The New Way of Handling Java Errors Before They Hit Production

How we realized the old way of solving errors in pre-production is not enough, and how we were able to change that

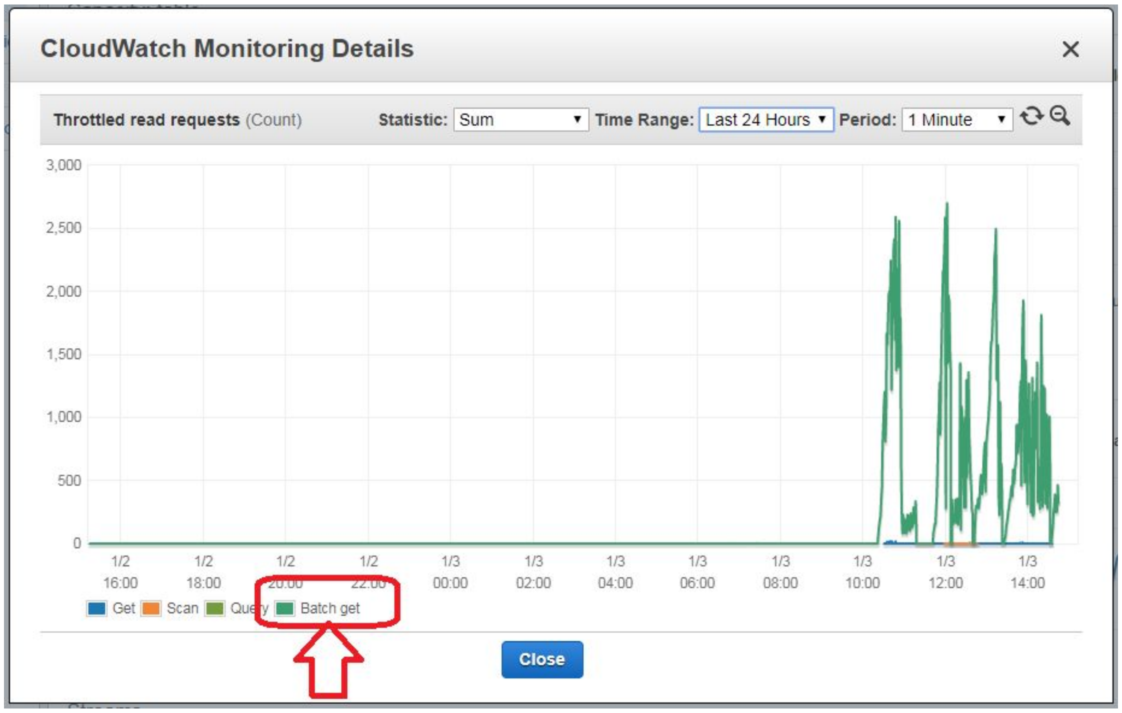
There's no such thing as perfect code on the first try, and we can all testify that we've learned that the hard way. It doesn't matter how many test cycles, code reviews or tools we use, there's always at least one sneaky bug that manages to surprise us.

In the following post, we'll share our own story of such a bug, the common workflow that developers use to solve it compared to the new way we do it at OverOps. Spoiler alert: log files don't cut it, and now it's time to understand why.

Act I: Detecting there's an issue

A couple of weeks ago our AWS server started sending out distress signals. Our QA team came across it during one of their tests, and complained that the server couldn't handle their load testing. They turned to our VP R&D seeking help. The VP R&D pulled out his credit card, increased the server load and told QA that the issue was fixed.

As you can guess, the problem was far from over. After the second cycle of complaint -> increasing AWS payment -> complaint, we realized that there's more to this issue and it needs further investigation. We had to figure out what went wrong, and our first step was to open the Amazon CloudWatch Monitoring Details, which gave us the following chart:



Throttled read requests on our server. Yikes.

As you can see, the “batch-get” request started to go haywire right after 10:00 AM, consuming an ever-growing amount of resources and failing operations, effectively causing an outage of the staging environment on which QA were running load tests.

While the data Amazon provides can tell us which type of operation is causing the issue, it can’t tell us where it is coming from and why is it happening in the first place. It was time to roll up the sleeves and start digging inside the code. Or at least, that’s what most teams would do at this point.

Act II: The old method of analyzing the issue

After verifying through Amazon that something is indeed not working as it should, the first traditional step is to search through the code and logs to see where this certain action is being called from. However, the chances of finding only one stack trace is similar to the chance you have of winning the lottery.

Usually, we find ourselves with numerous places in the code that might be relevant to a certain exception. We now need to map out the places that might be the cause of our main issue, and add some logging statements that will (hopefully) give us a clear picture of what’s going on. After laying out the traps, we deploy the code to staging and cross our fingers hoping that the error will happen again.

Assuming the issue has recurred, now it’s time to start sifting through the logs trying to understand what happened. This whole process repeats itself a few times, and it could take a while to pinpoint the area or method that are causing this error. This could take a few hours at best, but we all know that some errors bug us for days until we can detect and fix them.

While this might be a common workflow for some dev and ops teams, it’s not ideal (to say the least). It has too many downsides, especially if it happens in production, and it can hurt users and customers who will complain and might even leave your application. To put it in other words, it affects the reliability of your application, as well as the productivity of developers who have to waste days on solving issues.

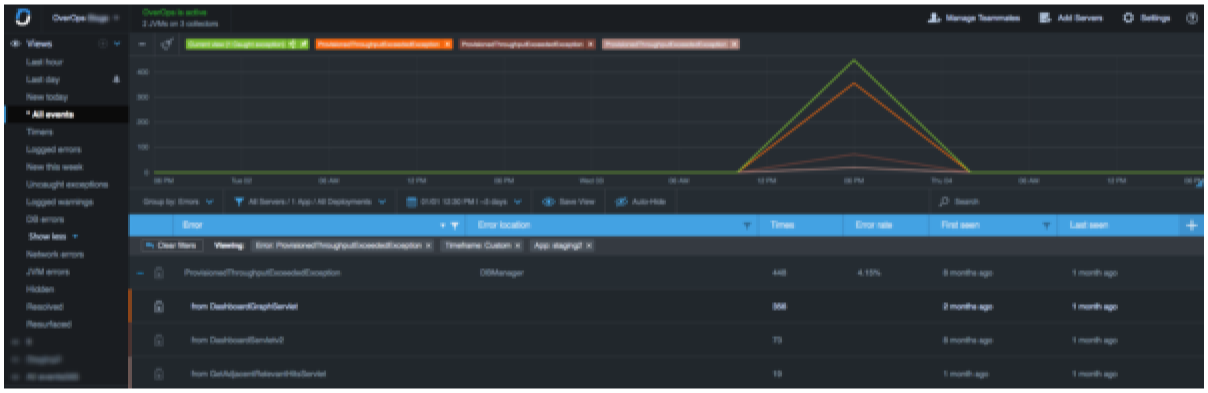
Steve Rogers, Software Development Director at Viator, a TripAdvisor company told us that their old troubleshooting process would take days, leading to very noisy and costly logs. It made their process of identifying, and investigating errors a real challenge for the team.

At this point TripAdvisor were thinking what we were thinking:

Act III: There must be a better way

Let's circle back to our personal issue. The staging servers were suffocating with a batch get call and we had to understand what's going on. Since we at OverOps want to make sure our developers are productive and are not wasting days on troubleshooting errors like the one found by our QA team, we use OverOps. We practice what we preach.

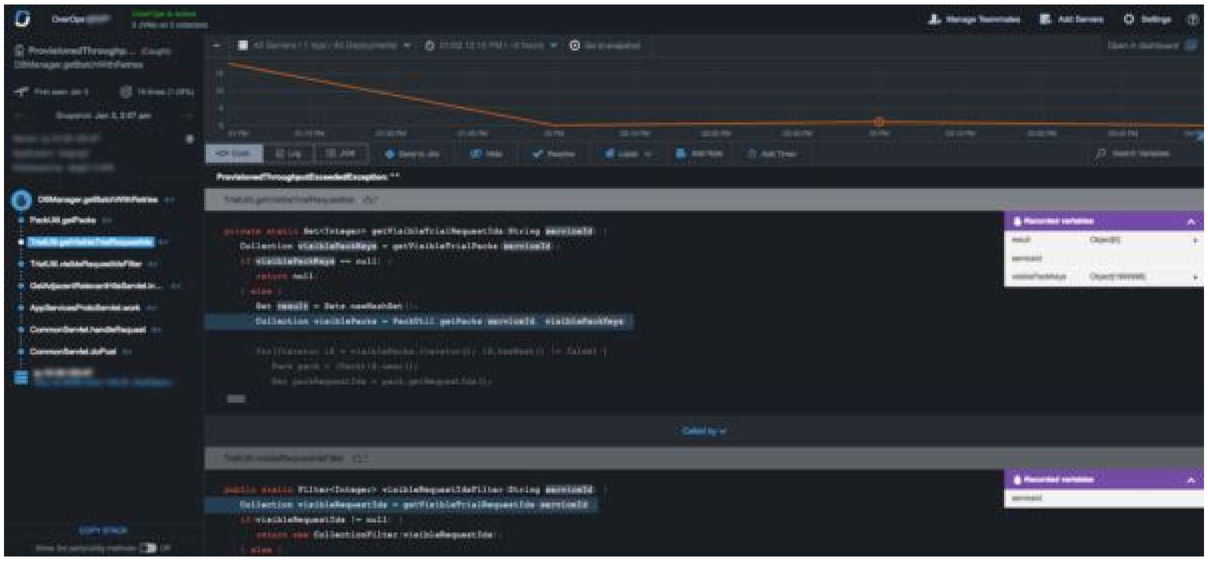
Getting throttled requests means that we'll be able to see a spike in a specific error, ProvisionedThroughputExceededException, and that's exactly what we were looking for in our dashboard:



Error	Error location	Times	Error rate	First seen	Last seen
ProvisionedThroughputExceededException	DBManager	448	4.15%	8 months ago	1 month ago
from DashboardGraphServlet		356		2 months ago	1 month ago
from DashboardServletv2		73		8 months ago	1 month ago
from GetAdjacentRelevantHitsServlet		19		1 month ago	1 month ago

These spikes mean that something is acting up on our server.

After detecting the error, we can click it and reach its Automated Root Cause (ARC) analysis that includes its complete source code and variable state, across the entire call stack:



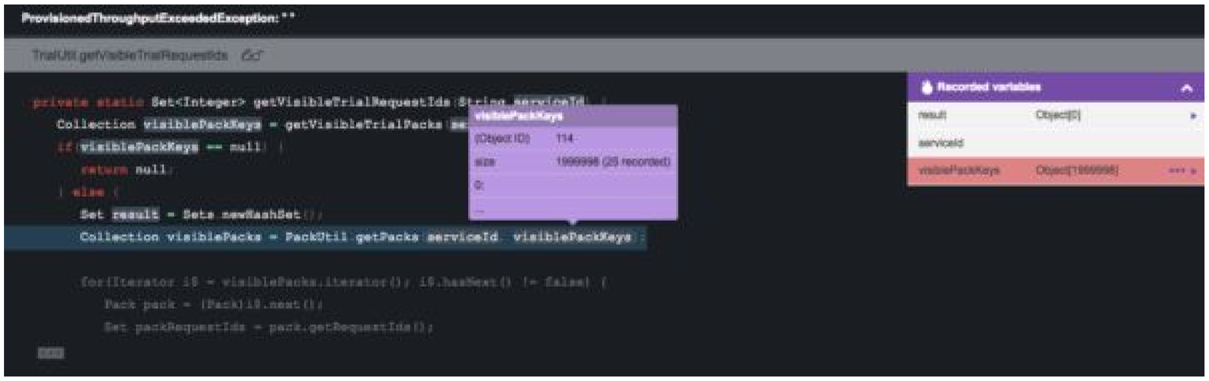
```
private static Set<Integer> getVisibleTrialRequestIds(String serviceId) {
  Collection visiblePackKeys = getVisibleTrialPacks(serviceId);
  if (visiblePackKeys == null) {
    return null;
  } else {
    Set result = Sets.newHashSet();
    Collection visiblePacks = PackUtil.getPacks(serviceId, visiblePackKeys);

    for (Iterator it = visiblePacks.iterator(); it.hasNext() != false;) {
      Pack pack = (Pack) it.next();
      Set packRequestIds = pack.getRequestIds();
```

```
public static Filter<Integer> visibleRequestIdsFilter(String serviceId) {
  Collection visibleRequestIds = getVisibleTrialRequestIds(serviceId);
  if (visibleRequestIds != null) {
    return new CollectionFilter(visibleRequestIds);
  } else {
```

Can you detect the error?

In the screenshot we can see a drill down into the exception, focusing on the method in which the error has occurred. The ARC analysis code view tab shows the code (middle of the screen), that corresponds with every method in the exception's stack trace (visualized on the left side of the screen), and the variable state at the moment that this exception was thrown. It also shows the severity of the error by displaying the number of times that it occured (top left), the error rate, and its occurrences over time.

It took our Chief Architect one look at the dashboard to realize what happened. As you can see on the right side of the dashboard, the visiblePackKeys variable is holding close to 2 million objects. This is also visible when hovering over the variable itself, as you can see in the following screenshot:



ProvisionedThroughputExceededException: ""

TrialUtil.getVisibleTrialRequestIds

```
private static Set<Integer> getVisibleTrialRequestIds(String serviceId) {
  Collection visiblePackKeys = getVisibleTrialPacks(se
  if(visiblePackKeys == null) {
    return null;
  } else {
    Set result = Sets.newHashSet();
    Collection visiblePacks = PackUtil.getPacks(serviceId, visiblePackKeys);

    for(Iterator i$ = visiblePacks.iterator(); i$.hasNext() != false) {
      Pack pack = (Pack)i$.next();
      Set packRequestIds = pack.getRequestIds();
```

visiblePackKeys

(Object ID)	114
size	1999998 (25 recorded)
0:	
...	

Recorded variables

result	Object[0]
serviceId	
visiblePackKeys	Object[1999998]

Almost 2 million events. Wowza.

This means that this function is trying to read almost 2 million events. Now, the database is doing as it's told, so that's not the issue here. The next step is realizing why the code is calling 2 million events in the first place, and why can't it handle the load. By looking at the dashboard we have enough information to reproduce the issue and fix it, without wasting resources for hours or days.

You're probably wondering what was the root cause of this issue. OverOps shows users all the exceptions and errors that are happening inside their application. Trial users can see a certain number of events on their dashboard, and our QA director was testing to see how many events the user can see.

Although the user should be able to see as many events they want, the error here was that the database in this particular instance didn't have 2 million events. The tested scenario and the database didn't match – and that was the root cause of the error.

It was immediately evident once we saw that variable state, we understood that the load testing process should have a filtering system (and now it does), changed the code logic to be more efficient and deployed it. This whole process took us less than an hour from detection, through investigation and down to deploying a fix.

This new way of handling errors is now getting widely adopted. Sumit Nagal, Principal Engineer in Quality at Intuit told us that they use OverOps across multiple environments, including QA, pre-production and staging. With OverOps, the team can detect an error before it impacts the user, which improves their application's reliability, and helps the company provide an outstanding user experience. Learn more on how Intuit automates root cause analysis at scale.

Final thoughts

We at OverOps want to show that there's a better way of handling application errors, in pre-production and production. OverOps help teams improve developer productivity, and accelerate the delivery rate of new releases, without compromising on application reliability.