testim

Automation Test Strategy & Design

for Agile Teams

Table of Contents

Automation Test Strategy & Design

Software test automation has existed in one form or another for many decades. The benefits of test automation are huge in terms of increasing product quality while reducing costs and time to market.

A few organizations have been immensely successful in automating their tests as part of their application development cycle. But many have had mixed results with test automation.

Oftentimes, teams still continue to manually test, or they otherwise struggle as they embark on their test automation journey. After all, this journey can raise difficult questions, such as, “Can every manual test really be automated?” and “What are some of the strategies and best practices of test automation?”

In this ebook, we’ll address these questions. Additionally, we’ll offer a constructive definition of test automation, its challenges, and how to get started. Finally, after discussing some test automation strategies and best practices, we’ll provide a brief overview of how artificial intelligence and machine learning is being introduced into software testing.

Test Automation: Challenges & Getting Started

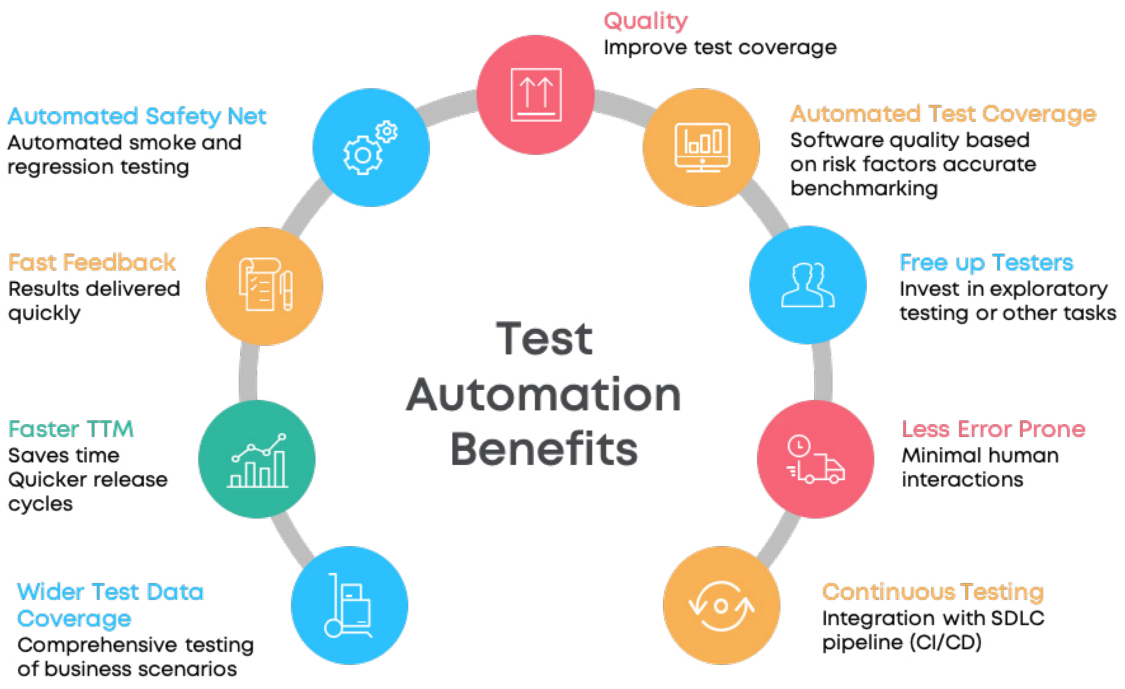
What Is Test Automation?

Test automation means testing software without any human intervention. You execute a test that's either scheduled for a specific time or triggered by an event, such as the completion of a build. The actual test result is then validated against expected results using assertions or validations.

Test automation reduces costs, time to market, and testing time. It also improves software quality since, with reduced testing costs and time, you can run more tests and increase test coverage.

There's a common misconception that test automation will replace manual testing. It isn't true. Not every test can be automated. For example, exploratory testing still needs to be done manually.

Test Automation Benefits

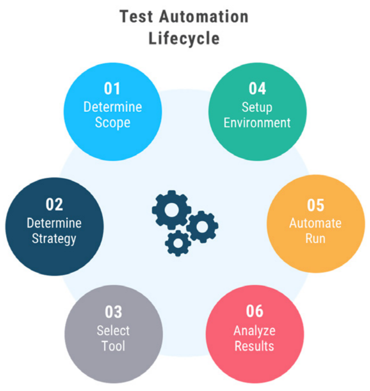

Test Automation Benefits
Quality
Improve test coverage
Automated Test Coverage
Software quality based on risk factors accurate benchmarking
Free up Testers
Invest in exploratory testing or other tasks
Less Error Prone
Minimal human interactions
Continuous Testing
Integration with SDLC pipeline (CI/CD)
Wider Test Data Coverage
Comprehensive testing of business scenarios
Faster TTM
Saves time Quicker release cycles
Fast Feedback
Results delivered quickly
Automated Safety Net
Automated smoke and regression testing

Other benefits of test automation include:

- A safety net of automated smoke and regression tests before you promote your software to production.
- The flexibility to do risk-based testing - choosing the tests that you want to run based on risks.
- You can also incorporate automated tests with your continuous integration/ continuous deployment processes to achieve continuous testing.

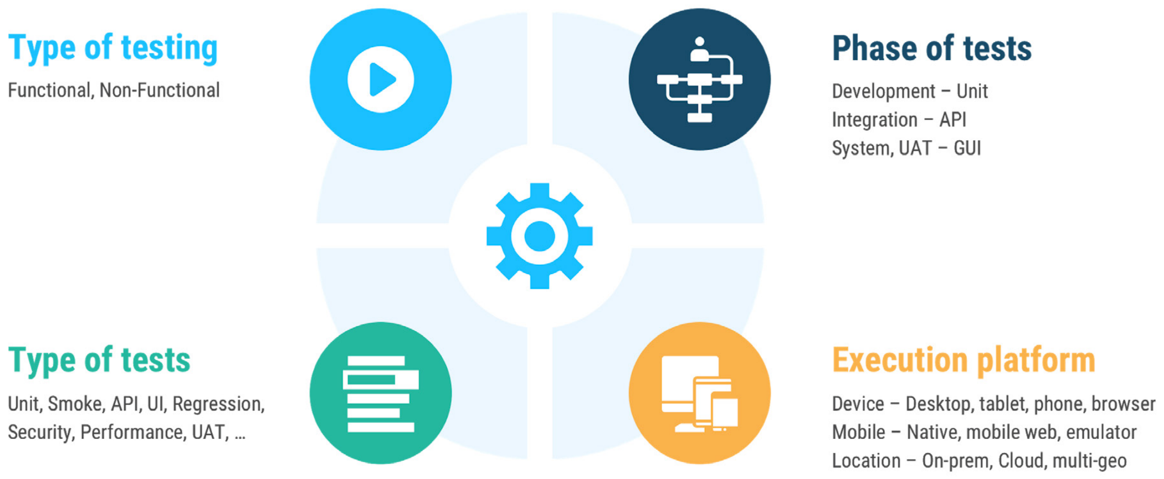
Test Automation Lifecycle

The test automation lifecycle is comprised of a few things. First, you determine the scope of testing. You also decide how you're going to test, what tools you'll use, and what the execution of those tests will look like (meaning you set up the environment and automate the running of the tests).
Finally, you'll analyze results.



Test Automation Classification

Test automation can be classified based on type of testing, type of tests, your SDLC phase and execution platform. This is important because it is required to identify the right strategy, right toolset and also the set the right expectations from the tests.


Type of testing
Functional, Non-Functional
Phase of tests
Development – Unit
Integration – API
System, UAT – GUI
Type of tests
Unit, Smoke, API, UI, Regression, Security, Performance, UAT, ...
Execution platform
Device – Desktop, tablet, phone, browser
Mobile – Native, mobile web, emulator
Location – On-prem, Cloud, multi-geo

Challenges of Test Automation

Test automation requires a software development mindset and follows the same set of best practices, like agile, to be successful

Test automation requires similar investment and priority as software development. It can start as a pet project but requires a corporate mandate for wider adoption. You need your testers and developers to have the right attitude. You also want some experience with test automation, the right toolset, and a stable test environment.

Resources &

- Initial investment (cost, time, effort)
- Competing corporate initiatives and priorities
- Lack of clear mandate and realistic goals
- Not treated like other software dev projects

Culture & Skillset

- Team's approach, attitude, resistance to change
- Lack of experience, false sense of security
- Relies on programming languages only
- Underestimate the amount of time needed
- Creating large, end-to-end tests

Tools &

- Not using proper tools & framework
- Legacy or constantly changing code
- Not having a controlled and stable test environment

Proces

- Not reusing automation code
- Not having a test data strategy in place
- Not making your automated tests readable

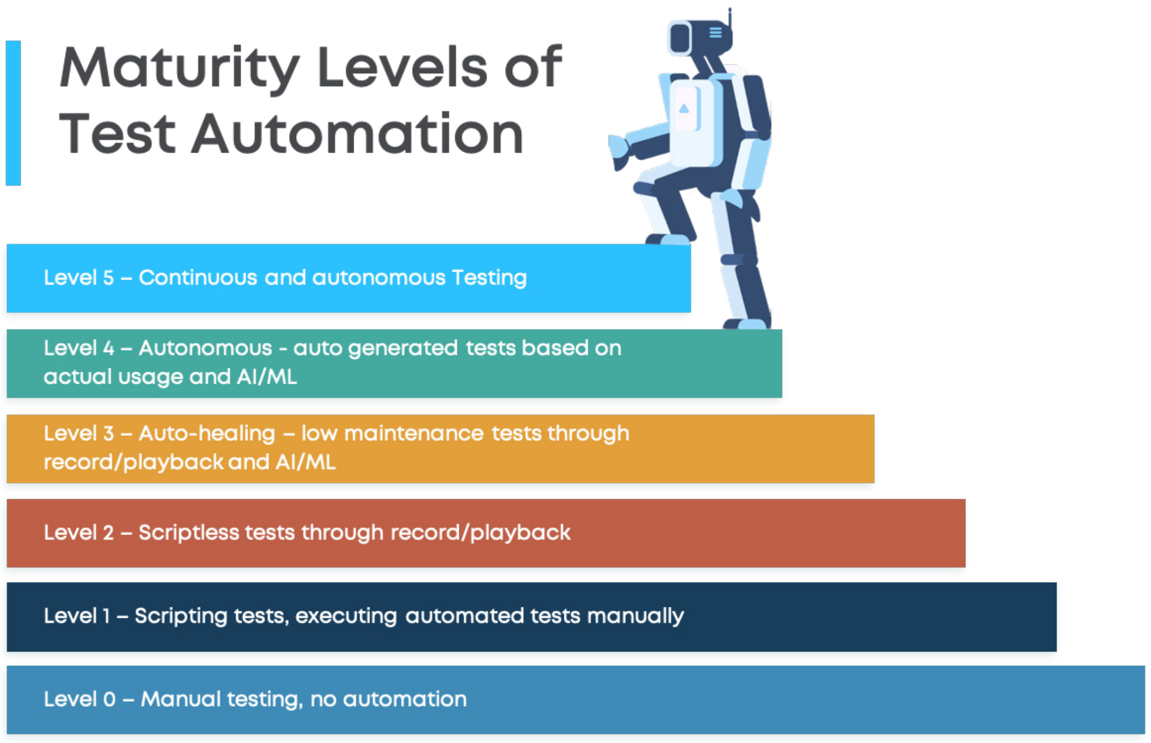
Top 10 reasons for Flaky Automated Tests

1. Not having at framework
2. Using hardcoded test data
3. Using X,Y coordinates or XPath for element recognition
4. Using shared test environments. Not using a stable test environment
5. Having tests that are dependent on one another
6. Test not starting in a known state
7. Tests not managing their own test data
8. Not treating automation like any other sofware development efort
9. Failure to use proper synchronization
10. Badly written tests

Maturity Levels of Test Automation

The maturity levels of test automation start from 100% manual testing to continuous testing where test cases are created automatically based on application usage. These levels of maturity mostly apply for UI testing, but can also be applied to other types of software testing.

In many cases organizations start with test automation by writing scripts to execute tests and execute them through a cron job or integrate with CI for continuous testing.


Maturity Levels of Test Automation
Level 5 – Continuous and autonomous Testing
Level 4 – Autonomous - auto generated tests based on actual usage and AI/ML
Level 3 – Auto-healing – low maintenance tests through record/playback and AI/ML
Level 2 – Scriptless tests through record/playback
Level 1 – Scripting tests, executing automated tests manually
Level 0 – Manual testing, no automation

For UI interactions, recording user interactions with the applicatinon under test (AUT) and playing it back to test the application would be the Level 2.

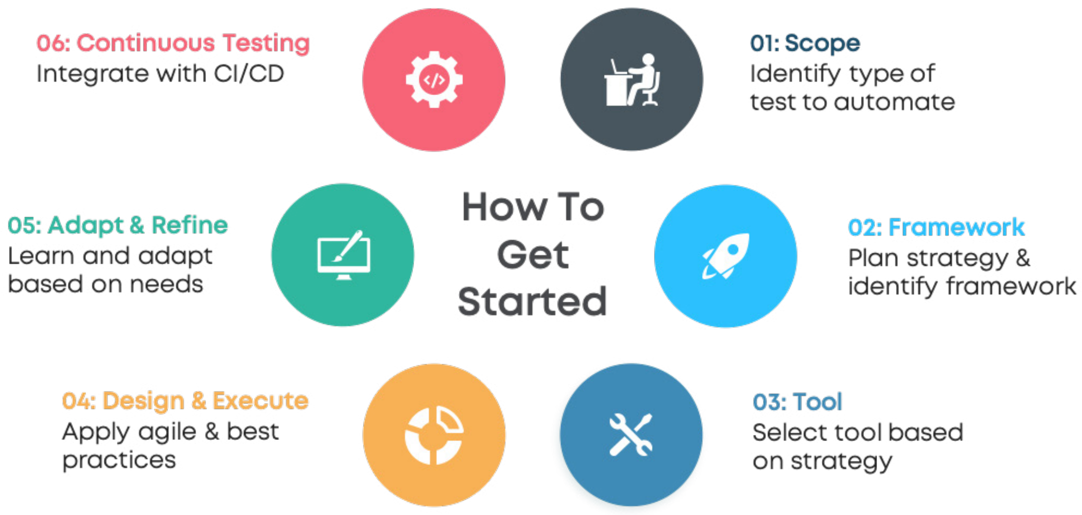
In Level 3 script based or script less tests are made immune to small changes in the UI – like changes in an UI element. Reinforced learning techniques in Artificial Intelligence (AI) and Machine Learning (ML) is used to locate the element even when few attributes of the element have changed in the DOM and then execute the test. This reduces the amount of maintenance needed for automated UI tests. This is also applicable for pixel level validation tools.

In Level 4, automated tests are created based on learning the usage of an application by actual users. This helps tests the paths used by the user while traversing a business scenario like booking a travel ticket or buying something from a shopping cart application.

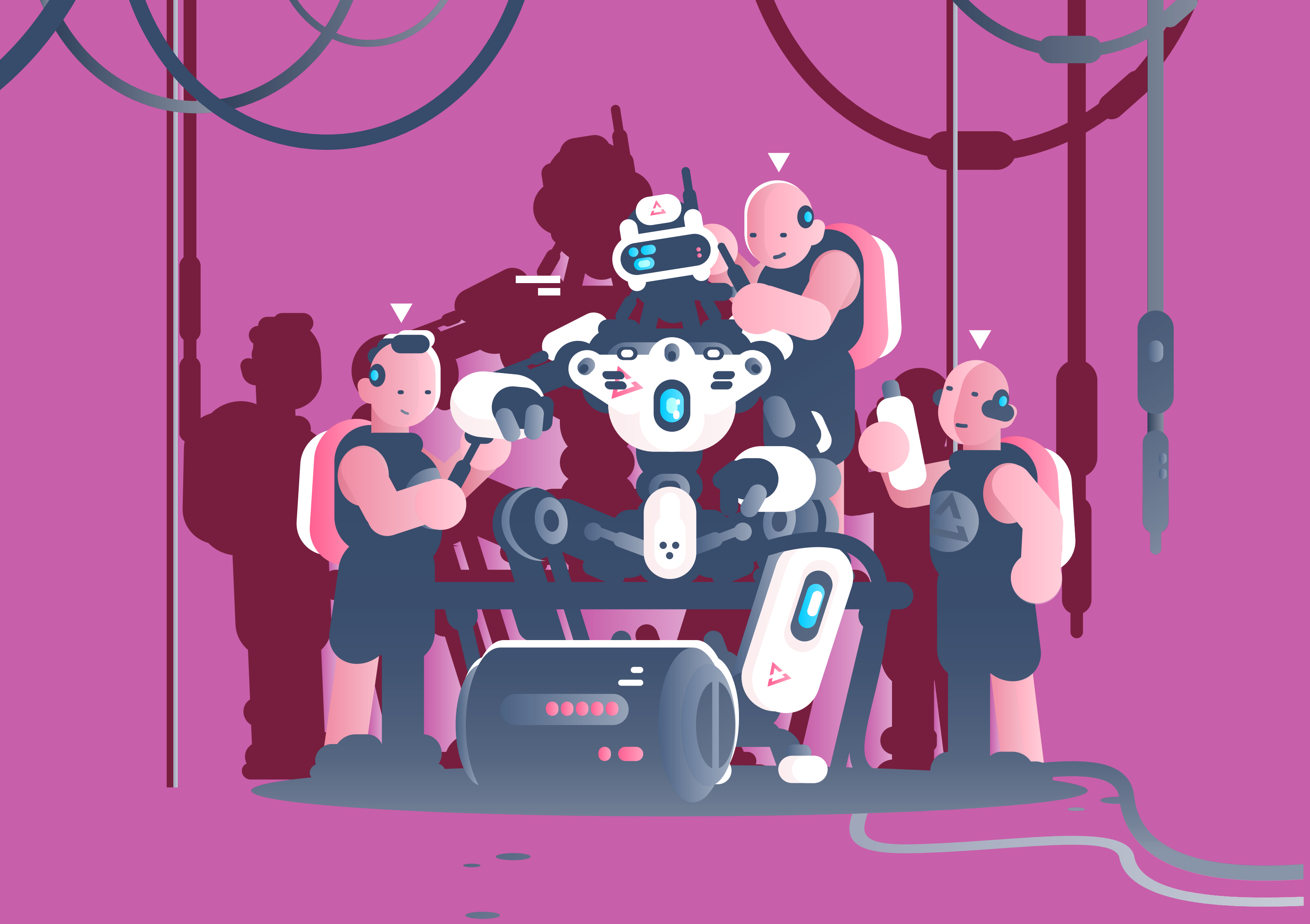
Getting Started

For those of you who are new to test automation, you may be wondering - how do I get started? To do so, you have to think about:

- The scope – what is it that you want to test and automate the testing of
- The framework and tool set – what tools will be appropriate for creating and executing your automated tests. In many cases you may be choosing multiple tools.
- Then you have to design and execute your tests followed by refining your tests to adapt to your environment and scenario
- The next step will be to integrate your tests with CI/CD processes that you may have in house to achieve continuous testing



How To Get Started
01: Scope
Identify type of test to automate
02: Framework
Plan strategy & identify framework
03: Tool
Select tool based on strategy
04: Design & Execute
Apply agile & best practices
05: Adapt & Refine
Learn and adapt based on needs
06: Continuous Testing
Integrate with CI/CD



Test Automation Strategy Considerations

Areas to Consider

The likely areas to consider as you get started with your test automation journey can be as follows:

1 | Where Do We Begin?

2 | Applying Agile Principles to Test Automation

3 | Test Automation Candidates

4 | Test Modeling and Test Coverage

5 | Choosing Automation Tools & Implementation

6 | Test Data & Service Virtualization

7 | Managing & Maintaining Automated Tests

8 | Test Automation Metrics

9 | Continuous Testing

10 | ROI: The Cost of Test Automation

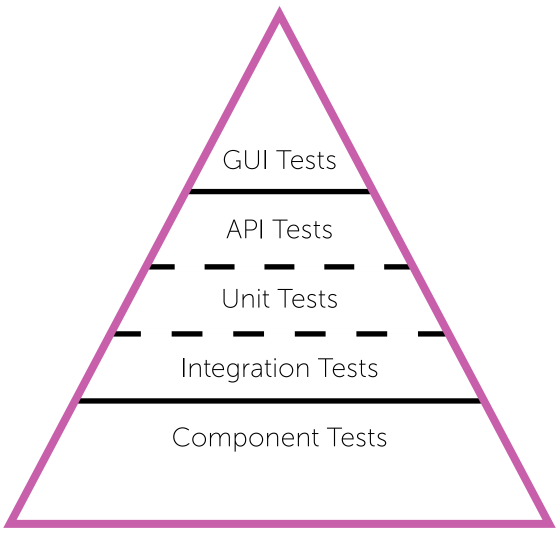
Where do we begin?

The first step is about identifying your scope. What do you want to automate first? Mike Cohn lays out a Testing Pyramid with Unit tests at the bottom and GUI tests at the top of the pyramid. Everything else like component level testing, integration testing, API testing is in between.

Typically, you identify the low hanging fruit that delivers the big bucks. Identify what hurts most and what delivers the biggest value to your organization. If GUI tests require a lot of manual effort, that may be your starting point. On the other hand, if your unit tests are not integrated and automated, that can be a starting point. If API is the service you provide to your customers, making sure those APIs deliver what they are expected to yields the biggest value.

You may not find a single tool that delivers all your test automation needs. Before choosing a tool or framework for your needs, try it out in your environment with your test use cases. If it meets your needs then expand adoption of that solution.

- **Use Multi-layered approach**
  Use "Testing Pyramid" (Mike Cohn) to locate your starting point
- **Identify what hurts the most?**
  Cost, TTM, Quality
- **Identify what delivers the biggest value?**
  Application, Pain Point
- **Choose Automation based on overall needs**
  Implement a steel thread for experience and validation
- **Continue with additional implementations**
  Additional Applications, Pain Points



Applying Agile Principles

Automated test development should be treated like any other software development project. All of your favorite agile principles apply when automating tests.

Each dev sprint will have its own test creation sprint to test the features built during that sprint. These tests then become part of the regression suite as you move to the next sprint cycle.

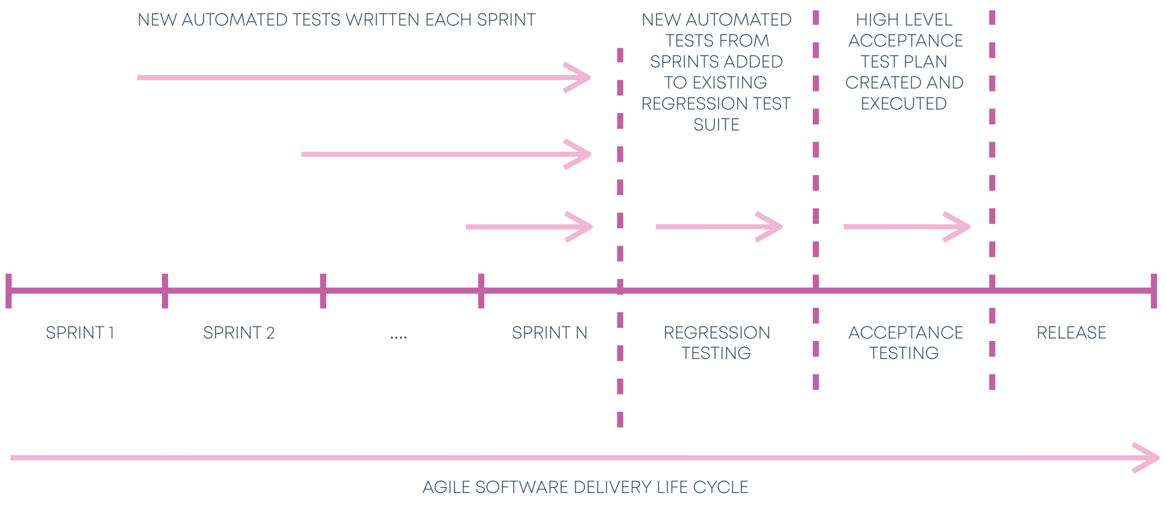
- **Keep it Simple**
  Use incremental and Iterative approach to test automation and test design
- **Every DEV iteration has its own testing phase**
  Implement regression testing every time new functions or logic released.
  User acceptance tests executed at the end of each sprint.
- **Whole team approach**
  Testers and developers work together
- **Apply Agile Coding practices to creating tests**
  Use BDD, TDD as appropriate for your environment
- **Invest time to do it right & Learn by doing**

Testing shouldn't be an activity done only at the end of a sprint by a designated tester. As with agile's emphasis on teams over processes, it's important to remember that test automation is a team effort.

Everyone on the team must be on board, and expectations for testing must be clearly outlined.

Agile Testing

Agile test automation



NEW AUTOMATED TESTS WRITTEN EACH SPRINT
NEW AUTOMATED TESTS FROM SPRINTS ADDED TO EXISTING REGRESSION TEST SUITE
HIGH LEVEL ACCEPTANCE TEST PLAN CREATED AND EXECUTED
SPRINT 1
SPRINT 2
....
SPRINT N
REGRESSION TESTING
ACCEPTANCE TESTING
RELEASE
AGILE SOFTWARE DELIVERY LIFE CYCLE

In any agile software development lifecycle (SDLC), you'll have automated tests created for a sprint while the sprint is being developed. That way, the tests are ready to execute toward the end of the sprint. These new tests are added to the regression suite and are executed during or after every sprint.

Some or all acceptance tests may be executed before code is released to production.

Test Automation Candidates

common problem I usually see is that teams start off by trying to automate everything. The problem is that, not every test can be automated. For test cases to be automated, you should look for tests that are deterministic and don't need human interaction. They are hard to test manually, and need to run more than once possibly using different data sets or on differentbrowsers or for load testing.

You should consider using automation for any software testing activity that saves time, improves quality of testing andtesting efficiency.

What tests shouldn't you automate?

In general, tests that you execute once or applications that are not testable unless you actually use it should not be automated. Exploratory tests, or tests that do not provide predictable results are additional examples, but there can be exceptions.

What should be automated?

- ✓ Safety net through automated smoke & regression
- ✓ Unit, component, API, Load/Perf, GUI level tests
- ✓ Repetitive, tests that run against different data sets
- ✓ Tests integrated with CI/CD builds and releases

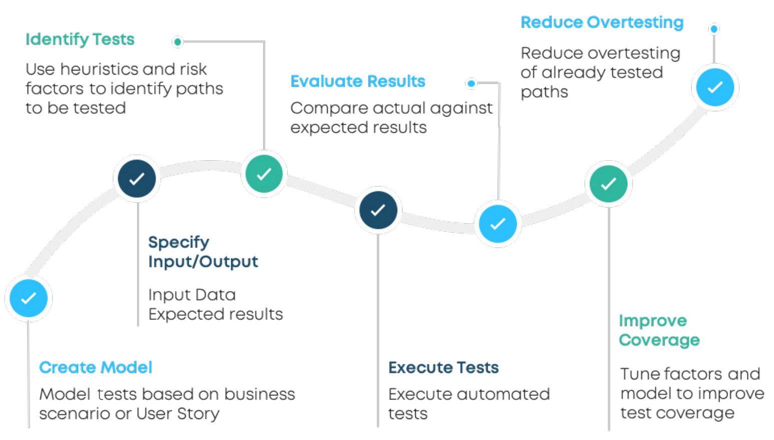
What shouldn't be automated?

- ✗ One-time tests, applications not testable
- ✗ Usability, exploratory, ad-hoc tests
- ✗ Tests that don't fail or have predictable results

Test Modeling and Test Coverage

Test models can be broadly categorized into three types.

- **Event-based model :** Based on GUI events that occur at least once.
- **State-based model :** Based on GUI states exercised at least once.
- **Domain model :** Based on the application functionality.


Identify Tests
Use heuristics and risk factors to identify paths to be tested
Evaluate Results
Compare actual against expected results
Reduce Overtesting
Reduce overtesting of already tested paths
Specify Input/Output
Input Data
Expected results
Create Model
Model tests based on business scenario or User Story
Execute Tests
Execute automated tests
Improve Coverage
Tune factors and model to improve test coverage

In model-based test automation, you create a model of your application and specify the inputs and outputs to the application. You identify the list of paths that needs testing. This can depend on the business scenario, events or states that you are trying to test.

You may apply heuristics and risk factors while trying to come up with a test list that can provide you with an optimal test coverage based on risks taken. Heuristics in this case is a way of prioritizing certain computation paths over others based on feature functions that you are trying to test and applied mainly when you add new capabilities or make changes to an existing application.

Next you execute tests and evaluate results. Many test automation solutions create test automation scripts and provide fit-for-purpose test data to optimize your tests and test data coverages. Software testing based on a model, improves test coverage without increasing over testing.

Automation Tools & Implementation

The tool selection process involves

- Understanding your requirements and key criteria in the decision-making process.
- If you already have some type of test automation tool in house, consider that as a baseline.
- You may leverage Pugh Matrix technique for analysis to make your decision. Pugh Matrix technique is based on key criteria, a baseline and weighting to arrive at a score.
- If you are an Agile development shop, it is always wise to pick an Agile friendly tool: an example will be to choose a tool that ties test cases and test results back with user stories and requirements in your agile management tool.

As for implementation of test automation, you have the option to go code based or code less. This will depend on the target users who will be using the tool. Test engineers will prefer a code based or hybrid solution whereas manual testers will prefer codeless solution. Your choice of tools will also depend on your test execution environment -desktop/server, web, mobile device, emulators etc. Not every test automation solution supports every environment.

Test Data

Test data is an important aspect of testing. In many cases it drives testing as in data driven testing. Before you start testing, you need to understand what your test data requirements are. Test data coverage is as important as test coverage itself. Testing teams typically derive test data from production databases through subset and masking. The test data is then made available in a database from where test data is provisioned to individual teams for testing. Test data may get burnt after testing. When that happens, you re-provision the test data.

Test data from production isn't the only source of test data. To reduce reliance on production data, for security reasons like GDPR or to improve test data coverage, I have seen my customers use synthetic test data generation techniques to create test data. TDM tools from CA Broadcom, Informatica, and others provide different sets of capabilities on creating test data.

Service and Database Virtualization

Today's applications are no longer monolithic and rely on various other components and applications that have their own SDLC time lines. When your application or component relies on other components and these components aren't available you virtualize these components using service virtualization. This can be done for APIs, databases and other services. Variety of tools are available in this space including tools from CA Broadcom, Parasoft, Delphix, Tricentis and others. Simple stubs and mocks are also used to simulate dependent services.

Test Management

It is important to deliver features as described in the requirements by business analysts.

How do your BA's know that your application has delivered the features that they had requested? This is where linking the user stories from requirements with the test cases and test results in an Agile requirements management tool becomes important.

BA's don't typically have access to test systems but they do have access to requirements management tools. Tying test results back to requirements makes it easier for BA's to have better visibility into where the feature is in the SDLC as well as whether the feature delivers the user stories outlined in the requirements

Maintaining Automated Tests

Test maintenance is one of the biggest challenges test engineers encounter when doing Test Automation, especially for applications with an UI. Minor changes to the UI break most automated tests. Advanced test automation frameworks like Testim allows you to create automated tests that reduce test maintenance. Testim self-heals automated tests whenever there are minor changes to the UI cutting down overall test maintenance to less than 10%. It does this using machine learning techniques in AI.

You can also reduce test maintenance by modularizing your tests and by using other best practices that we will cover later in this ebook.

Software Test Metrics

Software test metrics are used to measure and monitor your team's progress with test automation. These metrics can convey absolute data, like time taken to run a test, or information derived from absolute data.

Bugs found during testing can be helpful to determine if your software testing efforts including using test automation is bringing value.

Continuous Testing

One of the main reasons for Test Automation is to be able to achieve continuous testing. As you implement continuous integration, the next step is the ability to execute tests every time you have a new build. Automating your tests helps you achieve that.

You can then integrate your automated tests with your continuous delivery and continuous release tools to integrate testing into your entire SDLC pipeline enabling you to do continuous testing as you promote builds from one environment to another.

Test automation is the enabler for continuous testing. You can achieve continuous testing for both your functional and non-functional tests including component level performance tests.

Test Automation Metrics

Some of the most important test metrics to consider is what percent of manual tests you're executing and what your mean time to debug (MTTD) a failing automated test is.

The more manual tests you have, the longer it will take to verify that your application is ready for release. A high MTTD is an indicator that your automated test code is not of good quality.

Test automation flakiness should be zero. This is a good indicator as to whether your automation tests are reliable or not.

ROI & Cost of Test Automation

Determining the ROI of your test automation efforts can be tricky. A common calculation that some folks use to get a rough estimate of their test automation costs is:

Tools cost + Labor cost to create automated tests + cost of automated test maintenance

If the cost of executing the tests manually is more than automation costs, it makes sense to automate the tests.

This can help you decide whether a test case is even worth automating as opposed to testing it manually from a cost perspective. There are other benefits like time savings, ability to run with broader data sets, improved quality etc., which should also be factored in.

ROI quickly adds up as you re-run your automated test suites.

Test Design Best Practices

Now that we've covered some test automation strategy considerations, we'll look at some best practices.

Test Automation Process

It is quite normal to assume that your applications are going to change over time. And since you know change is going to happen, you should start off right from the beginning using best practices or design patterns. Doing so will make your automation more repeatable and maintainable. Common test automation design patterns that many teams use to help them create more reliable test automation are outlined below.

Single Responsibility Principle

Model behavior of application using page objects - e.g. login, home page. Reusability, easy debugging and maintenance

Screenplay Pattern

Breaks down page objects to smaller more maintainable, reliable and readable

Ports and Adapters

Decouple test code to swap slow components with fast simulators to run test and app being tested in the same process

Presenter First

Organize code and development behaviors to create completely tested software using TDD. Modification of MVC approach.

Single Responsibility Principle is a popular strategy to use when creating your test automation by modeling the behavior on your application. Creating simple page objects that model the pieces of your software that you are testing against, can do this. Say, for example, you would write a page object for login or a page object for a homepage. Following this approach correctly makes use of the single responsibility principle.

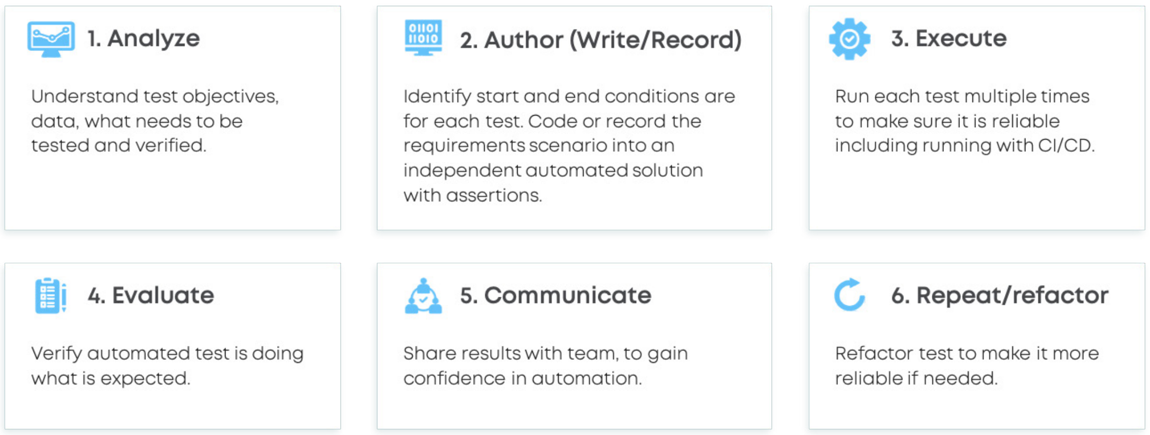
The Screenplay pattern takes page objects and chops them down into really tiny pieces for better maintainability and reliability.

The Ports and Adapters design strive to make sure that you are using the single responsibility principle so that an object should do only one thing and have one reason to change. You decouple your test to allow you to swap slow components with fast simulators to prevent slowing down your tests.

Presenter First is a modification of the model-view-controller (MVC) way of organizing code and development behaviors. This helps to create completely tested software using a test-driven development (TDD) approach.

Test Automation Process

The test automation process can be reduced to a six-step, cyclical process as shown in the diagram below:



1. Analyze

Understand test objectives, data, what needs to be tested and verified.

2. Author (Write/Record)

Identify start and end conditions are for each test. Code or record the requirements scenario into an independent automated solution with assertions.

3. Execute

Run each test multiple times to make sure it is reliable including running with CI/CD.

4. Evaluate

Verify automated test is doing what is expected.

5. Communicate

Share results with team, to gain confidence in automation.

6. Repeat/refactor

Refactor test to make it more reliable if needed.

You start with analyzing your testing requirements and objectives, followed by authoring your tests through scripting or recording or a combination, executing your tests to make sure they run reliably, evaluating the results, communicating the results to the team to gain confidence and fine tuning the tests to make them more reliable. If you notice a flaky test, refactor it to make it more reliable. Most importantly, delete any tests that aren't reliable and haven't been fixed within a given time frame.

Periodically ask the team if an automated regression test is still adding value. Pruning old tests will save you maintenance time and ensure you're only running ones that are useful.

Areas of Testing: GUI Testing

For web-based GUI testing, here's an example of what you may want to analyze and test:

GUI Testing - Sample Scope			
1	Size and position of GUI elements	8	Error messages
2	Clear and well-aligned images	9	Required fields
3	Font and alignment of text	10	Abbreviations inconsistencies
4	Date and numeric fields	11	Progress bars
5	Screen Validations	12	Shortcuts
6	Navigations (links)	13	Screen rendering
7	Usability conditions and data integrity		

In a similar fashion, you can classify different areas of testing for API testing, integration testing, security testing, and the like. Analyze your needs and then start implementing tests based on your objectives

Test Automation Design Strategy and Best Practices

There are many test automation design best practices. I have listed the ones that I found quite useful below:

- Prioritize what you want to test. You can't automate everything in your first attempt at test automation.
- Reduce, reuse, recycle. Revisit existing regression tests and recycle the ones that are no longer providing any value.
- Create short, structured, single-purpose tests that are independent and can be executed in parallel.
- Compose complex tests from simple ones.
- The initial state of a test should always be consistent.
- Use wait-for mechanisms instead of hard coded sleep to improve stability and synchronization between test execution and the application.
- Use abstractions where possible for reusability, clarity, and ease of maintenance.
- Use assertions to validate automated tests.
- Reduce the use of conditions whenever possible.
- Use setup and teardown steps to prepare for your test and cleanup after the test executes.
- Use data-driven tests instead of hardcoding data, use design patterns while designing your tests
- Use a stable test environment to run your test. This makes it easier to debug your tests if they're failing.
- For web UI testing, create tests that are resistant to small changes in the UI.
- Follow a test naming convention that is aligned with your source code naming convention so you can easily identify and locate your tests.
- Capture screenshots for easy debugging of your tests.
- Setup detailed reporting for your test results

UI is just the top of the Testing Pyramid. Just testing the UI isn't enough. Unit tests, integration and other tests must be done in conjunction with UI tests to improve quality of your software and be successful with your test automation efforts.

As you continue with your test automation journey, you'll learn and have your own best practices for your organization. It's important to capture, share, and use that knowledge across departments.

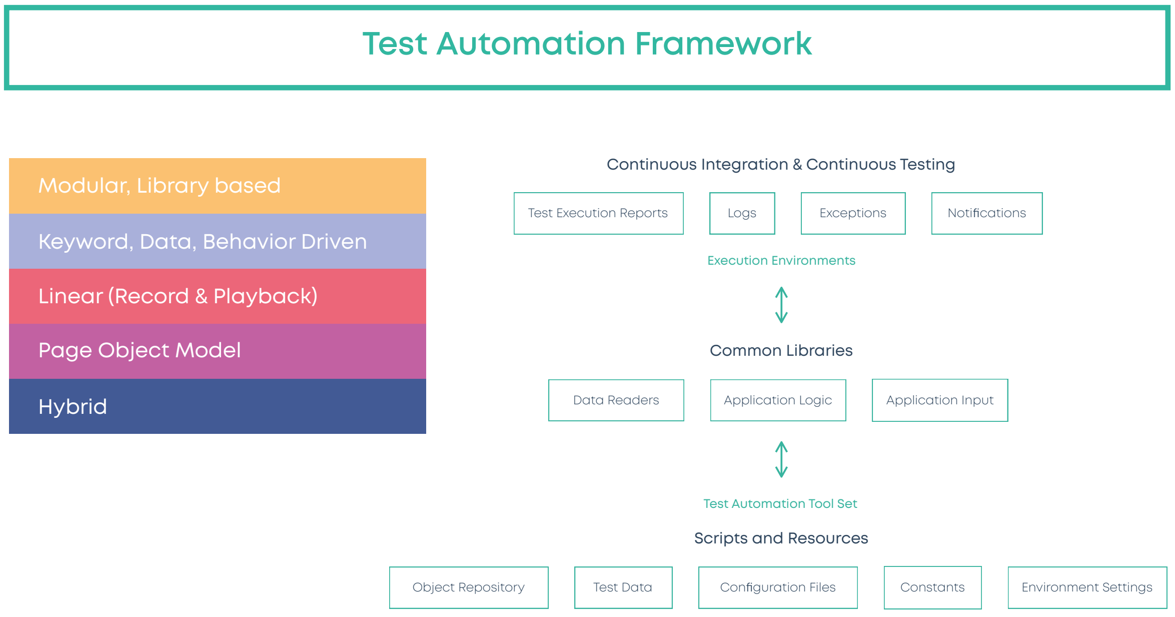
Choosing a Test Automation Framework and Tools

Test Automation Frameworks

A test automation framework is a combination of set protocols, rules, standards, and guidelines that together can be used to leverage the benefits of the scaffolding provided by the framework when implementing test automation. There are many test automation frameworks and I will try to cover few categories with some examples for each.

When Mercury pioneered performance testing in mid or late 1990s, it was just a load generating tool. Over the years it added support for more protocols, more execution platforms, more applications and adding enterprise level features around reporting, logging mechanisms, Exception handling, notification etc. Later LoadRunner, was tightly integrated with HP Unified Functional Test (QTP), Application Lifecycle Management (ALM) and the architecture was enhanced to support modern applications on the internet with millions of virtual users connecting from all over the world. It provided the authoring and execution environment, included libraries for integration with other applications and also featured UI to create your automated tests easily and reports. This is what I mean by test automation framework.

A test automation framework provides the execution environment for the automation test scripts in addition to capabilities for developing automated tests, executing tests, test reporting and integration with other CI/CD toolsets.



The advantages of test automation frameworks, can be in different forms like the ease of scripting, scalability, modularity, understandability, process definition, re-usability, cost, maintenance etc. To reap these benefits, developers and testers are advised to use one or more of the test automation framework as may be appropriate for your needs.

When you have many developers and test automation engineers working on different modules of the same application, it is advisable to select a single test automation framework to avoid situations where each of the developers implements his/her approach towards automation.

There are different types of test automation frameworks:

- Modular or library-based: primarily employed for their reusability.
- Keyword, data, or behavior-driven: allow you to control your tests based on feature or
  test data.
- Linear: records user scenarios and plays them back for testing.
- Page object model: reduces duplication and enhances maintenance.
- Hybrid: a combination of the above frameworks.

Today, most commercially available test automation solutions provide a test automation framework of some sort.

Below are some criteria to be considered before deciding on a test automation framework in addition to some examples of open-source and vendor sourced frameworks.

Criteria/What to consider?	Open Source	Vendor Sourced
Easy and fast authoring	Selenium	Testim
Reusability & test coverage	Carina	Tricentis
Script bases, scriptless or hybrid	Google EarlGrey	Mabl
Stability of tests, low cost maintenance	Cucumber	BlazeMeter
Minimal manual intervention	Watir	UFT/QTP
Root cause analysis/debugging	Appium	LeanFT
Reports	Robot Framework	Automation Anywhere
Integration with APM	JMeter	CodedUI
Testers (developers, QA, manual)	Gauge	TestComplete
Open source/non-open source	Robotium	Sikuli

Test Automation Tools

There is no “correct” test tool for automation testing. The best test automation tool that your team can benefit from depends on your team’s unique needs and skill set.

I always recommend that you run a two or three week proof of concept (POC) for each tool that you are considering and review your team’s feedback in the process before committing to a tool. Find out if the tool has an active user base and select tools that other companies are using. Determine how easy it is to hire folks that have the skills needed to create your automated tests. Review product roadmap and make sure the tools you select will handle future features and technologies. Finally, evaluate costs, not only the initial cost of deployment but also maintenance or subscription costs as appropriate.

Below is a summary of the guidelines for tool selection and different categories of software testing tools.

Criteria/Guidelines

- No “correct” tool for test automation
- Depends on your unique needs
- Execute a 2-week POC before selection
- Review extensibility, ease of use, reporting, debugging, integration, version control and other features
- Review team feedback
- Review product roadmap
- Evaluate cost including maintenance

Categories

- Test data management tools
- Test modeling tools
- Visual pixel validation
- Visual non-pixel validation
- Service virtualization
- Database virtualization
- Continuous Integration
- Continuous delivery
- API testing tools
- Test management tools
- Automated testing tools
- Cross-browser testing tools
- Web UI testing tools
- Mobile UI testing tools
- Load testing tools
- Defect tracking tools
- Security testing tools

Examples of Test Automation Tools

Below are examples of some test automation tools. This list is based on my use of some of the tools first hand, reviews and recommendations from colleagues and customers and domain thought leaders in this area.

I haven’t covered few categories like Service Virtualization, Test Data Management etc. A simple google search will provide you with list of tools in that area

Test Automation Tools

Test Management

- qTest
- TestPad
- PractiTest
- Qmetry
- Testrail
- TestCollab
- QAComplete
- TestLinkz

Cross-browser Testing

- Testim
- LambdaTest
- Browsera
- CrossBrowser Testing
- SauceLabs
- GhostLab
- BrowserShots

Defect Tracking

- JIRA
- Mantishub
- FogBugz
- Bugzilla
- BugNet
- BugGenie
- RedMine

Security Testing Tools

- NetSparker
- Snyk
- Acunetix

CSS Validator Tool

- W3C CSS Validator
- Code Beautify

Model Based Testing

- CA Agile Requirements
- Designer Gannett USA Today

Load Testing

- BlazeMeter
- LoadRunner
- WAPT
- LoadUI Pro
- SilkPerormer
- JMeter
- LoadFocus
- LoadImpact
- WebLoad

Automated Testing (functional, regression)

- Testim
- Ranorex
- Selenium
- QTP
- Watir
- Testim
- Applitools
- Telerik
- TestComplete
- Katalon

Mobile UI Testing

- Appium
- Expresso
- Perecto
- ExperiTest
- Robotium
- MonkeyRunner
- Ranorex
- UI Automator

API Testing Tools

- SoapUI
- SOAPSonar
- WebInject
- Tricentis

Web UI Testing

- Testim
- Ranorex
- Studio
- Rapise
- AutoIt
- CubicTest
- eggPlantUI
- Fitnesse
- Ascentialtest

Visual Validation

- Applitools (pixel)
- Percy.io (pixel)
- Chromatic (pixel)
- Screenr (pixel)
- Galenframework (non-pixel)

New Trends in Test Automation

AI/ML in Test Automation

Brief Overview

- Applications complexity has increased significantly & so has software testing
- AI/ML is expected to play a major role in software testing
- Software testing tools have started incorporating AI/ML
- AI enables non deterministic tests
- AI identifies problem areas based on past tests (defects, results, logs, test cases, source code etc)
- AI helps understand system behavior better

Scenarios

Auto heal test scripts for small application changes for effective regression testing and to reduce test maintenance.

Auto-validate test inputs based on ML without manual user inputs validations

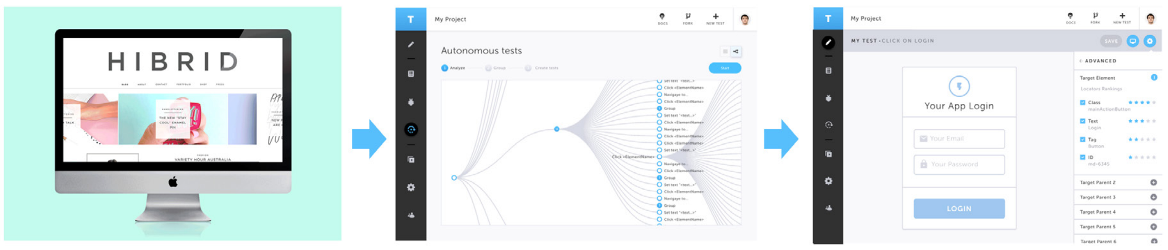
Auto-generate test cases based on user interactions & use cases

AI/ML based Tools

Testim
Applitools
TestCraft
AccelQ
Mabl
AutonomIQ
AppvanceIQ

Applications today are increasingly complex. They interact with many other applications using APIs, and user interactions with these applications can be on a variety of devices. As a result, the complexity of testing these applications has grown in a non-linear fashion.

AI and machine-based intelligence are expected to play a key role in solving the complexity of testing modern-day applications. AI identifies problem areas based on past tests (defects, results, logs, test cases, source code etc) and to help understand system behavior better. AI/ML is expected to make testing from driver based to driver-less, from monitored to non-monitored and from manual creation, execution, maintenance of test cases to automated without or with minimal human involvement.


LEARN BY OBSERVATION AGGREGATE USER- ACTION FLOWS CREAT TESTS FROM FLOWS

Our testing tools are already changing. Software testing tools have started incorporating AI/ML. For example, **Testim** allows test cases to execute successfully even when there have been minor changes in the application UI. Testim uses reinforced machine learning techniques to locate and identify elements in the UI of the application under test, even when certain attributevalues of the elements or properties of the web page have changed. Tests will no longer have to be deterministic. AI will be able to help test non deterministic scenarios. Testim is also spearheading the effort to create automated tests by observing actual usage of applications by real users. This will help increase coverage and focus on areas most prone to defects.

Automated tests know how to interact with the system, but they can't distinguish between correct and incorrect behaviors of the application under test. In AI/ML based testing there will be a range of possible outcomes. A test engineer would need to run a test many times and make sure that statistically the conclusion based on test results is correct. AI based learning from failures will help make decisions on how new tests will be created and executed even under slightly changed conditions.

Looking to the Future

Currently, autonomous solutions in software testing are still in their infancy. Functional testing tools have adopted various forms of autonomous capabilities from discovering an application structure to predictive self-healing to intelligent bug hunting. End-to-end autonomous testing solutions have yet to be widely adopted by large enterprises. However, this will change as people become more comfortable with test automation.

About the Author

Sudhrity Mondal

Sudhrity is a technology advisor and technical sales leader at Testim.io.

He specializes in Automated functional/performance Testing, Test Modeling, Test Data Management, Service Virtualization, Continuous Delivery/Testing and DevOps.

With over 28 years of software development, architecture and consulting experience, he is passionate about making sure that his customers see and derive value from successful adoption of technology.

LinkedIn
linkedin.com/in/sudhrity

Twitter
@sudhrity

Email
sudhrity@testim.io

testim

Thank You!

For more information
contact us at info@testim.io