OverOps

Whitepaper

Expect the Unexpected: How OverOps Detects Sev1 Issues Before They Hit Production

Sev1 issues in production cost companies money. In order to prevent them from ever happening, we need to be able to expect the unexpected using Anomaly Detection and Quality Gates.

We frequently hear questions such as: how do we separate signal from noise, which errors should we fix, how do we know if a release is good to go? These are all important questions to ask and, more important, to know how to answer.

The answer to all of these questions is based in Anomaly Detection. With the amount of noise that applications produce every minute, finding meaningful information or signals is a challenge. And without any distinction between noise and signal, understanding which errors to prioritize or if a release is ready for production is nearly impossible.

To help you focus on critical issues in any new release, here are three anomalies to watch for and how OverOps can help prioritize issues that are most likely to cause Sev1 incidents.

3 Anomalies to Watch For in Your Releases

Every application has errors. Thinking that we can eliminate 100% of errors in our applications sounds like a pipe dream, and it is. Fortunately, maintaining a high-quality application isn't about preventing or handling every single error that appears, it's about understanding how changes we make to the system change the volumes and rates of errors and how those changes affect performance.

Anomalous events such as unusual changes in error volume or performance slowdowns is a much better metric of application quality. In the end, the quality of a release or deployment should be based on (the absence of) 3 types of anomalies:

Anomaly #1 - New Errors:

Errors that were introduced by the most recent deployment or within a specific time window being observed (e.g. last 24 hours).

Anomaly #2 - Increasing Errors:

Errors that existed in previous deployments or before the time window that is being observed, but their rate has increased significantly not proportionally to throughput. This is determined by comparing the error rate to its dynamic baseline calculated by OverOps' Machine Learning Platform.

These errors can also be referred to as regressed errors, not to be confused with *resurfaced errors*, which is a term used to describe an error which the user marked as resolved (meaning they do not expect it to happen again), and is now happening again.

In a production environment, this more often has to do with something wrong with the environment setup / provisioning (i.e. *operational error* handled by the SRE/DevOps) versus in QA environments that are much more stable, in which case the reason will most likely be *programmatic* (and handled by the developer).

Anomaly #3 - Slowdowns:

When the code slows down during a specific time window or deployment. Again, if this happens in a lower environment it is most likely a programmatic issue (caused by the code, handled by devs) and in higher environments it is most likely operational (caused by env setup / provisioning, handled by DevOps/SRE).

How OverOps Prioritizes Anomalies to Prevent Sev1 Issues

For each one of these anomalies, OverOps determines its **severity level** (p1 or p2). The scoring of a release or overall application is done by deducing those scores and averaging that over time.

Prioritizing New Errors:

A new error may be considered a ***severe new issue*** by the OverOps Platform if any of the following are true:

- It is uncaught.
- Its type is defined in the critical exception types list in the setting page.
- Its volume AND rate exceed the defined thresholds (50 times and 10% of calls is the default)

Example -

Let's say a developer is writing code to process a new type of insurance claim. The claim report is a very complex data structure, with many inputs and sub-fields. The developer assumes that a specific field is mandatory, when in fact it is optional. As they are testing their code on a limited data set during unit testing (where all the mock data they are running the new code on have that field populated) everything is fine.

When the code goes into a staging environment, a broader data set of claims is passed into the code, and one of the test claims does not contain this field. This will make the code fail with a NullPointerException (i.e. attempt to access an empty field).

OverOps will detect this as a **new error**, and since its type (NullPointer) is defined as critical within the OverOps settings screen it will be marked as ***severe (p1)***.

Prioritizing Increasing Errors:

Increasing errors are prioritized according to the percentage increase that was observed. By default, an increase of more than 50% compared to the baseline is categorized as a ***p2-level issue***, and an increase of 100% or more is categorized as a ***p1-level issue***.

These thresholds are adjustable to each team's individual prioritization needs. Seasonal variance is also taken into account if, for example, a similar rate or volume was previously observed in the baseline.

Example -

In another situation, let's say a new server is spun up, and the queue configuration should have been set to 2mb per message, but it was not configured correctly causing the queue to run with a 512kb default message size. From that point, almost every claim that is inserted into the queue fails, which means the error that used to happen rarely (<1%) is now happening at a rate of 100% on that box.

If there are 10 servers, then this error is now happening 10% of the time in that env overall, where before it used to happen <1%. OverOps will mark this error as an **increasing error** - as its something that used to happen already but is now happening at a higher rate.

Since the error has more than doubled in rate (it’s happening 10X more than it did previously), it is marked as ***severe (p1).***

Prioritizing Slowdowns:

Like increasing errors, or regressions, slowdowns are identified by measuring transaction times against their own baseline.

OverOps automatically identifies all code entry points within an application and continuously tracks their response time. If the time difference between a call and its baseline is greater than a specific threshold, that call is considered slow. When the percentage of slow calls exceeds a critical percentage (by default, 60%), that is considered a ***severe (p1)*** slowdown.

Example -

Imagine one last scenario. A change is made in a piece of code that's used to calculate whether or not a database entry can get pulled from memory cache. Because of the change, the code no longer matches requests to entries in the cache. This causes a 65% increase in (unnecessary) calls into the physical underlying database which takes more time.

The code still works functionally-speaking, from the standpoint of providing the right data, but because in some cases the cache is no longer utilized, some of the requests are severely slow. OverOps automatically identifies this as a **slowdown** and marks it as ***severe (p1)***.

Using Quality Gates to Prevent Sev1 Issues from Reaching Production

With proper identification and classification of anomalies in code behavior and performance, we can preemptively block releases from being deployed to production if circumstances suggest that a Sev1 issue is likely to be introduced.

Based on the anomaly types and prioritization levels described above, here are the four quality gates to use to make sure your code is certified before being promoted to the next step:

- **Gate #1 - Error Volume:** Blocks release promotion if the normalized error rate of an application increased between releases.
- **Gate #2 - Unique Error Count:** Blocks promotion if the unique error count, especially in key applications or code tiers, increased between releases.
- **Gate #3 - New Errors:** Blocks promotion if new errors appear of a critical type or with a high frequency.
- **Gate #4 - Regressions & Slowdowns:** Blocks promotion if there are severe increases in any of the previous categories, or if there are severe performance slowdowns.

To learn more about using Quality Gates to prevent bad code from being deployed to production, read this free whitepaper or visit our website.