testim

abstracta

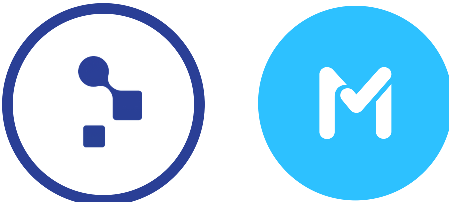
8 Mistakes Companies Make When Transitioning to CI/CD

White paper by: Federico Toledo, Ph.D., Lucia Lavagna, & Oren Rubin

www.testim.io

Introduction

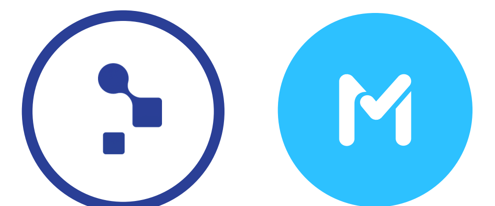
CI/CD, BDD, ATDD, DevOps, Agile, Continuous Testing—however you want to do it, companies today are all striving for faster release cycles and higher quality software. While helping our clients over the years, the teams behind Testim.io and Abstracta have witnessed several successful transitions to Continuous Integration and Continuous Delivery (CI/CD). It's important to remember that change requires not only a technical shift, but a cultural one at the same time in order to create a lasting impact. We've identified eight common mistakes that teams make during the transition that you can avoid while embarking on your own transition related to team culture, processes, expectation setting, and more



Mistake 1: Lacking Proper Management Buy In

A Bottom-Up Endeavor

More often than not, the driver of the shift to CI/CD is a bottom-up effort, meaning, seldom do leaders impress upon their teams the need to adopt these new practices, but rather, testers and developers make the case for it. The problem when implementing it bottom-up is that there comes a point where the team must obtain executive buy-in to move forward. Otherwise, the project may gain some temporary momentum, but it's unlikely to be sustainable over time. While partially making the shift is better than nothing at all, it's ideal to provide visibility into the team's success by management in order to receive its support and to spread it to other teams throughout the organization.



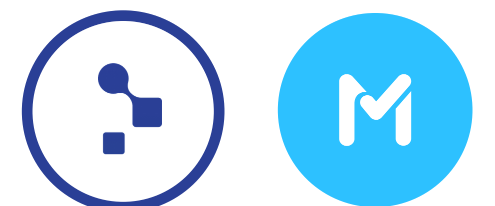
When it comes to a bottom-up implementation, the most vital factor for a team's successful shift to CI/CD is its ability to showcase the value of the adoption to management.

Increasing the Likelihood of Management Buy-In

Especially when working inside a fairly rigid organization, it's very rare to see the shift to CI/CD occurring top-down. In these cases, one effective strategy is for the technical team to align itself and sync up in terms of implementing continuous integration, adopting its practices in the search of adopting a DevOps culture, and agreeing on some of the key benefits that they will present to management that may be produced as a result of the changes. For example, one potential benefit may be finding issues earlier in a process, thus being able to identify and eliminate risks much sooner and at a lower cost.

It's important to do frequent reviews that allow the team to have better visibility of these issues in an earlier stage. It's also practical to properly implement a development strategy that allows the team to have a clear process and trackable metrics for improving it over time.

The shift to CI/CD is usually triggered from the bottom up, but at the end of the day, it's a transformation that requires management buy-in because it's about skill set, processes, and several areas to which management needs to allocate the proper time, resources, and attention.

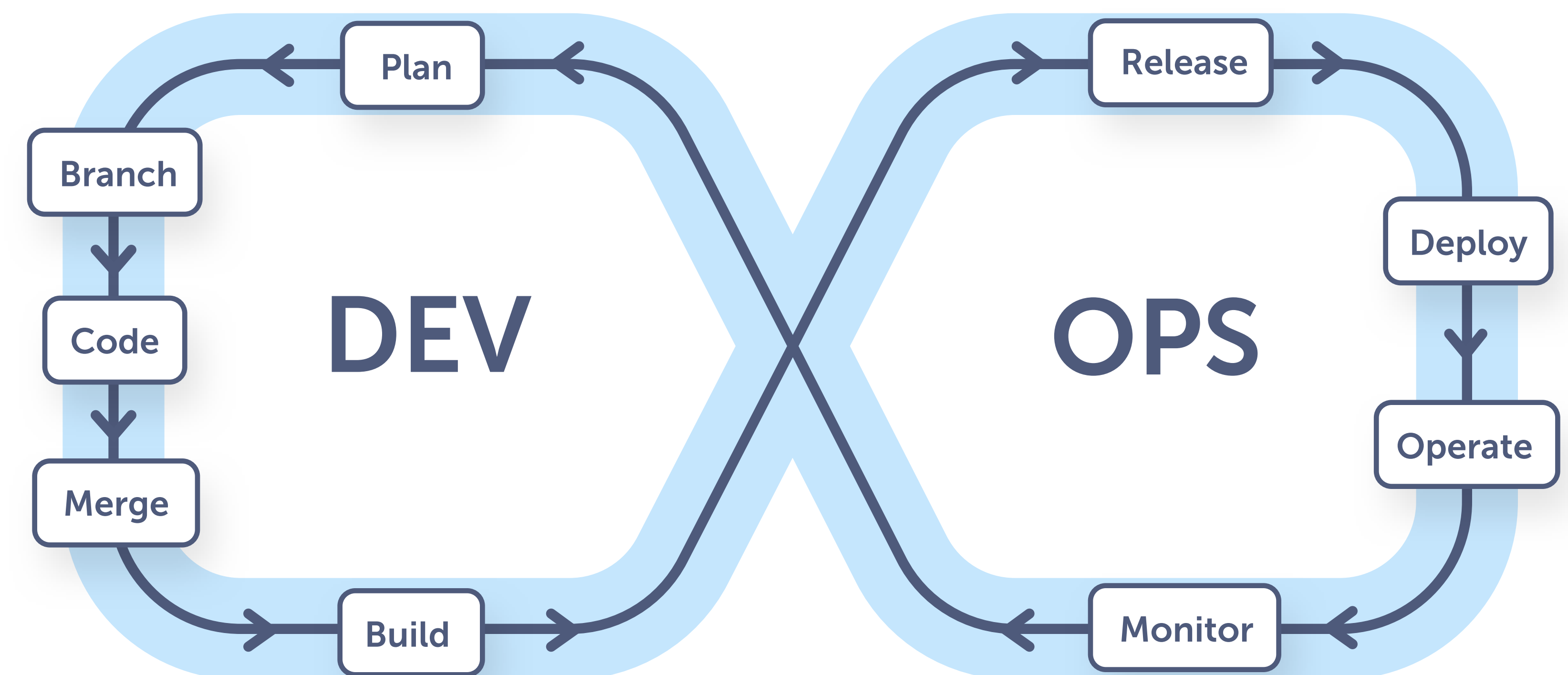


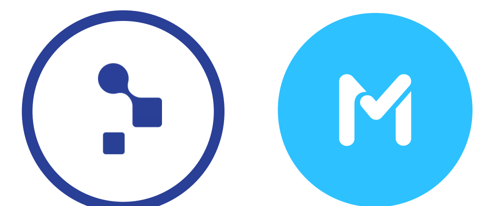
Mistake 2: Underestimating the Importance of the Necessary Cultural and Mindset Shifts

The Responsibility for Quality

In the original illustration by Dan Ashby below, one can see that there is no specific time for testing in DevOps. Meaning, testing should be a fundamental part of every single task, making it a continuous activity. In CI/CD, it's a part of everything that development teams do and may be part of the mindset shift that whatever phase of development one is currently in, there must be some sort of testing included as an integral part of it.

Testing should be something that everyone on the team is doing all the time, as a part of all their activities.





When it comes to software quality, all team members share ownership of it. For instance, the developers can write unit tests and also write the code with testability in mind, helping to mitigate risk from the start.

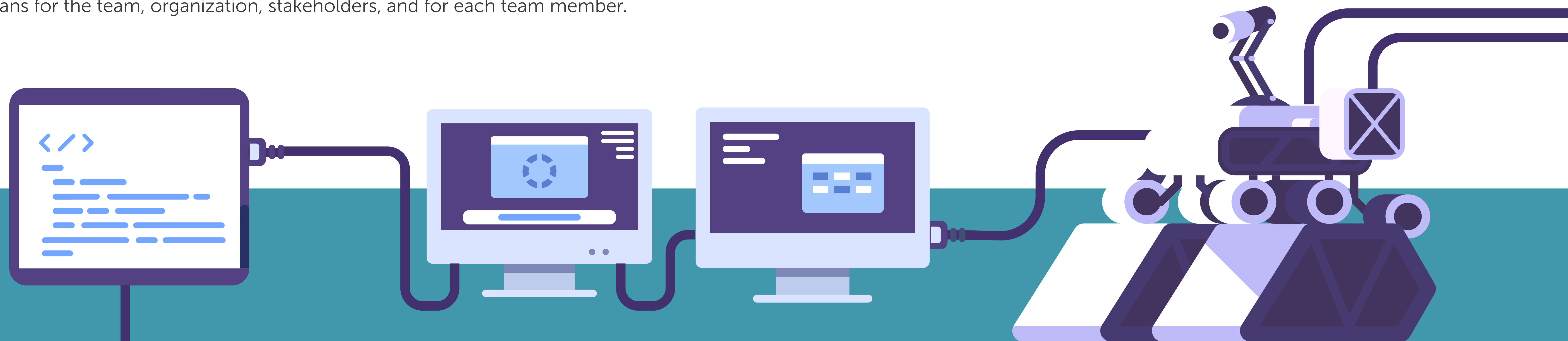
One simple way to help reflect the change in the view of testing is to change the testers' titles from QA to software tester or better yet, quality engineer. While this change may seem surface-level, too simple, or even silly, if someone has the title of "software quality assurance," it conveys the wrong message about who has the responsibility to ensure the product quality (which in Agile, CI/CD and DevOps is everyone).

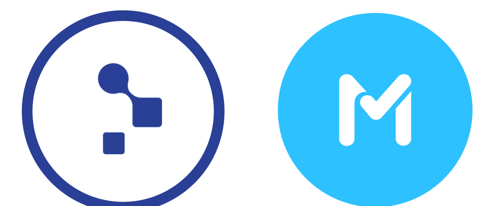
Another fundamental aspect is to be aligned in terms of quality, what quality means for the team, organization, stakeholders, and for each team member.

Continuous Feedback Culture

Culture is key, and it needs to be conducive to a high level of collaboration and communication. In DevOps, the most important element is the constant giving and receiving of feedback. In our experience, you cannot claim to have a culture of DevOps or have CI/CD in place if you aren't holding retrospective meetings.

Most likely if a team is trying to adopt CI/CD, then it is adopting an agile approach because it is not something that teams consider when following a more traditional approach such as waterfall. In Agile, particularly in Scrum, retrospective meetings





are fundamental for getting teams accustomed to giving and understanding feedback, allowing for continuous improvement.

The first step for adopting an agile methodology is to get into the habit of holding retrospectives, aka, "retros." In the first retro, one of the action items is to set up recurring retro meetings every two weeks. In that way, the practice of exchanging feedback becomes continuous.

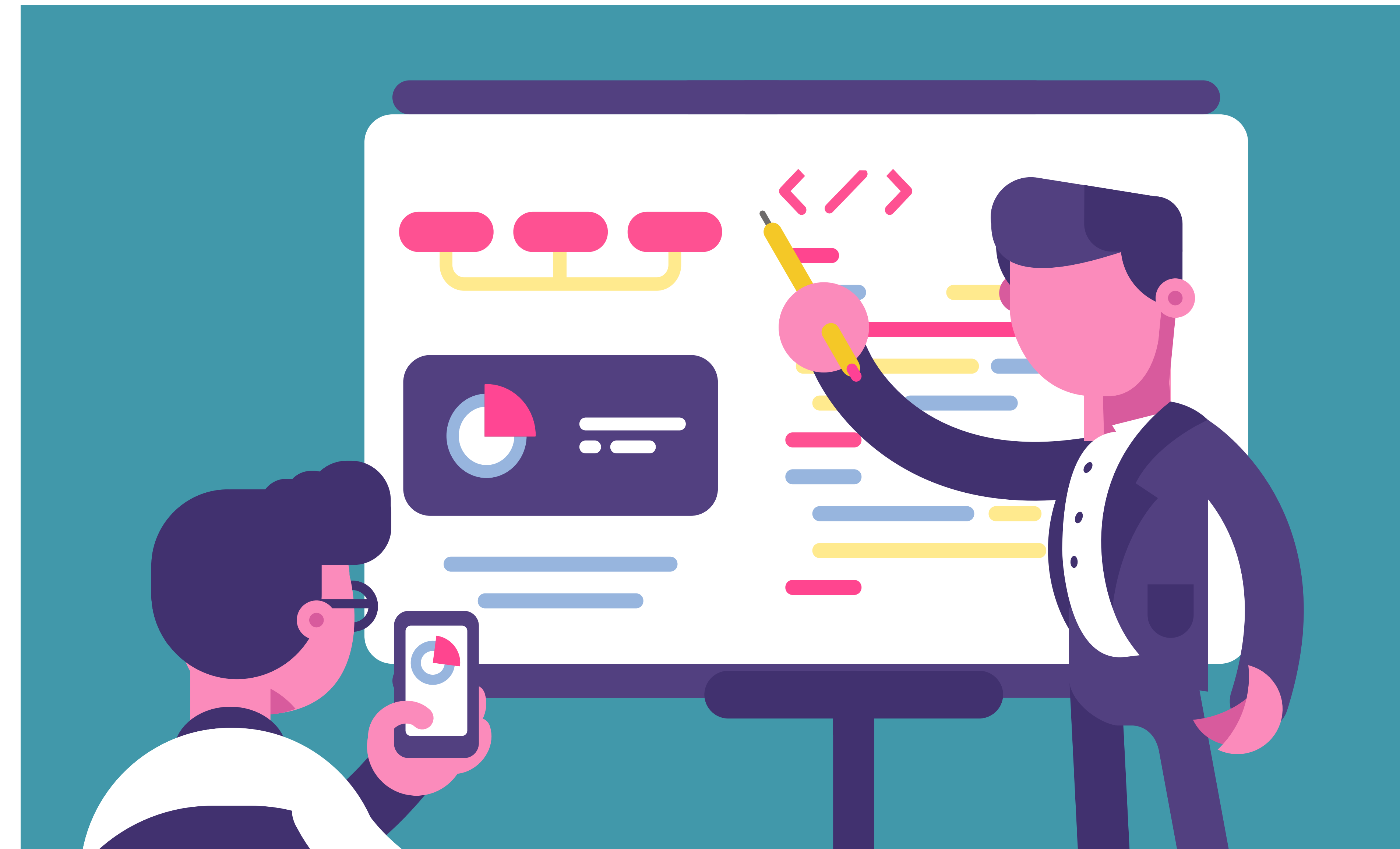
The Definition of Done

Here, the question of the matter is when there is an item in the team's Trello or Kanban Board in the "Done" column, what does everyone understand was executed for that item?

Does everyone have a common understanding of the team's "Definition of Done" (DoD)? Can it be assumed that the item was tested and how it was tested? Can it be assumed that someone reviewed the code? What else?

The DoD is a very powerful tool to share this understanding about the quality standards of what the team produces and how. It provides visibility into the process, even including how to implement CI/CD, as well as fosters trust. This is part of the mindset shift that must take place. If the DoD is unclear, the team must define it together and make it visible and accessible to everyone.

For a successful shift to CI/CD to occur, it's important for the team to adopt an agile mindset and culture, starting with a shared responsibility for software quality, fomenting a culture of continuous feedback firstly by conducting retro meetings, as well as having a transparent and universally agreed upon "Definition of Done."



Mistake 3: Not Setting Clear, Defined, and Realistic Goals

What to Aim For?

For the success of any type of transformation to occur, it should be measured and comprise tangible goals. When defining goals, it's very important to think about where the organization stands. If the team is currently releasing once a month, a realistic goal may be to cut down that time to just two weeks. A different, yet respectable goal for the same scenario would be to continue to release once a month, but to do so with greater quality and less risk.

While it may not be quite possible to copy a company like Amazon and release every 12 seconds, teams can consider where they are today, and at a very minimum, think about assessing their operating reality.

In thinking of their goals, teams need not to only think about what's realistic, (What is truly achievable?) but also **what will make the most sense in the time being**. All of the goals that are decided upon should be adjusted to the team's current context, for example, considering how frequently users need new versions, or new adjustments or fixes.

For some teams, it may not make sense to release more than three times in a year, and that's fine. In this case, it would be a mistake to over-engineer processes if they do not produce any added value.

Always keep in mind what is going to make the most sense. Will Continuous Delivery or just Continuous Integration be adequate? In the case of highly regulated organizations like banks or healthcare industry companies, CI alone may be sufficient, without the need for reaching Continuous Delivery.

In our opinion, Continuous Integration should be a must for every team, as it tends to become the golden standard for any team that experiences it.

Unlike CI, CD fundamentally changes the way companies deliver software as it impacts users, the business, and operations. Normally, teams should aim to implement CI as a starting point. Once implemented, it is possible to think about how to start releasing faster, and if doing so is ideal. Try it for a couple of sprints, measure it and evaluate how well it's working in order to make a long-term plan.

Determine whether your team will benefit from achieving Continuous Delivery or if Continuous Integration on its own is sufficient for your needs.

Mistake 4: Failing to Make a Scheduled Plan with Deadlines

The Blueprint

It can be really difficult to make a plan for something that implies a fair amount of research and the adoption of various tools. Before you make a plan, understand what the CI process entails, as illustrated in the following graphic:

In CI, there is a separate server which fetches the new changes from the source control server which makes a new build and tests it for each change in the code. This is done frequently, at least once a day, if not more.

When teams prepare the build automatically, they run the different tests that they've prepared and everyone receives a notification if anything has gone wrong or if everything is okay with the new build. This is part of the continuous feedback element.

This is the blueprint for the first step to undertake when making the transition

Create MVPs

In planning the transition, your team may ask, "What is a minimum viable product (MVP) for this change? What makes sense and adds value to the delivery pipeline?"

An initial goal could be Continuous Integration, starting with the automation of the build, some unique tests, and after that, the team can evaluate and plan the next MVP, adding some other automation to the process. Once CI is in place, the team can aim for Continuous Delivery.

It is important to note that the process itself must be tested in order for it to be trusted. Testers need to trust the delivery process and the automation that is involved there, since therein lies great risk. The second step may be to define a plan for testing the process.

There are countless aspects to consider when defining a pipeline. In this stage, testers can add a lot of value and contribute ideas about which verifications to add to the testing process. Also make sure to ask, “Who is going to prepare each test?” and, “In which environment will we run those tests?”

A good process to have in place is one in which each developer runs automated tests at different levels, (like at the API or UI level) in their own branch and after that, they ask for a merge request and someone else then performs a code review. After that, all the tests could be run in a fresh environment generated for the main branch. There the team can also include performance verifications or security checks or use tools to find issues related to accessibility, among other things.

Follow an Agile approach for planning the transition, creating a series of “MVPs” organized in different sprints, adding different levels of automation to the delivery pipeline in each.

When it comes to timing, it greatly depends on the technologies or the tools to be used and also the skills of those on the team.

Plan for Managing Costs

Another important consideration is cost when you're building the CI process. To manage this, some teams segment their tests into buckets, essentially running different buckets at different stages in the development process. It’s possible to run an entire regression suite on every commit or every merge but, unless the team is counting on a highly robust environment, it will take some time to get feedback to the developers. Many teams set out to have a small subset inside a branch and once they merge the master, they have a sanity suite which is robust enough but still such that the developer can get feedback in a few minutes and then once a night, run a full regression. Testim.io follows this practice itself to tradeoff the cost of infrastructure with agility.

Know what the CI/CD process will entail and create a plan for the transition involving a series of MVPs for each stage while keeping in mind how to manage costs. The timing will depend on each team and the tools they choose to use.

Mistake 5: Confusion Around the Role Changes and Responsibility Shifts

Each transition plan includes different people with different roles. Teams must manage the plethora of changes to be made and the shifting responsibilities. First, figure out what are the processes, people, tools and technology needed to reach your goals. Determine what will stay its course, what will be modified or eliminated, and what should be added.

Not everything needs to have a complete overhaul.

For teams that are used to working in Agile, there will be fewer changes to the roles and responsibilities. Teams not yet accustomed to being agile will first have to adapt to the changes that come with it. To reiterate, the main mindset that needs to change to be agile is that quality is the responsibility of the whole team. Otherwise, you will have a test team which will work as a quality gate with an inherent conflict of interest with the development team (developers being the builders while testers being the "destroyers" of their work). This conflict of interest between developers and testers was something that was made on purpose for traditional development environments to function, yet it's important to break out of this mindset in order to transition to CI/CD.

A testament to the shift-left movement gaining traction and developers being also responsible for quality is the fact that 30% of Testim.io users are actually developers, not testers

Essential Technology

Regarding technology, it's essential to use a git-based repository. It's not necessary to automate the process from the changes in the repo to the deploy, including all the automatic checks in the middle, the building process, and so on.

Everyone on the team will be responsible for understanding the process.

Continuous Delivery is also about getting feedback from users, not only from the tools with the automatic checks. For example, someone must be responsible for checking the logs to see if, for example, the newly released version created any negative impact to the system's performance or functionality. It's important to be familiar with application performance monitoring (APM) tools to see if there is any change in the way the resources of the service are being used, or if there is a new exception appearing in the log or a new javascript error. There are several well-designed tools on the market today to help process this information.

Mistake 6: Misunderstanding the Technical Requirements

Assessing Your Team's Maturity

How do you determine the technical requirements and the tools to make the shift to CI/CD?

An important first step is understanding where the team is today. Based on that, it's possible to plan what needs to change and what or who needs to be involved. The plan should also be linked, of course, to the team's goals and expectations. From experiences working with several clients, Abstracta has identified three pillars that teams must assess if they want to make improvements: **people, technology, and processes (and how they interact together).**

In doing this, software development teams assess their maturity, and what we will place the most emphasis on, their testing team and testing maturity, which is a a main driver of CI/CD and DevOps.

Abstracta has established a testing maturity model, wherein there are three main levels of testing maturity based on three basic characteristics; risks, quality, and costs. The model contains everything deemed necessary in order to lay the foundations for efficient testing, which is the minimum level of testing maturity in which it is possible to achieve CI/CD.

Using this chart is a great way to trace all the activities to carry out and to define some preconditions for making the transition. It allows for a broad overview of a team's technology landscape and it helps to determine which activities to engage in.

You can identify your testing maturity level using the chart below or take this online software testing maturity assessment.

	BASIC TESTING Aware Of Risks Measured Quality Measured Costs	EFFICIENT TESTING Controlled Risks Controlled Quality Controlled Costs	CONTINUOUS TESTING Reduced Risks Optimized Quality Optimized Costs
SOURCE CODE	Source code versioning	Code quality control	Continuous Integration / Continuous Delivery
ENVIRONMENT/ INFRASTRUCTURE	Separted testing/ dev environments Few devices/ browsers are covered	Test data management Full devices / browsers coverage Virtual servers	- Test environment management - Containers - Service virtualization
INCIDENTS/ BUGS	Bug tracking	Defects Causal Analysis	Traceability between features, issues and code
TEST MANAGEMENT	Basic test management Test planning with development	- Testing before development - Impact analysis	Agile management Testing and development as one unified team
FUNCTIONAL TESTS	Functionality inventory (backlog) Traceability between - TCs and features TCs, checklists, ET sessions	- Test design techniques - Test are prioritzed	Long-term code coverage strategy
AUTOMATED TESTS	Unit testing API automated tests	- UI automation - Unit testing with minimal coverage	Unit, API and UI automated test running continuously
PERFORMANCE TESTS	Client-side performance tests System performance testing (before go-live) Reactive monitoring in production	- Unit performance tests (during development) - Proactive monitoring in production	Performance tests running continuously
SECURITY TESTS	Basic testing of ccess control policies	OWASP TOP 10 Pentesting	Automated security checks
USABILITY TESTS	Usability testing	User testing	Accessibility testing

○ MANDATORY ◉ RECOMMENDED ● OPTIONAL

The Levels of Testing Maturity

Basic Testing

The first level of maturity in this model is "Basic Testing." At this level, there are different activities that need to happen in terms of the source code. For example, it's necessary to have versioning as well as a separate environment from which the code can be tested. Bug tracking should be in place as well as test management. Teams at this level know what it is they are testing and have achieved traceability between test cases and product features. In terms of automation, they may have at least some unit testing in place and test at the API level. This level of maturity may also comprise some performance and security testing.

Efficient Testing

One level ahead of Basic Testing, "Efficient Testing" is characterized by the achievement of a more efficient way of testing where various aspects like risk are not only identified, but controlled. At this level, teams begin to have more coverage in terms of testing, with more data management, and they may even have separate environments added. Some optional areas of quality here include defect causal analysis, greater prioritization in testing, and regarding automation, more end-to-end testing.

Continuous Testing

When a team reaches "Continuous Testing," Continuous Integration is already set in motion, and maybe even Continuous Deployment, depending on the team's needs. Teams at this level have a set of unit, API, and automated tests that are running continuously, generating the highly sought after continuous feedback loop. Performance tests are also continuous, and hopefully there are also security and accessibility checks included in the pipeline. At this level, risks are reduced and quality and costs are optimized.

Using this testing maturity scheme, it's possible to understand what are the necessary technical requirements and preconditions in order to advance to CI/CD, as continuous testing is a primary enabler of CI/CD.

Mistake 7: Forgoing an Effective Test Automation Strategy

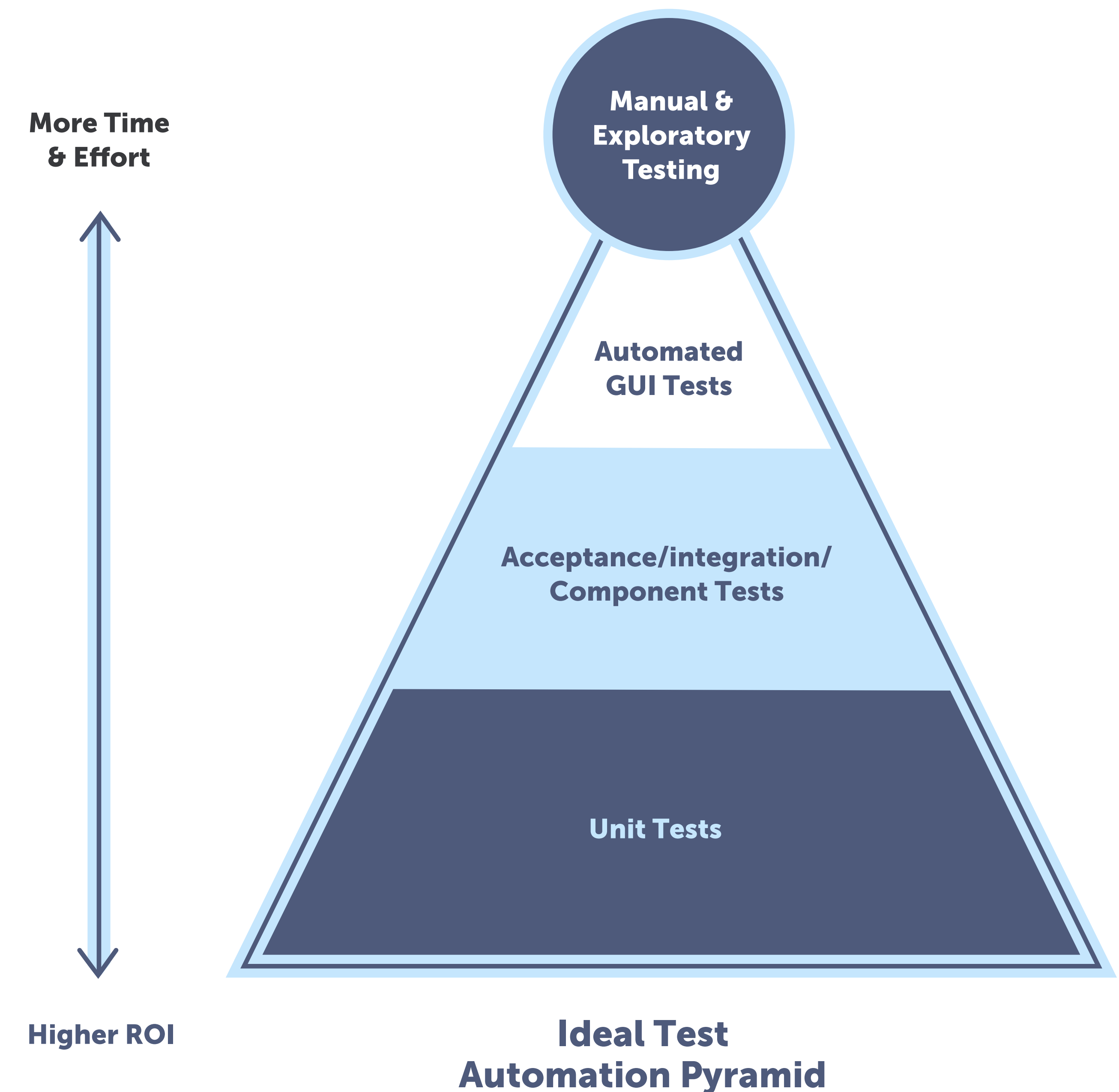
One of the mistakes that teams often make regarding test automation is wanting to automate everything and have every test running ALL the time. This might not be necessary, so it's important to define a clear strategy for how to approach automation.

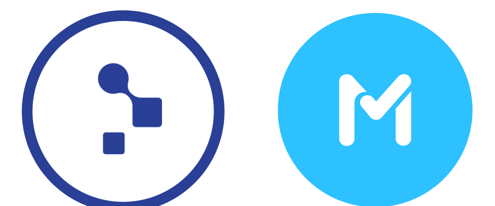
The Test Automation Pyramid

One guideline to follow is the Test Automation Pyramid by Michael Cohen, which illustrates the amount of automated tests to aim for across different test levels:

Unit Tests

At the base of the pyramid lie unit tests. The aim is to devote more effort to this kind of tests because they are faster to execute, easier to maintain and manage, and they enable teams to detect errors when they are still premature, leading to a higher ROI. Developers should be in charge of writing these whenever they're writing a new feature, cutting short the lifespan of any bugs that may appear.





Acceptance, Integration, and Component Tests

Moving to the next level is the middle of the pyramid. Considerable efforts should be directed here, which is automation at the service level. When it comes to API testing, only the most critical workflows should be automated, in order to test the logic before reaching the UI. At this level, the tests are slower than unit tests, but are still quicker than UI tests, so they will help to provide good feedback on how different services are working.

Automated GUI Tests

The top of the pyramid represents having a few automated tests at the UI level. These are the slowest and most difficult tests to maintain, but they serve the purpose of validating end-to-end functionality. These tests replicate how a user would actually use the system. When a team reaches this point and establishes a good foundation for its automation strategy, there will still be the need for manual and exploratory testing.

Cohen's pyramid is mostly focused on functional testing, but regarding performance, it is also recommended to include unit performance checks to the pipeline.

But, don't forget—It's also important to dedicate time and effort to maintaining the automated tests so that their results remain reliable.

Testim.io has helped several developers and testers to instill the habit of running automated functional and end-to-end tests because it uses artificial intelligence to keep the tests maintainable.

Take a Risk-Based Approach

Instead of boiling the ocean with your automation strategy, take a risk-based approach, focusing on the areas that will result in the highest return on investment (ROI).

For example, for an e-commerce business, a critical aspect to test is the online check-out process, as that user experience will have a direct impact on the business and its bottom line. For a search engine, the search functionality would be the main focus for testing

Focus on the user scenarios that are critical to your business and those that are critical to the user experience.

Larger organizations are putting more focus (once they have good baseline) on the visual aspect of their applications. Today, much more attention is placed on the user experience and the look and feel of applications. There's a multitude of tools to add different tests or checks to the pipeline, whether they be visual, performance, security, or accessibility checks. For all of these checks, it's important to take a risk-based approach because teams can get carried away and there is a considerable amount of preparation and maintenance to be wary of. Pay attention to what is truly adding value to the pipeline and helping to avoid risks.

Test automation is a crucial aspect of CI/CD. To know what to automate and where, it is a good practice to follow the approach illustrated by Michael Cohen's Automation Pyramid and prioritize the tests to be automated using a risk-based approach.

Mistake 8: Failing to Set the Right KPIs to Measure Success

How do you know if your transition is going well? How do you know if your Continuous Integration or Continuous Delivery environments are producing the expected results?

What we have seen at Testim.io and Abstracta, particularly in the CI/CD context, is that teams should pay attention to different metrics associated with the delivery pipeline. One of the most important questions related to CI/CD is, "How much time does it take for a change to reach the user?" With that information, teams can start thinking about how they can reduce this time.

There are some useful metrics related to the lean methodology which are lead time and cycle times. These are related to how much time is needed to go from an idea to the delivery of the implementation of that idea to the users.

Other metrics that are always paramount, regardless of whether a team is working in CI/CD or not, are how well the users rate the software and how happy the team is with itself and the way it works. Those are two fundamental key performance indicators (KPIs) as the basis of any company is a group of people whose mission is to deliver value to customers.

If a team wants to utilize a dashboard, it can turn to its CI engine (like Jenkins) which may provide more information related to the technical aspects of the Continuous Integration and Continuous Delivery pipeline, like the tests that are passing or failing, response times, code coverage, etc.

The important thing that teams and leaders should always be mindful of is how much time it takes to deliver value. This is the measure of success for implementing CI/CD.

About Abstracta

Abstracta is a world leader in software testing and quality engineering focused on improving the performance of software applications, implementing agile testing, and setting up automation stacks in CI/CD environments.

With offices in Latin America and Silicon Valley, Abstracta has expertise working not only with leading-edge proprietary and open source testing tools, but developing specialized tools for the financial, retail and technology sectors including companies such as BBVA Financial Group, CA Technologies, and Shutterfly.

Abstracta was recently named the fifth best software testing company for 2019 by Hacker Noon and is a top rated outsourcing company according to Clutch.co.

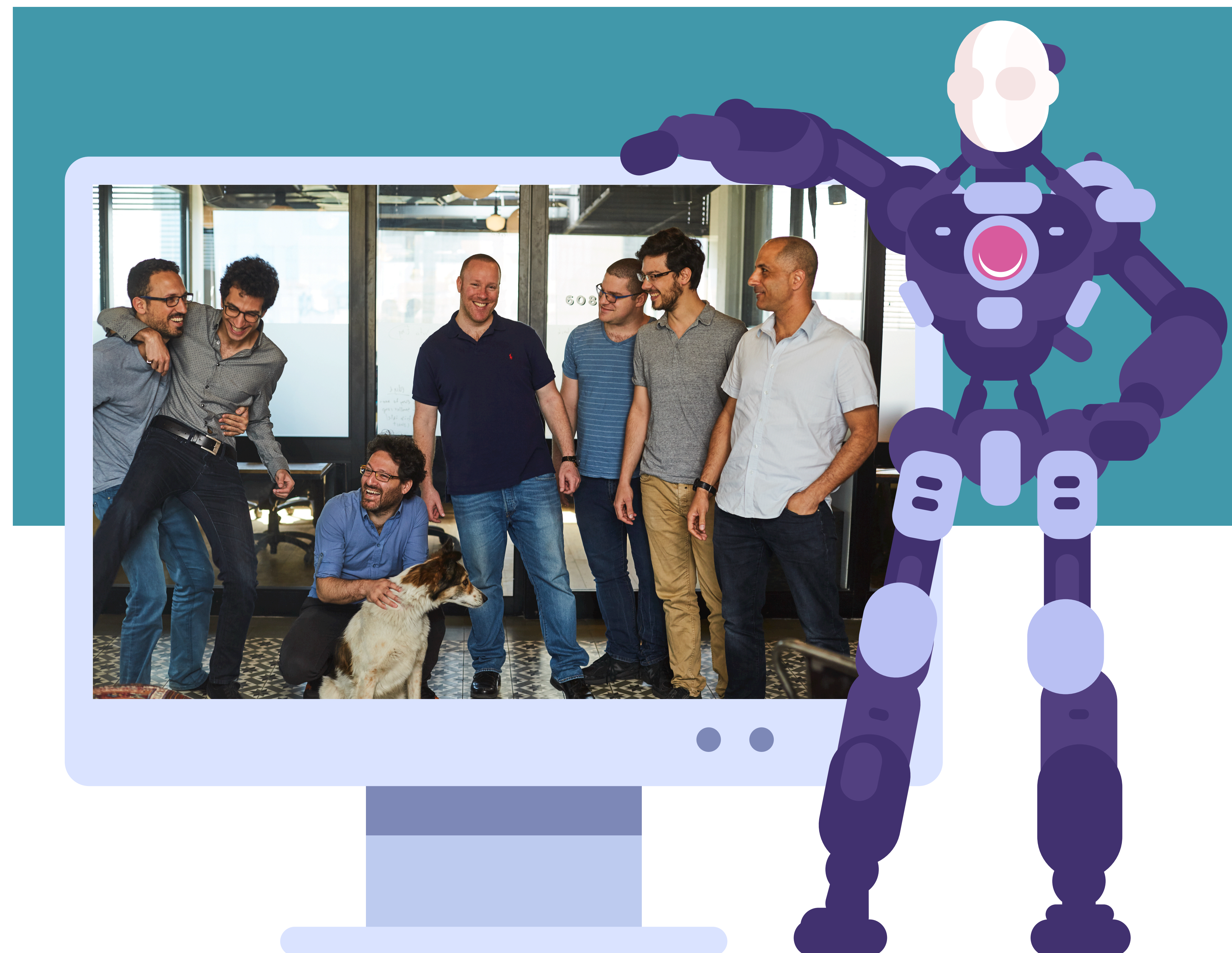
About Testim

Software is changing the world every day. Its behind everything we do; powering back office systems and websites, to robotics and mobile apps used across a wide range of devices. We want to help the people developing the powerful applications that are improving everyone's lives.

We created Testim because automated testing is still difficult. Being developers for the last 20 years, we recognized we spend time and energy maintaining our automated testing environments, and are still anxious with how a simple bug fix might break another part of our application.

Testim is software that's easy and works the way you want it to, supporting Agile transformations and shift left.

Our AI based software testing platform uses machine learning for the authoring, execution and maintenance of automated test suites to support your organizations user experience for every release..



About the Authors

Oren Rubin

Founder and CEO, Tesim.io

Oren has over 20 years of experience in the software industry, building mostly test-related products for developers at IBM, Wix, Cadence, Applitools, and Testim.io. In addition to being a busy entrepreneur, Oren is a community activist and and the co-organizer of the Selenium-Israel meetup and the Israeli Google Developer Group meetup. He has taught at Technion University, and mentored at the Google Launchpad Accelerator.

Federico Toledo, Ph.D.

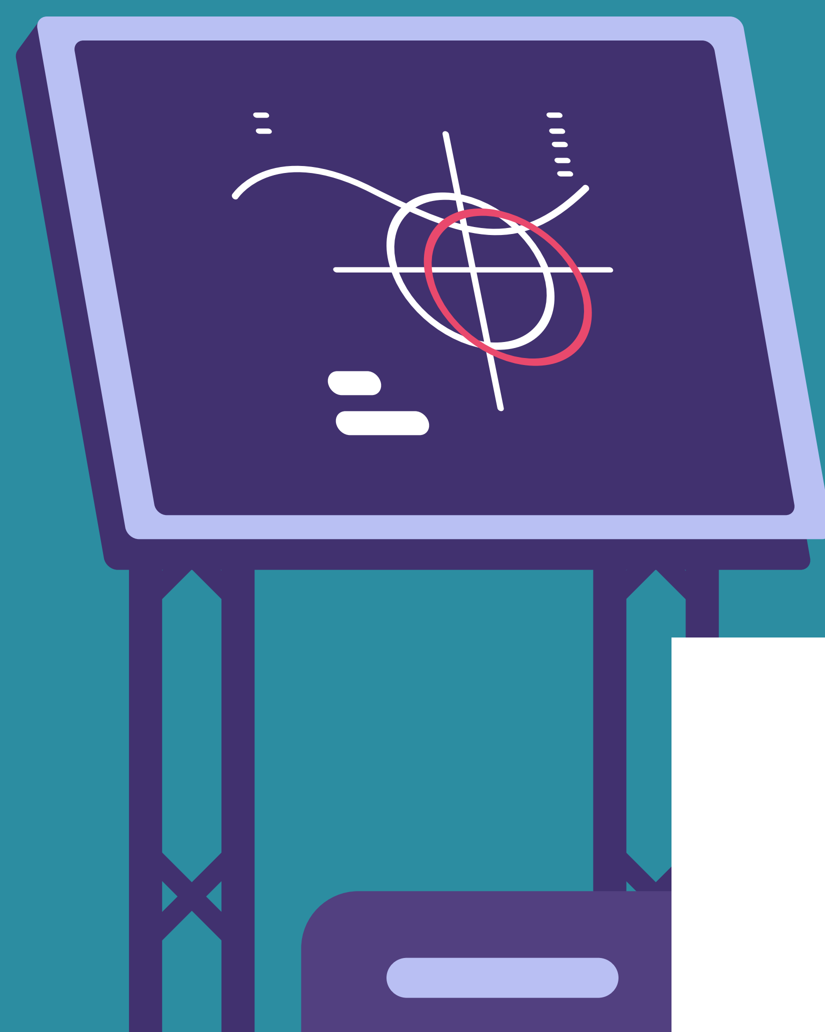
Founder & COO, Abstracta

Federico Toledo is the co-founder and director of the software testing company Abstracta, and holds a PhD in Computer Science from UCLM, Spain. With over 10 years of experience in Quality Engineering, he's helped hundreds of companies to successfully improve their application performance. He's dedicated to testing education as a professor, author and public speaker. He is also a co-organizer of TestingUY, the largest testing conference in Latin America

Lucia Lavagna

Chief Sales Officer, Abstracta

Lucia Lavagna is a performance engineer turned Chief Sales Officer at Abstracta, with over three years of experience overseeing client relations with companies like Benefit Cosmetics, The RealReal, and Singularity University, helping them to implement Agile testing and set up test automation stacks in CI/CD environments. As a performance engineer, she helped dozens of companies worldwide to improve system performance and reliability.



Thank You!

For more information contact us at

info@testim.io

or

hello@abstracta.us

www.testim.io