

- Build Task – This is required and will be used to build your code.
- Azure Resource Group – You'll want a Resource Group Deployment Task.
  - This is made easier by using pre-made templates with PowerShell (this is why we install PowerShell in the note above).
  - See Deploy an application with Azure Resource Manager template[17] for more information.
- Certificates -If you need Key vault and Test Certificates, make sure you deploy these as well.
- Test Data – You will want to utilize Azure File Copy to help get these up to the blob, container, or Azure VM itself.

Test agent deployment

You can use PowerShell on your build agent and a build task to deploy these agents to the machines. This will simplify your deployment process with one single process. Make sure to have these as part of a test machine group so you can pull from this pool later.

To get your testing to work, you'll have to configure:

- Test machine group. This is going to be the Azure group you're deploying to. It should be the group you used in the build task.
- Configuration. Debug or release build.
- Platform.x86 or x64.
- Run Settings.Location of the run settings file that defines the variables,paths, etc.
- Drop Location. The place where you put the files you need to deploy or test.
- Test assemblies and Dependent assemblies. Bring everything that you need for running tests with you.

[14] https://www.visualstudio.com/get-started/build/hosted-agent-pool
[15] https://msdn.microsoft.com/Library/vs/alm/Build/agents/windows
[16] https://msdn.microsoft.com/Library/vs/alm/Build/agents/xplat
[17] https://azure.microsoft.com/en-in/documentation/articles/resource-group-template-deploy/

- Test Matrix. This specifies what test cases go to which machines in the pool. You'll want to make sure that machine and test case are matched up for a particular run.
- Test Options. Code coverage, filters, parameters, etc.

Let the testing begin!

At this point, you should have your build agents up and running, your build task ready with the appropriate tasks, and know and what and where you'll want to deploy. Your Azure groups will be associated with VS Team Services and you can then execute your test cases against the pools you have created here.

Leveraging Azure for test lab

Challenges

Delivery teams require environments to validate functional and non-functional requirements and machines to run automated tests to validate this functionality. To achieve the level of quality desired, teams can run into infrastructure challenges. Some IT departments struggle with the demands that teams need to enable business agility. Requesting hardware for testing can take a long time. In some cases, it can take weeks or even months. Automated testing, mostly UI testing, can be slow and require a large number of resources, especially when you want to reduce the test run time and run tests concurrently. Teams might be challenged to justify more machines because they won't be fully utilized, except during runs. Finally, corporate policies often lock down the machines too much and this doesn't provide the team the flexibility to configure the systems as needed.

Lab environments are usually considered testing environments with a more limited role than a traditional shared and integrated testing environment. Often, these environments are isolated, single purpose environments that are specially designed to test an application and limit the integration with other systems. Lab environments consist of two general groups of systems. The first group of machines should be considered test clients to use for automated testing to validate the functionality. These machines should include the required clients and configuration combinations to represent the end users. The second group of machines are typically configured as the system under test and represent a test version of the system that is to be validated. Lab environments can consist of both or either one.

Testing machines

Microsoft Azure provides the necessary tools to enable teams to use the cloud to perform manual and automated testing for their systems. For test machines, some key benefits are:

- Elastic or scalable architecture for running tests allows teams to provision additional hardware as their automated testing needs grow. These machines can be shut down when not in use and turned on at the beginning of a test run and then shut back off at the end. Teams pay for only for what they use.
- Connectivity to on-premises resources without having to join Azure VMs to the domain. Azure provides site-to-site VPN connections that are easy to configure and provide the granularity so the VMs only have access to the resources they need. For example, this can be configured to give access to an internal test application only. Azure allows the testing and development teams to manage the resources that they need to make changes and business needs changes.
- MSDN for Dev/Test provides teams with a number of benefits. MSDN subscriptions provide discounted rates for many of the resources for dev/test usage. MSDN subscriptions can be set up on an individual basis with MSDN subscription benefits[18] and through Azure EA accounts to take advantage of the same benefits. MSDN Azure subscriptions also include additional client VMs

[18] http://azure.microsoft.com/en-us/pricing/member-offers/msdn-benefits/

that are not available otherwise. These subscriptions allow teams to create Windows 7, 8.1, and 10 client VMs, which are key to building many lab environments.

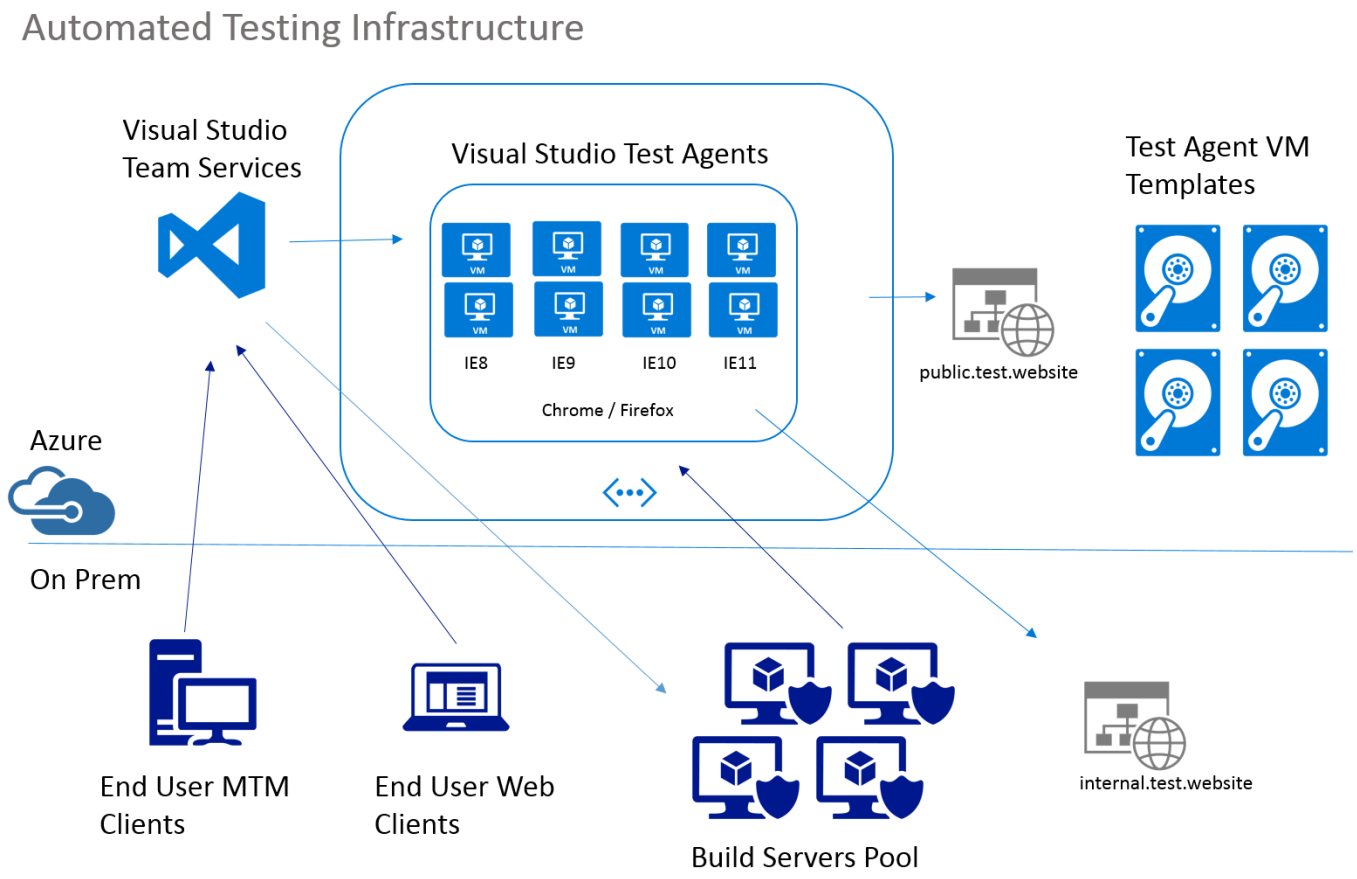


Figure 3 – Visual Studio Team Services integration with Azure test machines

Azure Dev/Test Lab

One exciting feature that isn't available at the time is of writing is the Dev/Test Lab product in Azure. This will be designed to quickly provision environments, provide built in mechanisms for shutting down environments, and include templates to help you get started. See the Azure Dev/Test Lab[19] site for more details and watch for this to become available in the future.

System under test

Azure also provides several benefits when it's used as a system under test for automated testing. Automated testing requires a known starting point, where an environment that can be created easily by using re-runnable scripts, such as Azure Resource Manager Templates, can be used. This also allows multiple team members to create individual test environments to test in isolation, starting with a known data set that is restored before each run. It also allows for environments to be created on the fly and deleted after automated testing runs are complete.

[19] http://azure.microsoft.com/en-us/campaigns/devtest-lab/

Infrastructure as code

Whether you are creating environments to run automated tests, provide the system under test, or both, configuring the environments manually is time-consuming and potentially error prone. To provide the benefits discussed in the previous section, the environments should be created using scripts, often referred to as Infrastructure as Code (IaC). Using IaC, the scripts can be run to provision multiple instances of the environment. Azure Resource Manager Template provide a great way to declaratively define the environment that is idempotent, or can be run multiple times with the same result, so there is no need to verify whether something is already installed before you install it.

Azure Resource Manager Templates can be created from scratch, can be modified from an example such as the Azure Quickstart Templates[20], or created using Azure Resource Group project template in Visual Studio (when the Azure SDK for the .Net Framework 2.6 or later version) is installed. Using the Azure Resource Group project template, you can choose from several templates to get started.

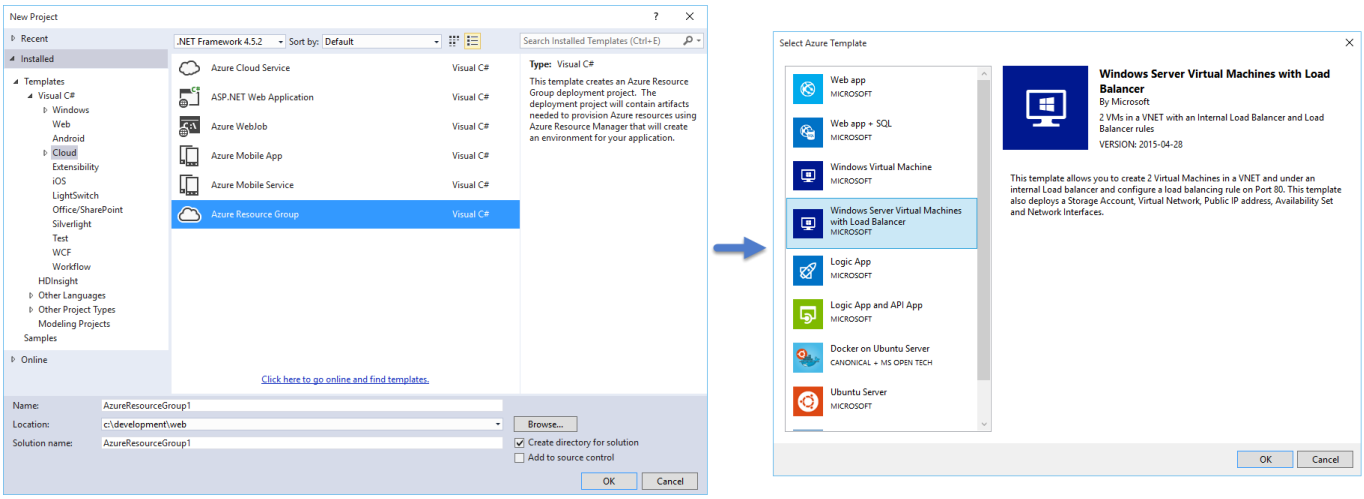


Figure 4 – Azure Resource Group project template allows you to configure resources to provision your environments.

Leveraging Azure for load testing

Load Testing has historically been an expensive endeavor requiring investing in and maintaining dedicated infrastructure. The elasticity of the Cloud makes it a natural fit to reduce this cost and cope with testing today's larger and more complex systems particularly as the demand for near real time response times increase.

This is especially true for load testing scenarios that need to replicate real world usage patterns including geographical distribution on the load. Virtual machines hosted in Azure as described in the "Automated Testing Infrastructure" section above can be used to host load test agents (please see: *Hosting TestController and TestAgents on Azure[21]);* however this has been made much easier with the Cloud-based load testing in Visual Studio Team Services which not only takes advantage of the elasticity of Azure but automatically provision these agents on demand and enables the use of build tasks to automate these runs as part of your automated processes.

[20] https://github.com/Azure/azure-quickstart-templates
[21] http://blogs.msdn.com/b/visualstudioalm/archive/2014/01/13/hosting-testcontroller-and-testagents-on-azure.aspx

While these load tests can be run as part of the CI release process due to the impact on the entire system it is more likely many types of Load Testing, such as stress and capacity scenarios be done at dedicated times as part of scheduled builds. That said using those same (capacity and stress) load test with much smaller durations and user load is an easy way to verify your latest build is running, externally available and meeting your performance SLA's (for that scale).

Similarly, operations team members can verify the health of the overall infrastructure by running these (smaller) Load Tests on a scheduled basis leveraging the same build automation and definitions. For more information on scheduling load tests to use as validation tests, please see: *Scheduling Load Test Execution*[22].

[22] http://blogs.msdn.com/b/visualstudioalm/archive/2015/11/23/scheduling-load-test-execution.aspx

Appendix

Why Selenium is a popular choice

Selenium is a suite of tools that are used to automate browsers across a variety of platforms.

- Selenium is supported across a wide range of browsers including Firefox, Safari, Opera, Internet Explorer, and Chrome. This enables the same tests to be re-used in a multi-platform environment.
- Selenium works on Windows, Linux, and iOS.
- Selenium also supports a large range of programming languages through its driver model. Supported languages include:
  - C#
  - Haskell
  - Java
  - JavaScript
  - Objective-C
  - Perl
  - PHP
  - Python
  - Ruby
- Selenium is open source so there is no upfront license cost. It also has a vibrant developer community, backed by Google.

Selenium products

The Selenium suite includes the following products:

- WebDriver is a code driven approach to automate a browser to conduct regression automation suites and tests. WebDriver makes direct calls to a browser using its native support for automation.
- IDE is a Firefox add-in that can be used to build test script via record and playback.
- Grid is used to scale across multiple machines and run tests in parallel.

How it is used

Selenium WebDriver is the recommended product for automated testing. It is added to a product using the appropriate package manager such as NuGet, GEM, etc.

WebDriver uses an object model to find elements on the page and drive navigation. Selenium supports the Page Object[23] pattern to encapsulate the details of the UI structure from the tests themselves.

References

Information	URL
A good way to catch bugs more quickly and efficiently is to include automated tests in the build process	http://www.martinfowler.com/articles/continuousIntegration.html

[23] http://martinfowler.com/bliki/PageObject.html

Information	URL
Azure Dev/Test Lab	http://azure.microsoft.com/en-us/campaigns/devtest-lab
Azure Quickstart Templates	https://github.com/Azure/azure-quickstart-templates
Cloud-based load testing	https://www.visualstudio.com/features/vso-cloud-load-testing-vs
Create Test Cases from an Assembly of Tests Methods	https://msdn.microsoft.com/en-us/library/dd380741.aspx#assembly
DevOps - Enabling DevOps on the Microsoft Stack	https://msdn.microsoft.com/en-ca/magazine/mt422586
DevOps and Application Lifecycle Management	https://www.visualstudio.com/features/alm-devops-vs
Eradicating Non-Determinism in Tests	http://martinfowler.com/articles/nonDeterminism.html
How to: Create Test Cases from an Assembly of Automated Tests Using tcm.exe	https://msdn.microsoft.com/en-us/library/dd465191(v=vs.110).aspx
Load test in the cloud	https://www.visualstudio.com/get-started/test/load-test-your-app-vs
MSDN Microsoft Test Manager Guide	http://msdn.microsoft.com/en-us/library/ms182409.aspx
MSDN subscription benefits	http://azure.microsoft.com/en-us/pricing/member-offers/msdn-benefits/
Our DevOps Journey	http://stories.visualstudio.com/devops
Page Object	http://martinfowler.com/bliki/PageObject.html
Running Tests from a Test Plan Using the Command Line Utility	https://msdn.microsoft.com/en-us/library/ff942469.aspx
Team Foundation Build environment variables	https://msdn.microsoft.com/en-us/library/Hh850448(v=vs.120).aspx
Visual Studio Test Tooling Guidance	http://vsartesttoolingguide.codeplex.com

In conclusion

This concludes our journey. The goal was to equip software testers with a firm understanding of the concepts of test automation that will enable them to develop better solutions to the testing challenges of their particular projects.

We hope you have found this guide useful and wish you success in your software adventures.

Sincerely,