



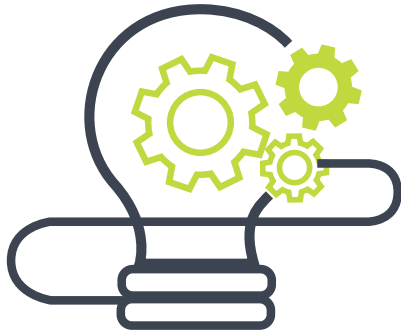
THE ROAD TO HYPERAUTOMATION:

Modern Quality Engineering Part 2 -
Automating Testing Dependencies

In our last eBook (Modern Quality Engineering for Agile and DevOps Delivery), we discussed how expectations of Software Quality and Testing have changed across the industry to reflect the requirements of Rapid Application Delivery in line with Agile and DevOps approaches. To recap some of the points made briefly:

- Testing needs to support increasingly faster application releases
- Testing needs to happen within smaller and smaller timeframes
- Testing needs to be more product-focused and collaborative, rather than siloed by function

So, we broke down the main principles that should be considered in building a quality organization that provides both the strategic and tactical functions.



From a strategy standpoint, Quality goals and the very definition of Quality in the software development lifecycle have become more malleable and adaptive, and at the very least, subjective. Instead of having absolute goals such as the number of defects in production, % testing coverage, or % defect leakage, the standards of quality are different for every organization and often times are different across initiatives. So, the question becomes - if the definition of Quality is flexible, what do we base it on? This was answered as well in our previous eBook through the philosophy of Just Enough Quality as well as risk thresholds and how they relate to the Total Cost of Quality.

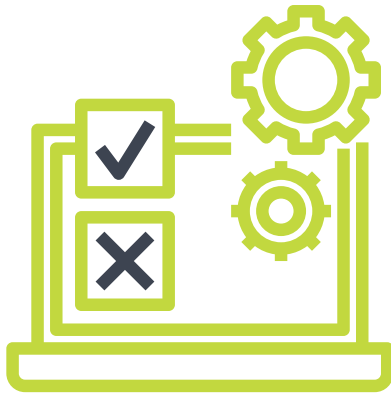
Finally, we pointed out that automation plays a focal role in enabling the testing needs of today and tomorrow and that next-gen automation expands on the opportunities with the use of robotic process automation (RPA) to automate dependencies for testing, and artificial intelligence and machine learning (AI/ML) to automate decision-making as well as creating and troubleshooting activities related to testing. Simply put, to realize the dream of in-sprint testing - or testing fast enough to truly catch up to the latest release - there is no other way but to automate.

In this sequel to the Modern Quality Engineering eBook, we will explore general testing dependencies as well as if and how to use automation to address them.

The Testing Maturity Journey

Before we dive into specifics of dependencies to testing, let us baseline our discussion with some fundamental commonalities of how organizations evolve and mature their quality and testing function. This is undoubtedly an oversimplification, and there are many nuances that one will find across most companies, but for the sake of discussion, we will start with some **key milestones that represent a basic Quality Maturity Journey** –





1. No testing

This is straightforward –an organization does not apply any testing to their software development and maintenance initiatives. I've seen many startups, skunkworks, and non-IT led initiatives operate this way – and of course, when your team is just enough to build your product, you often make sacrifices to the quality. Your cost and time to deliver are most likely fixed, and scope at this point is as much as humanly possible. Also, the entrepreneurial, even adventurous, spirit of an initiative with a well-defined vision also comes into play – the focus is to push what can be done, rather than how well it can be done.

As an aside, when I do organization maturity appraisals, the concept of nothing is often the starting point and corresponds to a score of 1 or 0. Sometimes, I even include the concept of a negative 1 to represent an activity that is not only unbeneficial but largely harmful to the organization. An example of this could be the unrestricted use of feature flags to justify not needing to test. While this does speed up work and reduce effort, this is a sure-fire way to open up the software to the immense risk and unrecoverable damage.



2. Basic Unit Testing and / or Acceptance Testing

A marked improvement over having no testing at all is **unit testing** where most bugs are caught, and it's a good place to start testing. Especially for developers applying a TDD (Test-Driven Development) practice, unit testing allows the most basic and fundamental issues to be found. Also known as **component testing**, unit testing focuses on the smallest unit of testable code – a button, a field, a condition, etc. – and makes sure that it executes correctly.

On the other hand, **acceptance testing** (also known as UAT or BAT for user or business acceptance testing) validates the fitness for purpose. Simply put, it addresses the question, "did the end result meet the original objective?" That is entirely different from QA functional testing that looks more at whether the application works (fitness for use). Some teams also use acceptance testing in their definition of done to signify that a user story has been completed. For more on the difference, do a Google search on 'Verification vs. Validation'. These two are often at the very beginning and the very end of testing cycles and represent the very basic requirements of quality – that every piece works, and it meets the end-users need.



3. Basic QA / SIT Testing

This is where it starts getting interesting. Having a dedicated team specifically for testing purposes is what separates those trying to deliver with the minimum amount of quality from those who are serious about bringing a quality experience to their end-users. The importance of QA SIT (Software Integration Testing) is also touched on in the last eBook and will not be elaborated here, but suffice it to say that until you have dedicated testers, you are not in control of your product's quality. Essentially, you are just rolling the dice by assuming that developers could be completely responsible for not making mistakes or to check their own work. This is not a ding on developers, but more a spotlight on the complexity of development - especially in an increasingly software-driven world. The reality is that checks and balances are needed because failures do happen and we often are most oblivious to our own mistakes, regardless of the role.

What also differentiates QA SIT from unit testing is the scope – as software becomes more and more complex, the need to test multiple paths, combinations of steps, different inputs for transactions, and the full interoperability across interfaces of different systems, different companies, and even different technologies become increasingly crucial. A unit test or UAT test alone cannot address the complexity of today's software.



4. Formal Testing Process & Metrics, Basic Automation

The key difference between doing an activity compared to executing a process mainly lies in the ability of a process to scale and to have some level of reliability. An activity that is not driven by a process is completely dependent on the ability of its doer, which by no means is an indication of failure (or success). However, it will fail to scale and it will not be repeatable and reliable. It also is rarely measurable, so it is hard to calibrate or improve. Formalizing testing processes is the main milestone to make any testing activity fit into an enterprise environment.

The other major milestone at this juncture is this is when organizations typically start their automation activities. There is good sense in defining a standard process before automation is started. Even before Agile became mainstream, I've seen many examples of teams and organizations trying to figure out automation on their own and jumping ahead when a few key people have learned how to do it. It is inevitable that they will run into a brick wall without a strategy or a process – whether it's consuming all of their capacity for test automation maintenance instead of creating new tests or not having standards, means no one else can troubleshoot another's script. Possibly the worst of it (and this happens in real life) is investing substantial effort in record-and-play scripting (not to be mistaken for Scriptless automation) and realizing that this is mostly a novelty and not real test automation. All in all, this milestone represents bringing in some level of formality and initiating automation in the organization.



5. Process Maturity and Automation Program

This is another similar-sounding milestone, but significantly different. The main differentiator between this and the previous milestone is that the process is mature, which in a very broad sense means it is both quantitatively managed and it is widely standardized. Wide standardization means, with few known exceptions, the process is applied across the organization. The exceptions must be few and known because there should be a reason acceptable to leadership as to why they are not applying the standard consistently. Quantitatively managed means that metrics govern and act as a basis for improvement – improvement cannot be based on subjective information or assumptions.

The automation program needs to have evolved as well towards these directions – from the basic functional automation,

- a. it needs to service more platforms (i.e. Web, Mobile, Cloud, Mainframe, etc.), and/or
- b. it needs to provide more testing types (Smoke, Regression, Performance, Monitoring, etc.), and/or
- c. it needs to test more components of the tech stack (Database, APIs, Integrations, etc.).

Usually this results in the use of multiple automation tools to service those needs.



6. Continuous Testing

This milestone is what separates standard automation and HyperAutomation. We start leveraging advanced technologies (some process automation – business or robotic) in automating dependencies and requirements to testing. There are six basic dependencies to testing, and this milestone is interested in automating the fulfillment of some or all of them.

- a. the code or application to test,
- b. test data,
- c. test environments,
- d. integrations,
- e. the test window, and
- f. the execution agent.

As the six dependencies to testing are automated, the process becomes more and more autonomous from the moment the tests to be run are ready. This is a form of TestOps although TestOps can also be defined as a combination of Continuous Testing and Continuous Validation in production – more on that later.



7. AI and ML-Based Automation

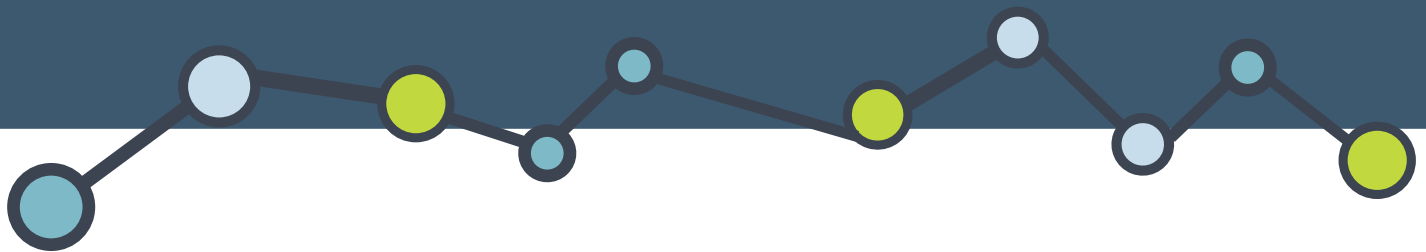
One can surmise from the previous milestone that the main area left to automate is about test creation and selection, and that would be accurate. This stage is where AI and Machine Learning concepts are applied – in design and update of automated test scripts and in selecting and prioritizing them. These are pretty much all the decision-making and cognition needed for testing. Since testing windows are always limited (and typically less than planned!), the challenge then becomes what are the most important tests to run given the available time. At the same time, creativity, knowledge, and skill come into play for designing tests and keeping them up to date. By leveraging AI and ML technologies to automate these activities, we have now fully automated all activities related to testing.

What then is HyperAutomation and how does it affect Quality Engineering?

Gartner provides a good definition of HyperAutomation in that it is not too rigidly defined, and it also offers some basic anchor points. According to Gartner in its [article about the top strategic technology trends](#):

"Hyperautomation deals with the application of advanced technologies, including artificial intelligence (AI) and machine learning (ML), to increasingly automate processes and augment humans. Hyperautomation extends across a range of tools that can be automated, but also refers to the sophistication of the automation (i.e., discover, analyze, design, automate, measure, monitor, reassess.) As no single tool can replace humans, hyperautomation today involves a combination of tools, including robotic process automation (RPA), intelligent business management software (iBPMS) and AI, with a goal of increasingly AI-driven decision making."

By this definition, Milestones 6 and 7 are HyperAutomation. By defining the components and discrete pieces of capability in those milestones, we are able to understand which tools provide which capabilities. The reality is, each of the pieces that we are referring to is in itself, a major undertaking. To date, no single tool has even come remotely close to addressing them all; thus, we need to combine them in order to create fully autonomous automation. For the rest of this eBook we will consider only Milestone 6 to define the discrete pieces we will need. Milestone 7 will be discussed in a separate eBook.

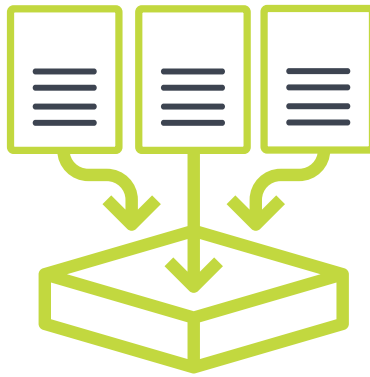


Automating the 6 Dependencies for Continuous Testing: The Foundation for HyperAutomation

Each of the six dependencies we mentioned in Milestone 6 are historical components of the quality assurance process. Our goal is to become increasingly more autonomous and achieve HyperAutomation.

Code or Application to Test

The first one is straightforward; it is what you test. Outside of experimental technology, there is not yet anything out there that can automatically create code or an application to test. There are certainly advances in tools that help in this regard, but for the most part the only automation we can reliably expect for this part is automating the availability of the code to test. That is where the Continuous Integration / Continuous Deployment (CI/CD) Pipeline comes in – it automates the packaging and deployment of a build to the right environment for testing.



Test Data

Test Data is of course about the different parameters and input combinations that you need to test. Test data automation will help provide the data needed for testing either by looking for the said data or by generating it synthetically, on-demand, as tests are run. All organizations with testing activities need test data, but not all organizations require the dedicated automation of this process since this requires a lot of cost, not to mention specific expertise. The most common need comes from Financial Services, as these organizations deal with a large number of data combinations (just think for each transaction type, multiple layers of payment types, transaction sizes, customer info types, geographies, settlement types, etc. - you get the picture) but anywhere where testing deals with similar scenarios or transactions and many different combinations of inputs is a viable candidate for automation. If you are not sure, a general way to find out is just ask testers – how much of the testing are you not able to do because you do not have the right data? And if the gap is significant, then it may require dedicated test data automation. That said, there are tools that are not dedicated to test data automation but can be used towards this purpose – a good example is leveraging existing APIs to create the data by creating scripts that trigger before a test is run. This counts as test data automation but does not require a separate tool to accomplish, thus not requiring investment in additional tools.



The Test Environment

The test environment is the configuration (device, browser, OS, SDE) of the right technical ecosystem to test in, including its integrations to other systems. Test environment provisioning and external environment dependencies can also be addressed by automation using containerization and provisioning software. Containers essentially pre-configure the environment needed and allow for its automated provisioning and decommissioning, as well as the ability to scale it as needed. This is important for organizations heavily leveraging cloud infrastructure as these resources are easy to spin up whenever they are needed. However, it's important to note that most organizations do a poor job of taking down temporary use resources, especially environments for testing. If you don't keep track of them it is easy to get blindsided by a much bigger subscription bill because of all the environments that are still active but no longer needed. This is why automated provisioning, including taking down environments after use in testing, is crucial.



Integrations

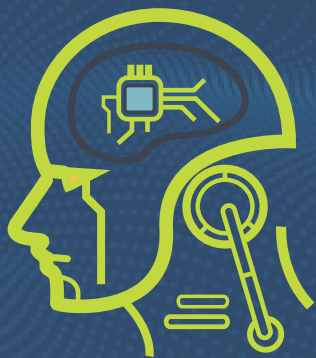
The test environment may or may not have integrations to third-party dependencies or may have a different version of the services in other environments, or in production. Since non-production environments can have, at any time, any version of an external API, there is a risk that the new release breaks because it is expecting to consume a different version of the service. Service virtualization uses synthetic services to act as a dynamic test harness to catch any outgoing requests and incoming responses to external systems from the system being tested, and will enable the testing of the newly added feature against a previous version of an API, or even virtualize it if not available in the non-production environment.



The Test Window

The Test Window is the allocated sequence and duration defined by the software development/engineering lifecycle and the process in play. Most companies already have a defined process that tells you when to do what type of test, and for how long. The reality is that this predefined test window is actually one of the first policies to get compromised, and this is true of DevOps, Agile and even Waterfall (when it was more common), across all organizations. QA and QE Teams have since worked around these challenges with the use of modular testing, or risk-based testing to know when to push back, negotiate, or adapt to moving and decreasing test windows. As it is defined, there is no way that a

test window can be automated per se, but automation can be used to facilitate the process expectations and entry criteria to testing. Again, the CI/CD pipeline can be used to enforce this and is typically done through a conditional smoke test that automatically executes when the build moves from the Dev to the Test environment. If it passes a certain threshold (tip: do not require 100% passed as the threshold, start on the lower end at first), then the build is allowed into the test environment. If not, Dev gets a notification and is able to act on it. This is certainly preferable to waiting for a manual smoke test and having a back and forth discussion that may take up to a few days.



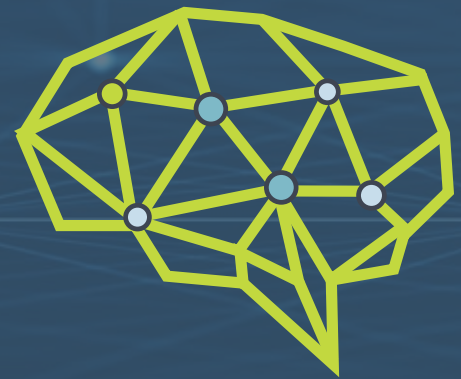
The Execution Agent

The Execution Agent can be a machine or a person that triggers the running of the test. As described in the previous point, triggering tests, specifically smoke tests, as part of the build and release process is standard feature in a CI/CD pipeline, and is typically automated. Although it's not clear-cut: a major consideration is the fact that only a small handful of organizations have 100% automated tests and those are mostly unit tests (see Milestone 2) that are part of the development process. So in reality, while the execution of automated tests can be triggered through the pipeline, it either rarely covers all of the testing needed, or the existing regression suite is too large and time-consuming to execute all at

once and should be scaled down to just the right amount that fits the testing window. One of the ways that we have addressed this in the past is by building a test automation execution portal or a simplified UI layer that allows any user to select and run tests manually across different test automation tools. So, we can remove the requirement of knowing how to use the automation tool itself, which is especially useful in a large team with many different automation tools in use. We have also expanded its use to developers and product owners who want to run tests at their leisure. While this is not true automation but more a pseudo-automated process for on-demand testing, it does help bridge the gap of the limited test window and the non-automated tests.

What else is there to automate?

The obvious dependency to testing that we did not cover are the actual tests themselves – designing and writing tests, test script creation, test maintenance as well as selecting which scripts to run (as hinted in our Test Windows discussion). This requires data-driven analysis and skill, and we did not tackle them in this eBook because the existing solutions that address these are AI and ML-based, which is our Milestone 7. The focus here is dependencies to start testing, after the test suite has been set up, which is a combination of selecting existing tests, creation of new tests, and maintenance of outdated tests.





Also, after execution, one of the realities of automation is that there will generally be two types of failed results – failures because of actual application defects which are true defects, and all the other failures (which are false positives). A case can be made for more sub-definitions (such as failed because of environment, because of outdated test script, because of undocumented feature, etc.), but fundamentally, there are always true defects and false positives as a result of running automated test suites. There is effort that goes into figuring out which is which, and reporting and retesting true defects. For this end-to-end process to be truly 'zero touch', this analysis needs to be automated as well. That automation capability is again provided by AI tools, and as such, is going to be covered in our discussion of Milestone 7.

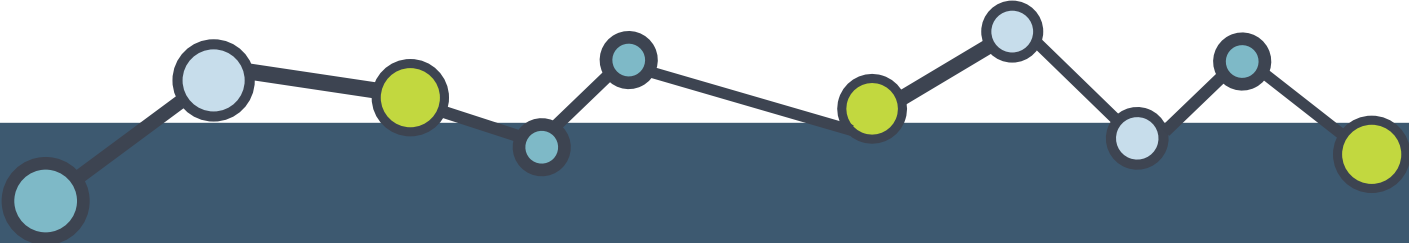
A note about Continuous Validation

Some readers versed in DevOps and TestOps may point out that Continuous Validation is not covered in this discussion. Continuous Validation deals with testing that runs on a regular, scheduled format instead of being triggered by an event, and has the same dependencies as Continuous Testing. This concept applies to testing in non-production environments such as ETL testing and Data Validation testing, but also mostly refers to continuous monitoring and validation in production as well.



The previous eBook discusses the importance of production monitoring in Quality Engineering, but there are still many QA and QE organizations that currently do not own, manage, or are even aware of production validation, including operational monitoring and alerting. As more and more organizations move into the concept of TestOps, testing needs to come full circle and consider the insights from actual production use into the way testing is done. After all, testing is a simulation of what might happen in production, so what better way to refine the testing approach than to learn from production itself?

To be able to automate this process, QA teams need access to production usage data and real user insights, and need to make sense of patterns in the data to bubble up the gaps in test coverage as well as the priority and relevancy of specific test cases. This will inform the decisions that need to be made during the selection and prioritization process and any approach being implemented (see the Test Window discussion, Risk-Based and Modular testing). This requires sophisticated analysis that is mostly done manually today but can be automated using Machine Learning as well.



I certainly hope that this eBook (in conjunction to the one that precedes it – Modern Quality Engineering: Testing for Agile and DevOps) is a helpful reference. My next eBook will expand on the final automation piece and dive into Milestone 7, moving into the territory of AI and ML and how these concepts are being used by testing software companies to assist or even replace activities that require cognitive and creative steps in testing – test design and creation, test script maintenance, and test selection and prioritization.

To apply these Quality Strategy and Test Automation concepts in your organization, reach out to RCG Global Services through our Contact Page. RCG offers a rich portfolio of services and deep expertise built from working with organizations including the biggest brands in the world, regional and mid-size businesses, and even cutting edge and promising startups. For any questions, comments, and feedback about this eBook, I would be happy to hear from you – feel free to reach out to me on LinkedIn (see About the Author Page).

Until then, Happy Testing!

About RCG Global Services

RCG is a global provider of digital solutions across mobile, web, cloud, and legacy platforms, with a focus on actionable data and analytics. We have a rich history of enabling clients in the Global 1000 marketplace to realize their digital ambitions — serving clients across a range of markets, with particular emphasis on financial services, insurance, healthcare, and consumer industries.

As your end-to-end digital innovation partner, we empower you to tackle the challenges you face in customer engagement, workforce enablement, and operations optimization. From customized strategy to implementation and sustainment, our seasoned experts collaborate with your team on solutions that deliver measurable impacts quickly and reliably.

In today's digitally driven world, transformation is essential if your company wants to disrupt the status quo and be respected as a leader in your field. RCG is the partner you can trust to help you realize your objectives and turn ideas into action. RCG is based in Iselin, New Jersey, with offices throughout the United States and offshore delivery centers in the Philippines and India.

About the Author

Niko has over 14 years of Quality and Process Strategy Consulting for Information Systems across virtually every industry. After starting as a developer, he found his true passion in taking on testing management, project management and process appraisal roles. Niko started and led Hewlett-Packard Asia-Pacific's QA solution architecture team, growing the team, revenue, and client accounts more than tenfold through expanded pursuit efforts, impactful capabilities and exciting innovations - touching all aspects of software testing and developed global Software-as-a-Service delivery models. He also started and led the Enterprise Quality program for CareerBuilder, where he continues to be a strategic advisor.



Niko now leads the Quality Engineering Consulting Practice for RCG GlobalServices, where he drives best practices, standards, innovation, and capability building internally and through technology partners. Before that he was Director of QA, building and running TCOEs, roadmaps, architecture and automation programs for enterprise clients. He is passionate about leveraging AI and RPA technologies in Quality Engineering.