The Complete Guide to

Automated Root Cause Analysis

By Tali Soroker

Table of Contents

Introduction

The Importance of Automated Root Cause Analysis in Testing and Production

We spend our nights here at OverOps dreaming of a fully-automated world. It might seem a bit weird, but can you imagine how much happier you and your team would be minus those sleepless nights and endless days sifting through log files trying to figure out what went wrong?

Those mundane manual tasks are slowing us down as we aim to build innovative technologies. So, what if they were eliminated from your day-to-day life altogether? Imagine automated alerting and an empty exceptions inbox. It's all possible, and we're sharing all of the best practices for automating your root cause analysis right here.

Handling exceptions isn't just about fixing a problem, it's about understanding the inner workings of your application. With automated root cause analysis, time spent identifying and solving production errors within Java applications is cut down from days to just minutes.

Chapter 1

Reverse Engineering the Ideal Monitoring Setup for Your Production Environment

A decision maker's guide to the tooling ecosystem

Over the last couple of years we've had the opportunity to talk with hundreds, if not thousands, of engineering teams. Looking back at those discussions and at our own engineering team's pains, a few similarities started to surface.

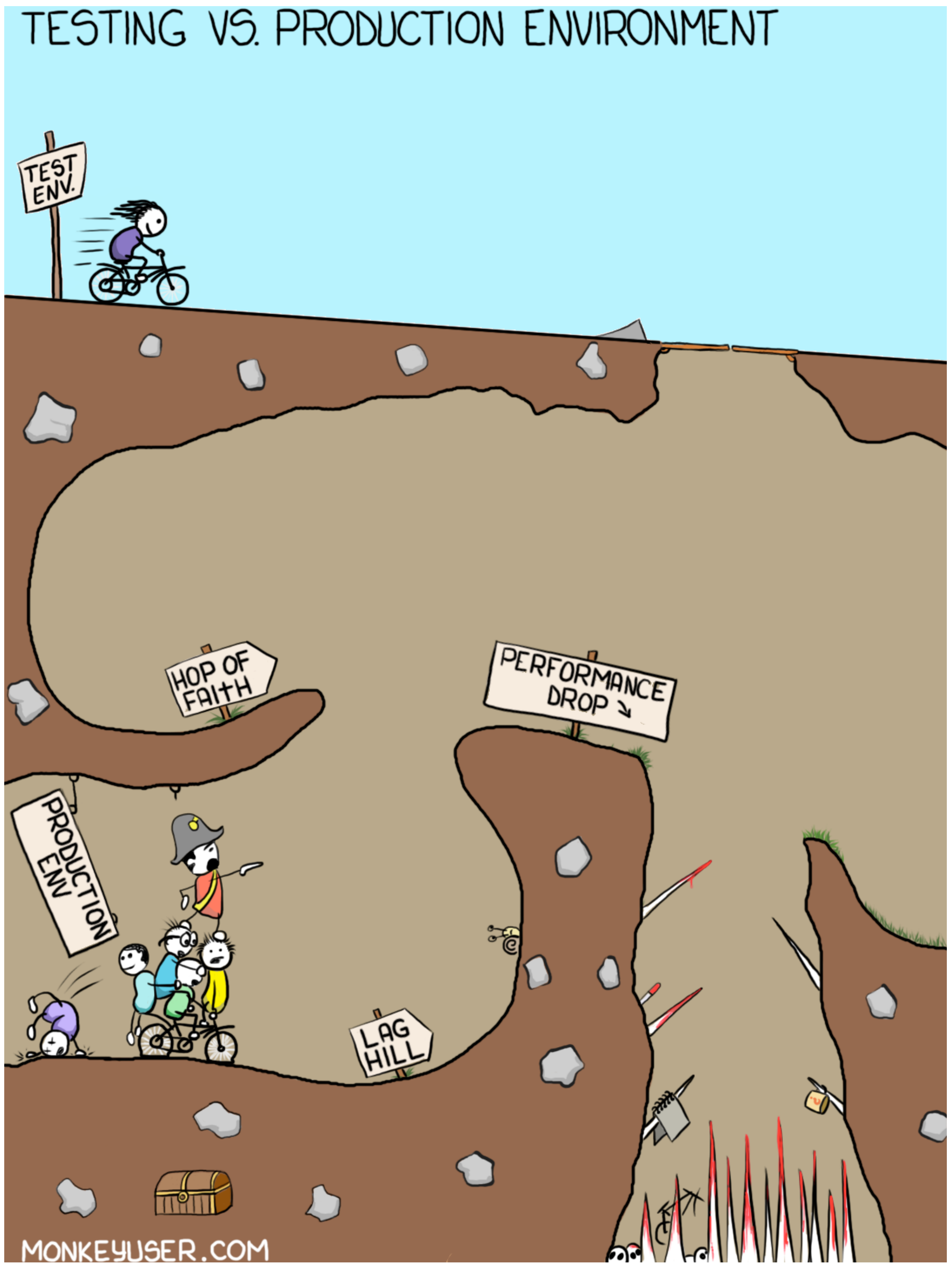
It all comes down to this: Monitoring is a tough job, and a big part of it comes down to the tools and processes you build around it. First, you have to know what you want to monitor, and this will correlate to one of 3 main tooling categories. Next, we'll talk about choosing the right tools for building an automated framework for your production monitoring stack.

Plan for Observability

First, let's talk about what we're trying to achieve with monitoring, Observability.

Testing and pre-production environments are great for making sure your application is ready to be deployed, but they never cover 100% of the errors that your user will experience. Plus, you can't rely on users to report every issue they encounter. And when they do report it, you can't rely on their report to reproduce the error. That's where the importance of planning for Observability comes in.

This comic strip from monkeyuser.com illustrates the concept pretty well.



In your testing environment, the horizon is clear and you're covering the obstacles you're planning ahead for, but in the underground environment of production, things can get pretty dark. The obstacles are different and you can't see that far ahead.

It's amazing how even today, the question…

"How do you know a new deployment introduced errors into your system?"

…is usually answered by saying:

"Well, we deploy our code, and then, we wait. If a day or two pass by and no one starts screaming about something, we know that we're in the clear."

The good news is that it doesn't have to be like this. Enter Observability.

What is Observability?

The term Observability comes from the field of control theory. An interdisciplinary branch of engineering and computational mathematics. It's a measure for how well internal states of a system can be inferred by knowledge of its external outputs.

Adapted to operating complex applications in production, we can think of it as a measure for how well we understand what's going on under the hood of an application once it leaves the safe and warm dev or testing environment and real data is thrown at it at scale. And a lot does happen there. This is a topic that we frequently explore and covered extensively in another eBook which you can read right here.

The bottom line is that we need to have excellent visibility into the inner workings of our application in order to solve any problems as soon as they arise. Creating an automated monitoring system is the best way to architect for Observability which allows you to identify the issues with the highest priorities so that you can tackle them without wasting any time.

Basically, Observability is how you know WHEN, WHERE, and most importantly WHY code breaks in production. It's easy in development where you're basically Neo after he figures out how to manipulate the matrix. Everything is under control, and you have debugger to step through the code. In production, the situation is different.

The Symptoms of Not Planning for Observability

Creating an automated monitoring system for your production environment is really about being proactive, rather than reactive. Without it, understanding the ins and outs of your application requires manual user reports and other, more time consuming, processes like debugging with log files.

And that kind of sucks. It's actually one of the top 10 causes for unhappiness in developers, which we'll talk about more in Chapter 4.

Let's see what we can do to gain better Observability for our applications.

Reverse Engineering Root Cause Detection

We talked with engineering teams in companies like Fox, Comcast, Intuit, Zynga and others, and took a closer look into what they're doing to maximize the Observability of their applications in production.

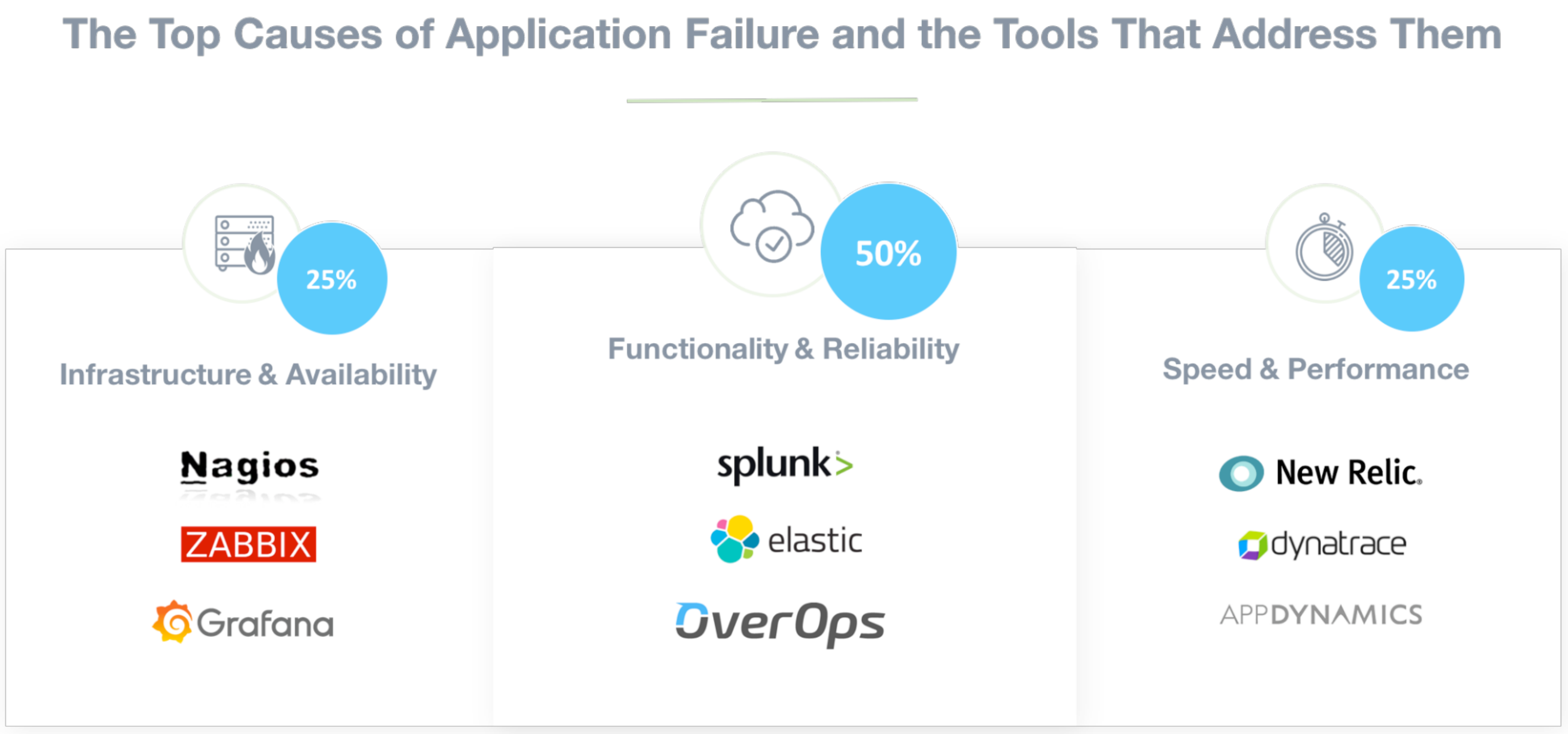
What they found is that their tooling and processes didn't give them a good enough view into the heart of the application. They were often late to detect critical issues. There was just too much noise out there and when a new or critical error was introduced into staging or production, it wasn't detected amidst thousands, if not millions, of other noisy events.

Frequently, as much as 75% of the time or more, it is the end user or customer that first detects the problem when the application or service is no longer acting reliably or is no longer available. These teams are drowning in a sea of noise that, more often than not, is missing the actual root cause of the errors that they're investigating.

Rather than accepting the current state as it is, we tried to find the ideal monitoring stack by reverse engineering it from the top sources for application failure. To do so, we asked engineering teams what their top sources of failure were.

It turns out there's actually a pretty definitive answer to that question.

We found out that when production applications experience failure of any kind, the root causes behind these failures are derived from 3 possible sources:


The Top Causes of Application Failure and the Tools That Address Them
25%
Infrastructure & Availability
Nagios
ZABBIX
Grafana
50%
Functionality & Reliability
splunk>
elastic
OverOps
25%
Speed & Performance
New Relic.
dynatrace
APPDYNAMICS

1. First, there is a whole set of application failure problems that are due to **infrastructure issues**. And developer teams are seeing that this bucket is responsible for approximately 25% of their application problems.

2. Then, there is a set of application failures that are due to **speed and performance**. And if they would have pulled their tickets and examined them, they would see that approximately 25% of application failures are due to this particular problem area.

3. And lastly, the area of **functionality and reliability** or, in other words, "bugs." This group is responsible for the largest amount of failures, approximately 50% of them. It is in this third bucket, the one that is responsible for the largest amount of application failures, that OverOps plays in.

We feel that this space is ripe for disruption and we have a technology that can radically improve the standard approach today. By automating the process around detection and resolution, OverOps' technology perfectly complements engineering efforts in resolving speed and performance issues as well. A personal walkthrough of the technology is available right here.

Core Use Cases for Popular Monitoring Stacks

To wrap up the introduction of this approach for building a monitoring stack, let's briefly cover some examples and use cases in each of the buckets we just introduced. While these tools have many features, we see that they're mainly used for specific problems.

Infrastructure & Availability – 25% of Production Errors

This category includes tools to monitor the infrastructure and availability of your servers. Think of things like network health checks, response time, and uptime. Things like CPU, disk space, and memory utilization also fit in here.

Speed & Performance – 25% of Production Errors

Application Performance Monitoring (APM) tools fit in this bucket. We also published an in-depth eBook that covers the most popular ones, like AppDynamics, New Relic and Dynatrace, together with open-source alternatives. You can check it out right here.

The core use case for APMs is identifying performance bottlenecks, identifying transactions that take longer than usual to execute, and showing which methods within them are to blame.

Functionality & Reliability – 50% of Production Errors

In this bucket we have log management tools, and this is also where we chose to focus most of the efforts with OverOps.

Logs have been around for so long without much real change and innovation. We have log management tools that help make them searchable, but they're only as good as the information that was printed to them in the first place. If something is missing, you need to add additional logging statements, deploy the code, and hope that the situation you're trying to remedy will actually happen again! That's a bit paradoxical.

In discussions we had, RJ Lim, a Senior Software Engineer at Zynga, told us "some problems are very hard to debug even if you have logs, and on some occasions they might not be debuggable at all. We needed something that could help us debug in production."

And Dmitry Erman, Executive Director, Development and Architecture at Fox, told us: "As opposed to having developers search through logs or attach debuggers to the production or pre-production environment, OverOps gives us the exact conditions behind each error. We can see why it happened and if it's critical for us."

That's basically the new thing OverOps offers in a nutshell, that's what makes it unique. It enhances and complements your existing log analyzers and performance management tools, by automating root cause detection. Basically showing you the variable state behind all of your errors in staging and production. OverOps gives your team the complete source code and variable state needed to fix errors 90% faster.

Click here to watch a live demo.

Final Thoughts

Building a monitoring strategy is a tough job to get done right. Yet, we believe that automating your monitoring stack makes troubleshooting in production much simpler. Get better Observability into your application and solve any problems faster than before.

Chapter 2

The 7 Undeniable Benefits of Implementing Automated Alerting

What's the ultimate alerting strategy to make sure your alerts are meaningful and not just noise?

Production monitoring is critical for your application's success, we know that now. But how can you be sure that the right information is getting to the right people? Automating the monitoring process can only be effective when actionable information gets to the person it needs to. The answer is automated alerting. However, there are some elements and guidelines that can help us get the most out of our monitoring techniques, no matter what they are.

In order to help you develop a better workflow, we've identified the top benefits that your alerts can offer you. Let's check them out.

1. Timeliness – Know as soon as something bad happens

Our applications and servers are always running and working, and there's a lot going on at any given moment. That's why it's important to stay on top of new errors when they're first introduced into the system.

Even if you're a big fan of sifting through log files, they only give you a retroactive perspective of what happened to the application, servers or users. Some would say that timing is everything, and getting alerts in real time is critical for your business. We want to fix issues before they severely impact users or our application.

This is where 3rd party tools and integrations are valuable, notifying us the minute something happens. This concept might not sound as nice when an alert goes off at 03:00 AM or during your night out, but you still can't deny its importance.

TL;DR – When it comes to production environment every second counts, and you want to know the minute an error is introduced.

2. Context is key to understanding issues

Knowing when an error has occurred is important, and the next step is understanding where is it happening. Aleksey Vorona, Senior Java Developer at xMatters, told us that for his company, context is the most important ingredient when it comes to alerts; "Once an error is introduced into the application, you want to have as much information as possible so you can understand it. This context could be the machine on which the application was running on, user IDs and the developer that owns the error. The more information you have, the easier it is to understand the issue".

Context is everything. And when it comes to alerts, it's about the different values and elements that will help you understand exactly what happened. For example, it would benefit you to know if a new deployment introduced new errors, or to get alerts when the number of logged errors or uncaught exceptions exceeds a certain threshold. You'll also want to know whether a certain error is new or recurring, and what made it appear or reappear.

Breaking it down further, there are 5 critical values we want to see in each error:

• *What* error was introduced into the system

• *Where* it happened within the code

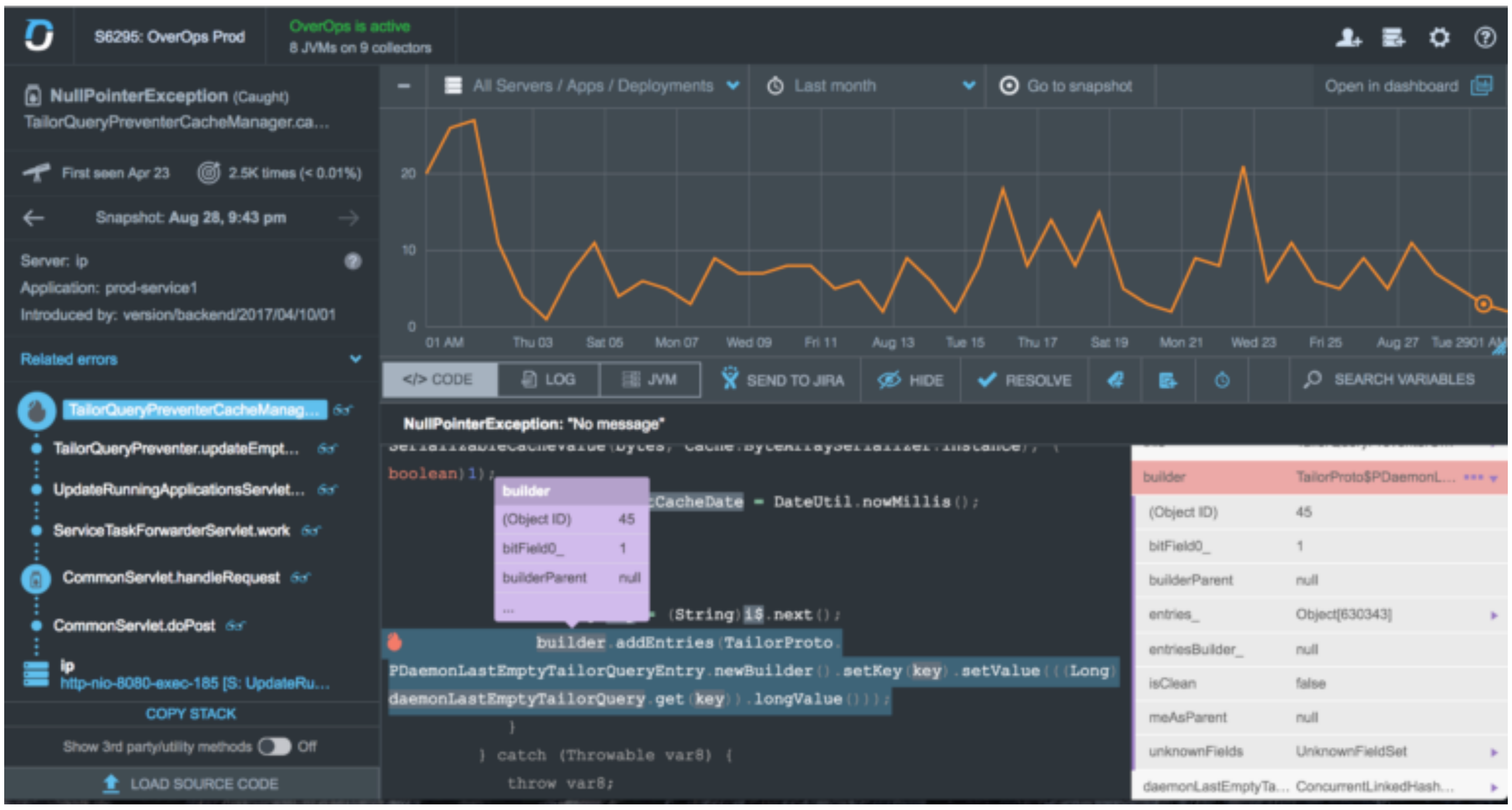
- *How many* time each error happened and what is its urgency
- *When* was the first time this error was seen
- *When* was the last time this error occurred

These were some of the issues we had to face ourselves here at OverOps, trying to help developers, managers and DevOps teams automate their manual error handling processes. Since each team has its own unique way of handling issues, we created a customizable dashboard in which you can quickly see the top 5 values for each error. OverOps allows you to identify critical errors quickly, understand where they happened within the code and know if they're critical or not.

TL;DR – You need to know when, where, what and how many errors and exceptions happen to understand their importance and urgency.

3. Root cause detection – Why did it happen in the first place?

Now that we're getting automated alerts with the right context, it's time to understand why they happened in the first place. For most engineering teams, this is the time to hit the log files and start searching for that needle in the haystack. That is, if the error was logged in the first place. However, we see that the top performing teams have a different way of doing things.


S6295: OverOps Prod
OverOps is active
8 JVMs on 9 collectors

NullPointerException (Caught)
TailorQueryPreventerCacheManager.ca...
First seen Apr 23 | 2.5K times (< 0.01%)
Snapshot: Aug 28, 9:43 pm
Server: ip
Application: prod-service1
Introduced by: version/backend/2017/04/10/01

Related errors
- TailorQueryPreventerCacheManag...
- TailorQueryPreventer.updateEmpt...
- UpdateRunningApplicationsServlet...
- ServiceTaskForwarderServlet.work
- CommonServlet.handleRequest
- CommonServlet.doPost
- ip
  http-nio-8080-exec-185 [S: UpdateRu...

NullPointerException: "No message"

```
boolean)1);
                    CacheDate = DateUtil.nowMillis();

                      (String)i$.next();
                builder.addEntries(TailorProto.
PDaemonLastEmptyTailorQueryEntry.newBuilder().setKey(key).setValue(((Long)
daemonLastEmptyTailorQuery.get(key)).longValue()));
            }
        } catch (Throwable var8) {
            throw var8;
```

builder	
(Object ID)	45
bitField0_	1
builderParent	null
...	

builder	TailorProto$PDaemonL...
(Object ID)	45
bitField0_	1
builderParent	null
entries_	Object[630343]
entriesBuilder_	null
isClean	false
meAsParent	null
unknownFields	UnknownFieldSet
daemonLastEmptyTa...	ConcurrentLinkedHash...

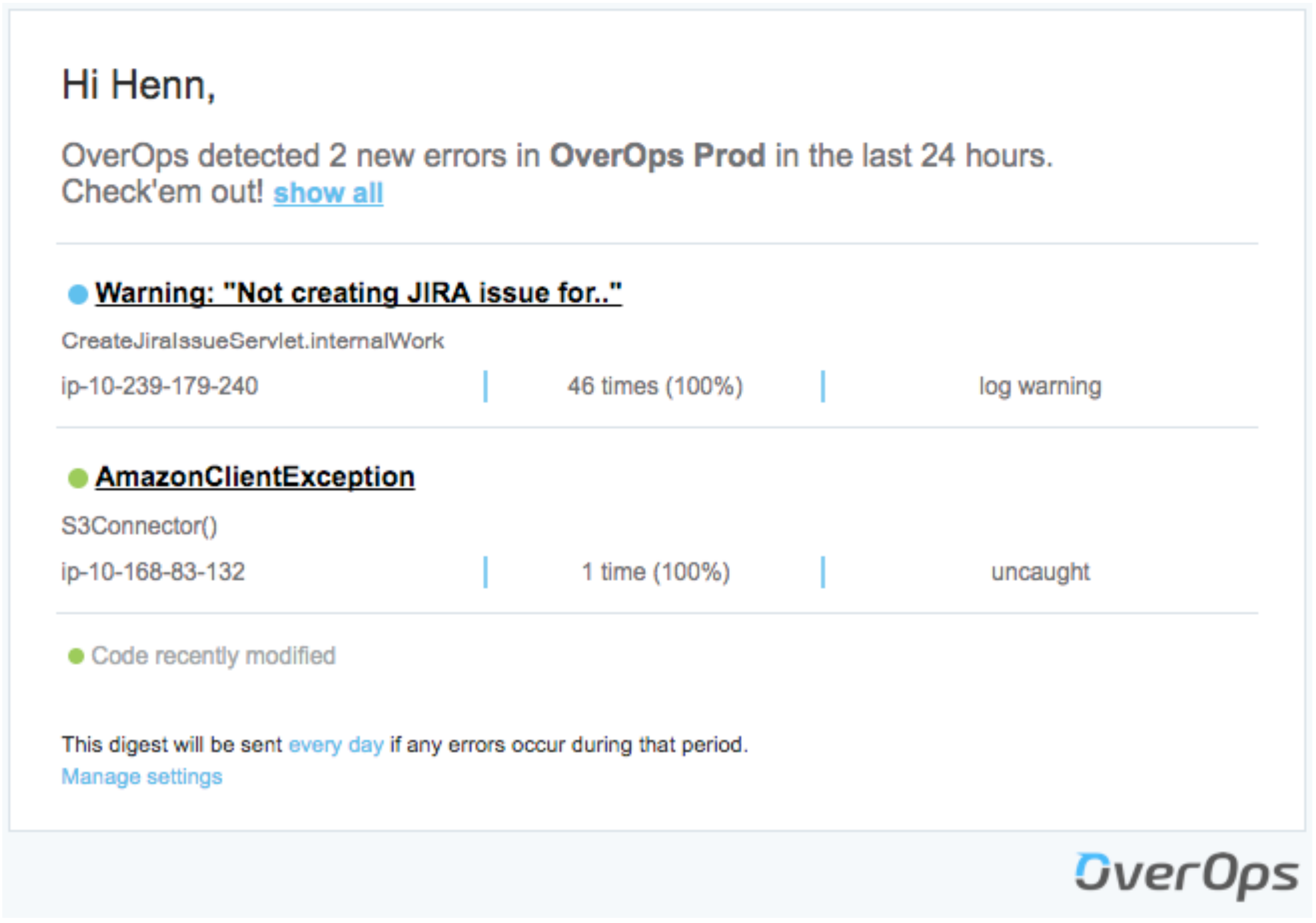
Usually, applications fire hundreds of thousands or even millions of errors each day, and it becomes a real challenge to get down to their real root in a scalable manner without wasting days on finding it. For large companies such as Intuit, searching through the logs wasn't helpful; Sumit Nagal, Principal Engineer in Quality by Intuit points out that "Even if we did find the issues within the logs, some of them were not reproducible. Finding, reproducing and solving issues within these areas is a real challenge."

Instead of sifting through logs trying to find critical issues and closing tickets with a label stating "could not reproduce", Intuit chose to use OverOps. With OverOps, the development team is able to immediately identify the cause of each exception, along with the variables that caused it. The company is able to improve the development team's productivity significantly by giving them the root cause with just a single click.

TL;DR – Getting to the root cause, along with the full source code and variables, will help you understand why errors happened in the first place.

4. Communication – Keeping the team synced

You can't handle alerts without having everyone on the development team on board. That's why communication is a key aspect when it comes to alerts. First of all, it's important to assign the alert to the right person. The team should all be on the same page, knowing what each one of them is responsible for and who's working on which element of the application.



Hi Henn,

OverOps detected 2 new errors in **OverOps Prod** in the last 24 hours.
Check'em out! show all

● **Warning: "Not creating JIRA issue for.."**
CreateJiraIssueServlet.internalWork

ip-10-239-179-240	46 times (100%)	log warning

● **AmazonClientException**
S3Connector()

ip-10-168-83-132	1 time (100%)	uncaught

● Code recently modified

This digest will be sent every day if any errors occur during that period.
Manage settings

Some teams might think that this process is not as important as it should, and they assign different team members to handle alerts only after they “go off”. However, that’s bad practice and it isn’t as effective as some would hope.

Imagine the following scenario: it’s a Saturday night and the application crashes. Alerts are being sent to various people across the company and some team members are trying to help. However, they didn’t handle that part of the application or the code. You now have 7 team members trying to talk to each other, trying to understand what needs to be done in order to solve it.

This is caused due to lack of communication in earlier parts of the project, leading to team members to not being aware of who’s in charge, what was deployed or how to handle incidents when alerts are sent out.

TL;DR – Communication is important, and you should work on making it better as part of your error handling process.

5. Accountability – Making sure the right person is handling the alert

Continuing our theme of communication from the previous paragraph, an important part of this concept is knowing that the alert reaches the right person, and that he or she is taking care of it. We might know which team member was the last one to handle the code before it broke, but is he the one responsible for fixing it right now?

On our interview him, Aleksey Vorona pointed out that it’s important for him to know who’s the person in charge of every alert or issue that arises. The person who wrote the code may be more likely to handle it better than other members of the team, or there may be another team member be equipped to resolve it.

The bottom line is that by automating your alerts, you can direct exception handling tasks directly to the team member that is responsible for them. Otherwise, the right people might miss important information and accountability goes out the window. It’s problems like these that can lead to unhappy users, performance issues or even a complete crash of servers and systems.

TL;DR – Team members should be alerted to issues that they are responsible for maintaining, so that it’s always clear who’s accountable for which tasks.

6. Processing – Alerting handling cycle

You have your team members communicating and working together, which is great. However, you still need to create a game plan that the team will aspire to achieve. A good example of a game plan is having an informed exception handling strategy rather than treating each event in isolation.

Exceptions are one of the core elements of a production environment, and they usually indicate a warning signal that requires attention. When exceptions are misused, they may lead to performance issues, hurting the application and its users without your knowledge.

How do you prevent it from happening? One way is to implement a “game plan” of an Inbox Zero policy in the company. It’s a process in which unique exceptions are acknowledged, taken care of and eventually eliminated as soon as they’re introduced.

We’ve researched how companies handle their exceptions, and found out that some have the tendency to push them off to a “later” date, just like emails. We found that companies that implement an inbox zero policy have better understanding of how their application works, clearer log files and developers focused on important and new projects. We’ll cover this more in the next chapter.

TL;DR – Find the right game plans for you and implement them as part of a better alerting handling process.

7. Integrations? Yes please

Handling alerts on your own might work, but it’s not scalable in the long run. For companies such as Comcast, servicing over 23 million X1 XFINITY devices, it’s almost impossible to know which alerts are critical and should be handled ASAP. This is where 3rd party tools and integrations will be your best friends.

After integrating OverOps with their automated deployment model, Comcast was able to instrument their application servers. The company deploys a new version of their application on a weekly basis, and OverOps helps them identify the unknown error conditions that Comcast didn’t foresee. Watch John McCann, Executive Director of Product Engineering at Comcast Cable explain how OverOps helps the company automate their deployments.

Integrations can also be helpful in your current alerting workflow. For example, Aleksey Vorona from xMatters works on developing a unified platform for IT alerting, and developed an integration with OverOps. The integration allows the company to get access to critical information, such as the variable state that caused each error and alert the right team member.

TL;DR – Use third party tools and integration to supercharge your alerts and make them meaningful.

Final thoughts

Alerts are important, but there's much more to it than just adding them to your application. You want to make sure you have information on why they happened in the first place, how you should handle them and how can you make the most out of them (vs. just knowing that something bad happened). Automated alerting is an important part of the monitoring system. We need the right people to know when, where and why things go wrong in production so that they can fix it as soon as possible.

Chapter 3

Getting to Inbox Zero With Automated Exception Handling

How and why should you apply an inbox zero policy when it comes to your exceptions

Inbox zero is a concept that has been around for a while, and one that tries to help you keep a clear email inbox and a focused mind. Imagine if you could take this concept, and apply it to your exception handling process.

Automating the debugging process boosts efficiency allowing teams to implement an inbox zero policy for their exception handling. With Automated Root Cause detection, teams are able to see the root cause behind every error and exception and then resolve them so that they aren't hanging around. In this way, development teams also gain a better understanding of their application by seeing meaningful insight into every exception that's thrown.

An inbox for exceptions

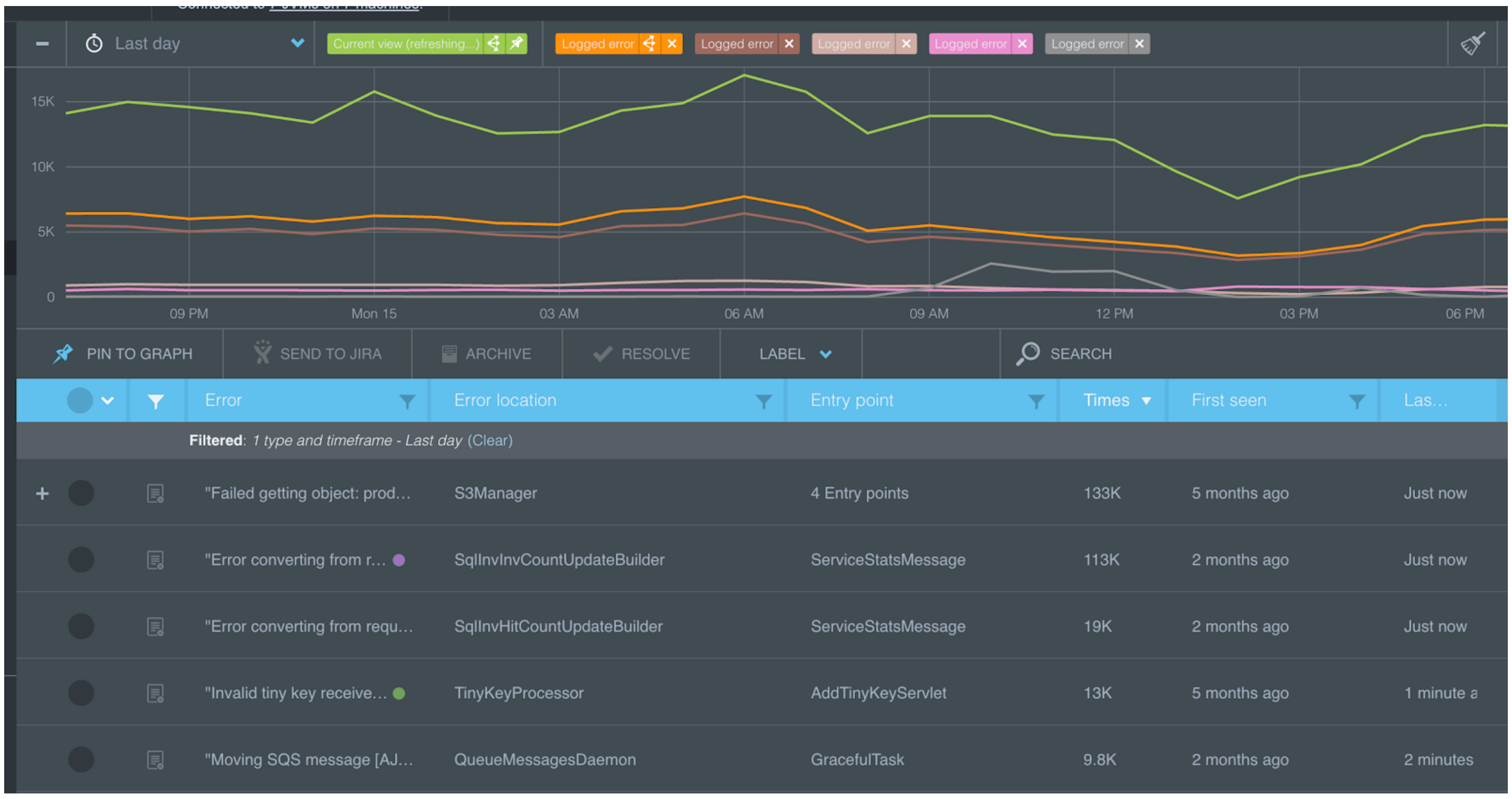
A production environment is a lot like a living organism, and that's where the real-time action is happening. There's a lot going on behind the scenes, and we need to be aware of everything that might damage this delicate operation.

One of the core elements of a production environment is exceptions. No matter if they're new or recurring; they indicate a warning signal that requires attention. In Joshua Bloch's book "Effective Java" he says that "When used to best advantage, exceptions can improve a program's readability, reliability, and maintainability. When used improperly, they can have the opposite effect."

In addition to that, when exceptions are misused, their sheer number may cause performance issues and leave you unaware of critical issues.

That's why we need a better way of managing exceptions, and this is where the concept of an exception inbox shines. We heard about this developing practice from engineering teams we spoke to, and think that it should be the golden standard for more teams to strive to achieve.

An exception inbox is an actual inbox that holds all of the events that happened in a certain time frame, with the aggregate information about each unique event. It allows us to go through different issues and manage them as needed. On OverOps, it looks like this:



Filtered: *1 type and timeframe - Last day* (Clear)

Error	Error location	Entry point	Times	First seen	Las…
"Failed getting object: prod…	S3Manager	4 Entry points	133K	5 months ago	Just now
"Error converting from r…	SqlInvInvCountUpdateBuilder	ServiceStatsMessage	113K	2 months ago	Just now
"Error converting from requ…	SqlInvHitCountUpdateBuilder	ServiceStatsMessage	19K	2 months ago	Just now
"Invalid tiny key receive…	TinyKeyProcessor	AddTinyKeyServlet	13K	5 months ago	1 minute a
"Moving SQS message [AJ…	QueueMessagesDaemon	GracefulTask	9.8K	2 months ago	2 minutes

Exception Inbox Zero: A similar concept for a similar issue

The Inbox Zero concept was developed by the productivity expert Merlin Mann, and it centers on handling every email as an immediate task, until there are no emails left in the inbox folder. Emails, regardless of what they hold, correspond to a to do list, of sorts, since they require our attention.

Each email should be translated to an action: acknowledge and archive, delete, reply, or defer for a later date when you can take care of it. Yet, sometimes we choose to push aside the "known" and "familiar" email, to be taken care of at a later unknown date. Or in other words, never.

Our exception inbox will be where we collect, view and handle exceptions, and to that we'll apply the inbox zero concept. Just like emails, we have the tendency to push them to a "later" date, but we need to acknowledge each exception, take care of it and eventually eliminate it. Otherwise, we'll have to face major downsides, some we might not even be aware of:

The top 5 disadvantages of not handling exceptions

Disadvantage #1: Experiencing unnecessary overhead

Every application's boogieman is inefficient use of memory and CPU resources, and it makes sense. It can be downright destructive. Whenever an exception is thrown, it creates an object and sends data to the log. It's not much when it comes to a single occurrence of an exception, but what happens if it fails millions of times? Before we know it, the heap gets clogged, the CPU is wasting precious cycles on meaningless tasks, and we can kiss performance goodbye.

Aleksey Shipilëv, Principal Software Engineer at RedHat, who has been working on Java performance for more than 10 years, has an optimistic rule-of-thumb. According to him, the frequency of common exceptions should be 10^-4, which is a single occurrence of an exception for 10,000 calls into the method it's thrown from. An error rate of 0.0001%. Anything beyond that deserves your attention.

What should we do? Keep an eye out for overhead. If an exception happens more often than 0.01% of the times the method it's in has been called (at the very least), its root cause should be taken care of. Poof.

Disadvantage #2: Not understanding how the application really works

The best way to figure out how a toy works is by taking it apart. Same for applications. We need to increase our application's Observability (From Chapter 1). In control theory, Observability is a measure for how well internal states of a system can be inferred by knowledge of its external outputs.

Exceptions are the external outputs that can give us knowledge regarding the state of the application. They give us an exclusive look backstage, which could be especially handy when debugging someone else's code or working on legacy code.

What should we do? Promote Observability. Exceptions are a big part of this process, and you need to use the proper tools and techniques in order to understand what's going on within the application.

Disadvantage #3: Filling your logs with noisy events

When an error happens, everyone's go-to-solution is usually to look at the log. There, they hope to find the information needed to understand why the error happened and if there's an exception that could help shed more light on it.

The thing is, logs usually contain massive amounts of information that often has no real use. If something breaks unexpectedly you'll have to sift through the logs, and it feels like looking for a needle in a haystack.

What should we do? Well, as their name suggests, exceptions should only be used for exceptional scenarios that need to be dealt with as soon as possible. It could be a ticket, a hotfix or even just acknowledgment of the issue – but it should get the right attention.

Disadvantage #4: Causing developers to be unhappy (more on this later - Chapter 4)

According to a study focused on "The Unhappiness of Software Developers", the number one reason leading to unhappiness is being stuck on an issue. Unfortunately, it's one of the most common tasks we face throughout the day, usually when debugging issues.

Being stuck on issues and debugging means that there's no time to work on features or to write new code. We can end up feeling like we're playing Whac-A-Mole with exceptions, trying to eliminate them before they pop-up again.

What should we do? Apply better exception handling techniques. It will require a deep dive into your existing exceptions and taking care of each one, a required step towards applying an Inbox Zero technique to managing your exceptions.

Disadvantage #5: Inability to focus on what actually matters

Exceptions cloud the developer's view. As exception numbers increase, it's harder to know which exceptions are more important than others. This could lead to missing a major issue or dismissing an exception that requires immediate attention.

What should we do? Unexplained performance issues that have been haunting your application could be easily solved when you have your exceptions under control. It's like the cholesterol of your application. And you should keep their level under control.

Next step: Run a check up for your application

You should be aware of when an exception happens more often than 0.01% of the times the method they're in is called. But how can you tell what's your exception rate? Our search for the answer to that question is part of what lead us to create OverOps.

The tool gives you a complete overview of your exceptions, pinpointing the root cause for each error that occurred in production. With one click, you can see the complete source code and the variable state that caused them, across the entire call stack.

With OverOps you'll be able to see how many times each exception happened, and get the detailed error rate of how many times it failed. In less than 5 minutes you can start counting (and solving!) your exceptions. Get a personal walkthrough of the Exception Inbox Zero concept and start calculating your exception rate within minutes.

Final thoughts

The official Java documentation states that "An exception is an event that occurs during the execution of a program that DISRUPTS the normal flow of instructions". But more often than not, exceptions are abused and used as part of the normal application flow, and that's less than ideal to say the least.

Just as the name implies, exceptions should be exceptional. Each one should be actionable and result in code changes, in order to eliminate the exception altogether. Reaching a zero-exception environment will lead to a better performing application and a cutting edge user experience, and Automated Root Cause detection can help get you there.

Chapter 4

Study: The Top 10 Causes for Unhappiness Among Developers

A new study lays out the main reasons that lead to unhappy software developers. But why is it so important?

Are you happy as a software developer? If that question sounds weird or out of place, it shouldn't be. It's a pressing issue for employers, HR teams and companies in general. A happy developer is often a productive developer, and keeping developers happy should be a top priority for companies who produce software.

A new paper titled "On the Unhappiness of Software Developers" aims to solve that challenge. The researchers behind it wanted to see how to keep developers happy, and how to filter out the main reasons for their "unhappiness" in the workplace.

TL;DR; Developers are pretty happy, but...

A research team recently wanted to see what the main causes are that could lead to unhappiness among software developers. The researchers were able to isolate over 200 causes of unhappiness and divide them into categories. The result? Somewhat of a guideline that could help make your teams happier.

Why should developers be happy?

A fun and happy work environment is something most companies try to offer. It could be perks like… slides that connect different floors, jam sessions during lunch or even unlimited vacation days. There's always that little extra something, to keep employees happy.

It's no surprise that companies spend time, energy and money to make sure developers are happy whenever they're at work. But why is it so important?

The idea that happy developers are better at their job is not new, and it's not surprising. It makes sense that if we're happy, having fun and continue learning we are more productive in our work.

It's not only a theory. In the study "Happy Developers Solve Problems Faster", the researchers were able to prove this assumption.

The study, conducted by Daniel Graziotin, Xiaofeng Wang and Pekka Abrahamsson, sets out to see how productivity and software quality could improve. It focused on the developers themselves, providing incentives to make them satisfied and happy with their work.

The results showed that the happiest developers were able to solve an analytical problem better than their "unhappy" co-workers. The conclusion also pointed out that keeping developers happy could reduce job burnout, anxiety and remove negative experience in the workplace.

Now that we know the "why", it's time to get to the most important part – the "how".

Why are developers unhappy?

If you're looking for a clear answer, we have some news for you. According to the current study, "On the Unhappiness of Software Developers", there are many causes for unhappiness. 219 causes, to be exact. On the bright side, the researchers were able to group them into 2 main categories:

- Developer’s own being (i.e., internal causes) – Focusing on personal state and actions originating from the developers own behaviors
- External causes – How developers are affected by something they have no control of

Yet, these 2 main categories are still too broad for us to understand the exact causes of unhappiness. That’s why the researchers dissected them into 18 subcategories, so it will be easier to pinpoint the cause that might lead to unhappiness. Then, they extracted the top 10 causes of unhappiness among software developers.

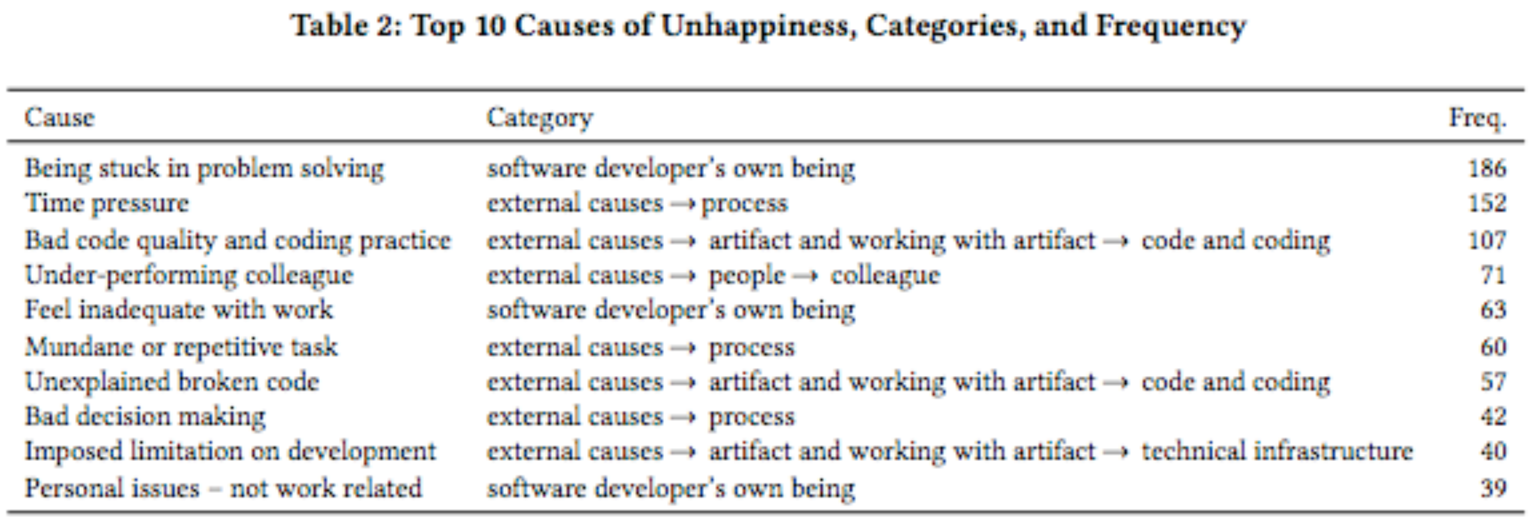


Table 2: Top 10 Causes of Unhappiness, Categories, and Frequency

Cause	Category	Freq.
Being stuck in problem solving	software developer’s own being	186
Time pressure	external causes → process	152
Bad code quality and coding practice	external causes → artifact and working with artifact → code and coding	107
Under-performing colleague	external causes → people → colleague	71
Feel inadequate with work	software developer’s own being	63
Mundane or repetitive task	external causes → process	60
Unexplained broken code	external causes → artifact and working with artifact → code and coding	57
Bad decision making	external causes → process	42
Imposed limitation on development	external causes → artifact and working with artifact → technical infrastructure	40
Personal issues – not work related	software developer’s own being	39

Before diving into the causes themselves, this chart shows us that out of the top 10 causes that make developers unhappy, 7 are external.

Among the top causes of unhappiness, we can see issues we’ve all faced: tight deadlines, bad code quality (written by other members of the team), underperforming colleagues, repetitive tasks and so on.

On the bright side, this means that in most cases, we can affect the unhappiness of a developer. It also means that we can be aware of certain issues that might lead to unhappiness, and react. We can help turn that frown upside down.

But there’s also some not-so-happy-news. The chart holds 3 internal causes for unhappiness. The most significant cause that makes developers unhappy is being stuck on a problem. Unfortunately, it’s a common issue that’s hard to avoid.

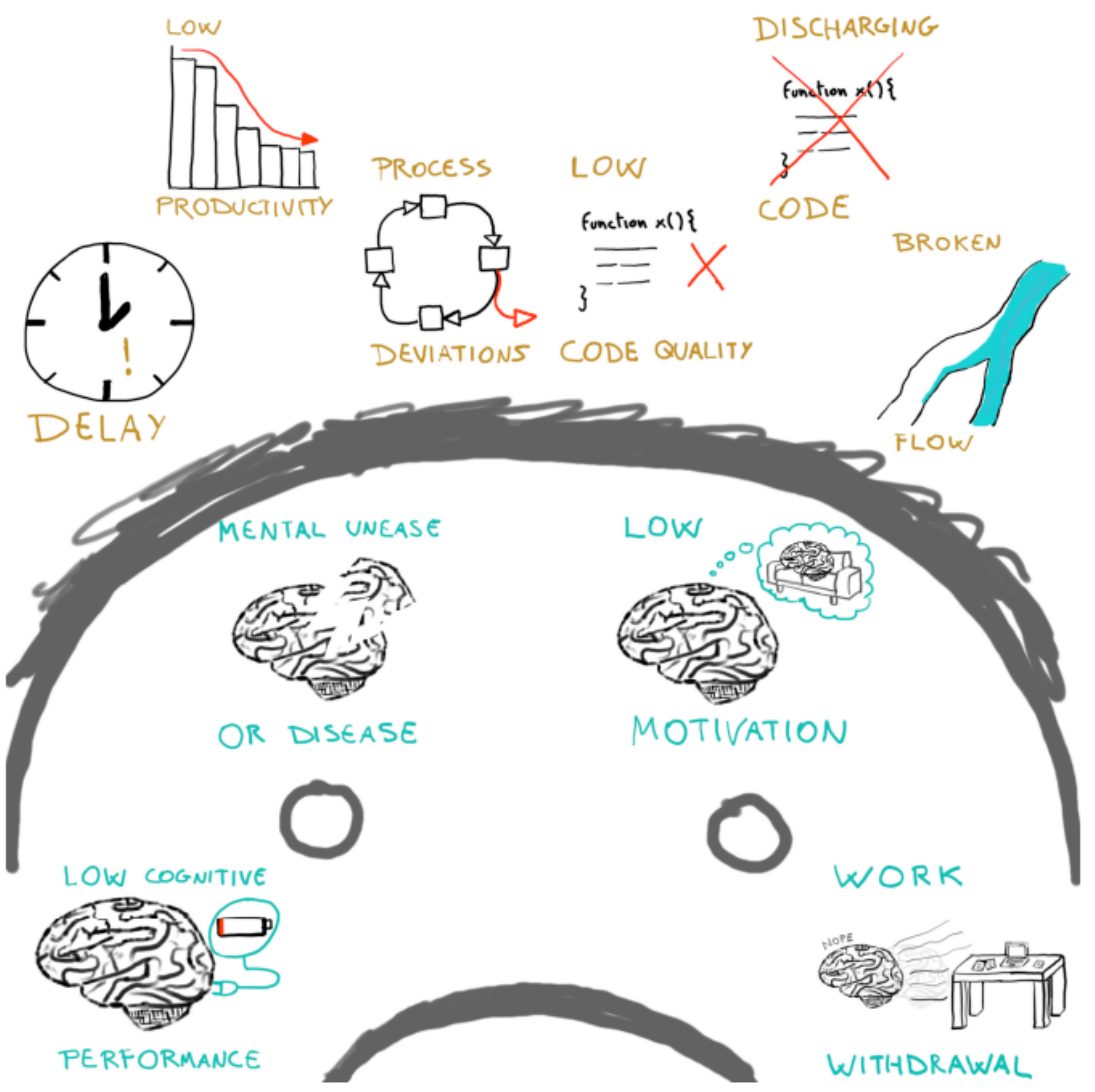
Another internal cause that appears in the chart, is feeling inadequate with work. This feeling could arise when a developer is unfamiliar with his/her work environment. This could include tools, languages, frameworks, or development methods used by the team.

The last internal reason has nothing to do with work, but it still affects it – personal issues. According to the researchers, in most cases personal issues are family related.

The bottom of the list is a bit surprising if you ask us. Causes such as misuse of software by a customer, being too technical or git conflicts lead to the least amount of unhappiness among developers.

If you're interested in the full list of 219 causes of unhappiness, you can view it here.

Helping developers stay happy



The number one reason that leads to unhappiness among developers is being stuck on an issue. Unfortunately, it's one of the most common tasks any developer faces throughout the day. Automating the debugging process with full root cause analysis reduce time stuck on troubleshooting.

Since one of the most time consuming tasks developers face is debugging, and being stuck on bug related problems 20-40% of your time on average is no fun at all, we thought there must be a better way to tackle these errors. That's where OverOps comes in.

When it comes to debugging in production, most of the wasted time is spent on trying to reproduce the variable state that caused the error. Sifting through logs is just plain annoying. Most of the time, they're super noisy and the information you're looking for isn't even there.

With OverOps, you can see the exact variable state behind any exception, logged error or warning, without relying on the information that was actually logged. It lets you immediately see the complete source code and variable state across the entire call stack of the event. Even if it wasn't printed to the log file.

How can we deal with unhappiness?

There's no doubt that the unhappiness of developers is an important issue that should be handled. However, the causes detailed in this research could seem a bit broad, and each developer has a different take on the various causes. That's why we've decided to ask other developers how they feel about this research, and the causes lined up in it.

Eugen Paraschiv, an engineer, architect and the person behind Baeldung found the first cause surprising. While being "stuck on a problem" is familiar to any developer and is a cause for temporary frustration, he wouldn't label it as being "unhappy". Baeldung's opinion is that when the problem is under our control, working on it is generally rewarding, even if it's sometimes frustrating to be stuck.

Oleg Shelajev, a product engineer and developer advocate at ZeroTurnaround, pointed out that most of the causes lined out in this research have the same base concept: They all relate to being unproductive, or having to sacrifice seemingly achievable quality of the project they work on for some reasons.

This could be the result of bad planning and time pressure, inadequate team skills, already rotting code base, or not having access to modern tools that make life easier, etc. In other words, when developers know they can be better but something prevents them from achieving it, they feel unhappy.

Shelajev adds that some of the things that make us unhappy are hard to solve: issues with the management, the team, the planning, making everyone invest in code quality. Still, the fact is that some of them could be solved with the proper tooling: a proper IDE, a debugger you know how to use, an APM solution, a powerful dedicated Jenkins cluster, a crash monitoring solution, sane logs, JRebel, thought through libraries and frameworks or any other combination that works for you.

Peter Cooper, a software developer, podcaster, blogger, author and more believes that the causes for developer unhappiness are not significantly different to those for people doing other types of job. While poor management and poor setting of expectations and requirements seem to be behind most unhappiness, those things will make most jobs hard to tolerate and will lead to unhappiness.

Looking at the numbers

This research is based on a survey that included 2,220 developers, spread across 88 countries, and the most represented nationality was American, with 24% of respondents. The majority of participants were men (94%), with an average of 8.2 years of experience, and people who were born at 1984.

Out of the total participants, 75% are professional software developers. Only 8% are in management roles, such as CEO, CTO and other c-level positions. 10% of participants develop software as a hobby, passion or volunteer without pay.

Final Thoughts

We can't say it's a surprise that happy developers are better developers. This research gives us an idea as to what the impact may be of continuing to rely on manual processes for debugging purposes.

Implementing Automated Root Cause analysis won't rid the world of developer unhappiness, but it's undoubtedly a step in the right direction.

Chapter 5

How Comcast Automates Production Debugging to Serve over 10M Video Customers

Comcast's Executive Director of Product, John McCann, explains how OverOps helps his team stay on top of every new error introduced to the X1 XFINITY platform

We sat down with John McCann, the Executive Director of Product Development at Comcast, to see how he and his team are using automation to improve their debugging process. First, we'll get to know their team a little bit more and then we'll jump right into some highlights from our interview with John.

XFINITY's X1 is the flagship application for Comcast, providing users with a different television experience. The application offers an interactive platform combining universal search results from live TV, Comcast's On Demand programming, and DVR recordings, in addition to personalized recommendations and apps. It runs on 23 million devices, across dozens of different data centers.

What are some of your key challenges and pain points?

"We offer services for over 23 million boxes, which means that an issue in production impacts a lot of users. Since we deploy a new version of our application on a weekly basis, we have to stay on top of every new error and exception that might impact the application's performance.

Our monitoring method was inconsistent. We had a log management tool with predefined queries to detect errors and exceptions, and we would spend a lot of time going through the logs trying to identify their severity level and which are worth investigating. One person's set of go-to queries wasn't the same as the next person. Often times different people were looking at different things. And this was relevant only for a handful of the alerts. Our scale led to these alerts not being great in terms of their effectiveness, and sometimes we disregarded them since they were noisy.

This was a tedious process, that involved manual effort from our team. Since there are millions of devices that run our application, pinpointing a single error or trying to reproduce it takes up too much time and resources.

When issues hit production, it would impact our customers and it was up to us to try and figure out what went wrong and how to quickly fix it."

How OverOps has helped you solve issues?

"We've integrated OverOps with our automated deployment model, that helps us instrument our application servers.

We use OverOps regularly for all of the unknown error conditions that we didn't foresee. It helps us automate the process of sifting through log files, making it easier to detect issues as soon as they appear.

In fact, we had a full day where the whole team did what we called an "exception burn-down day", where we basically spent an entire day fixing exceptions and log errors that were identified by OverOps. We spent a good amount of time essentially reducing the noise in our application and in a lot of cases fixing problems that had eluded us in the past.

Thanks to OverOps, we now have visibility into the long tail of problems that the system has, that we otherwise wouldn't have visibility into. We know as soon as an error occurs and have the ability to react fast to every issue, error or exception."

How are you integrating OverOps with your daily workflow?

"After installing OverOps, we almost immediately saw detailed data about our application's performance. We were able to detect where exceptions were thrown, and could display changes in the application's behavior.

OverOps is especially helpful when it comes to issues that might impact our users. We use the OverOps dashboard to see our application's behavior and look for trends along with some of the aggregated metrics. That way, we can look for highly problematic areas and quickly detect and fix any error without harming the user's experience."

Final Thoughts

Comcast needed to implement Automated Root Cause analysis to support their high frequency release cycle. With less time in between deployments, it became even more critical for their team to be able to identify production errors as soon as they occurred. Hear more about Comcast's story with automated deployments here.

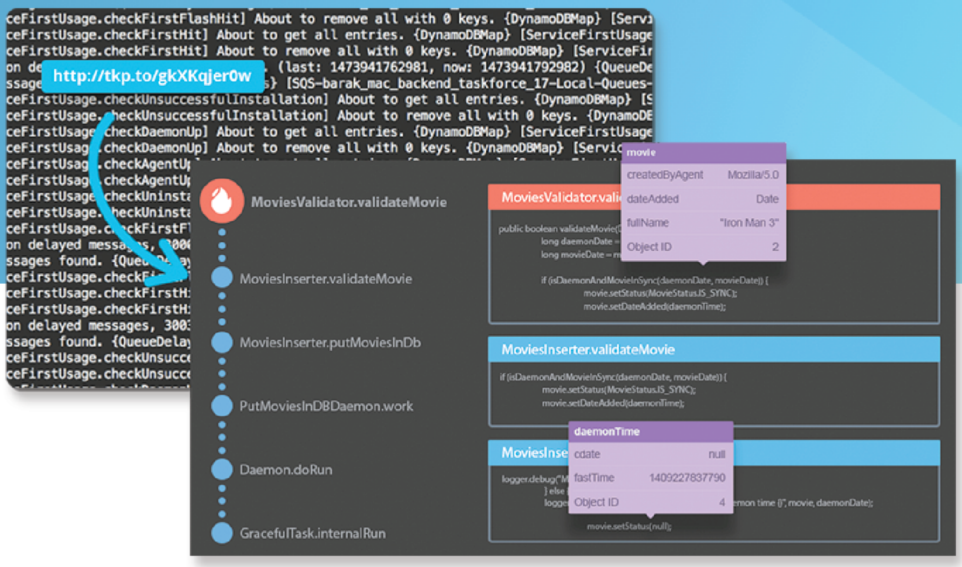
OverOps

Datasheet

Know **When**, **Where** and **Why** Java Code Breaks in Production

See the complete source code and variable state for any error across the entire call stack.

Available as SaaS, hybrid and on-premises



```
ceFirstUsage.checkFirstFlashHit] About to remove all with 0 keys. {DynamoDBMap} [Servi
ceFirstUsage.checkFirstHit] About to get all entries. {DynamoDBMap} [ServiceFirstUsage
ceFirstUsage.checkFirstHit] About to remove all with 0 keys. {DynamoDBMap} [ServiceFir
on de                              . (last: 1473941762981, now: 1473941792982) {QueueDe
ssage                              s} [SQS-barak_mac_backend_taskforce_17-Local-Queues-
ceFirstUsage.checkUnsuccessfulInstallation] About to get all entries. {DynamoDBMap} [S
ceFirstUsage.checkUnsuccessfulInstallation] About to remove all with 0 keys. {DynamoDE
ceFirstUsage.checkDaemonUp] About to get all entries. {DynamoDBMap} [ServiceFirstUsage
ceFirstUsag .checkDaemonUp] About to remove all with 0 keys. {DynamoDBMap} [Servic
ceFirstUsag .checkAgentUp
ceFirstUsag .checkAgentUp
ceFirstUsage.checkUninsta
ceFirstUsage.checkUninsta
ceFirstUsage.checkFirstFl
on delayed messages, 300
ssages found. {QueueDelay
ceFirstUsage.checkFi
ceFirstUsage.checkFirstH
ceFirstUsage.checkFirstH
on delayed messages, 300
ssages found. {QueueDelay
ceFirstUsage.checkUnsucc
ceFirstUsage.checkUnsucc
```

movie	
createdByAgent	Mozilla/5.0
dateAdded	Date
fullName	"Iron Man 3"
Object ID	2

```
public boolean validateMovie(
        long daemonDate =
        long movieDate = m

        if (isDaemonAndMovieInSync(daemonDate, movieDate)) {
                movie.setStatus(MovieStatus.IS_SYNC);
                movie.setDateAdded(daemonTime);
```

```
if (isDaemonAndMovieInSync(daemonDate, movieDate)) {
        movie.setStatus(MovieStatus.IS_SYNC);
        movie.setDateAdded(daemonTime);
```

daemonTime	
cdate	null
fastTime	1409227837790
Object ID	4

```
logger.debug("M
        } else {
        logge                    emon time {}", movie, daemonDate);
                movie.setStatus(null);
```

Fast Paced Innovation

Complex Java applications operate mission critical capabilities at the core of your business. Forward thinking companies who use OverOps, such as TripAdvisor, Fox, Nielsen, Zynga and Cotiviti, plan ahead and create automated processes to identify and solve critical production issues the moment they occur. Traditional troubleshooting workflows take days and weeks to reach a solution, create negative user experiences and cause delays to release cycles and product roadmaps.
When developers spend over 20% of their time debugging production issues - the business impact is impossible to ignore.

Automated Error Resolution

OverOps is the only solution that provides business with actionable root cause visibility that cuts down the time it takes to troubleshoot production errors by over 90%. Unlike logs and performance monitoring solutions, OverOps immediately identifies any new error that's introduced to the application, shows the complete source code that caused it, and the exact variable state that led to the error. It is built to be secure and to operate in scale, supporting thousands of JVMs through SaaS and On-Premise environments, under strict PCI and HIPAA compliant regulations.

"When we release a new version, OverOps alerts us about errors in real time, shows us the variables and lets us easily reproduce and solve the issue. OverOps turned days of work into minutes."

Avg. Issue Resolution time: **3 Days** → **5 Minutes** = **99% Reduction** ✓

tripadvisor

Steve Rogers,
Software Development Director
at Viator, a TripAdvisor company

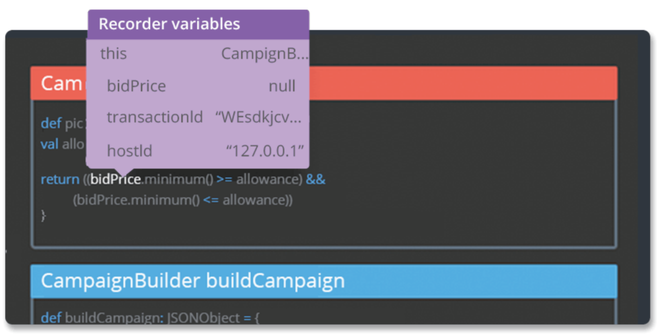
OverOps **Datasheet**

Code and Variable State

- **Code View:** See the complete source code and variable state across the entire call stack
- **Log View**: See the last 250 DEBUG level statements leading to the error, even if they weren't logged
- **JVM View**: See the memory state at the moment of error, including active threads and process metrics
- Open a JIRA ticket with the exact state that caused the error
- See multiple snapshots of recurring errors

Real Time Detection

- See all caught and uncaught exceptions, logged warnings, logged errors, and HTTP errors
- Group errors by microservice, server, app, or deployment
- Drill into the root cause of each event through the error analysis view
- Create custom views and alerts to focus on the errors your team cares about the most
- Receive a push notification for critical errors through Slack, HipChat, PagerDuty, JIRA or Email

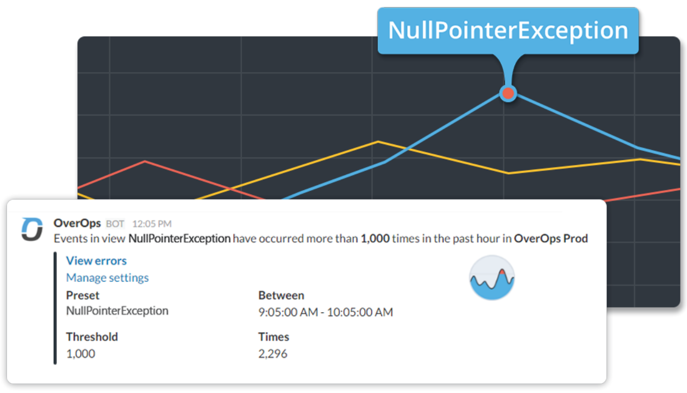


this	CampignB...
bidPrice	null
transactionId	"WEsdkjcv...
hostId	"127.0.0.1"

```
def pic
val allo

return ((bidPrice.minimum() >= allowance) &&
    (bidPrice.minimum() <= allowance))
}
```

```
def buildCampaign: JSONObject = {
```



OverOps BOT 12:05 PM
Events in view **NullPointerException** have occurred more than **1,000** times in the past hour in **OverOps Prod**

Preset	**Between**
NullPointerException	9:05:00 AM - 10:05:00 AM
Threshold	**Times**
1,000	2,296

Full Code and Variable State

Immediately reproduce any error.
No need to manually reproduce issues by searching for information in logs.

<1% overhead in production

OverOps operates between the JVM and processor level, enabling it to run in staging and production.

Reliability and Availability

Proactively detect all new and critical errors. Prevent negative impact to your users and business.

No Changes to Code or Build

Immediately deployed in minutes.
Every new code release or microservice is automatically monitored for new errors.

Fits Into Existing Workflow

Native integration with current tooling ecosystem. New errors are automatically routed to the right developer.

PII Redaction and AES Encryption

Source code and variable state are redacted for PII and privately encrypted with 256-bit AES keys. HIPAA and PCI compliant.

Supported Platforms:

JDK 1.6 and above | HotSpot, OpenJDK, IBM JVM | Java, Scala, Clojure, Groovy | Linux, OS X, Windows | Docker, Chef, Puppet, Ansible | Coming soon: .NET

Integrations:

SLF4J, Log4j, Logback, Apache Commons Logging, Java Logger | Splunk, ELK, SumoLogic, and any other log management tool | AppDynamics, New Relic, Dynatrace | Workflow automation: Slack, HipChat, JIRA, Pagerduty | Webhooks | StatsD

Tel: 1-800-324-3053 www.overops.com
343 Sansome Street San Francisco, CA 94107

Final Thoughts

Debugging is crucial, but there's no reason why we should be spending so much of our time on manual tasks like sifting through log files and trying to reproduce user-reported errors. Without visibility into the real root cause of what's going on in your application, it's hard to fix and maintain it.

Plus, identifying and handling exceptions should be about more than just fixing a problem. By automating root cause analysis, you can achieve better visibility of your application in production so that debugging doesn't waste your days or keep you up at night.

We hope you've found this guide useful and would be happy to hear your feedback on twitter @overopshq and over email: hello@overops.com