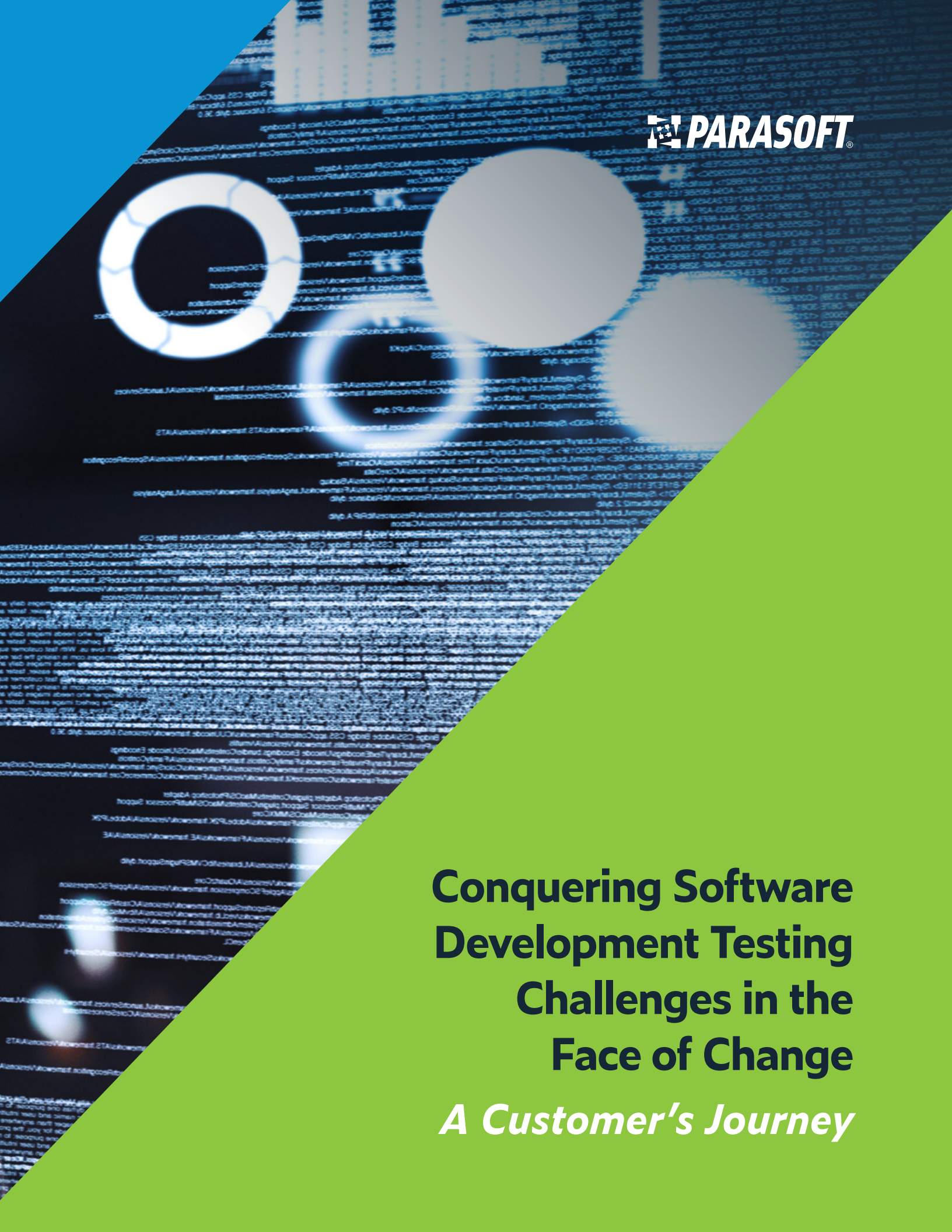


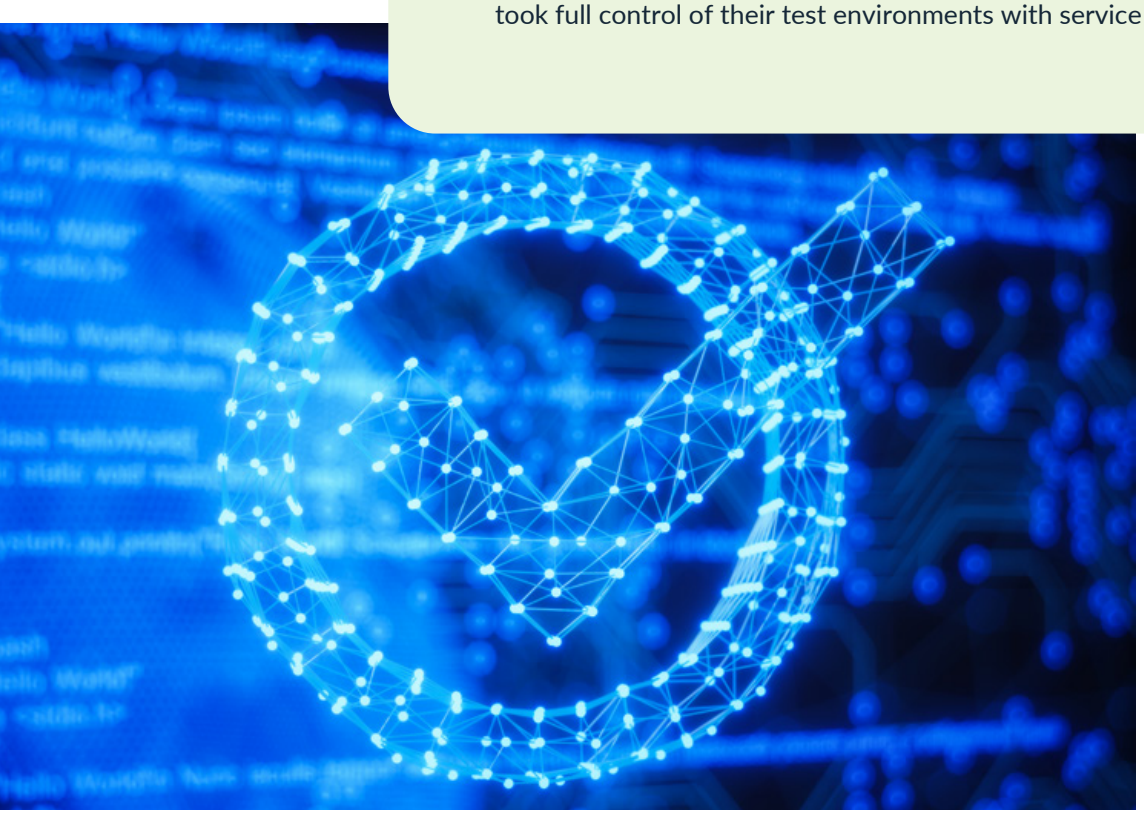
PARASOFT®

Conquering Software Development Testing Challenges in the Face of Change

A Customer's Journey

Financial services are dynamic. Multiple channels and a broad span of technologies — both old and new — must work together under highly prescriptive regulations. Delivering high-quality applications to the lines of business requires testing software early and often. All while managing a complex array of dependencies and integrations of data and interfaces.

Herminio Vazquez is an IOVIO consultant in the ING Netherlands Mortgages Tribe in a specialized squad named Data Rangers. In this follow-up piece to *Testing Alone is Losing the Battle Against Change*, he offers a strategy checklist to help you overcome your software development testing challenges. It's inspired by how ING Mortgages took full control of their test environments with service virtualization.



A STRATEGY CHECKLIST TO HELP OVERCOME YOUR SOFTWARE DEVELOPMENT TESTING CHALLENGES

The motivation to write about **data orchestration** and **service virtualization** in the first place came as a reflection of the challenges we had to overcome when deciding what to test and when for ING Mortgages Netherlands. This is especially relevant when immersed in a complex and heavily integrated enterprise environment.

With **change** and releases accelerating, it was necessary to revisit our approach to testing and come up with ideas about how to prevent defects sneaking into production.

We recognized the problem and asked a question, perhaps you can ask the same question to prepare for the checklist.

CAN YOU TEST YOUR BUSINESS APPLICATIONS AT ANY GIVEN POINT IN TIME?

And for us, the answer was clear: **NO**.

Some of the reasons were due to:

- Restrictions on external applications or environments.
- The inability to consume certain data sets being concurrently used by other teams.
- Waiting times on the `release traffic control` to allow our testing window landing in an environment.

There was no chance to launch a full cycle of testing on-demand at the press of a button, or even just at the call of "Let's Go, Team!"

This event triggered me to think of those many times I have visited IKEA to purchase that extra thing needed at home — walking through its corridors and finding those comfortable chairs displayed in crystal cubicles with a robotic arm pushing them down and stressing their wooden structure. I also remembered a gentler mechanic setup for continuously opening and closing drawers from a cabinet. What I was thinking is that IKEA produces home furniture, nevertheless, they have robots to test.

At this point, I know what you're thinking: "Welcome to the 21st century, son. This is called `test automation` and even your neighbor's son knows at least three test automation tools to get the job done nowadays." Indeed, I agree that the testing industry has brought automation so much forward that it's unthinkable for software delivery teams to operate without it.

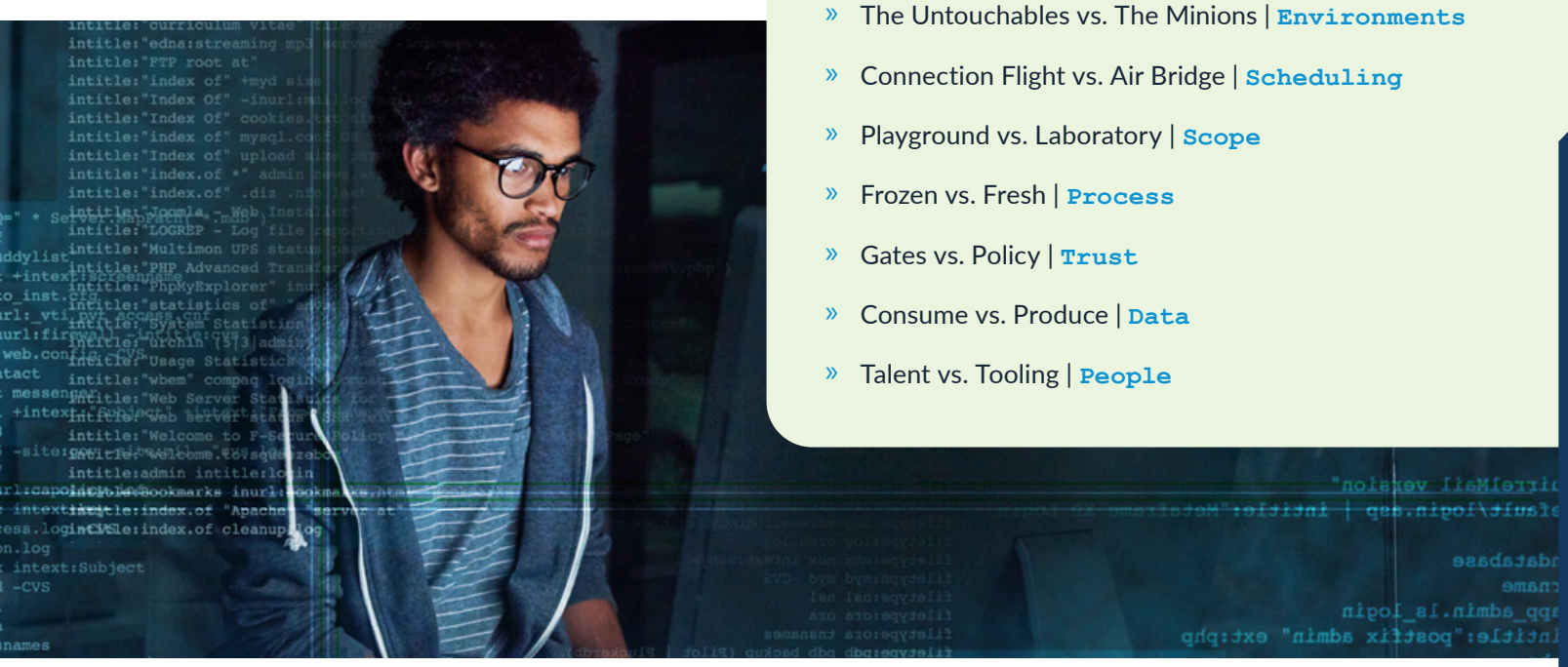
But automation is not the catch here. It's the idea that for every chair produced, a sample population of them will have to get the robot set up, mounted, running, and available for testing at any given point in time so that testing will never be compromised. They were virtualizing people sitting `continuously` in their chairs!

Back to the real world. If software is not chairs, what are the common denominators of software delivery teams that prevent them from accelerating delivery? And how does recognizing those common denominators help you determine whether your testing strategy is ready for the word `continuous`?

Well, no worries. This article presents a curated list of symptoms and their associated (suggested) solutions, as mentioned in the previous article, to change the shape of testing as we know it.

The list goes like this:

- The Untouchables vs. The Minions | `Environments`
- Connection Flight vs. Air Bridge | `Scheduling`
- Playground vs. Laboratory | `Scope`
- Frozen vs. Fresh | `Process`
- Gates vs. Policy | `Trust`
- Consume vs. Produce | `Data`
- Talent vs. Tooling | `People`



THE UNTOUCHABLES VS. THE MINIONS

When your environments become disposable and not indispensable, then you will be ready for running tests at any given time.

If something is holding you back from destroying an environment, that means that you don't have enough control, and therefore the conditions for your entire testing strategy could be compromised.

The motivation to lose the fear of your environments — as if they were gangsters or members of another league that nobody could touch — is the main challenge here. We encourage you to visualize them as small little toys that can be brought in or out of the picture with no drama.

If you think about it, there is a current approach to dispose of entire technology stacks, which is `containers`. Removing dependencies with other environments require you to **virtualize** those interfaces and manage the data dependencies inside them.

Virtualizing environments is a strategic decision that spans from testing to hardware, software licenses, and maintenance. It was hard for us to convey that message to our stakeholders, as they understood that no more conceivable budget could be allocated to testing tools. However, when we made clear that environments and configuration will be side benefits, our simplification approach landed more gently.

Reducing the need for one more integration environment in a complex enterprise could bring one zero less in the monthly bills.

> Virtualizing environments is a strategic decision that spans from testing to hardware, software licenses, and maintenance.

Suggestion

- Virtualize services.
- Containerize when possible.
- Work with smaller data sets.
- Don't let complexity prevail.

How to Get Started?

- Identify data flows.
- Observability and tracing.
- Service catalog.
- Schema registry.
- Enterprise canonical model.

> Readiness requires planning and scheduling to be on a continuum. No booking of time or resources should be required. It means access to everything you need to go from A to B in your testing, without disruptions.

CONNECTION FLIGHT VS. AIR BRIDGE

Opportunities are commonly associated with doing the right things at the right time. In software delivery, this is not an exception. Time is of the essence and bearing the cost of waiting is something that all software delivery teams must deal with when removing roadblocks in their delivery pipelines.

`Readiness` requires planning and scheduling to be on a continuum. No booking of time or resources should be required. It means access to everything you need to go from A to B in your testing, without disruptions.

Working as a consultant requires a lot of traveling, and my commute to the workplace in many occasions required me to wake up early on Mondays and take an airplane to the destination city of my project. I remember when the concept of air bridges was born — when commutes between certain cities became more frequent, and no bookings or reservations were required, you could just show up and travel.

> Scheduling flexibility does not mean a lack of planning. What it means is the ability to perform any activity without reservations.

Air bridges are a great concept for continuous testing because as soon as you think about connections or dependencies, then you automatically lock your strategy to work on one thing at a time or establish hard dependencies that will end up in waiting times.

Scheduling flexibility does not mean a lack of planning. What it means is the ability to perform any activity without reservations. Boarding the plane still requires a queue, even for those world travelers and families going first.

Frequently, a delivery team associates a time window for a certain activity, like running regression suites unattended during the night, as we do in our team (4 times to develop confidence and gather more data). This is just a preference, but it shouldn't act as a mandate, otherwise, you lose your ability to test at any given time.

I need to agree that this is not easy, and by no means will I attempt to provide a trivial solution that can be bullet-pointed. We are still finding ways to time travel certain business processes that need a lot of data preparation or dependencies to be able to run them on the fly as opposed to orchestrating the entire data or process dependencies every time. An example of this in the Mortgage application ecosystem is to simulate signed offers moving to contracts without the data collection ceremony.

Suggestion

- No timesheet for testing.
- Environment isolation.
- Self-provision of data.
- Business process time travel.

How to Get Started?

- Process facets.
- Identify master data sets.
- Canned process data set.

PLAYGROUND VS. LABORATORY

Embarking on the advantages of continuous testing may lead to a lot of exploration that, if not prevented, could lead to reduced time for collecting the benefits of it. Not knowing what success looks like, is what I refer to as a playground. However, if success looks like removing a data dependency or simplifying one component of the architecture then we are talking about the tangible benefits of your continuous testing approach.

Don't let the fun of creation, derived by impersonating other systems, drive you away from your main focus. Which is to enable continuous testing.

I remember when we containerized and deployed our first set of services. We were only two people working full time in the authoring of the virtual assets and their data requirements. However, we worried about other members being able to support us and created requirements to a larger team at the time, without any special need. In reality, the main objective was to expand the `coverage` of our virtual assets and managing data requirements for each of them. Scaling up the virtual solution proved to be the wrong decision at that time.

At a later stage, we found that keeping the solution simple, and expanding the number of virtualized services paid better dividends than exploring auto-scaling, authorization for certain services, notification on renewal of certificates, and many others. All these things when looking from outside, were one-off activities that were not necessary to keep our testing moving forward to change, instead, they increased the technical testing debt.

Our learnings in this area, of playground against laboratory, is to allocate resources and time with clear objectives. Set a number of services virtualized over time, and data dependencies removed across the business process. If this is not feasible, or difficult to estimate, then most likely the benefits of this entire approach will never surface as enablers for faster delivery or facilitators of continuity.

If your team is new to the concept of virtualization, then spend some time setting a minimum set of guidelines and conventions with a minimum bank of knowledge and references, so that everyone on your team can understand the ultimate objective of your continuous testing strategy.

Suggestion

- Create a virtualization matrix.
- Allocate time and resources to it.
- Set expected coverage in time.
- Baseline virtualization activities.

How to Get Started?

- List critical interfaces.
- Time-box virtualization activities.
- Break complex ones into smaller pieces.
- The job is finished when positive and negative cases are covered.

FROZEN VS. FRESH

Freezing releases will never contribute to a continuous testing approach.

Parallel streams of work are the new norm, and with it comes to the ability for everyone and everything to work seamlessly, without disruptions. Time to ax human semaphores.

> Everyone should be able to access early releases or beta versions of software without being stuck into the Dev-Test-Acceptance-Production highway.

If the term *freeze* is around your delivery pipelines, it is time to rethink about enabling any type of activity associated with software delivery in `concurrent` mode, which boils down to operating in smaller and less monolithic pieces.

Everyone should be able to access early releases or beta versions of software without being stuck into the Dev-Test-Acceptance-Production highway. And that `fresh` software produce is always at reach for teams conducting the activities needed to validate if it meets its purpose.

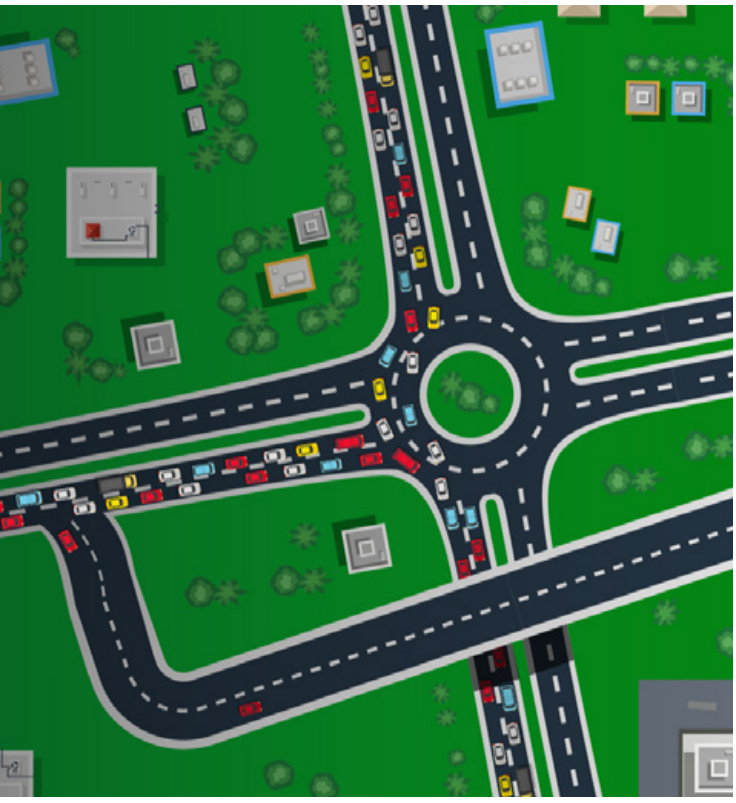


Figure 1: Don't get stuck on the Dev-Test-Acceptance-Production highway.

Fresh releases mean configuring pieces of your environment dynamically, or at speed. This modularity plays a key role in not blocking the pace of fast releases. This also brings new challenges, like the ability to deploy configurations or maintain a `service mesh` to route traffic from one point to another with no configuration or deployment changes. This infrastructure level capability is fundamental to keep your test assets in ready mode all the time.

An example will be thinking in endpoints, where version 1 and version 2 of that endpoint are accessible to an environment or team, without having to target a full release on an environment.

Suggestion

- Smaller environments.
- Service proxies.
- Zero-config routing.

How to Get Started?

- Automated deployments.
- Branch policies in repositories.
- Configuration-only builds.

GATES VS. POLICY

For this idea, there is not much to say. The end of the Change Advisory Board as a group of humans deciding on the risks associated with software delivery is certainly becoming more inclined to an **algorithmic risk policy** instead.

Stages and preconditions are built-in features in modern continuous delivery pipelines, which implement policies responsible for assessing risk, adhering to a standard process, or for compliance. If your delivery is subject to audits or a set of controls, then it is necessary to digitalize them all together with all the assets mentioned so far in this checklist.

For us, semantic versioning and tagging were strategies that helped us to maintain compliance and traceability across releases, and especially helped us triangulating anomalies or simplifying the eventual root cause analysis when failures occur.

Your target for policymaking is to reverse engineer your decisions, and most of all, to establish a computation of risk. If the risk is in the hands of individuals and their gut feeling, most likely the outcome will be arbitrary, and its result will consume more time, resources, and slow down change.

Testing at any given point in time requires making decisions consciously and not arbitrarily.

Suggestion

- Semantic versions.
- Branch/release tagging.
- Digital risk policies.
- Reversible decision process.

How to Get Started?

- Create risk policies (risk = impact x probability).
- Compute risk.
- Back up decisions with data points.

CONSUME VS. PRODUCE

This is perhaps the most well-known artifact for continuous testing. There are limited insight and low probabilities to execute testing activities without the right data sets.

Typically, software delivery teams resource data extracts from production environments. However, this approach leads to additional steps like depersonalizing data sets due to acts on data protection or data governance. In any case, you guessed it well, it takes time, and it can block your ability to test.

One kind reminder is that your systems of record hold the so-called `good data`, data that passed through the checks and validations (typically implemented in middleware or various layers of your application).

What this means is that you still will have to work out on the generation of `dirty data` or data that can help you trigger the not-so-happy flows, or those exercising the error handling or exception handling of your applications.

> A gentle word of caution to stress that the quality of your data sets influences the quality of your test results. And it is through test results that you incentivize confidence, knowledge, and trust.

In any case, we are talking about the ability to produce data at the lower cost, and in a reasonable time, so that testing activities are not influenced by this dependency.

For regression test suites, a core set of data is a viable solution to reduce preparation time, and move testing activities at a rapid pace. Consolidating such a data set is the result of once more, minimizing and modularizing areas of your application and limiting data requirements for specific business scenarios.

I guess you want to think about your team as a **producer** rather than as a consumer of data.

I acknowledge that the production of large data sets might be time-consuming or not feasible in the skill sets or toolsets available to your teams, hence the idea of reducing the scope of the data requirements according to the minimum requirements of your interfaces or systems of record.

A gentle word of caution to stress that the quality of your data sets influences the quality of your test results. And it is through test results that you incentivize confidence, knowledge, and trust.

Suggestions

- Test automation for data generation.
- Synthetic data sets.
- Don't forget about error handling data.
- Minimum data requirements.

How to Get Started?

- Complement extractions with small self-production.
- Services with smaller data sets.
- Fake data generators (people, numbers, and so on).

TALENT VS. TOOLING

At some point in your journey, your rockstars will start challenging you. Change will always bring resistance and friction. Whilst some members will foster change, it will also be an arguing catalyst for emerging experts and seasoned professionals.

Your approach may be challenged by members with crafty mindsets, arguing that programmatically they can produce similar results in less time, without having to resource the right toolsets. Of course, *if all you have is a hammer, everything looks like a nail.*

As mentioned previously, creating a `mock` is not rocket science. Orchestrating a set of virtual services, controlling and versioning data sets, and scaling the solution at a press of a button is what is called `operationalizing service virtualization` which is far from putting three lines of code, with a `REST` endpoint that does one thing.

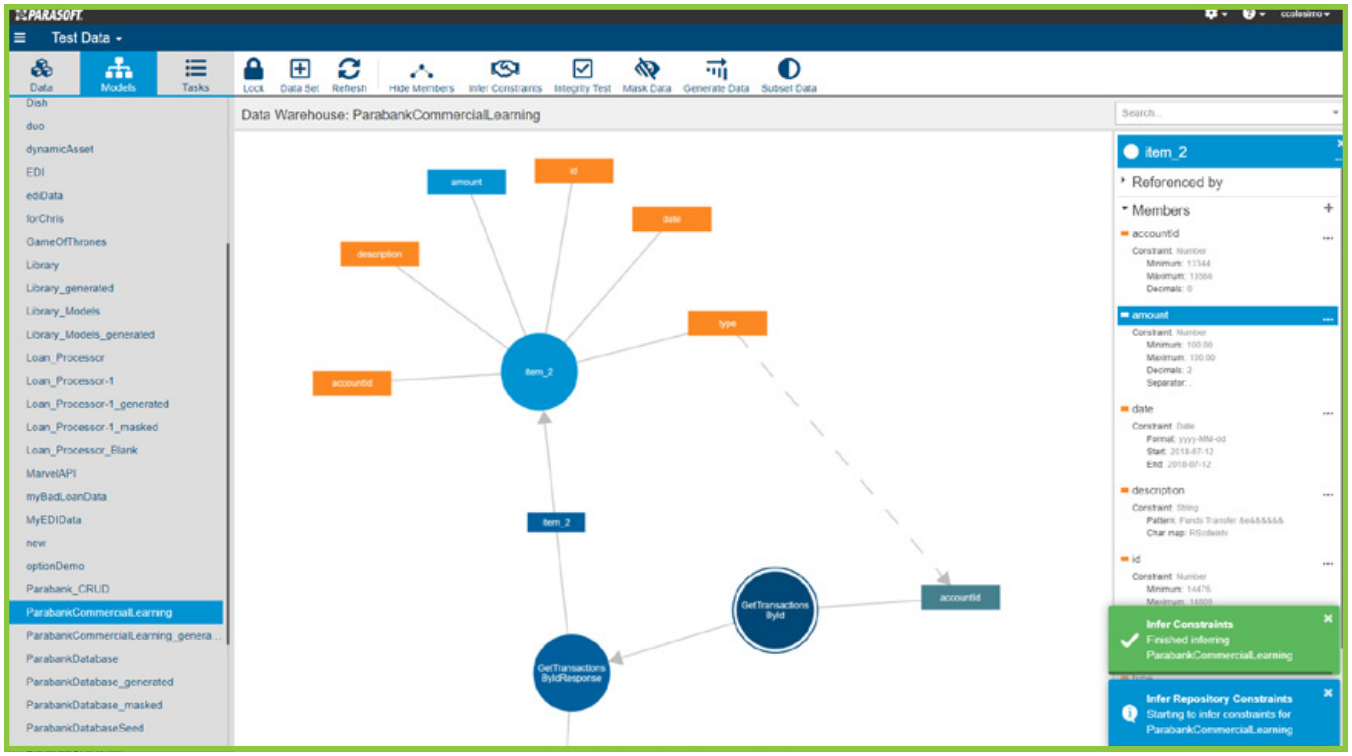


Figure 2: This data model is automatically generated by recording traffic for a virtual service. It shows a data object, item-2, with children elements like account ID, amount, date, and so on. To fill the object with data, you can infer constraints to determine the kind of data in the data repository . A string? A number? What are the minimum and maximum sizes of those values? Once understood, you can generate data within those constraints. A relationship between type and account ID is also set up. Masking one of those values makes sure they're consistent between the two elements.

Get your teammates involved and use their expertise to boost your continuous testing journey. Their inclusion is relevant for the success of your efforts. Collective work will end up helping you to remove technical roadblocks in the implementation.

Be aware that you want to reduce dependencies, if all your solutions rely on one person, then it's time to rethink about spreading knowledge and diversifying skills so that the sum of all your capabilities leads to the speed and continuity you are aiming for.

Suggestions

- Listen to your team, but don't overload them with new work.
- Process and people first, tooling after.
- Think big, not one service only.
- Produce synergies, a leader also needs followers.

How to Get Started?

- Quick turnaround with examples.
- Facilitate material and onboarding.
- Bring experts with experience to share their journeys.
- Use the entire checklist as a defense of your approach.

CHECKLIST SUMMARY

1. The Untouchables vs. The Minions

`Environments`

Suggestion

- Virtualize services.
- Containerize when possible.
- Work with smaller data sets.
- Don't let complexity prevail.

How to Get Started?

- Identify data flows.
- Observability and tracing.
- Service catalog.
- Schema registry.
- Enterprise canonical model.

2. Connection Flight vs. Air Bridge

`Scheduling`

Suggestion

- No timesheet for testing.
- Environment isolation.
- Self-provision of data.
- Business process time travel.

How to Get Started?

- Process facets.
- Identify master data sets.
- Canned process data set.

3. Playground vs. Laboratory

`Scope`

Suggestion

- Create a virtualization matrix.
- Allocate time and resources to it.
- Set expected coverage in time.
- Baseline virtualization activities.

How to Get Started?

- List critical interfaces.
- Time-box virtualization activities.
- Break complex ones into smaller pieces.
- The job is finished when positive and negative cases are covered.

4. Frozen vs. Fresh

`Process`

Suggestion

- Smaller environments.
- Service proxies.
- Zero-config routing.

How to Get Started?

- Automated deployments.
- Branch policies in repositories.
- Configuration-only builds.

5. Gates vs. Policy

`Trust`

Suggestion

- Semantic versions.
- Branch/release tagging.
- Digital risk policies.
- Reversible decision process.

How to Get Started?

- Create risk policies (risk = impact x probability).
- Compute risk.
- Back up decisions with data points.

6. Consume vs. Produce

`Data`

Suggestions

- Test automation for data generation.
- Synthetic data sets.
- Don't forget about error handling data.
- Minimum data requirements.

How to Get Started?

- Complement extractions with small self-production.
- Services with smaller data sets.
- Fake data generators (people, numbers, and so on).

7. Talent vs. Tooling

`People`

Suggestions

- Listen to your team, but don't overload them with new work.
- Process and people first, tooling after.
- Think big, not one service only.
- Produce synergies, a leader also needs followers.

How to Get Started?

- Quick turnaround with examples.
- Facilitate material and onboarding.
- Bring experts with experience to share their journeys.
- Use the entire checklist as a defense of your approach.

TAKE THE NEXT STEP

Ok, a lengthy list, but hopefully plenty to invite you to think about changing the face of testing in the advent of change.

If you're interested to know more about Parasoft Virtualize and its benefits, you can always reach out to Parasoft and IOVIO.

ABOUT PARASOFT

Parasoft helps organizations continuously deliver quality software with its market-proven, integrated suite of automated software testing tools. Supporting the embedded, enterprise, and IoT markets, Parasoft's technologies reduce the time, effort, and cost of delivering secure, reliable, and compliant software by integrating everything from deep code analysis and unit testing to web UI and API testing, plus service virtualization and complete code coverage, into the delivery pipeline. Bringing all this together, Parasoft's award winning reporting and analytics dashboard delivers a centralized view of quality enabling organizations to deliver with confidence and succeed in today's most strategic ecosystems and development initiatives—cybersecure, safety-critical, agile, DevOps, and continuous testing.