Test Engineering Framework

Test Engineering is one of the critical phases of product engineering and delivery life cycle, which play a critical role to accelerate the pace in delivering to market with ultimate quality standards in meeting customer experiences through the consistent and continuous feedback loop.
A typical continuous delivery pipeline requires certain foundational practices from the test engineering team or roles within a scrum or product engineering team.

1. Test As Isolated as possible
2. Automate as much as possible
3. Zero tolerance on flakiness & NFRs are just as significant as functional
4. Fast Feedback (test as early as possible) - Shift Left Approach
5. Regression failure = broken build
6. Measurement and Metrics are important too

Test As Isolated as possible -

The Lower-level tests allow you to better narrow down errors and replicate them in an isolated way. Lower-level Test like Unit Test and Component Test provides an opportunity to execute your test as isolated as possible. They'll run faster and less bloated when you're debugging the issue at hand. Also, they serve as a good regression test for the future. If you have tested all conditions confidently on a lower-level test, there's no need to invest heavily on higher-level test in your test suite. Generally higher level test like end to end UI test are costlier to build, manage, and execute for all combinations of the test. It just doesn't add more confidence that everything's working. Having redundant tests becomes annoying in your daily work. Your test suite gets slower, and you need to change more tests when the behavior of the code changes. Another critical aspect of achieving the effectiveness of testing in isolation is the test environments. How often you can create or provisioning an environment to run your lower-level test, which can be possible through infrastructure as code with templates to create and provision small environments on-demand basis.

Automate as much as possible -

Successful Continuous Testing is a competitive advantage, enabling businesses to deliver value to their customers with higher quality faster. Test automation provides quick and consistent feedback to the development team about the health of the components, system and applications. This is an important capability that is necessary to improve on quality and acceleration of the delivery of the product.

The number of unit tests, component tests, integration test and acceptance tests should increase automation test coverage. The low-level tests will provide a safety net to ensure the code is working as intended and helps prevent defects escaping to later stage of the delivery cycle. Depending on product and team maturity, its' recommended to maintain a 80% to 90% Unit Test Coverage and 70% to 85% of functional coverage for a given application component. This can be achieved with adaptive implementation of automation for functional validation of sprint and regression coverage.

With automation in place, test activities can be seamless and continuous testing goals can be achieved. Automation test scripts may be extended for both functional and non functional regression suites provided they collaborate and plan the strategy of automation effort.

Testers should work closely with the developers to ensure that quality index is maintained and that defects are addressed immediately. To achieve a better quality index, it is necessary to automate your test that are repeatable and critical to get iterative feedback whenever there is a change in your code. Testers can work closely with DevOps team as part of the build & deploy pipeline setup/configuration and support for continuous improvement of the pipeline. The automated regression test suites will be added to the pipeline to get incremental feedback as when the code commit is made.

Automation Approach

The success of test automation requires careful planning and design work. An interim approach to adopt automation in brown field projects is to first automated identified sanity suite for various levels of test which would determine the quality with baseline as current in production. Incremental automation to address coverage of P1, P2 and P3 test cases across Unit, Component, Integration, System and End to End UI test would be roadmap till 85%+ code coverage is achieved. This activity involves automation of test across multiple levels -

Unit Test – The smallest piece of code that is logically isolated in a system under test can be automated using IntelliTest or NUnit or JUnit or JSUnit framework. Recommendation is to use IntelliTest to accelerate the automation in Microsoft based Backend services, whereas JUnit for Java and JSUnit for Javascript based solutions. As there are tools providing the capability to generate test scripts dynamically while analyzing the code stream; we can use them smartly to address the backlogs in brown field projects. The goal is to address all critical and foundational modules to be covered as part of addressing the automation backlog while enforcing the practice of automating the unit test for new feature development activity which is governed by a checklist in user story acceptance criteria.

Component/API Test – The test relevant to the individual components or interfaces should be automated using data driven approach to validate the different permutation and combinations of business data that are processed by these components. It is trivial to automate, generally the execution of these test cases are rapid. The component changes relevant the user story implementation should be tested and automated as part of the sprint activities. TDD approach is the best one to be adopted, which require developers and testers to collobratively build these test scripts.

Functional Integration Test – These tests can be Webservice or GUI based that verify the functionality and integration aspect of the application. Test cases are automated using BDD approach to ensure the GWT principles are validated. Most of the positive user scenarios are tested in sprint using exploratory testing and are fully automated as backlog activities by lean automation certification team.

End to End Test – GUI based Tests that are mostly part of the user acceptance tests, to validate the end-to-end product workflow. These tests are "light-weight"; verifying state transitions and a selected set of scenarios or user journeys of a business feature in production. Minimal set of test are automated and maintained by the dedicated lean automation certification team. Test should be build and maintain in collaboration with business or product owners and cross functional team members.

Non Functional Test – These type of tests are specifically designed to validate non functional aspects like Security, Performance, Resiliency, Accessibility features of the application. The requirements for these factors are defined and driven by the solution architects and product owners. A specialized team or engineers as part of the scrum team will design, automate, execute and maintain the test as needed.

Regression Test – Unit, Component, Functional Integration, E2E and Non-Functional Test are identified to be regressed for multiple iteration or releases. These test constitutes the regression test pack and will be executed during certification phases of a release and regression testing for user stories functional sign off.

Ownership with	Test Types	Execution Phase	Automation Coverage	Overall Test Coverage
Scrum Team	Unit Test	Sprint	99%	60%
Scrum Team	Component Tests	Sprint	95%	25%
Scrum Team/Automation Team	Integration	Continuous (S + 1)	75%	10%
Scrum Team/Automation Team	End -To- End	Continuous (S + 1)	50%	5%

In addition to the automation elements defined above – it's recommended to incorporate the following suites to provide comprehensive testing capabilities throughout the delivery cycles.

Test Suite	Description
Sanity Suite	Basic acceptance checks that the application core functional flows are intact. Smoke Test pack should catch the most obvious issues and test duration should be less than 5 minutes to provide early feedback.
Environment and Infrastructure Verification Suite	A limited amount of environment and infrastructure checks (10 to 15) to validate environment availability related to 3rd party services, disk space, memory, processor, network bandwidth, and other middleware components health checks
Heartbeat Suite	Tests which can be run on Production and Non Production environment to validate the business critical flows exercised by the customers or product owners.
Featured Regression suite	This pack contains the relevant regression suite of the component or functional integration or non-functional tests for the feature or bugfix to be tested. The goal is to get quick feedback with a larger set of tests. If the feedback takes more than 1 hour, it is considered slow. Either the number of tests has to be reduced by using pairwise test technique.
Full Regression suite	This pack contains tests to cover the comprehensive coverage of the component or functional integration or End to End or Non-Functional tests for the certification activity of a major or bug fix release.

The automation testing approach should address two key aspects, the product platform/components and business validations. The target approach can incorporate one or both **TDD (Test Driven Development) and BDD (Behavioral Driven Development)** models to drive their automation across engineering driven unit, component, integration test and business driven User Acceptance Test and End to End business Flow Test.

To adopt a customer and engineer driven approach, **ATDD (Acceptance Test Driven Development)** should be utilized, capturing requirements in acceptance tests and utilizing them to drive the development. The approach will be similar to BDD (GWT Approach - Given When Then) and closer to TDD model. Acceptance tests are the customer tests that demonstrate the business intent of a system which can be reused for UAT testing. The Acceptance test should be built in collaboration with Business Team (Product Owner and Integration Engineer when it comes with multiple stakeholders involved across organization) for the business flows.

Component tests are technical acceptance tests developed by an architect that specifies the behaviour of modules or components pertaining to an engineering platform/application solution. The developer is responsible for creating unit tests to drive easy-to-maintain code.

One of the main issues with our current state is frequent change in requirements and scope creep. With the ATDD approach, it will be imperative that once the developer starts working on a story, the requirements become locked. Any changes must be prioritized alongside other upcoming stories and added to the upcoming sprints. This would reduce the workload of both developers and testers to improve the productivity and prevent stakeholders from having unrealistic expectations of completed features.

Zero tolerance on flakiness

The success of the test automation lies on stability/reliability and speed of the test execution. With flakiness in your automated test execution, the false failures breaks the build process in the continuous delivery pipeline and involves huge manual effort spent on the failure analysis with manual test execution to conform the false failures. These glitches will derail the automation effort if they are not addressed as part of our

automation framework selection/design and standards setup.

As part of the framework design and automation coding standards, few critical factors to be addressed -

- The value of writing proper test code to be established and standards documented for reference
- Setting up and customizing a test factory to help on modular approach
- Using the test factory to create test code is emphasized
- Best practices for writing test code is published
- Adding system asserts within framework as validate internals and externals kit

Here are the attributes associated in defining or measuring an automation framework. They are:

- Core Functionalities
- Adoptability
- Tools
- Value adds
- Execution
- Metrics
- Community
- Maintainability
- Non-functional

Core Functionalities

These are core features to be present in any automation framework apart from the functionality. Automation needs by most organizations are around `Web automation` or `Mobile automation` or `RESTful APIs` support for all of them is an absolute need.

- For `UI - Web Automation` the framework is expected to support multi-browser toggling via configurations (browser flavors to be supported can be determined by the user base of an organization's product).
- For `UI - Mobile Automation` the framework is expected to support multiple platforms (`ANDROID/IOS/WINDOWS`) and also various forms of the application (For e.g., `Native/Mobile Web/Hybrid`) with the least amount of effort by a test case automation engineer.

Most of them all, the framework should have a regular release cycle. The frequency and the mode (Need basis release or a well defined release cycle) can vary across different maturity levels.

Adoptability

Various different things such as `Get started guide` or having a proper support model all determines how easy is it to get started with an automation framework. Of-course it goes without saying that the Automation framework APIs or guidelines should be kept as simple as possible. But depending upon the choice of language, what is deemed as simple would change and is usually a subjective term.

- `On-boarding`: This usually represents the ask *How easy is it for an automation engineer to get started with using the automation framework*. It could be via :
  - Well defined documentation
  - Templated projects (Either a sample project in a version control system or as a bundled project)
- `Documentation`: From technical documentation (for e.g., `javadocs` in the Java world or `godocs` in the GoLang world) to user facing documentation that explains how to achieve a functionality using a framework, documentation plays a very huge role. Most of the times even the most sophisticated frameworks lack in this area. They either have poor documentation or have outdated/defunct documentation which is as good as not having documentation. Simplicity is the theme when it comes to documentation and has to be kept in mind. The other aspect of documentation is also an FAQ (Frequently Asked Questions) section that explains the most commonly asked aspects about an automation framework. The more easier it is to find information, the more successful a framework becomes.
- `User-support`: We all can agree that times there's always a situation wherein a user of an automation framework is stuck with a problem for which they are looking for help. Having a well defined user support channel is always a must have. It could be via emails (or) working hours sessions (or) via chat mechanisms such as a Slack channel or a Teams channel.

Tools

Most often beyond a point an automation framework would need to span into peripheral avenues wherein an automation engineer needs help. For e.g., it could be on a trivial basis, tools to augment an automation effort and at other times it could even span to the level of building sophisticated code generators.

Some examples for support tools could be:

- Record and play back tools which generate test code adhering to the automation framework's syntax and semantics.
- DSL plugins for IDEs
- Passwordless setup for UNIX machines.
- Browser based tools that make it easy to capture HTML element attributes of a web page or something similar for a mobile app.

Value adds

The amount of value adds that an automation framework provides speaks in leaps and bounds in terms of how much a framework has

evolved/matured.

Some of the value adds could be around the following:

- `Fosters customization` - An automation framework should follow the same design principle that is usually emphasised when building code viz., the automation framework should be closed for modification but at the same time, it should be flexible enough for a user to plug-in customizations that are needed for specific usecases.
- `Traceability friendly` - One of the asks from an automation engineer is to be able to figure out/map the list of automation tests back to a logical functionality within an application. The automation framework in this aspect should be able to support/provide this meta data information and also a proper reporting mechanism which can basically help answer the question *Which are the set of automation tests that are aligned to a particular functionality*.
- `Test Management tool integration` - Most often an automation engineer would want to post back test results into a Test case management system so that it gives them a holistic view of what tests were selected for execution, how many were executed and what are the results for the same. Some other times there may be a need to automatically file bugs when an automation test fails. A framework should provide mechanisms for these integrations depending upon the relevant TCMS tool being used and depending upon feasibility.
- `Environment readiness assertion` - The most commonly heard failures in automation tests is when a test runs against a partially setup application in a test environment. The automation framework if provides a simplistic way of determining environment readiness, it goes a long way into avoiding `fake failures` of automation tests. Some of the aspects that this readiness assertion could do can be :
    - Is the test environment box(es) up and running.
    - Are the required components for a particular application deployed.
    - Are the required components for a particular application up and running.
- `BDD support` - Most often teams realize that an alternative approach for automation is probably the need of the hour (There can be various reasons behind this realization and is beyond the scope of this post). In those cases, it would be good if an automation framework provided support for this migration on a very easy basis with as little change as needed.
- `Contextual reporting` - An automation framework should keep in mind the various reporting asks from its users and provide reporting mechanisms for the respective asks. For e.g., an automation engineer might need to share test results with management as a pdf or as a one pager email with embedded contents/screenshots etc., At other times an automation engineer might need to generate reports that add more debugging information around failed tests which he/she can circulate amongst the automation engineers team to quickly fix what failed.
- `Failure analysis and Root cause analysis behind failures` - Automation tests will fail all the time. But to fix the failure an engineer must not have to spend hours together trying to figure out what caused the failure and where did the failure occur. A matured automation framework would always provide means to help find the root cause via different mechanisms such as per test execution logs, payloads sent as requests and responses etc.,

Execution

An automation framework must always be geared towards supporting faster test execution. To that aspect, its a mandatory ask that an automation framework is **concurrency friendly**. It must support parallel test execution in a thread safe manner. An automation framework if is aimed at UI and Mobile platforms, must also provide for ways such that test execution can be easily routed to third party execution environments such as BrowserStack or SauceLabs for e.g.,

Metrics

As a framework scales and grows, it needs to start providing ways wherein an automation engineer can get some insights around trends.

- `Execution data gatherer` - An automation framework can for starters try capturing essential execution data for every test, details about the environment it is running against, the infrastructure details, the browser details etc., as part of gathering a holistic view of a given run. This information would become very useful for an automation engineer to start looking for patterns, identify bottlenecks, look for tests that fail consistently etc.,.
- `Execution trend analyzer` - Its not sufficient that an automation framework gathers data, it also needs to provide ways of consuming the data as information and help users get insights into this data to help them make deicisions. Either the framework could build the entire eco system of store and display data for users and even provide analytical capabilities or use out of the box solutions which are either paid or open source.

Community

An automation framework should never confine itself to just catering to some functional asks that it satisfies from an automation standpoint. An automation framework matures and grows only as good as its users grow in these aspects. To that affect an automation framework should always find ways to encourage the following amongst its users.

- `Foster giving back attitude` - The automation framework should encourage its users to come forward to contribute in terms of bug fixes or new features. Only by getting the community involved is when an automation framework gains support and love from its users. This also encourages its users to start gaining confidence into giving back to the respective open source libraries that a framework uses and thus help the entire community thrive.
- `Fosters open source contribution` - The ultimate goal of an automation framework should be to inculcate the culture of grooming open source contributors. This fosters the culture of expanding the network of connected engineers and can also start becoming an avenue of attracting talent to the organization.
- `Fosters enrich over silo mode` - The automation framework should basically strive against its users having to build tools/side frameworks etc., in silo modes but instead provide a platform for its users to be able to contribute to the framework and grow it holistically.

Intelligence

An automation framework as part of becoming advanced and sophisticated can consider providing some AI powered capabilities to its engineers.

Some of them could include:

- Ability to parse change-sets and decide on the set of tests to be executed as regression.
- Given the set of changes going in as functionality, execute a subset of tests that are aligned against those functionalities.
- Based on past history of areas wherein bugs tend to get introduced a lot, decide on tests that mandatory need to be run all the time.

With AI the possibilities are endless.

Maintainability

An automation framework should always keep in mind the maintainability aspects associated with itself, because at the end of the day code is code and needs a lot of love and care from its engineers so that it lives long enough and is still relevant.

- `Accountability for correctness` - No one likes to use an automation framework that has zero accountability in terms of its correctness. To that aspect an automation framework should be robustly guarded by in-depth unit and also functional tests of its own that always ascertain its correctness for every change made.
- `Well defined SLAs for new feature/bug fix requests` - It also does not make any sense for an automation framework to just sit on a pile of asks (be it new features or be it a bunch of bugs that need to be fixed) without any finite timelines as to when they would get addressed. Any staleness detected in this is a warning sign of a deteriorating framework.
- `Design driven` - One of the most important attributes of a framework that is change friendly is when its an easy answer to the question *How soon can I impregnate a change into the framework*. This is usually ONLY possible when an automation framework is heavily guided by good design principles and follows a proper design.
- `Adherence to best practices` - The industry prescribes a lot of best practices for good code management. Some of them could include the following:
    - `Code reviews` - All change-sets being delivered into an automation framework should go through a rigourous cycle of code reviews.
    - `Unit tests` - An automation framework should have a de-facto standard that all changes need exhaustive unit tests which ascertain the correctness of features/functionalities being delivered.
    - `Code coverage` - Although a 100% code coverage is both a myth and still does not guarantee correctiveness. But that should not be used as an excuse for not striving to have a good amount of code coverage mandated for an automation framework and efforts made to improvize upon previously established numbers.
    - `Static code analysis` - Many of the bugs surface due to the inability of their effects being ascertained either at coding phase or at review phase or due to the inability to build tests that test against them. These bugs could sometimes easily be caught if one were to use static code analysis tools which skim through the codebase for badly written code (which violate good practices or use error prone techniques) and call them out under various different categories. Again the focus should be to strike a balance in terms of how many of the warnings do we heed to, instead of blindly trying to keep the tool happy.
- `Modular capabilities` - The capabilities of a framework should be structured such that its users have the luxury of picking and choosing only those things that they need. To that matter a framework should be modularized as much as possible. For e.g., a typical framework could provide modules that specifically support only web, only mobile and only api testing so that depending upon a user's ask, one can pick and choose only those specific modules.

Non-functional

An automation framework should never confine itself to addressing only the functional needs from an automation engineer. It should also facilitate catering to the non-functional automation needs to be driven with ease by fosting reusing of the same functional tests for non-functional test needs.

To that aspect an automation framework could provide ways in which it can support the following non-functional asks:

- Performance testing integration
- Security testing
- Accessibility testing

Fast Feedback (test as early as possible) - Shift Left Approach

Shift Left testing provides effective means to perform testing in parallel to development processes, enabling faster, better and quality applications and enhancing collaboration between development, operation, and testing team. TDD provides the best opportunity to implement early testing within the engineering cycle. With effective Continuous Integration and Delivery pipeline, early and continuous testing gets inevitable.

Regression failure = broken build

Integration the regression automation suites to build pipeline jobs to embed quality checks early in the cycle to get early feedback. If you have implemented right automation strategy and adaptive framework, integration of your regression suites is seamless with build jobs to enable continuous testing. Any issues created as part of code changes or test scripts not updated for the new enhancements would break the build which forces the developer or testers to attend the changes to deliver and overcome the build failures.

Measurement and Metrics are Important too

The measurement through quality and automation KPI are important to validate the progress over implementing continuous testing within our product engineering life cycle. Monitoring the testing engineering metrics define the effectiveness of your development and testing efforts. With automation testing becomes significant to business productivity, testing metrics provide insight into the status of automated testing and making further improvements.

The typical automated testing metrics quantify the performance of the implemented automated test process which includes past, present and future efforts.

The KPI and Metrics are defined with clear goals pertaining to the success rate of the deployment, pre-integration vs post integration bugs found ratio, automation improvement criteria (Pass rate, Execute rate, Invalid failure rate, code coverage, MTTR, etc) and technical debt.

Automation Code Coverage %:

The objective of this metric is to report % of line of Development code covered by the automation script or code. This helps the management to assess the ops cycle time and testing velocity of the team. Automation Code coverage % is inversely proportionate to ops cycle time.

Example:

In the above stated example, Test/Ops Cycle time is higher for sprint 3 than other 2 sprints, as the automation code coverage is less in sprint 3 (ie 60% of the development code is tested through functional manual testing).

Lead indicator & outcomes:

- Reduced Ops cycle time
- High velocity
- More defects, as the team can perform exploratory testing

Automation coverage %:

The objective of this metric is to report total number of tests automated from the overall test cases. This helps the management to assess the ops cycle time, testing velocity of the team and automation capability. Automation coverage % is inversely proportionate to ops cycle time.

Example:

In the above stated example, Test/Ops Cycle time is higher for sprint 2 than other 2 sprints, as the automation coverage is less in sprint 2 (ie 70% of the functional testcases are tested through manual testing).

Lead indicator & outcomes:

- Reduced Ops cycle time
- High velocity
- More defects, as the team can perform exploratory testing.

Automation Effectiveness %:

The objective of this metric is to report the effectiveness of Automation scripts. This helps the management to assess the effectiveness of scripts developed. Automation effectiveness % is directly proportionate to defects detected by the script.

Example:

In the above stated example, the effectiveness is high is sprint 1 where 60 defects detected from 100 automation scripts.

Lead indicator & outcomes:

- Less defect leakage
- High quality product

Average Automation execution rate:

The objective of this metric is to report the average time taken to execute one test script. This helps the management to assess the time taken to run the test suits. Average Automation execution rate is inversely proportionate to certification cycle time.

Example:

In the above stated example, the average execution rate is high in Sprint 1 which reduce the certification time and ensure quick time to market.

Lead indicator & outcomes:

- Less time to market and certification time

Pass rate %:

The objective of this metric is to report the pass rate % of the Automation Test execution. This helps the management to assess the product readiness to the productions.

Example:

In the above stated example, sprint 2 has a very low pass rate which may stop the production/ live push.

Lead indicator & outcomes:

- High pass rate means high chances of pushing the code to production.

The Assessment Questionnaire is provided below to understand the foundational and best practice from Test Engineering team as part of devops assessment to discover and evaluate the maturity and capability of modern devops pipeline and cloud infrastructure implemented by the customer delivery organization.

Test Engineering - Assessment Questions