

The Complete Guide to Logging Mistakes CTO's Must Avoid

By Henn Idan

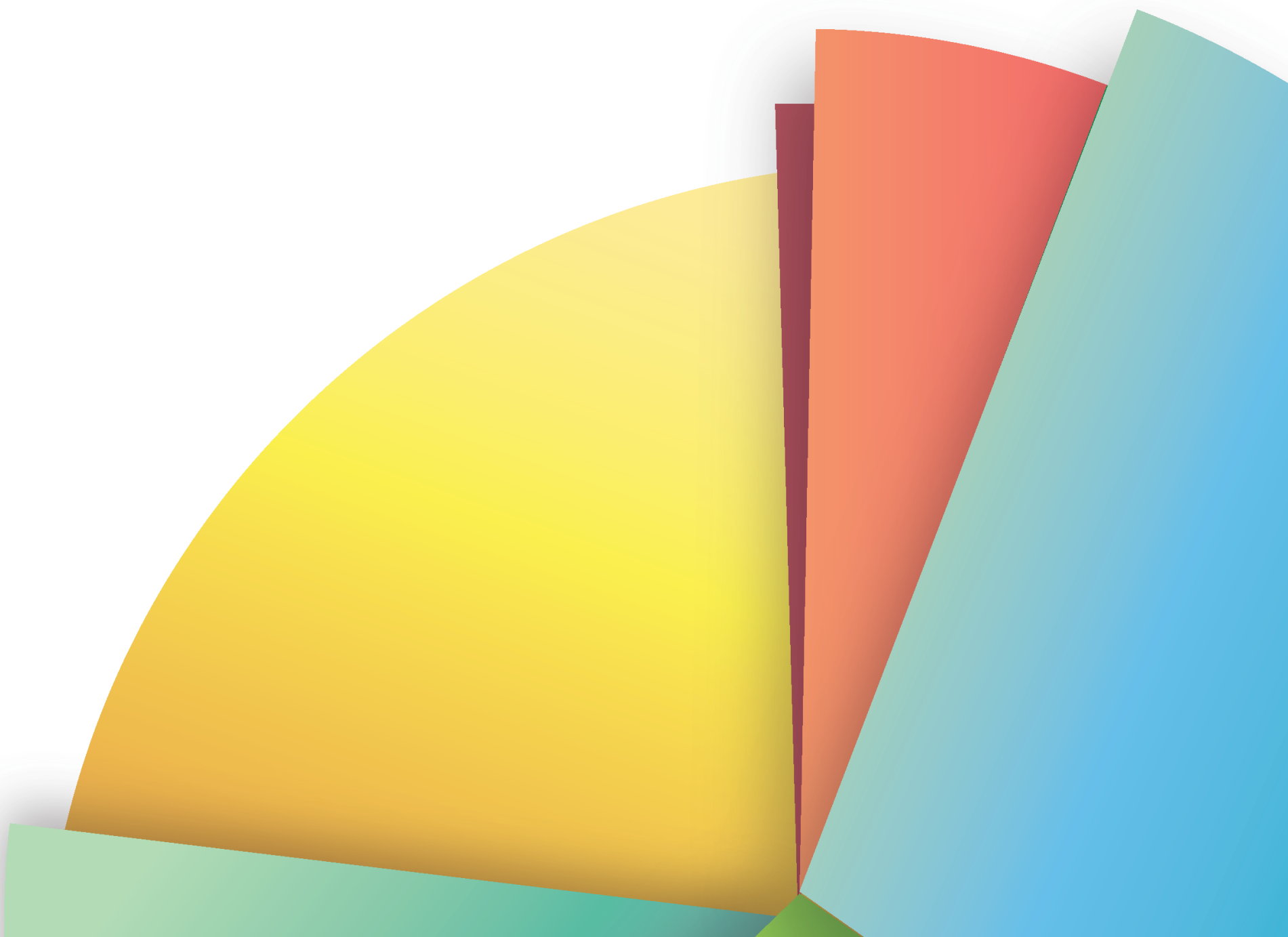


Table of Contents

Introduction.....	Page 2
Chapter 1.....	Page 3
779,236 Java Logging Statements, 1,313 GitHub Repositories: ERROR, WARN or FATAL? <i>Java logs data crunch: How GitHub's top Java projects use logs?</i>	
Chapter 2.....	Page 9
Is Standard Java Logging Dead? Log4j vs Log4j2 vs Logback vs java.util.logging <i>The Java log levels showdown: SEVERE FATAL ERROR OMG PANIC</i>	
Chapter 3.....	Page 16
Over 50% of Java Logging Statements Are Written Wrong <i>Why can't production logs help you find the real root cause of your errors?</i>	
Chapter 4.....	Page 22
What's the Top Java Logging Method on GitHub? String Concatenation vs Parameterized Logging <i>Parameters, concatenations or both; which logging method should you use?</i>	
Chapter 5.....	Page 29
How Did We Get the Data? Google BigQuery <i>To reach the first set of Java projects and the breakdown of their logging, we completely relied on Google BigQuery and the GitHub archive database.</i>	
Meet OverOps.....	Page 36
Final Thoughts.....	Page 38

Introduction

The Complete GitHub Logging Research: How Are We Using Logs?

One of the things we like most at [OverOps](#) is crunching data and learning new things. Yes, we're that fun at parties. If you feel the same, then you've reached the right place.

In our following eBook, we've decided to see how developers use their logs, along with how we can make them better for the entire development process.

For this data crunch we used Google BigQuery and GitHub's database - the top 400,000 repositories by number of stars they were given in 2016, with some SQL on top.

Now it's time to slice, dice and start crunching these repositories by their logs.

Chapter 1

779,236 Java Logging Statements, 1,313 GitHub Repositories: ERROR, WARN or FATAL?

Java logs data crunch: How GitHub's top Java projects use logs?

The starting point for this research is the [GitHub archive](#), and its datasets on Google BigQuery. We wanted to focus on qualified Java projects, excluding android, sample projects, and simple testers. A natural choice was to look at the most starred projects, taking in the database of the top 400,000 repositories.

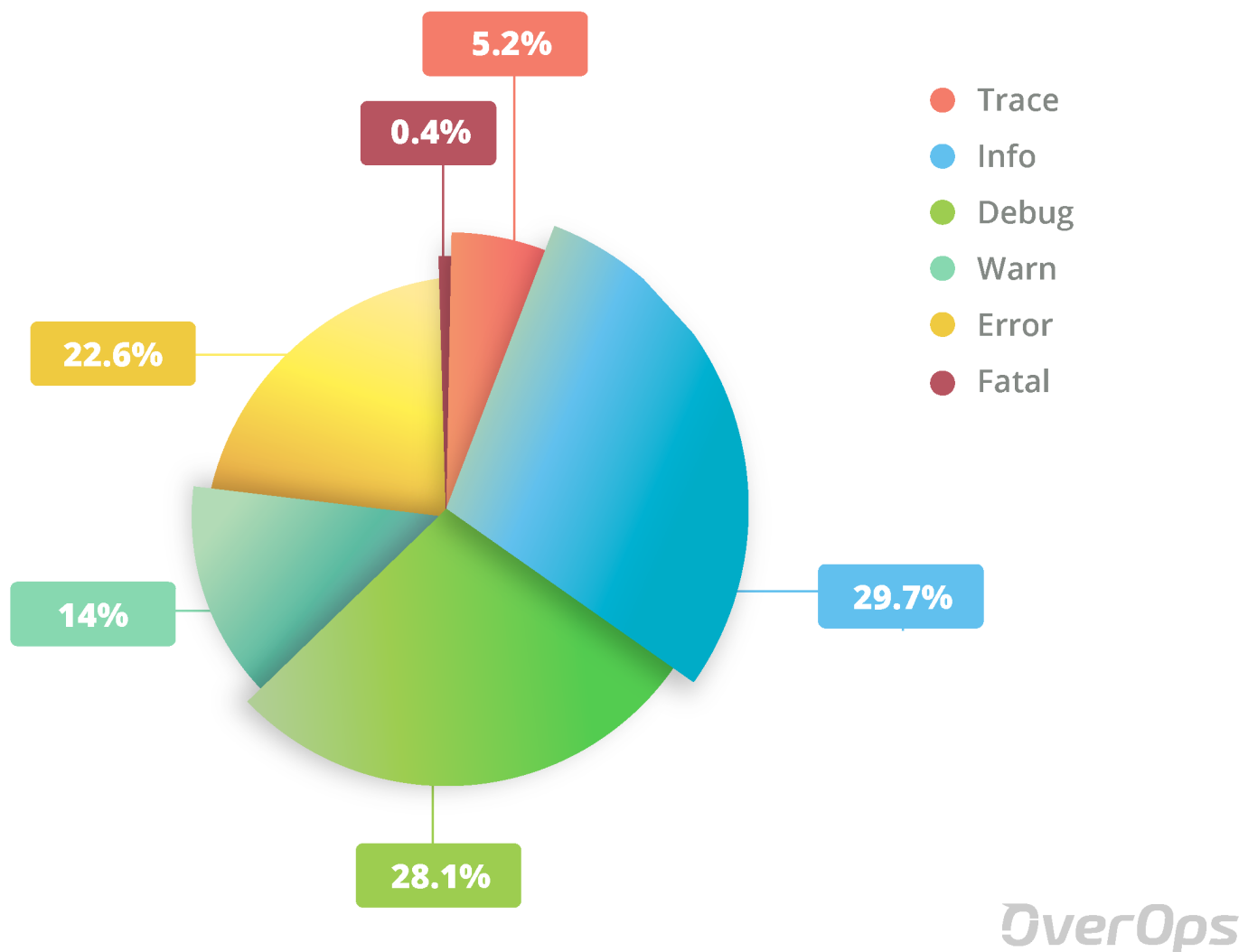
We ended up with 15,797 repositories with Java source files, 4% of the initial dataset. But it didn't stop there. Looking at the number of logging statements, we decided to only focus on projects with at least a 100 different statements, and only those who use the standard Logback / Log4j / Log4j2 / SLF4J levels: TRACE, INFO, DEBUG, WARN, ERROR and FATAL. JUL, java.util.logging, was ignored. More details on that later.

This left us with 1,313 Java project data vectors to play with. We believe this to be a fairly representative sample of what we were trying to achieve.

Result Highlights

The Average Java Log Level Distribution

The Average Project by Logging Levels



The Average Java Log Level Distribution

And the winner is... INFO, making up 29.7% of the logging statements in the average project. Followed by DEBUG with 28.1%, and ERROR with 22.6%.

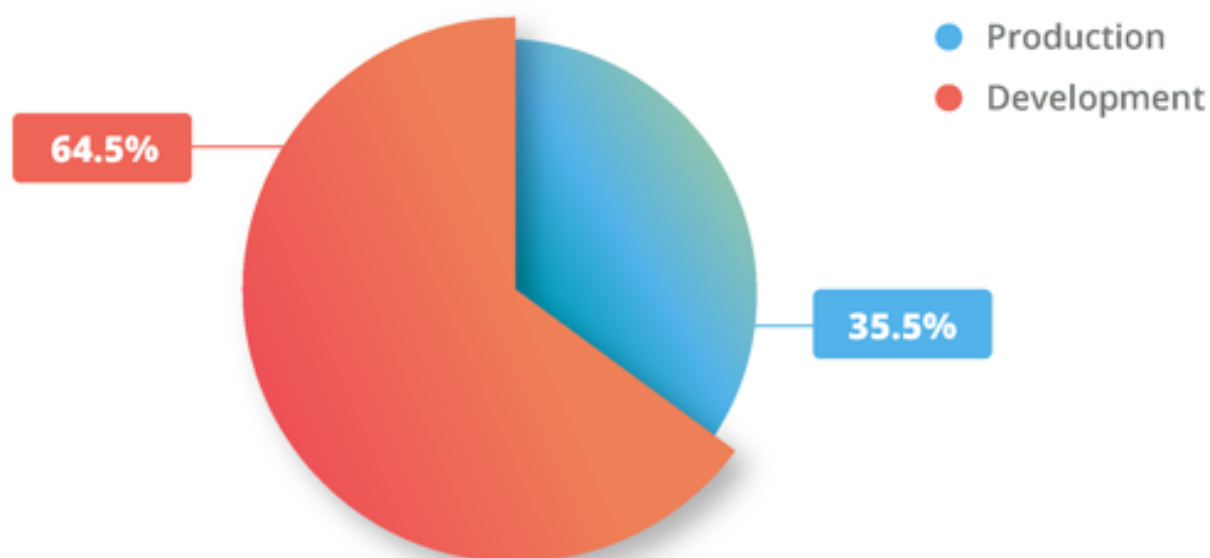
Closing the list are WARN (14%), TRACE (5.2%), and FATAL (0.4%).

On our own codebase, we got TRACE (0.55%), INFO (33.42%), DEBUG (7.92%), WARN (10.85%), and ERROR (47.26%).

Production vs. Development Logging

Based on this data, the next logical step was to look at the production logging vs. development logging breakdown. Since most (sane) people turn off logging for any level below WARN, this is where we drew the line.

Production Logging vs. Development Logging



OverOps

Production vs. Development Logging

For the average Java application, there are 35.5% unique logging statements that have the potential to be activated in production, and 64.5% statements that are only activated in development. That's almost DOUBLE. And not only that, naturally, log levels below WARN happen much more often, throw this into production and the difference would be much bigger.

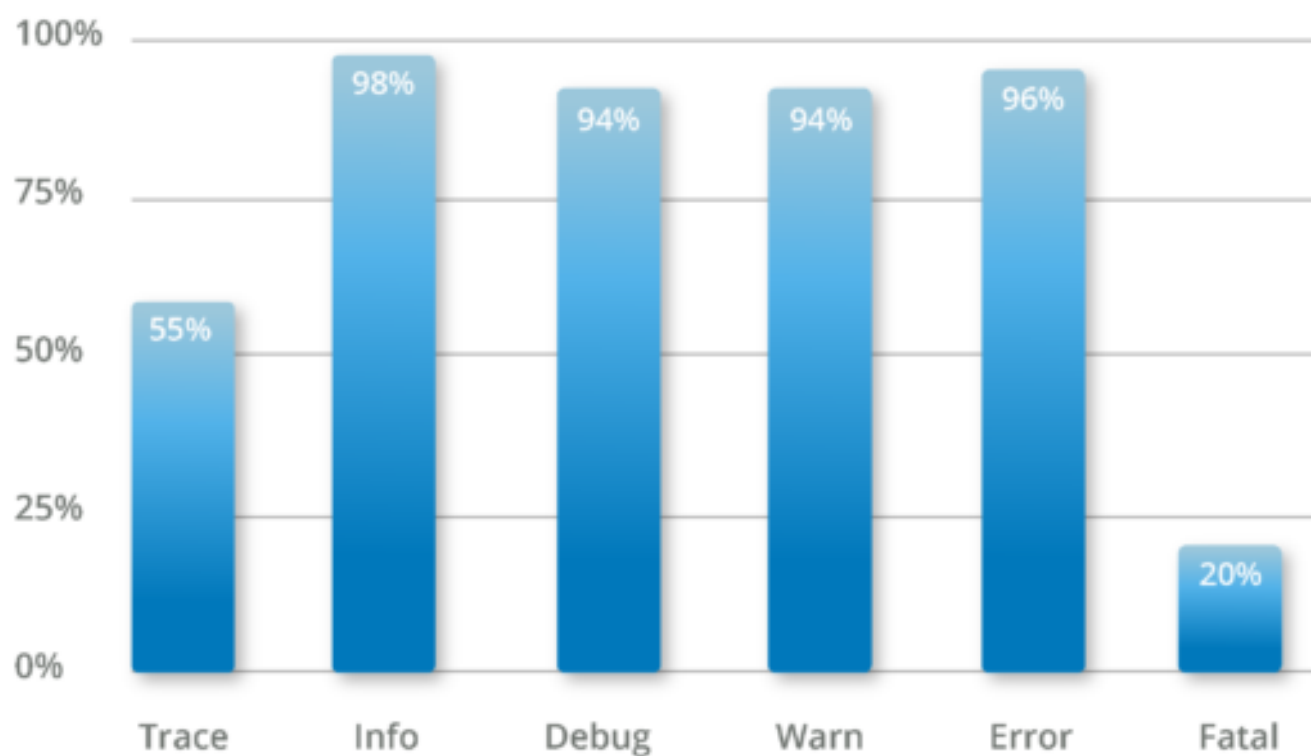
Excessive logging can produce a lot of overhead in terms of both storage, and performance, which quickly translates to lots of money – since log management tools would charge you by the log's volume. In one of our previous research posts, we also found out that 3% of the top unique log events, produce on average 97% of the total log size.

btw, regardless of which logging levels you're using in production, you can get the last 250 DEBUG, TRACE and INFO statements for any issue in production with our tool.

Actual Log Levels in Use

Another cool thing the data allowed us to look at was the popularity of different log levels. How often is TRACE used compared to other levels? We're not big fans of TRACE but looks like 55% of the projects use it:

Percentage of Projects Using Each Log Level



OverOps

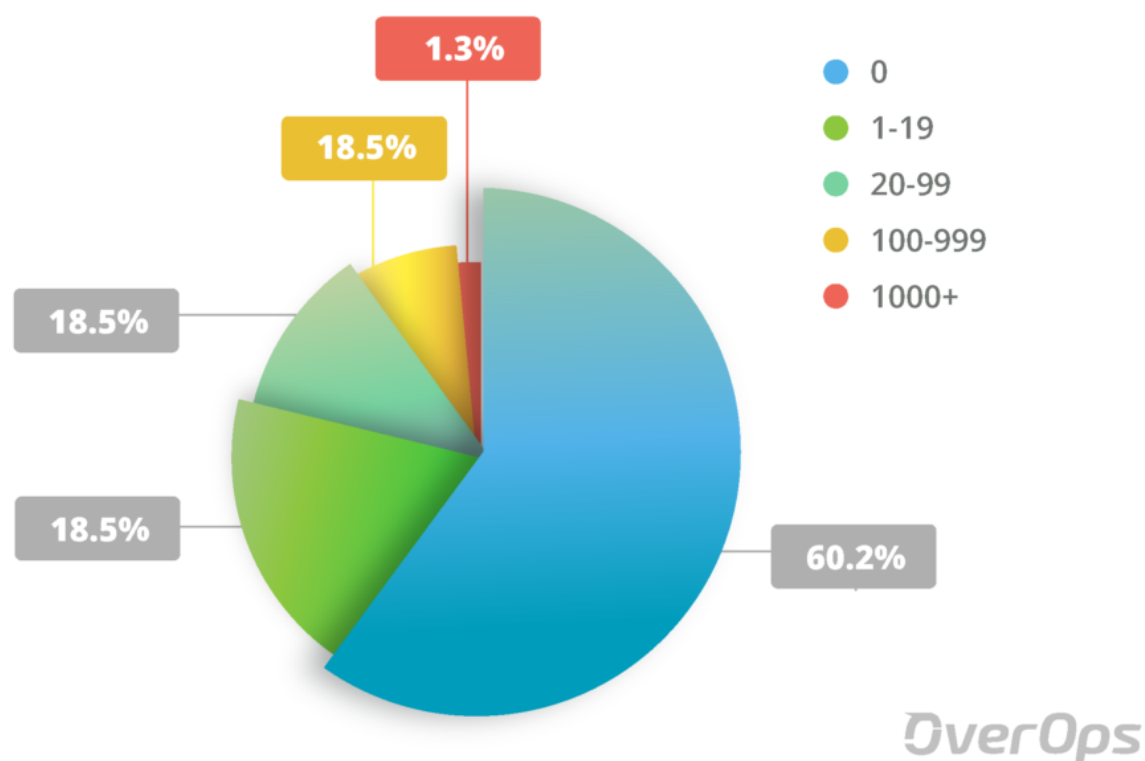
Another interesting insight is that FATAL is only used by 20% of the projects. INFO, DEBUG, WARN, and ERROR are kings.

Examining the Data

Looking at the results from the GitHub data crunch on Google BigQuery, we first ended up with 15,797 repositories with Java source files. 60% of those, didn't use logging at all, so they weren't relevant to this research. These repositories mostly included test projects, small experiments, utilities, learning materials, etc.

We broke those down to projects by number of logging statements:

Number of Logging Statements Per Project

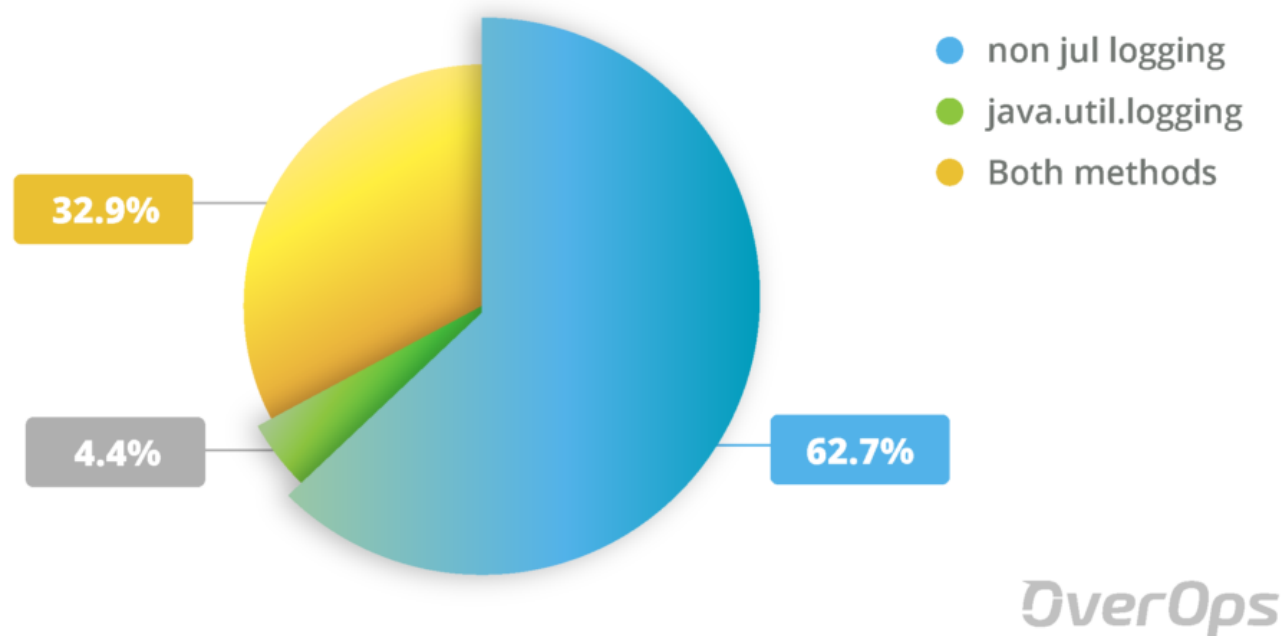


Number of Logging Statements per Project

Another criteria for the research was that we only use projects with a considerable amount of logging baked in. Based on this breakdown, we decided to focus on projects with 100+ logging statements.

Digging in further, we looked into java.util.logging levels versus Logback / Log4j / SLF4J and decided to focus on the latter:

Logging Levels by Type



Logging Levels by Type

The 1,313 Java projects we used for the research were those that had at least a 100 logging statements, excluding those that were pure JUL, java.util.logging (FINE, FINER, FINEST, SEVERE, etc.).

Final Thoughts

We've learned that INFO makes up 29.7% of the logging statements in the average project, and that there are 35.5% unique logging statements with the potential to be activated in production, while 64.5% of statements are only activated in development. Now, let's break our logging down into levels.

Chapter 2

Is Standard Java Logging Dead? Log4j vs Log4j2 vs Logback vs `java.util.logging`

The Java log levels showdown: SEVERE FATAL ERROR OMG PANIC

Capitalized log levels induce high levels of stress. What if, instead of ERROR we'd just use "oops"? After our data crunch over GitHub's top Java projects and the logging statements they use, we now know the log level breakdown of the average Java project.

Now, it's time to explore the data set from another angle, shed some more light on the dataset, and put the focus on the use of standard `java.util.logging` levels versus more popular frameworks like Log4j (+ Log4j 2), and Logback.

Step right in.

Meet the players

Logging utilities can be roughly divided to 2 categories: the logging facade and the logging engine.

As far as logging facades go, you pretty much have 2 choices: slf4j and Apache's commons-logging. In practice, **4 out of 5 Java projects choose to go with slf4j**. Based on data from [the top Java libraries in 2016 on Github](#). The motivation for using a logging facade is pretty definitive and straightforward, an abstraction on top of your logging engine of choice – allowing you to replace it without changing the actual code and logging statements.

As to the logging engine, the most popular picks are Logback, [which is an evolved version of Log4j](#), Log4j itself, and its new version since the development was passed on to the Apache Software Foundation, Log4j2. Trailing behind is Java's default logging engine, java.util.logging aka JUL.

Pointing fingers and calling names

On the “superficial” side of things, each of the logging frameworks has slightly different names for their logging levels.

slf4j	Log4j	Log4j2	Logback	java.util.logging
FATAL	FATAL	FATAL		
ERROR	ERROR	ERROR	ERROR	SEVERE
WARN	WARN	WARN	WARN	WARNING
INFO	INFO	INFO	INFO	INFO
				CONFIG
DEBUG	DEBUG	DEBUG	DEBUG	FINE
				FINER
TRACE	TRACE	TRACE	TRACE	FINEST

OverOps

In the rare case where slf4j is used with java.util.logging, the following mapping takes place:

FINEST -> TRACE

FINER -> DEBUG

FINE -> DEBUG

INFO -> INFO

WARNING -> WARN

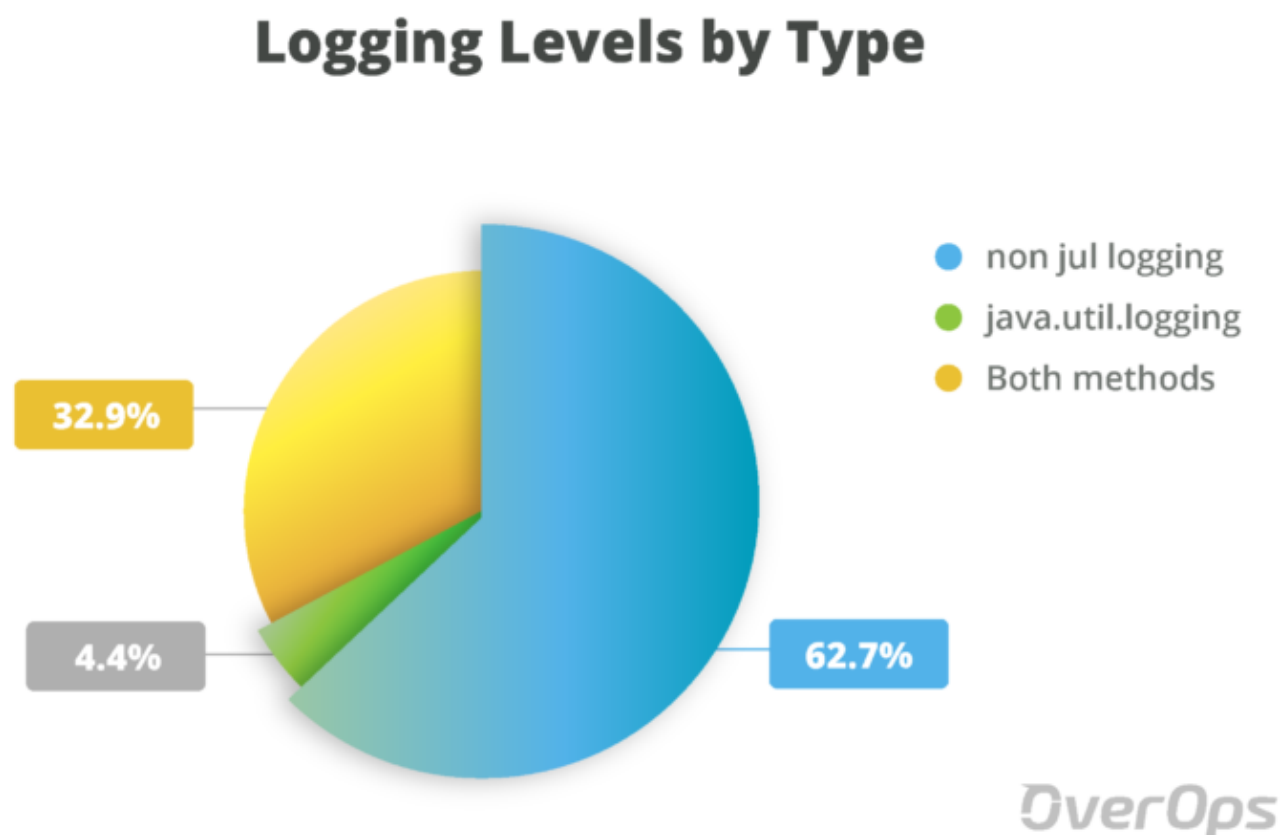
SEVERE -> ERROR

Another thing to notice here is that Logback and java.util.logging have no FATAL equivalent. Behind those error names, are simple integer values, that help control the logging level in a running application. Each library also contains values for OFF and ALL, which basically set the logger level to actually transmit everything, or nothing. Setting a logger level at WARN for instance, would only log WARN messages and above – Its practically the default setting for production environments.

btw, one of the cool things about [the tool that we're building](#), is that you can get log messages lower than WARN in production, even if you've set the logger level to WARN. [Check out this video for a quick \(25 sec\) demonstration](#).

How does the level naming breakdown look in practice?

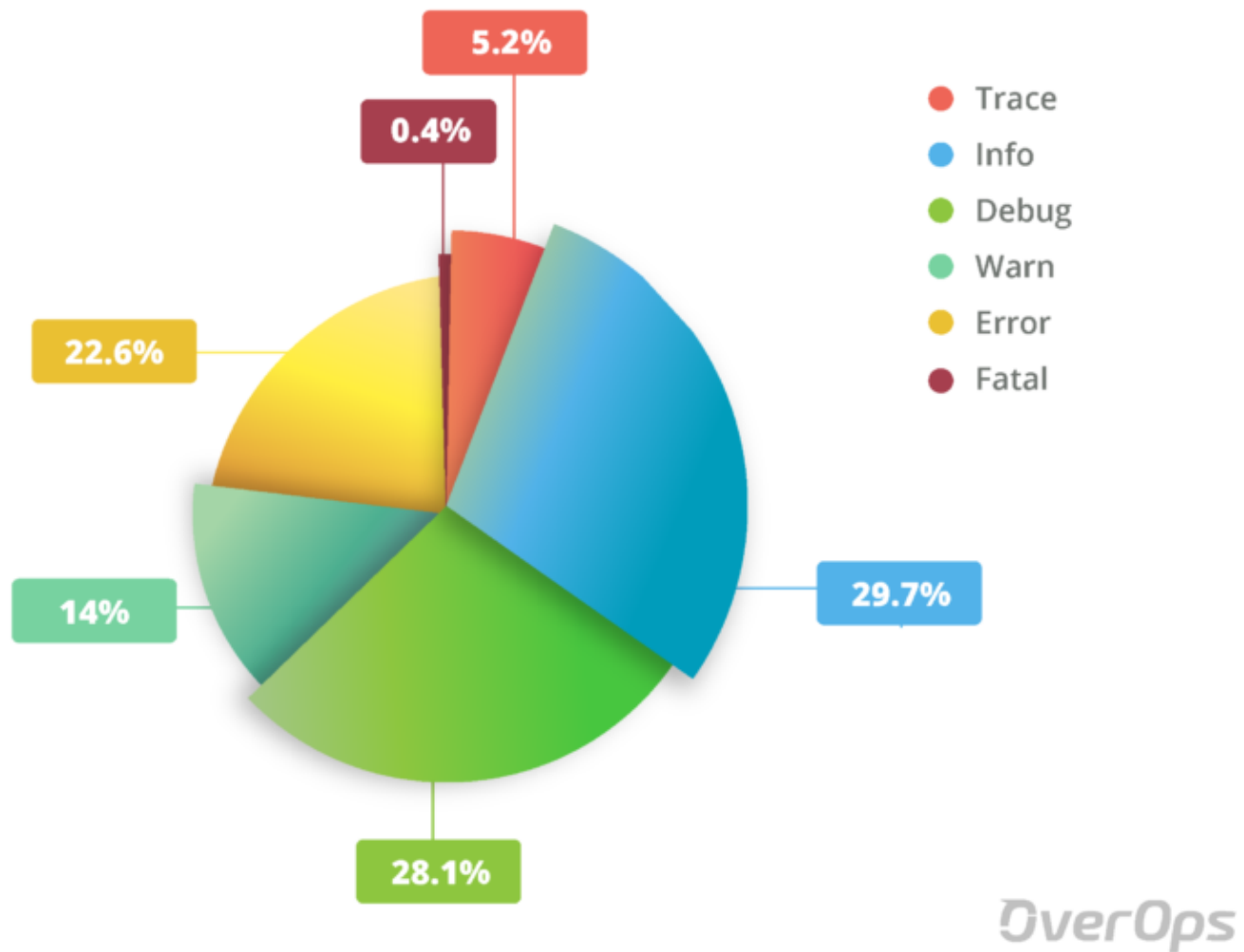
For the data crunch, we focused on the top starred Java projects with at least 100 logging statements in either of the methods. Examining the data set of projects, here's what we found:



Only 4.4% of projects exclusively used the java.util.logging naming scheme.

The average non jul logging project, looked like this (examining 1,313 projects):

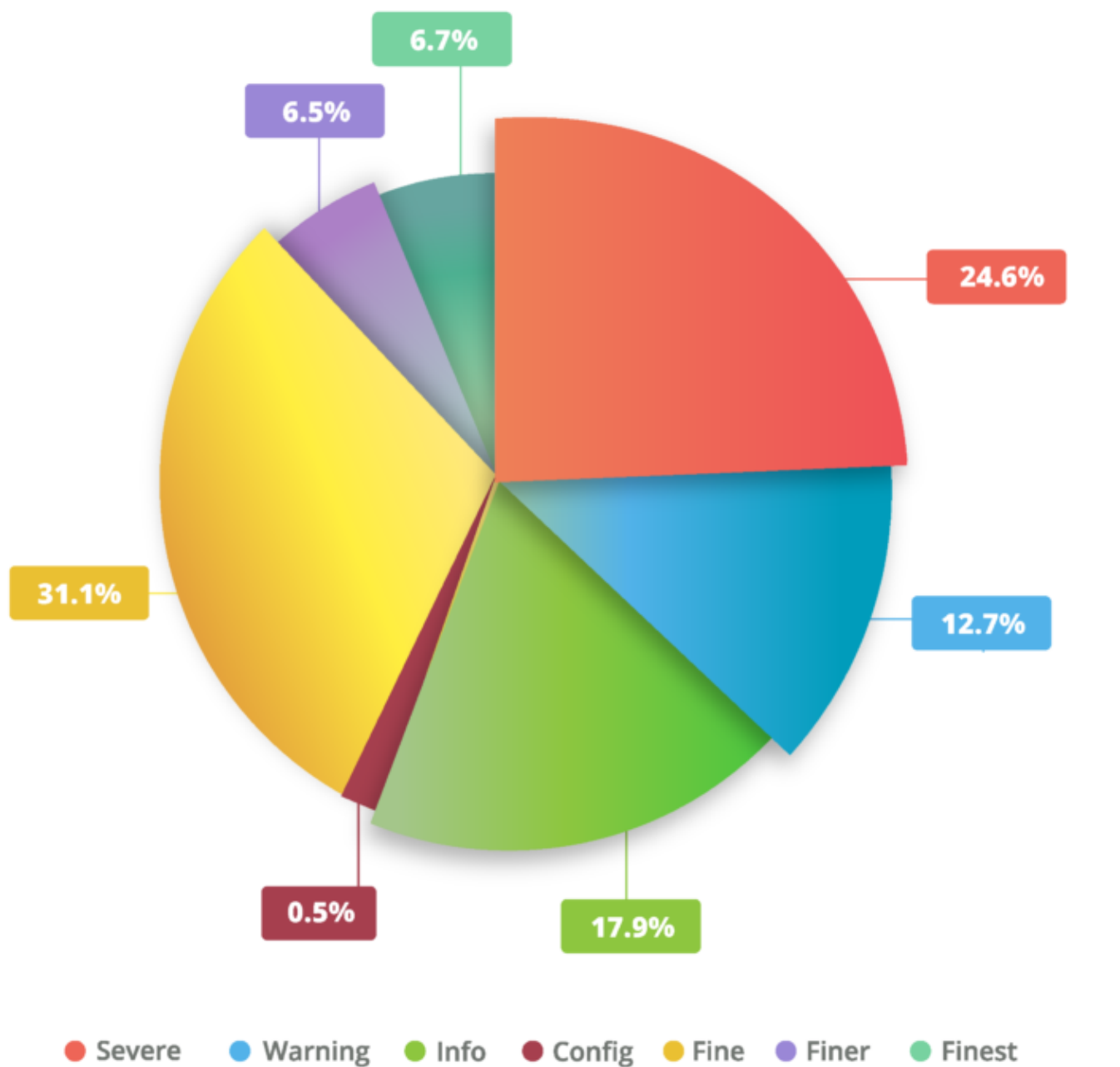
The Average Project by Logging Levels



To look at the average java.util.logging project, we filtered it down to include only projects who had at least 100 statements from levels that don't overlap with the non-JUL naming scheme (WARNING and INFO).

This left us with a smaller dataset, so it might not be big enough to make definite conclusions from:

The Average Project by JUL Logging Levels



OverOps

With that said, it looks like in both situations, **roughly 2/3 of logging statements are disabled in production**, since only WARN and above are activated in that case.

Fun fact: As an extra datapoint, we also looked at ALL / OFF levels. Turns out only 8.6% of the projects examined used them both.

The data stress that java.util.logging is, well, practically dead. Most serious projects choose to go with 3rd party logging frameworks.

Final Thoughts

It seems that `java.util.logging` is, well, practically dead. Most serious projects choose to go with 3rd party logging frameworks.

But do we even know how to use these 3rd party logging frameworks? Apparently not, since most of our logging statements are written wrong.

Chapter 3

Over 50% of Java Logging Statements Are Written Wrong

Why can't production logs help you find the real root cause of your errors?

Asking if you're using log files to monitor your application is almost like asking... do you drink water. We all use logs, but HOW we use them is a whole different question.

Let's take a deeper look into logs and see how they are used and what's written to them. Let's go.

TL;DR: Main Takeaways

If you're not into pie, column or bar charts and want to skip the main course and head straight for the dessert, here are the 5 key points we learned about logging and how it's really done:

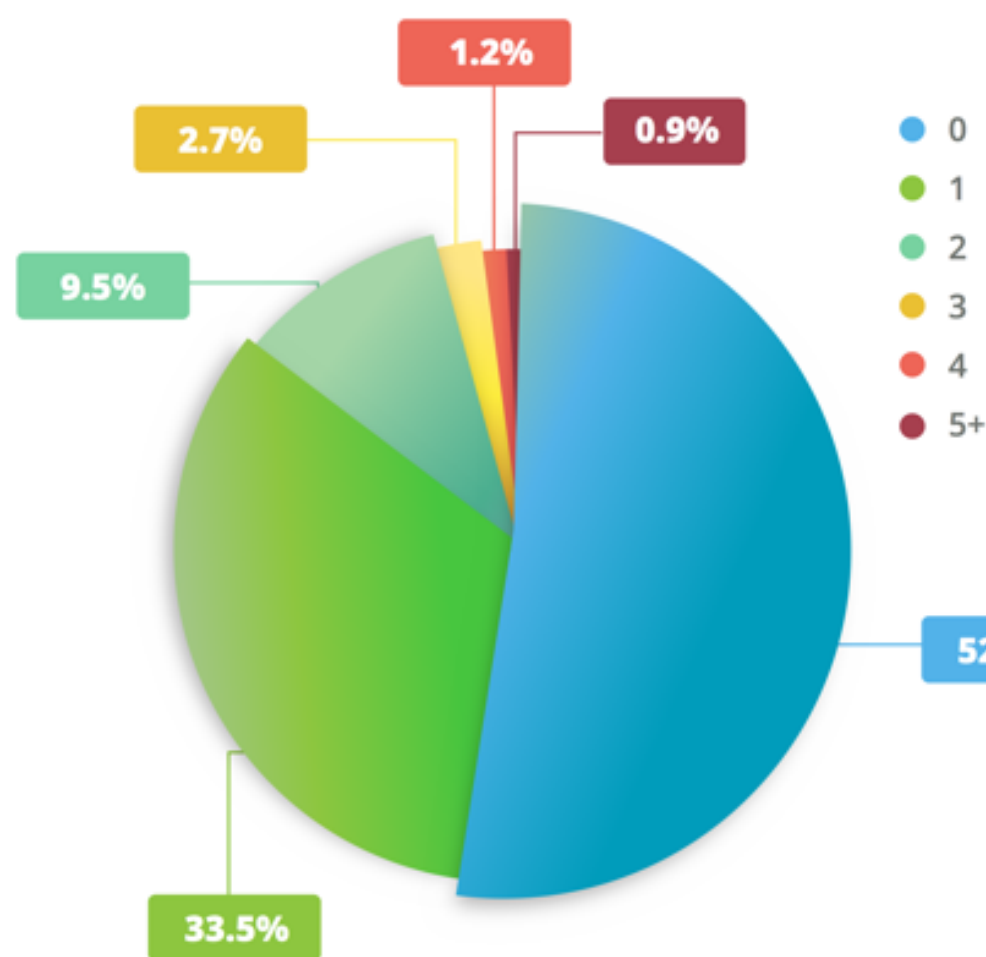
1. Logs don't really have as much information as we think, even though they can add up to hundreds of GBs per day. Over 50% of statements have no information about the variable state of the application
2. In production, 64% of overall logging statements are deactivated
3. The logging statements that do reach production have 35% less variables than the average development level logging statement
4. "This should never happen" always happens
5. There's a better way to troubleshoot errors in production

Now let's back up these points with some data.

1. How Many Logging Statements Actually Contain Variables?

The first thing we wanted to check is how many variables are sent out in each statement. We chose to slice the data on a scale from 0 variables up to 5 and above, in each repository. We then took the total count, and got a sense of the average breakdown over all of the projects in the research.

The Average Java Project by Number of Variables in Each Logging Statement



OverOps

Average Java Project by Number of Variables

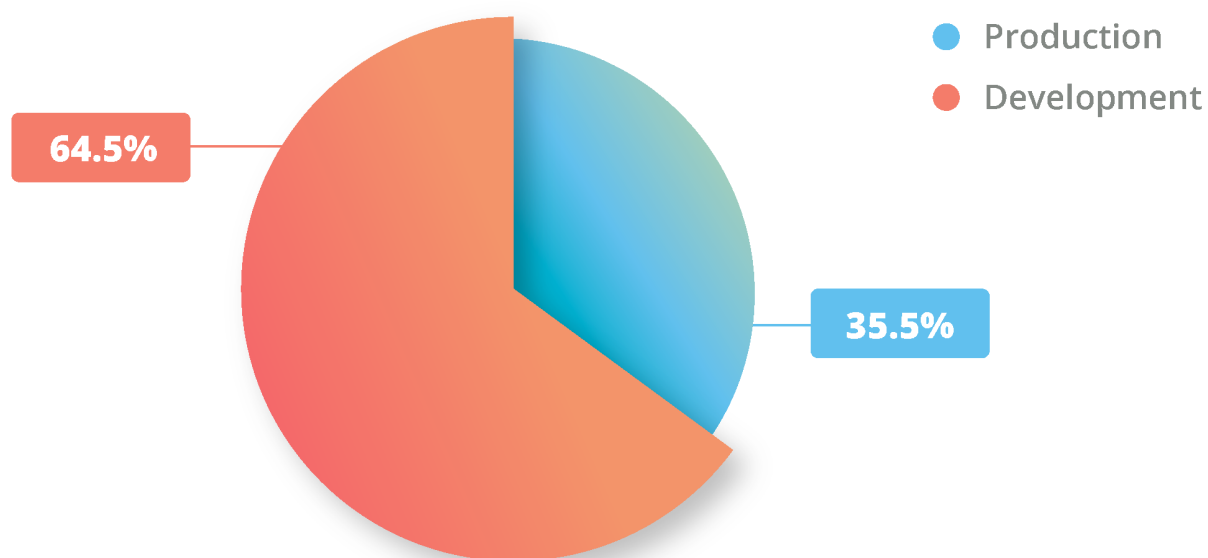
As you can see, the average Java project doesn't log any variables in over 50% of its logging statements. We can also see that only 0.95% of logging statements send out 5 variables or more.

This means that there's limited information about the application that is captured by the log, and finding out what actually happened might feel like searching for a needle in a log file.

2. How Many Logging Statements Are Activated in Production?

Development and production environments are different for many reasons, one of them is their relation to logging. In development, all log levels are activated. However, in production only ERROR and WARN are activated. Let's see how this breakdown looks like.

Production Logging vs. Development Logging



Production vs. Development Logging

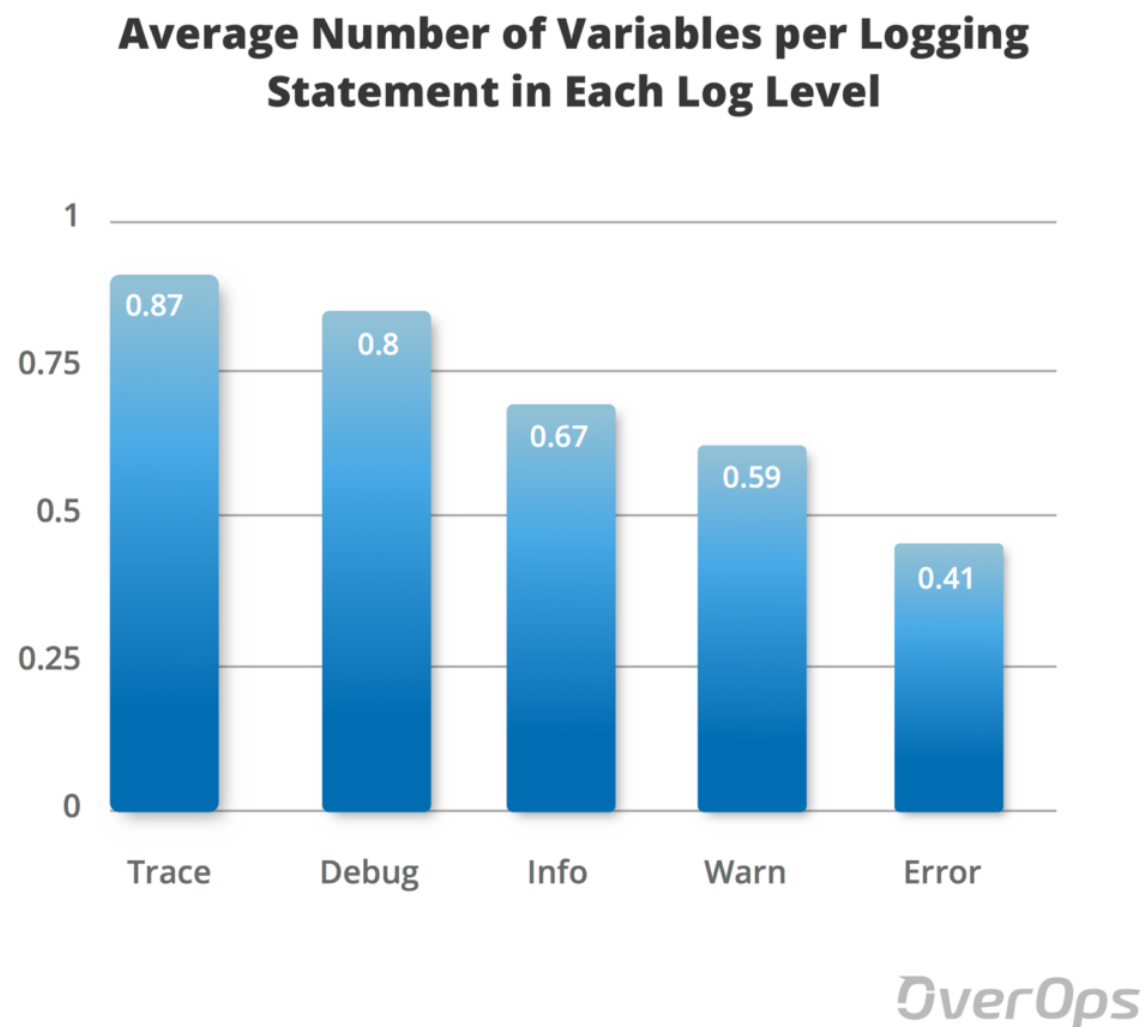
The chart shows that the average Java application has 35.5% unique logging statements that have the potential to be activated in production (ERROR, WARN), and 64.5% of statements that are only activated in development (TRACE, INFO, DEBUG).

Most information is lost. Ouch.

3. What's the Average Number of Variables per Each Log Level?

So, not only do developers skimp on variables in their statements, the average Java application doesn't send out that much statements to production logs in the first place.

Now, we've decided to look at each log level individually and calculate the average number of variables in the corresponding statements.



Average Number of Variables per Logging Statement

The average shows that TRACE, DEBUG and INFO statements contain more variables than WARN and ERROR. “More” is a polite word, considering the average number of variables in the first three is 0.78, and 0.5 in the last 2.

That means that production logging statements hold 35% less variables than development logging statements. In addition, as we've seen earlier, their overall number is also much lower.

If you're searching the log for clues as to what happened to your application, but come up blank – this is why it happens. Not to worry, there's a better way.

[OverOps](#) lets you see the variables behind any exception, logged error or warning, without relying on the information that was actually logged. You'll be able to 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. OverOps also shows you the 250 DEBUG, TRACE and INFO level statements that were logged prior to the error, in production, even if they're turned off and never reach the log file.

We'd be happy to show you how it works, [click here to schedule a demo](#).

4. This Should Never Happen

Since we already have information about all of those logging statements, we've decided to have a little fun. We found 58 mentions to “This should never happen”.

All we can say is that if it should never happen, at least have the decency to print out a variable or 2, so you'll be able to see why it happened anyway :)

Final Thoughts

We all use log files, but it seems that most of us take them for granted. With the numerous log management tools out there we forget to take control of our own code – and make it meaningful for us to understand, debug and fix.

[There's a better way.](#)

Chapter 4

What's the Top Java Logging Method on GitHub? String Concatenation vs Parameterized Logging

Parameters, concatenations or both; which logging method should you use?

A log is a log is a log. Some of us use dedicated log tools to monitor it, while others prefer going through the raw log lines with their very own eyes. It doesn't matter how we consume it – it's an inseparable part of our development cycle.

There are a lot of methods in which we can write to our logs, when endless debates, opinions and arguments surround the logging world. We've decided to focus on one of those topics – and understand which method is better: String concatenation or parameterized logging. How do most developers write variables to their logs?

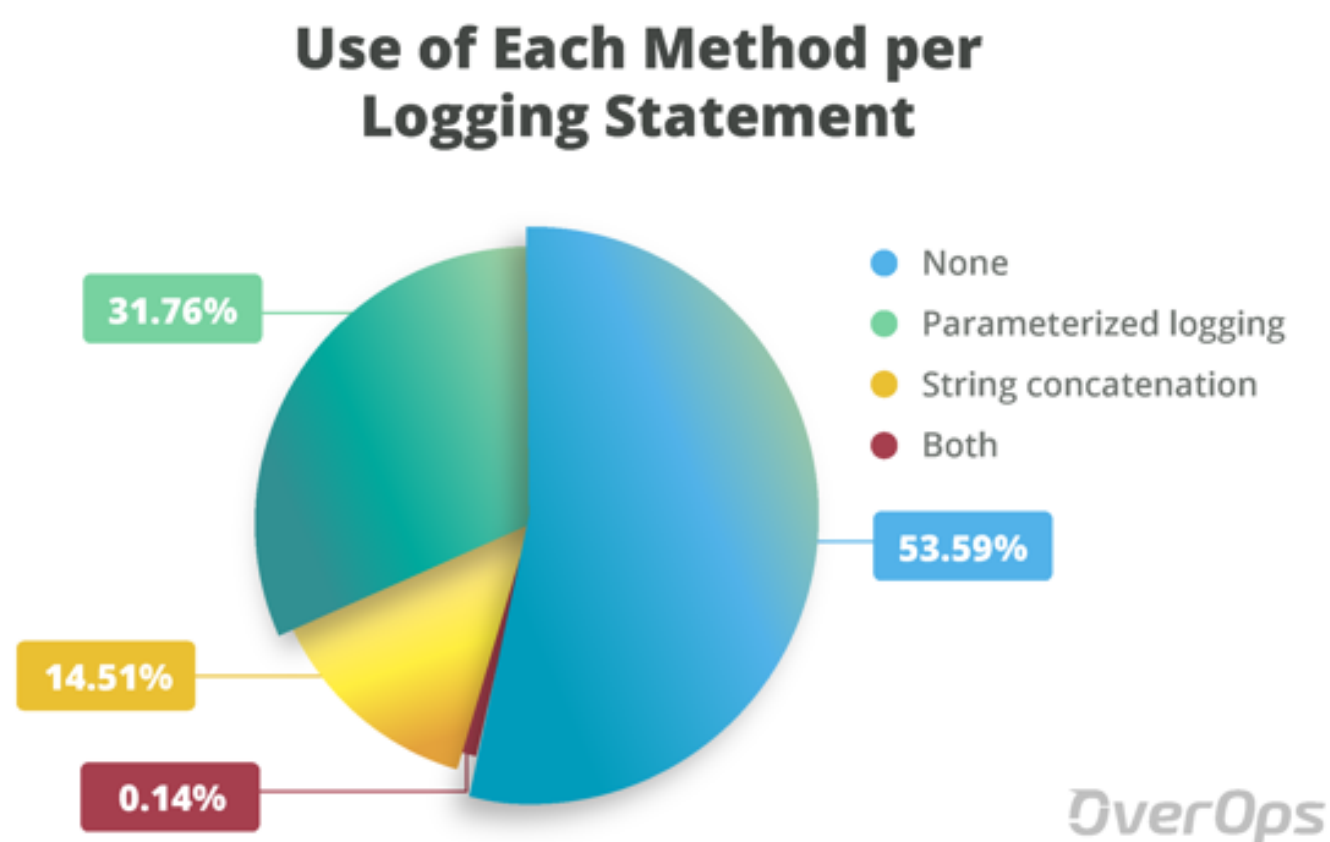
Let's find out.

Adding Up the Numbers

In the previous chapter we tried to understand why production logs can't help us find the real cause of errors and exceptions. Our research returned a lot of data, and we could see a pattern regarding the use of String concatenation and parameterized logging. That lead us to the following question:

How do most developers write to their logs – String concatenation, parameterized logging or both?

And the Winner Is...

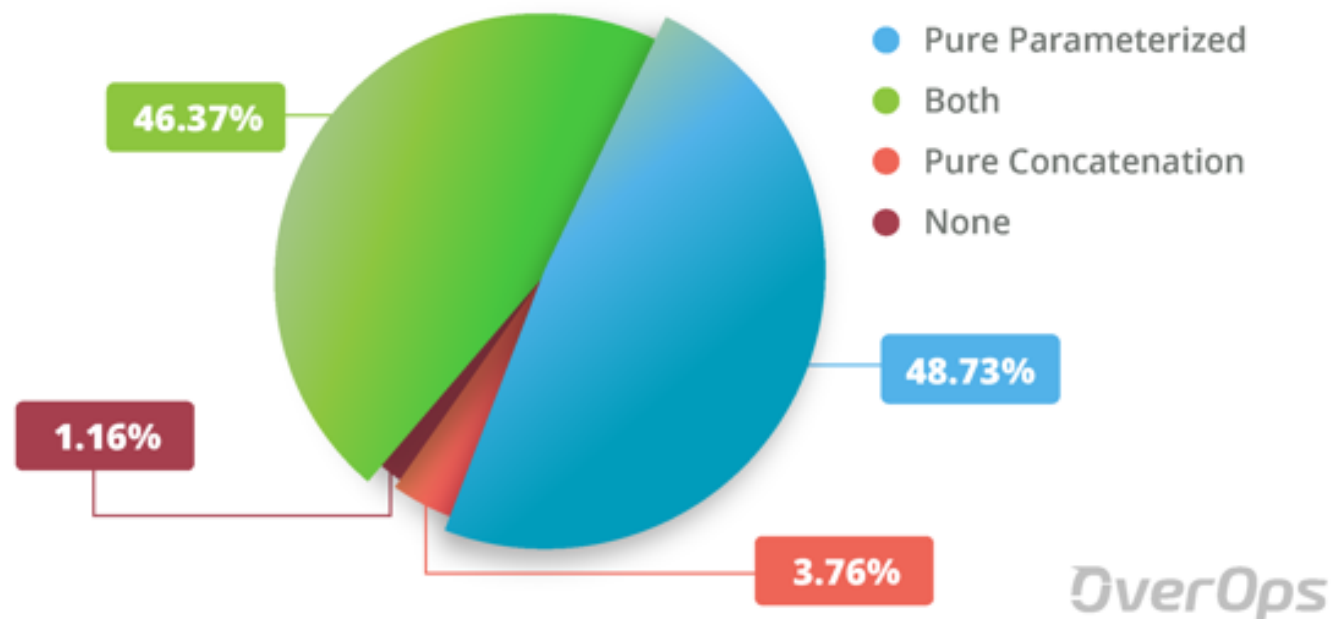


Use of each method across all statements

Well well well, what do we have here? The biggest winner is the use of no method at all. Or in other words, it seems that over 50% of statements don't contain variables. The runner up is parameterized logging, with an appearance in a little over 30% of statements.

Now, let's take a deeper look at the use of these methods across repositories:

Parameterized Logging vs String Concatenation vs Both vs None



Use of each method across repositories

Parameterized logging takes the cake (or the pie chart in this case), but it's a small win. We can see that over 46% of repositories use a mix between the parameterized and the concatenation methods. This means that too many developers didn't choose one method and ended up using both in the same project.

Now that we know who the clear winner is, it's time to understand how it got the crown.

Strings or Parameters?

First thing's first – why do we even need these methods?

When it comes to our log file, we often find ourselves wanting to merge or combine different strings and variables into a single log message. Doing so makes it easier to monitor and understand what happened when the application encounters an error or throws an exception. The 2 methods we've checked are:

- String concatenation – Adding parameters via the “+” operator
- Parameterized logging – In which we use the {} variant, also known as curly brackets, braces or formatting anchor

Now that we know how they're actually used, let's take a deeper look and see what are the main differences between the two:

String Concatenation

String concatenation lets us add variables using the “+” operator, as you can see in the following example:

```
1 logger.debug("This " + this + " and " + that + "with " + compute());
```

It can get a little messy, not only due to how the code looks but also since the variables are converted to strings, regardless of whether the message will be logged or not. In other words, they are evaluated immediately in every log level, even if we're not using it, which in return might affect the overhead of the application.

For example, if we use String concatenation for DEBUG level while running in production, the variables will still convert to strings, even though DEBUG level statements are not logged.

There are ways in which we can overcome this issue, such as using the `logger.isDebugEnabled()` function. As you can guess, this function checks whether debug is enabled before formatting the message.

This can prevent String concatenation from happening when it's not necessary, reducing the overhead, but in return it will make the code look like this:

```
1  if (logger.isDebugEnabled()) {  
2      logger.debug("This " + this + " and " + that + "with " + compute());  
3  }
```

Now think how long your code will be if you use this function with hundreds of log messages. Yikes.

Parameterized Logging

When using parameterized logging, the variables are added with the {} variant:

```
1  logger.debug("This {} and {} with {}", this, that, compute());
```

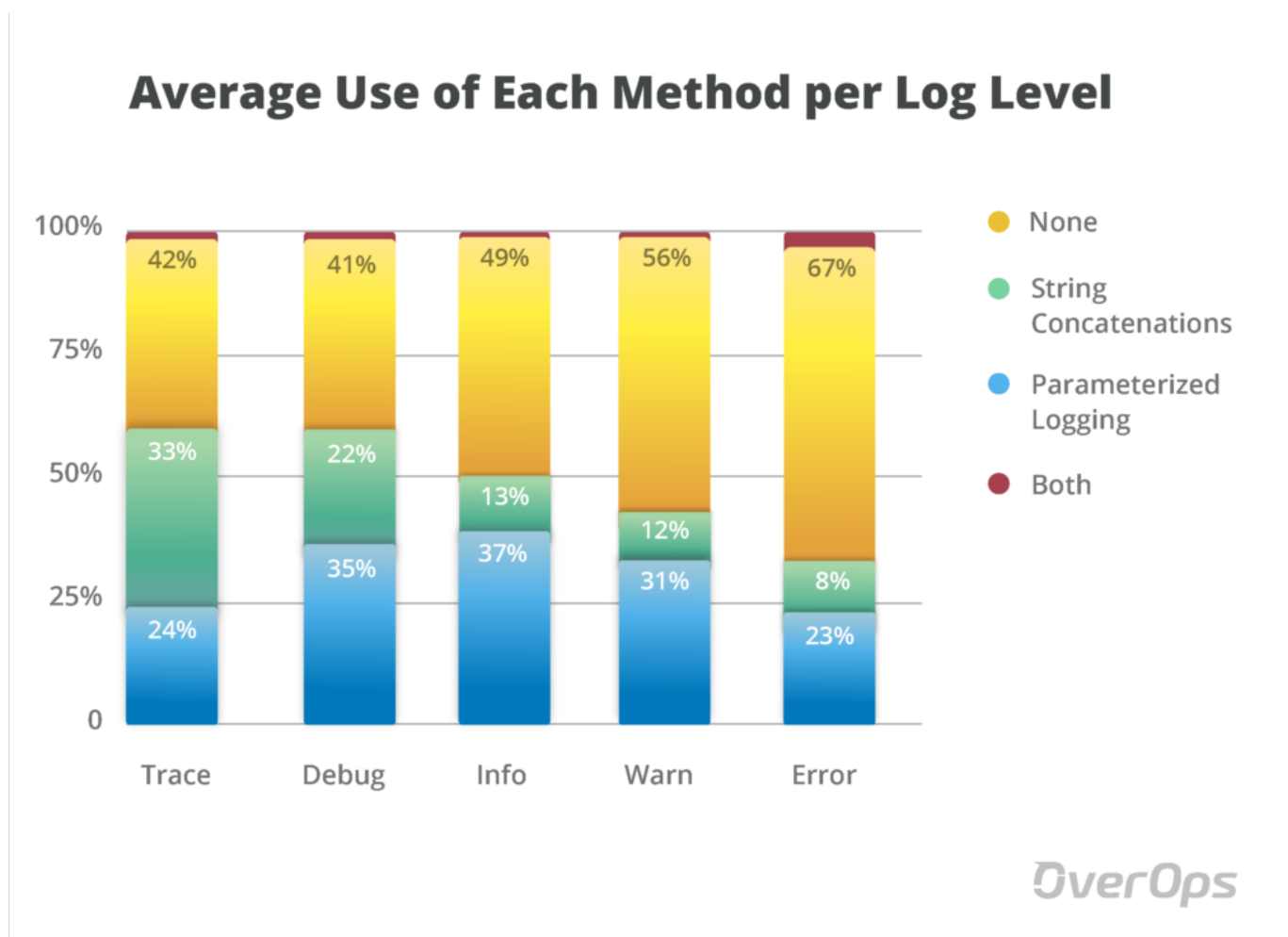
This method helps clean up the code, but what's even better is that the parameters are evaluated only if the statement is needed. Using this messaging format eliminates the need to call `isDebugEnabled()`, since the variables won't convert unless they're called.

Going back to our DEBUG log level example, it means that when we run our application in production the variables included in the DEBUG log message will not be converted to string, making this method much more efficient than String concatenation.

Diving into the Log Level

Going back to our data crunch, now that we know which method might be better for low production overhead, we've decided to see if it affects the choice of which one to choose per log level.

We know that the amount of variables sent to the log is pretty low, and the average number runs between 0.8 variables for an average DEBUG level message, and 0.4 variables for ERROR. So while we're at it, we've decided to see which percentage of statements are sent in each method to each log level.



Average use of each method per log level in GitHub's top Java projects

Many developers use String concatenations in their local environment (TRACE, DEBUG, INFO), with 33% statements for concatenations in TRACE and 22% statements in DEBUG level. Parameterized logging is the popular choice in INFO log level, with 37% of statements. The clear pattern here is that as we move towards production, the use of either method goes down.

Or in other words, developers don't send out enough variables when it comes to their production environment. It doesn't mean they don't use their logs, since they might use Strings to monitor what's going on – but wouldn't you want to know more about when and why your code broke?

Keeping Track of the Logs

Searching through your log files for certain strings might feel like... searching for a needle in a log file. If you find yourself wasting hours or even days trying to debug using your logs, you'd be happy to know that there's a better way.

[OverOps](#) tells you when, where and why your code breaks in production. It lets you see the variables behind any exception, logged error or warning, without relying on the information that was actually logged.

It gives you the complete source code and variable state across the entire call stack for each event, error or exception – Even if it wasn't printed to the log file. OverOps also shows you the last 250 DEBUG, TRACE and INFO level statements that were logged prior to the error, in production, even if they're turned off and never reach the log file.

Make your logs better. [Click here to schedule a demo.](#)

Final Thoughts

As we've said before, there isn't a right (or wrong) way to write to your log files, but there are methods you should apply if you want to get the most out of it.

The most important thing is that you'll keep track of what's going on in your code so you'll be able to understand it. If not for your teammates, then for your future self that won't have to deal with riddles just to solve a bug.

Chapter 5

How Did We Get the Data? Google BigQuery

Our data crunch cookbook is based on GitHub's top 40,000 repositories, which we examined, analyzed and broke down to understand how to use Java logging in production.

First stop, getting the Java projects out of GitHub's top 400,000 projects by stars:

```
1 SELECT id,  
2     content,  
3     sample_repo_name,  
4     sample_path  
5 FROM [fh-bigquery:github_extracts.contents_top_repos_top_langs]  
6 WHERE REGEXP_EXTRACT(sample_path, r'\.([^\.]*)$') IN ('java')  
7
```

This basically gave us all repos with their respective Java source files.

New Query ?

```

1 SELECT id,
2       content,
3       sample_repo_name,
4       sample_path
5 FROM [fh-bigquery:github_extracts.contents_top_repos_top_langs]
6 WHERE REGEXP_EXTRACT(sample_path, r'\.([\.\.]*)$') IN ('java')

```

Allow Large Results ✕

Destination Table: java-log-levels-usage:java_log_level_usage.top_repos_java_contents ✕

Overwrite Table ✕

RUN QUERY ▾

Save Query

Save View

Format Query

Show Options

Query complete (11.2s elapsed, 134 GB processed)

And... only took 11.2s, to process 134 GB. W00t. BigQuery magic right there.

Next step, getting the contents of those source files, excluding ones with android packages and android repo names. We noticed there were some Arduino projects, so we excluded those as well:

```

1 SELECT SPLIT(content, '\n') line,
2       id,
3       sample_repo_name
4 FROM [java_log_level_usage.top_repos_java_contents]
5 WHERE NOT (content CONTAINS 'import android') AND
6       NOT (sample_repo_name CONTAINS 'android') AND
7       NOT (sample_repo_name CONTAINS 'Android') AND
8       NOT (sample_repo_name CONTAINS 'arduino') AND
9       NOT (sample_repo_name CONTAINS 'Arduino')
10

```

7.2s, for 12.8 GB.

Hold on tight, the final query is a bit crazy. Including all the regex to extract the count for the different logging levels:

```

1 SELECT sample_repo_name,
2       SUM(CASE WHEN (REGEXP_MATCH(line, r'.*(((LOGGER|Logger|logger|LOG|Log|log)[.]trace)|((Level|Priority)[.]
   (TRACE|TRACE_INT|X_TRACE_INT)).*')) THEN 1 ELSE 0 END) trace_count,
3       SUM(CASE WHEN (REGEXP_MATCH(line, r'.*(((LOGGER|Logger|logger|LOG|Log|log)[.]info)|((Level|Priority)[.]
   (INFO|INFO_INT)).*')) THEN 1 ELSE 0 END) info_count,
4       SUM(CASE WHEN (REGEXP_MATCH(line, r'.*(((LOGGER|Logger|logger|LOG|Log|log)[.]debug)|((Level|Priority)[.]
   (DEBUG|DEBUG_INT)).*')) THEN 1 ELSE 0 END) debug_count,
5       SUM(CASE WHEN (REGEXP_MATCH(line, r'.*(((LOGGER|Logger|logger|LOG|Log|log)[.]warn|warning)|((Level|Priority)[.]
   (WARN|WARN_INT|WARNING|WARNING_INT)).*')) THEN 1 ELSE 0 END) warn_count,
6       SUM(CASE WHEN (REGEXP_MATCH(line, r'.*(((LOGGER|Logger|logger|LOG|Log|log)[.]error)|((Level|Priority)[.]
   (ERROR|ERROR_INT)).*')) THEN 1 ELSE 0 END) error_count,
7       SUM(CASE WHEN (REGEXP_MATCH(line, r'.*(((LOGGER|Logger|logger|LOG|Log|log)[.]fatal)|((Level|Priority)[.]
   (FATAL|FATAL_INT)).*')) THEN 1 ELSE 0 END) fatal_count,
8       SUM(CASE WHEN (REGEXP_MATCH(line, r'.*(((LOGGER|Logger|logger|LOG|Log|log)[.]severe)|((Level|Priority)[.]
   (SEVERE|SEVERE_INT)).*')) THEN 1 ELSE 0 END) severe_count,
9       SUM(CASE WHEN (REGEXP_MATCH(line, r'.*(((LOGGER|Logger|logger|LOG|Log|log)[.]config)|((Level|Priority)[.]
   (CONFIG|CONFIG_INT)).*')) THEN 1 ELSE 0 END) config_count,
10      SUM(CASE WHEN (REGEXP_MATCH(line, r'.*(((LOGGER|Logger|logger|LOG|Log|log)[.]fine)|((Level|Priority)[.]
   (FINE|FINE_INT)).*')) THEN 1 ELSE 0 END) fine_count,
11      SUM(CASE WHEN (REGEXP_MATCH(line, r'.*(((LOGGER|Logger|logger|LOG|Log|log)[.]finer)|((Level|Priority)[.]
   (FINER|FINER_INT)).*')) THEN 1 ELSE 0 END) finer_count,
12      SUM(CASE WHEN (REGEXP_MATCH(line, r'.*(((LOGGER|Logger|logger|LOG|Log|log)[.]finest)|((Level|Priority)[.]
   (FINEST|FINEST_INT)).*')) THEN 1 ELSE 0 END) finest_count,
13      SUM(CASE WHEN (REGEXP_MATCH(line, r'.*(Level[.]ALL).*')) THEN 1 ELSE 0 END) all_count,
14      SUM(CASE WHEN (REGEXP_MATCH(line, r'.*(Level[.]OFF).*')) THEN 1 ELSE 0 END) off_count
15 FROM   [java_log_level_usage.top_repos_java_contents_lines_no_android_no_arduino]
16 GROUP BY sample_repo_name;

```

16.9s, for 18.1 GB.

Then, we added some regex magic and pulled out all of the log lines:

```

1 SELECT *
2   FROM [java-log-levels-usage:java_log_level_usage.top_repos_java_contents_lines_no_android_no_arduino]
3  WHERE REGEXP_MATCH(line, r'.*((LOGGER|Logger|logger|LOG|Log|log)[.]
4         (trace|info|debug|warn|warning|error|fatal|severe|config|fine|finer|finest)).*')
5         OR REGEXP_MATCH(line, r'.*((Level|Priority)[.]
6         (TRACE|TRACE_INT|X_TRACE_INT|INFO|INFO_INT|DEBUG|DEBUG_INT|WARN|WARN_INT|WARNING|WARNING_INT|ERROR|ERROR_INT)).*')
7         OR REGEXP_MATCH(line, r'.*((Level|Priority)[.]
8         (FATAL|FATAL_INT|SEVERE|SEVERE_INT|CONFIG|CONFIG_INT|FINE|FINE_INT|FINER|FINER_INT|FINEST|FINEST_INT|ALL|OFF)).*')

```

Now that we had the data, we started slicing it up. First we filtered out the number of variables per log level:

```

1 SELECT sample_repo_name
2        ,log_level
3        ,CASE WHEN parametersCount + concatenationCount = 0 THEN "0"
4              WHEN parametersCount + concatenationCount = 1 THEN "1"
5              WHEN parametersCount + concatenationCount = 2 THEN "2"
6              WHEN parametersCount + concatenationCount = 3 THEN "3"
7              WHEN parametersCount + concatenationCount = 4 THEN "4"
8              WHEN parametersCount + concatenationCount >= 5 THEN "5+"
9        END total_params_tier
10       ,SUM(parametersCount + concatenationCount) total_params
11       ,SUM(CASE WHEN parametersCount > 0 THEN 1 ELSE 0 END) has_parameters
12       ,SUM(CASE WHEN concatenationCount > 0 THEN 1 ELSE 0 END) has_concatenation
13       ,SUM(CASE WHEN parametersCount = 0 AND concatenationCount = 0 THEN 1 ELSE 0 END) has_none
14       ,SUM(CASE WHEN parametersCount > 0 AND concatenationCount > 0 THEN 1 ELSE 0 END) has_both
15       ,COUNT(1) logging_statements
16       ,SUM(parametersCount) parameters_count
17       ,SUM(concatenationCount) concatenation_count
18       ,SUM(CASE WHEN isComment = true THEN 1 ELSE 0 END) comment_count
19       ,SUM(CASE WHEN shouldNeverHappen = true THEN 1 ELSE 0 END) should_never_happen_count
20   FROM [java-log-levels-usage:java_log_level_usage.top_repos_java_log_lines_no_android_no_arduino_attributes]
21  GROUP BY sample_repo_name
22         ,log_level
23         ,total_params_tier

```

Then calculated the average use of each tier. That's how we got the average percent of total repositories statements.

```

1  SELECT total_params_tier
2      ,AVG(logging_statements / total_repo_logging_statements) percent_out_of_total_repo_statements
3      ,SUM(total_params) total_params
4      ,SUM(logging_statements) logging_statements
5      ,SUM(has_parameters) has_parameters
6      ,SUM(has_concatenation) has_concatenation
7      ,SUM(has_none) has_none
8      ,SUM(has_both) has_both
9      ,SUM(parameters_count) parameters_count
10     ,SUM(concatenation_count) concatenation_count
11     ,SUM(comment_count) comment_count
12     ,SUM(should_never_happen_count) should_never_happen_count
13 FROM (SELECT sample_repo_name
14         ,total_params_tier
15         ,SUM(total_params) total_params
16         ,SUM(logging_statements) logging_statements
17         ,SUM(logging_statements) OVER (PARTITION BY sample_repo_name) total_repo_logging_statements
18         ,SUM(has_parameters) has_parameters
19         ,SUM(has_concatenation) has_concatenation
20         ,SUM(has_none) has_none
21         ,SUM(has_both) has_both
22         ,SUM(parameters_count) parameters_count
23         ,SUM(concatenation_count) concatenation_count
24         ,SUM(comment_count) comment_count
25         ,SUM(should_never_happen_count) should_never_happen_count
26     FROM [java-log-levels-
usage:java_log_level_usage.top_repos_java_log_log_level_usage.no_android_no_arduino_attributes_counters_with_params_count]
27     GROUP BY sample_repo_name
28             ,total_params_tier)
29 WHERE total_repo_logging_statements >= 100
30 GROUP BY total_params_tier
31 ORDER BY 1,2

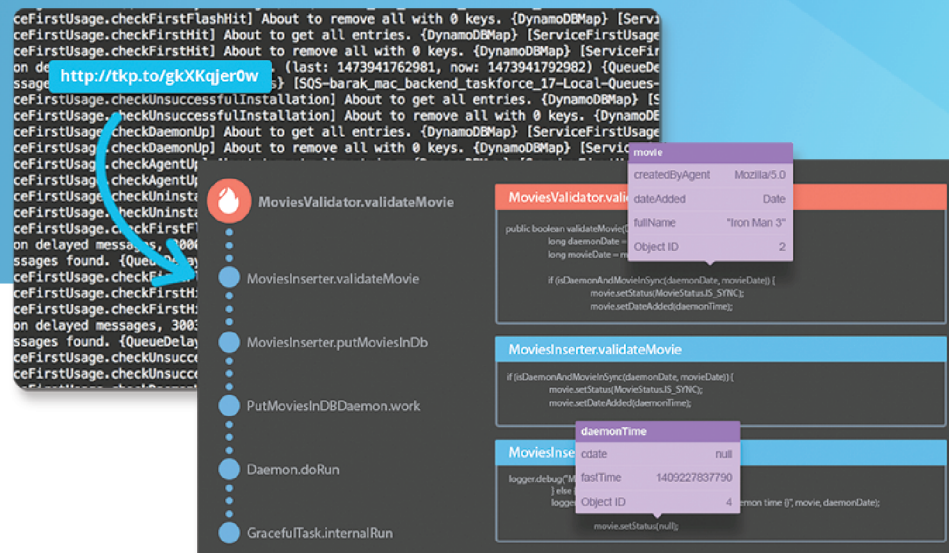
```

The final spreadsheet with all the data and some additional calculations is available [right here](#), along with the [raw data file](#).

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



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



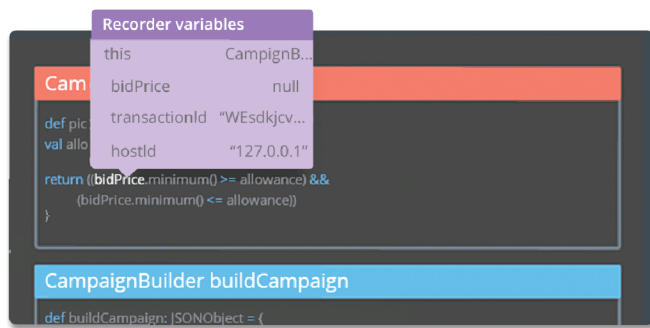
99%
Reduction



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

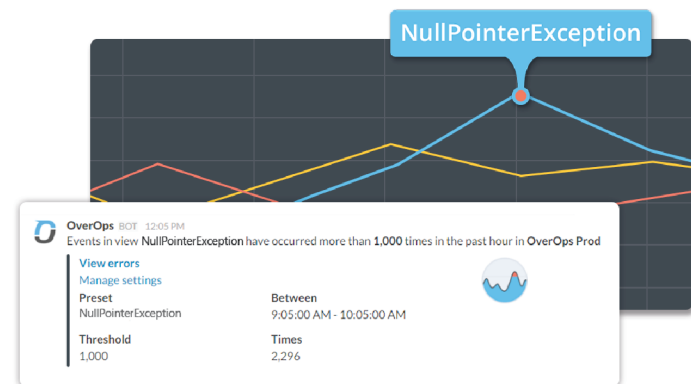
Code and Variable State

- **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, application, 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



Full code and variable state to immediately reproduce any error.

No need to manually reproduce issues by searching for information in logs. Reduce MTTI by 90%+



Proactive detection of all new and critical errors.

New issues are detected and routed to the right developer vs. discovered by users. Each error receives a unique code fingerprint unique it across the app.



Noise reduction: deduplication of all log errors and exceptions.

No need to sift through logs to search for errors. OverOps detects all events as distinct entities at the JVM level vs. "reverse-engineer" them from text logs.



<1% overhead in production

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



No change to code or build

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



PII Redaction. Cloud & On-Premises

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

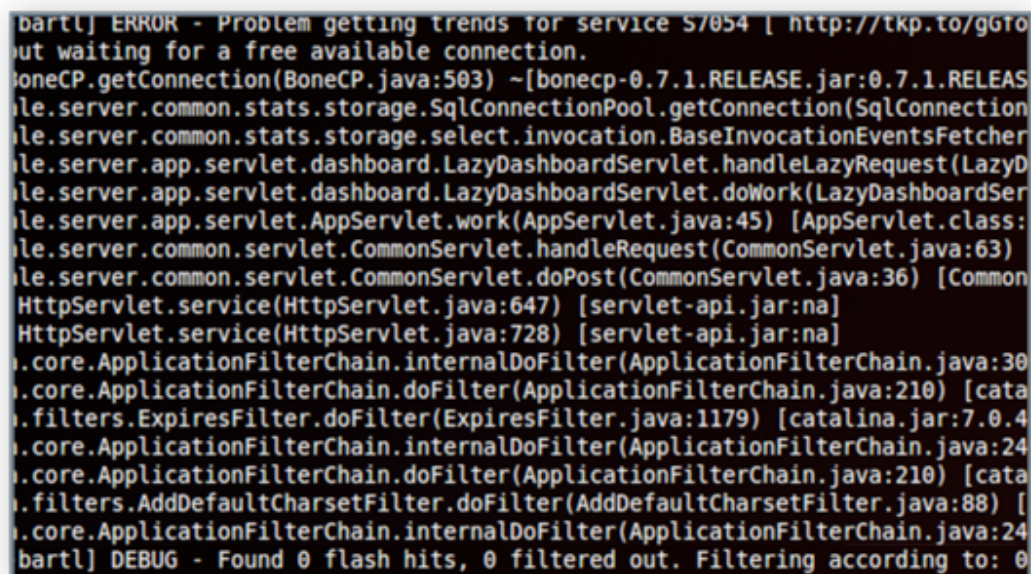
Integrations:

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

Final Thoughts

We all use log files, but it seems that most of us take them for granted. With the numerous log management tools out there we forget to take control of our own code – and make it meaningful for us to understand, debug and fix.

We've grown accustomed to digging in log files to find out how our applications behave in production. But it doesn't have to be like this. Generally speaking, log files suck. Tons of unstructured text, sometimes the information you're looking for wasn't even logged, and then there's the debugging paradox, adding logging statements and hoping the error that sent you on that chase would happen AGAIN. But... Java debugging doesn't have to look like this:



```

bartl] ERROR - Problem getting trends for service 57054 [ http://tkp.to/ggTo
out waiting for a free available connection.
boneCP.getConnection(BoneCP.java:503) ~[bonecp-0.7.1.RELEASE.jar:0.7.1.RELEAS
le.server.common.stats.storage.SqlConnectionPool.getConnection(SqlConnection
le.server.common.stats.storage.select.invocation.BaseInvocationEventsFetcher
le.server.app.servlet.dashboard.LazyDashboardServlet.handleLazyRequest(LazyD
le.server.app.servlet.dashboard.LazyDashboardServlet.doWork(LazyDashboardSer
le.server.app.servlet.AppServlet.work(AppServlet.java:45) [AppServlet.class:
le.server.common.servlet.CommonServlet.handleRequest(CommonServlet.java:63)
le.server.common.servlet.CommonServlet.doPost(CommonServlet.java:36) [Common
HttpServlet.service(HttpServlet.java:647) [servlet-api.jar:na]
HttpServlet.service(HttpServlet.java:728) [servlet-api.jar:na]
.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:30
.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:210) [cata
.filters.ExpiresFilter.doFilter(ExpiresFilter.java:1179) [catalina.jar:7.0.4
.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:24
.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:210) [cata
.filters.AddDefaultCharsetFilter.doFilter(AddDefaultCharsetFilter.java:88) [
.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:24
bartl] DEBUG - Found 0 flash hits, 0 filtered out. Filtering according to: 0

```

[There's another way.](#)

This eBook by inspired by Google's developer advocate [Felipe Hoffa](#), and his [tabs vs spaces post](#).

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