WHITE PAPER

THE QUALITY PIPELINE

What is a high performing software development team, and what can be done to improve performance? In this white paper, Principal Consultant Stephen Masters shares his recommendations for achieving consistent quality outputs from teams working across an organisation.

digiterre

WHITE PAPER

Typically, at a large company, there are multiple teams building and maintaining software systems. It is not unusual for each team to have its own approach to development when it comes to things like project methodology, technology stack, architecture, and testing. With such a variety of approaches, how do we know which systems are more likely to have critical bugs? Which systems might cause outages when they are released? Which systems are most vulnerable to attack?

MEASURING PERFORMANCE

When evaluating an IT department or a programme of work, all too often, management have a very subjective view of the quality of systems and the performance of the teams developing them. This is inevitable in the absence of objective measures. To help remedy this, Google's DevOps Research & Assessment (DORA) programme, proposed the concept of Four Key Metrics as a guide to the performance of a software development team. The most influential metrics they identified are:

- **Deployment Frequency** - How often a team successfully releases to production.
- **Lead Time for Changes** - How long does it take from code committed to running successfully in production?
- **Change Failure Rate** - The percentage of deployments causing failure in production.
- **Time to Restore Service** - How long it takes to recover from a failure in production.

By measuring these values and continuously iterating to improve them, we can deliver higher quality systems at increased velocity.

The DORA team have developed Four Keys; an open source pipeline which gathers these metrics. This is a great start, and in order to capture this data it encourages teams to be transparent and make use of tools to track events such as change requests, commits, deployments, and issues. It relies on teams being well-organised in how they track these things - for example, by linking a commit to a change request, a deployment to a commit, and an issue ticket to a failed deployment.

HOW DO WE IMPROVE?

Given these metrics that we would like to improve, what are some practical steps that we can follow? We will be discussing the following guiding principles in the next section, to help achieve these goals:

- **Gather metrics** - Make the metrics visible and transparent. The DORA team developed Four Keys; an open source pipeline which gathers these metrics from common tools and provides dashboards.
- **Maximise automation** - The process from commit, through testing, and into deployment should be automated, with minimal manual steps.
- **Shift-left** - The sooner you can identify an issue in the project lifecycle, the faster and easier it is to fix.

START WITH THE REQUIREMENTS

Following the shift-left principle, we should consider what we can do to improve how we capture requirements. One of the most valuable activities here is to hold three amigos discussions, which include the business or product owner, a developer, and a tester. By ensuring that these perspectives are represented in these sessions, we avoid misunderstandings, and importantly, we capture acceptance criteria which can be built into tests. Regular demonstrations of progress help ensure that we are on track. By improving how we do these things, we avoid spending time building the wrong thing.

THE DEVELOPER'S LOCAL ENVIRONMENT

Once requirements are understood, the focus moves to developers in their local environments. Here, a consistent style across a project helps all developers to work on it more efficiently, and can be enforced using a linter in a developer's IDE. When building, static analysis tools can highlight issues such as complexity, or unreachable code. Developers should be writing extensive unit tests and some integration tests. The build tooling can raise warnings if the test coverage does not meet a minimum agreed for the project.

By performing these checks locally, the developer gets instant feedback on a number of issues. By enforcing agreed styles and test coverage targets, there is a reduced likelihood that a colleague will need to reject a peer review of the commits, which saves reviewer time, and avoids a frustrating cycle of rejections or arguments over code style. Depending on the team structure, it can also be useful to include a tool such as ArchUnit, which performs checks on the application architecture when the Unit tests run.

Tools such as Docker-Compose and Kubernetes, enable us to create local environments which replicate production closely. This is incredibly useful when building systems comprised of multiple microservices.

AUTOMATE QUALITY GATES

A quality gate is a project milestone where we evaluate whether criteria have been met to be ready to go live. Unfortunately, all too often they are carried out when a project is "dev complete", which can lead to a checklist culture and delays when huge lists of bugs and vulnerabilities are given to a development team to resolve. Usually the amount of time needed to resolve does not match the remaining project timeline and budget, which often leads to unreasonable demands for developers to work unpaid overtime, overruns, and inevitably the release of software with large numbers of known bugs and vulnerabilities of varying criticality.

Following Agile principles, we know that if we identify issues against a small change early, then it's much easier to identify the cause and to resolve them. How can we move such quality gates earlier in the project lifecycle? Can we automate them in order to shorten the feedback cycle? In order to do this, the cornerstone of any modern software delivery lifecycle is the Continuous Integration / Continuous Deployment (CI/CD) pipeline, using tools such as Azure Pipelines, Jenkins, or GitHub Actions. A CI/CD pipeline is generally triggered when a developer pushes code from their local branch to a shared version control repository such as Git.

> *Every project is different, so the specific metrics you evaluate against can vary.*
>
> Stephen Masters
> Principal Consultant with Digiterre

Within a pipeline we can use a tool such as SonarQube to evaluate code against a set of quality metrics such as:

- **Maintainability:** Is the code readable? Does the change introduce "code smells" - features of the code which indicate deeper issues?
- **Security:** Does the change introduce any new vulnerabilities?
- **Reliability:** Is the automated test coverage sufficient? Are all tests passing?

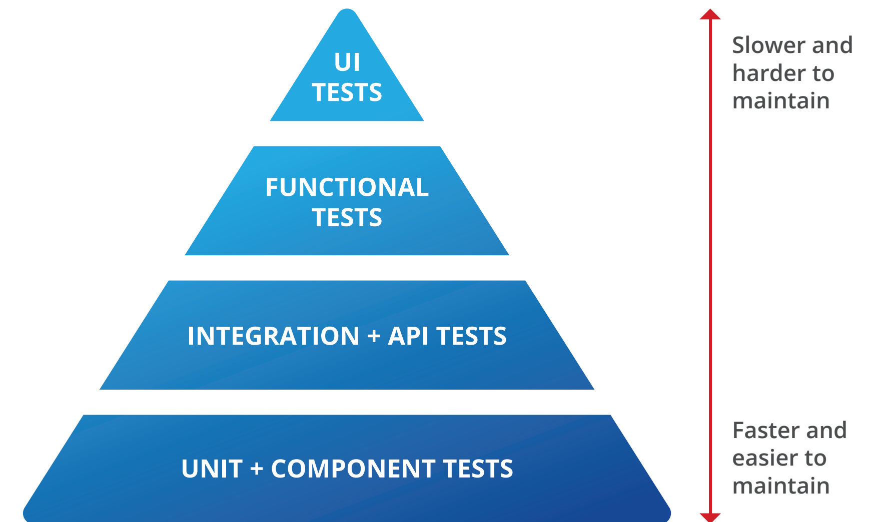
Every project is different, so the specific metrics you evaluate against can vary. If you're developing a business critical system then you might want to enforce more code coverage. If the system is public facing and connects to personal data, then it will need more stringent security checks. These gates should be agreed early, and when being introduced to an existing codebase, they should focus on changes, rather than the overall product. That way we ensure that new changes are good quality, but we're not rejecting good changes because they don't fix all existing issues.

AUTOMATE TESTS

In addition to testing early, we also want to be able to test often. By automating tests, we can run them fast, and integrate them into the CI pipeline so that they are executed every time a change is committed to the version control repository. Ideally, the majority of our tests should be runnable in a developer's local environment for even earlier feedback. They should also be able to run fast, so that a developer can see quickly whether they have broken something.

A popular concept in the world of testing is the **test automation pyramid**, which guides us to produce a larger number of fast tests that are easy to develop and maintain, whilst developing smaller quantities of tests that are slower and harder to maintain.

As tests cover more integrated components they tend to be slower to create and to run. Therefore we should aim to produce a lot of unit tests, which are written directly against functions in the code, are fast to execute and easy to debug. At the other end of the spectrum, it is possible to create end-to-end UI tests, which can simulate activities that a person might perform on a website by interactions with a web browser. These can be valuable, but they can also take a lot of time to execute, require a full running environment, take a long time to run, and can be very fragile to changes. Tools such as Cypress.io help a lot, but they still take a significant amount of time to develop, and benefit a great deal from the skills of test automation specialists. Between these extremes, should be varying amounts of more integrated tests where we test APIs and interactions between components.



As we are aiming to produce a reduced number of slower tests that take more effort to maintain, we need to prioritise tests that add most value. Therefore we need to be considering factors such as :

- Which parts of the system are most likely to experience bugs?
- Which parts of the system have the biggest impact if they fail?
- What are the key end-to-end user journeys through the application?

With all this automation, there is still a place for manual testing. However this should be limited to more exploratory testing, which doesn't need to be carried out for every commit.

AUTOMATE SECURITY CHECKS

Recently, a critical vulnerability was identified in the popular open source logging library, Log4J. Unfortunately, in many companies, teams were unable to say whether their applications used a vulnerable version. There is also the increasing threat from supply chain attacks, where bad actors deliberately modify a dependency to give themselves a backdoor into systems which use it.

A tool such as Anchore Syft can be used to generate a Software Bill of Materials (SBOM) containing a list of dependencies within a container, which is incredibly valuable when a vulnerability is discovered. If stored with build artefacts in a repository such as Sonatype Nexus or Artifactory, this can be a powerful way to identify all vulnerable systems across an organisation and when particular vulnerabilities were introduced.

Tools such as Grype compare an SBOM with a list of known vulnerabilities. Another popular tool is Snyk, which has particular strength identifying vulnerabilities in open source libraries. Any time the CI pipeline runs, we can identify whether a new vulnerability has been identified or if the change has introduced a vulnerable dependency, and the release can be prevented.

> *By identifying these issues early, we reduce the risk of attacks in production, and we avoid needing to go through a full development lifecycle to fix the vulnerabilities when they are discovered.*
>
> Stephen Masters
> Principal Consultant with Digiterre

AUTOMATE DEPLOYMENTS

The final stage of a pipeline is Continuous Deployment, where we establish confidence in our ability to deploy the changes to test and production environments in a fast and repeatable way, without errors.
This is where the use of Infrastructure as Code (IaC) tools such as Kubernetes, Terraform, Azure Resource Manager Templates, and AWS CloudFormation come into play. The initial effort in setting up IaC can be daunting at first, but it pays back that investment many times. It ensures that we are able to deploy small changes frequently to environments where we can run API and UI tests. IaC also makes it possible for a tester to spin up their own environment for them to perform exploratory testing without impacting anyone else. If the deployment to the test environment has been successful, and the tests against that environment pass, then we can deploy to production with confidence.

FINALLY...CONTINUE IMPROVING

There is no perfect process that works for all teams and all systems, but I think it is safe to say that all teams would like to be able to deliver higher quality software at greater velocity. To achieve that, start gathering metrics to give you visibility and transparency into the current state of your projects, and introduce some of the techniques and tools that I have discussed. The tools I have mentioned are not intended to be an exhaustive list. There are more that our teams at Digiterre make use of, and new tools appear all the time. So gather some metrics, evaluate which are the most important for you, shift-left, automate, and keep iterating and improving.

WHITE PAPER

ABOUT THE AUTHOR

Stephen Masters is a Principal Consultant with Digiterre. He has over 25 years' experience as a solutions architect and lead developer, specialising in Big Data Cloud Technology, Web Applications, Systems Integration and Service Oriented Architecture. Stephen's career has focused on the banking and energy trading sectors, where he has been instrumental in the successful delivery of a broad range of projects; from developing foreign exchange pricing systems, compliance rules engines, to optimising power stations.

FIND OUT MORE ABOUT DIGITERRE AND HOW WE DELIVER QUALITY OUTCOMES FOR OUR CLIENTS