OverOps

Whitepaper

The 4 Quality Gates That'll Stop You From Promoting Bad Code

To help you and your team score each deployment and make sure it's ready to be promoted, we present you with our 4 quality gates that must be passed in order to have a certified and approved release.

Every QA, DevOps and SRE team face two main challenges when promoting code to production: **Coverage** and **Anomaly Detection.**

Our coverage is never 100%, and that means that our testing will never cover our entire environment. In complex systems, it's tough to foresee all the different code and infra combinations, especially with the release of a new feature.

In addition to the coverage issue, large-scale systems, even in test environments, have a high noise to signal ratio. It makes detecting anomalies almost impossible, and we're left to rely on our users and customers to alert us when something goes wrong.

These two challenges impact the entire industry, and there's a need to craft an effective approach; one that lies in being able to converge **Machine Learning** and **AI** to automatically gate bad code from moving up the chain. That's why we need an objective measure to quantify the quality of a release, and to set the gates a release must pass before moving from staging into production.

To help you identify whether your code is ready for a promotion, here are the four quality gates to use to make sure our code is certified to go to the next step:

Gate #1 - Error Volume

The purpose of this gate is to understand whether this release increases the number of errors, compared to its predecessor, or not. The challenge here is that we need to be able to capture this data correctly and incorporate both errors that are logged, as well as those that aren't such as HTTP errors, swallowed and uncaught exceptions, as those might actually be more important than those that are in the logs.

Once we capture the data, we need to **normalize** it. Volume only makes sense when compared with **throughput**. You'll undoubtedly have a higher volume when more throughput is pumped into the system, so normalizing the error volume into a percentage is critical. The next thing we need is the ability to **duplicate** the data, so we can easily see what makes the bulk of the volume, and whether or not it's benign or severe.

Gate: The normalized error rate of an application should never increase between releases

Gate #2 - Unique Error Count

Here we want to have the ability to classify which errors comprise the volume and if that **count has gone up** or down since our last release. We want to transform multiple, separate events into a **time series** that we can chart and split into core components. Once we transform this mass of code and log events into a set of analytics, we can see where these errors are coming from - which apps, containers, locations in the code, and under which conditions.

This will allow us to evaluate the **quality of the code**, the **performance cost** associated with having those events in the code, in addition to their impact on the reliability of the app. This becomes more important when looking at key applications or reusable components (i.e. **tiers**) in the code such as payment processing or DAL, when more errors can be a very negative indicator.

Gate: The number of unique error counts, especially in key applications or code tiers, should not increase between releases.

Gate #3 - New Errors

Once we've broken down all the errors in our environments into components, we want to be able to quickly separate from hundreds, or sometimes even thousands of locations in the code, the ones that are new and have just been **introduced by** this release.

This is critical for both a basic application or an enterprise application with a 15 year legacy code, that has existing errors; In both cases, we shouldn't be introducing new errors into the environment. However, when dealing with a massive release or major infrastructure change, you might encounter a large number of these "micro fractures", in the form of a dozen new errors.

That’s why it’s important to be able to **prioritize** events, to make sure that even when new errors are introduced into the environment, they aren’t **severe** ones.

Gate: New errors of a **critical type** or occurring at a high rate should block a release.

Gate #4 - Regressions & Slowdowns

Here we’re using the data obtained from “Error Volume” and “Unique Error Count” gates, to look at the behavior of existing errors within the system. In this case, we’re looking for regressions in the form of errors that already existed but are happening at a **higher rate**, or **slowdowns** that deviate from previous performance.

This is more complex, as we need to look at errors in a **relative** way. For something to be considered regressed or slow, it needs to be compared against itself. For this, a **baseline** must be established before Machine Learning can be applied. The second ingredient needed is **tolerance** - what is the level of regression or slowdown we are willing to tolerate?

We want to be able to say that an increase of more than 50% is said to be a regression, and an increase of more than 100% is said to be a **severe regression/slowdown**, in which case a release should not be promoted - at least not without inspection of the anomaly.

This is the most complex gate, but also the one that has the strongest **predictive quality** with respect to potential outages and severity 1 incidents.

Gate: Severe regressions and slowdowns should block a release

Final Thoughts

With these four gates, we wanted to define a benchmark that is powerful and broadly applicable to complex environments, but also isn't complex to the point that it remains an academic exercise. They will help you understand whether your code is ready to be deployed to your pre-production or production environment, and help reduce or even eliminate errors, issues, slowdowns and anomalies within your application.