OverOps

AN OVEROPS WHITE PAPER

Net New Machine Data and Code-Aware Insight with OverOps

SECTION ONE: THE PROBLEM

Data differentiates

Data has evolved to become the ultimate tool to differentiate from your competition. Whether it be IoT, social or even just the transactional data found in our persistent stores, capturing all of it, extracting insights from it and using it to fuel business decision is the new norm for the modern enterprise. And typically, the more data you have the more insight you can gather. We rarely see data exhaust anymore as we move to instrument everything. But this isn't completely true.

Ironically, the software we deploy in these modern applications is also leaving massive amounts of information on the data center floor. There is a layer of data that nobody has really thought to capture because it seemed impossible. It's the data that is created by the actual execution of the lines of code that power just about everything we do. It's the data that's inside the engine, not just the exhaust we choose to emit.

Humans are human, logs are logs

For decades we have relied upon log data as our main source for gathering insight from our applications. They are incredibly useful but are not without limitation. While log files help identify when something has gone wrong, they only provide limited insight into "what" has actually happened. The root of this problem isn't the logging mechanism; it is the fact that we

rely on a manual, human process to extract the data we need from our applications and services to gain insight into "what" has actually gone wrong.

There are two consumers of log files, humans and computers. The human group is typically a developer who needs log files to identify where something went wrong in their code. They'll use logging statements to extract a few variables from the moment of execution and hope this is enough information to glean insight into the problem. For some of the best developers, they "know" their code and it works pretty well, but what about junior programmers and new members of the team? What about third party code and legacy code?

To illustrate this fundamental defect, let's consider a log statement which surfaced because of a failed call into an Amazon Web Service. This caused an exception to be thrown and the developer "caught" this using the following:

```
"ERROR: Failed to complete AWS operation APPROVE_CREDIT
for user jim@acme.com with error code AMAZON_FAIL"
```

The first issue, it is nearly impossible to understand just by reading the statement which part of it is the template and which are the variables that were used within, so how are we to apply consistent structure? Also, without further context you cannot discern the location in the code where this happened or which transaction was involved.

In this case, we are lucky that the developer chose to actually "catch" this exception. Without this manual check, this error would have gone undetected. These uncaught or not logged exceptions could be especially dangerous because you'll never know what you don't know and this could have had significant impact on your customers.

The second consumer of log files are machines. Included in this group are the log analyzer tools, the shell scripts built by sys admins and the grep functions we use to sift through these massive files. This group lives and dies by any sort of structure they can extract from the log. Unfortunately, when it comes to a common approach to logging application issues, we still rely on the humans to write logging statements and even with the most complete and effective best practices, the process will still fall short. What's worse, without any sort of common language and structure, it is nearly impossible to categorize events. What's worse, is that current log analysis tools have no way to deduplicate these items other than rely on full text of an event and there is no way to discern similar code methods based on where they are executed in the code base.

In summary, code does not comment itself and certainly doesn't log itself either (or does it?). There are four main challenges with the human element involved in our logging process:

1. *Log statements are manual and shallow*
   Logging frameworks are a core part of every programming language because they allow developers to record information from within an app to a file so they can use it for some purpose, typically troubleshooting. Herein lies the problem. The developer manually chooses where log statements are used and they also decide what information will be sent to the log file. This information is typically shallow, providing just a few variables that are often just representations of values or shorthand of what needs to be communicated. It is a tiny portion relative to the complete context of what is happening in the app at that moment. We'll talk more about verbosity in a bit.

2. *Searching through log files and classification of entries is tricky*
   Log files lack much structure and in order to sift through them you are required to use grep, grok, regex, etc. or other more advanced tools. Regardless of how you process them, you are still challenged by their shallow nature and your ability to find new errors or distinguish one error form the next is difficult. Classification and de-duplication of log entries is nearly impossible. Further, it is nearly impossible to correlate a log statement with a specific version of code.

3. *Log files provide limited visibility*
   Related to the manual nature of a log file, they are also not comprehensive across every error and exception. You only get what you log and further, you only get what you catch. Log files give you no visibility into uncaught or swallowed exceptions.

4. *Tracking down and tracing errors in microservices is impossible*
   As the world transitions to microservices, we are also introducing complexity. What is an application in this new world and how do we troubleshoot errors across multiple services? What is the execution stack in this new world? Some frameworks have been introduced to trace events across services, but they are good for performance tracking only.

Log files have been with us for ages and are valuable but how much time have we spent sifting through them to find what we need? How much time have we spent deciphering log information in order to troubleshoot a complex problem when we could have spent that time delivering new features? How often do we rollback a release, add logging verbosity, deploy a hotfix, and wait for the issue to happen again? How much can we rely on a log file to inform us how safe it is to promote code? Do we really know?

It's not just humans, It's also the machines

To further complicate this challenge, there is a set of logging levels that we use to determine what log statements might be executed in what environment. We have created this framework so that we can save on system performance. However, the more verbose your logging is, the more significant impact it will have on the performance of your application as it wastes CPU

cycles and slows down systems because of the time needed to collect additional information or display additional logging details.

Verbose logging also increases the size of the log file which can make it difficult to find the signal in the noise. (NOTE: And in the world of microservices, this issue is exaggerated as we are creating an exponential amount of application exhaust across multiple, disconnected distributed services that all need to be aggregated and then searched through to gain insight into where our systems are failing.) And again, without structure, this becomes intolerable.

To effectively address some of these issues, developers use log levels that can be turned on/off depending on which environment code is running. In production, you don't need everything and want to optimize for performance. The basic logging levels we define include:

- ERROR/FATAL – At this level, something terribly wrong has happened and the issue must be resolved immediately. Some examples: an unavailable database or a critical service is unresponsive. And even ERROR is often used to warn or present lower level priority issues. There is little or no standardization in most organizations. Is an ERROR an error?
- WARN – At this level, the system might continue, but with caution as the system can tolerate the issue and justify continuing as there might be a workaround. Some examples might be a lookup has failed but cached values can be used or a queue is near full.
- INFO – typically used to communicate information about an event to inform the business logic . some examples might include a statement when an order has been placed or fulfilled or when an event is sent to an external system.
- DEBUG – This is the world of the developer and typically the most valuable information used in the troubleshooting process. Typically, they will include a few variables and a comment.
- TRACE – This is typically very detailed information that is intended for development use only and is often confused in use with DEBUG. They should be temporary information that should be stripped once code is committed. The distinction between DEBUG and TRACE is the most difficult to discern.

With all this granularity, you would think we could get pretty efficient in our use of logging; however, it's widely accepted that developers generally use DEBUG and INFO only.

Logging frameworks are a foundational element of every language, but they were created over thirty years ago, back when the cost of CPU was severely limiting . And more than anything, performance is the critical gating factor for the amount of application exhaust we create.

There has to be a better way

This logging stuff all seems so very thirty years old. We've advanced nearly every part of the software supply chain but log files haven't changed much at all since they were first

introduced. We've created tools and frameworks that allow us to use them more effectively, but the fundamental challenges noted above still remain.

So, what if we took a step back and looked at this problem in a new way. What if we could intelligently capture and structure the entirety of information related to an error or exception. What if this information was code-aware and had contextual insight into the whole application or service it was found in? This isn't just about getting more data, **it is about getting the right data and delivering in a more useful format.**

SECTION TWO: OVEROPS DATA

OverOps developed a unique approach to gathering machine data and it changes the way we think about log files and how we use them to both troubleshoot and to derive the overall quality of an application or service. OverOps combines static and dynamic analysis to collect complete contextual data for every error and exception thrown in an application with minimal performance impact, securely and without any requirement to modify code. It is code-aware and delivers net new and structured machine data that provides granular detail about every error, its related application and the environment in which it was found.

This section outlines the data and the methods we use to extract it from your applications and services and it is organized into three parts. We start with a description of the unique elements OverOps uses. We then talk through the explicit data we are able to extract, process and create from these elements and then provide an overview of how this data can be used.

Code Graphs and Snapshots NOT log files

First and foremost, OverOps is completely independent of log files. In fact, our technology allows us to precede the creation of a log file entry so we can augment them. As we will speak to later, this allows us to place links to our platform in a log file so they can be connected to the OverOps data.

There are two foundational elements to our data collection strategy, a code graph and a snapshot. Deployed using a combination of an agent, collector and analysis server (see our architecture doc for more details), these foundational elements work together to not only extract data but to do so without considerable impact on performance or development.

A **Code Graph** is an index of all possible execution paths within explicit application code and is unique to every build or release. It's a map of the application or service. OverOps creates this secure code graph every time code is loaded from a company's "build" pipeline into a live software virtual machine (SVM) - in this case a JVM or the .NET CLR. The code graph is a

unique representation of that bytecode that can also be used to map every build and tie back to a build number.

To build the code graph, the code is analyzed to create a map of where and why the code may log or encounter exceptions (i.e. break), and it is used to determine an optimal strategy to collect information from the code running in the SVM. This map of all execution paths not only provides the critical context for each error, but it also allows OverOps to predict when an event might happen so it can optimize performance of this function and even catch the uncaught.

And as with all things in the OverOps Platform, a code graph is built with security and data privacy as a primitive goal. OverOps uses a redaction technique called "One-way hashing" to make sure that the bytecode we use can never be reverse-engineered into human readable source code by us, or anyone else. The code graph is an abstract graph of connections between variables that lacks knowledge of the data types or values themselves. It is also this code graph that allows us to proactively redact private information when we collect data. More on these security aspects later.

A **Snapshot** is the physical collection of the machine data at the moment an error is logged or an exception is thrown. This is the net new data that has never been captured before. With the code graph as a map, each snapshot is analyzed and processed to extract code, contextual values of variables and the state of the virtual machine and physical host. It is the dynamic element of this process and it is what captures the data that flows through the code pipeline. Some of the data we collect directly from each snapshot includes the following:

- *Variable State Across Entire Call Stack*
  OverOps extracts the value of every variable across the entire execution stack so you have complete context across every function involved in every issue. As opposed to logs where you will only get what you choose to place in a logging statement and generally this is just a few items, a fraction of what's really happening.

- *Execution Pipeline Source Code*
  OverOps intelligently reconstructs full source code from the bytecode that was captured in the execution pipeline. The source code is never exposed outside the system until an authorized user (your developer) needs to see it. OverOps also overlays the full stack variable state on top of the source code to make it easy to inspect what has happened in code. Without OverOps, you might waste time finding the right code versions, getting access, checking it out and then reconstructing the code pipeline.

- *Memory State*
  OverOps captures data 10 layers deep into the heap and presents this to you in an easy to consume format in the context of the error or exception so you can can readily identify the objects that are filling up memory. Without OverOps you would

have to be manually collecting this information, and for a production error getting this information is typically impossible.

- *State of Garbage Collection*
  OverOps provides the state of your Garbage Collection when an error occurs so you can determine if they are taking too long or if too much time is being spent in a pause and if you need to readjust your garbage collection strategy. You can only try to recreate these issues in dev and can typically get this in production.

- *Operating System, Environment Variables and System Properties*
  OverOps captures the operating system type, version number, number of processors and system load percentages as well as a complete list of all environment variables /system properties so you can detect related issues to the machine (or vm or container) the application is running. Visibility into these items might be easy in development, but again, in staging and production this is often difficult information to capture.

- *Active Threads*
  OverOps also provides with a complete list of all threads, their status and timing so that you can quickly identify thread contention or deadlock issues. While relatively straightforward to identify these in development, this is near impossible to obtain in production.

While each of these items can be extremely valuable, this isn't all the data that OverOps can capture. As we will see in the next section, there is a much deeper layer of contextual insight that can be gathered from snapshots because of the unique "code-aware" approach to gathering this data.

The benefits of the OverOps code graph: being "code-aware"

The code graph not only allows for performance gains, but it is also critical in the collection of data from your applications and services. It is what makes OverOps code-aware, allowing us to know what has happened and sometimes what hasn't happened. This deep level of insight is completely unique to OverOps and it also allows us to understand the structure for each event. It allows us to treat machine data as code.

The code graph allows OverOps to deliver a lot of net new insight, but the most valuable benefit is it allows us to **identify uncaught and swallowed exceptions**. The code graph provides the intelligence to allow OverOps to understand when it should capture an event that a developer never thought to capture or where they may have stubbed in a response and never came back and addressed it. These are issues that were once impossible to identify and are often the most painful production issues to deal with. They simply aren't in log files so you

might never chase them down until it is too late after a customer complains or a business indicator slips.

Being code aware also allows OverOps to treat the code pipeline execution data as code. It **provides schema and structure** to investigate what is happening so we can apply "schema" to the stream and deliver this as rich structured output to your developers and other tools that can use the OverOps data. The stream of information typically written to a log file is limited in form and, unless there is a strict best practice, is widely varied from developer to developer. (might we add, they are also extremely limited in valuable data/insight). Events in a log file are usually formatted by a logging framework and are a schema-less string representation of an event, with limited information stored in plain text. They are difficult to structure and classify and often lost within the noise of millions of messages.

Understanding code before it is executed also allows OverOps to deliver all this value with only a **minimal impact on application performance**. The code graph allows the platform to proactively know an error or exception is going to be encountered so it just acts without having to inform the SVM and slow it down. This ultimately means that OverOps can cover and provide value in every environment, from dev to test to staging and especially, in production. Without this critical capability, capturing this information would require you to stop the SVM and effectively kill your application performance.

Finally, the code graph allows OverOps to **distinguish one event from another** more easily because it knows where in the code an error or exception was encountered. In a log file, one error will often look very much like another and even with a full stack trace it may take time to sort out differences. This capability along with an ability to create a unique digital fingerprint of an event, ultimately allows OverOps to **effectively deduplicate events**.

The code graph is a map that provides an understanding of what was meant to be executed within the code and using this in context of the code execution pipeline is what allows OverOps to extract hidden or difficult to ascertain data, including:

- *Deployment/Build/Release Information*
  The code graph for each build is unique and it allows OverOps to map every error and exception it captures to a particular build. This can be used to track down the pull request and person associated with the event and if you look at all events related to a build, it can help determine the overall quality of a release. This information is not found in a log file unless a developer chooses to write it.

- *Previous 250 Log Statements (including DEBUG in prod)*
  While collecting log statements seems mundane, this is very different. Since we have the code graph, we can identify and collect log statements that may not actually be executed by the SVM because of verbosity settings. OverOps knows what the

developer meant to log so we can collect all warnings, debug/trace in all environments, even production, regardless of whether they were written to the log or not. This gives you developer level insight in production.

- *Entry Points*
  A function can fail for many reasons but getting insight into all the different entry points that may have caused the issue gives you better context into not just that event but others that have caused the same error. More importantly it will help you track back the origination of an end-to-end business transaction. The code graph in OverOps allows you to gain benefit from this broad approach and helps you recognize patterns that have resulted in the error. In order to do this with a log file would take some serious work.

- *Error Type*
  OverOps code graph is intelligent enough to proactively categorize database, network, JVM, HTTP and AWS errors before they are ever encountered. Beyond these standards, it can be customized to categorize by any package or library. With a log file, you could get two errors in the same method and not be able to discern if it was a functional error or system related.

OverOps can collect a lot of information that was previously impossible to capture and can be incredibly useful to the developer or to operations teams. Using this data in the context of a single event can certainly help with troubleshooting but looking at all errors in a build or by application opens up a new world of use cases that ultimately allow us to deliver more reliable software and establish a culture of accountability in our organizations. This is ultimately made possible by the static and dynamic analysis that OverOps provides.

The digital fingerprint of an error or exception

With the code graph and all the data that OverOps is able to capture, it can also create a unique digital fingerprint of every event. And with unique signatures, it can analyze the event in context of other events, applications and releases to create net new metadata for every event.

As noted in the last section, the code graph enables OverOps to help deduplicate events based on where in the overall application the code is being executed. With fingerprints, OverOps can further **deduplicate and count events based on unique signature**. This capability enables OverOps to selectively capture complete context for an individual error. The platform is intelligent to know if it needs to collect every incident or throttle/selectively choose over time what it collects and maintain only a count of the number of times this has happened. This approach has two side effects, it improves overall performance and reduces the amount of information that has to be stored, making the entire platform more efficient.

Signatures also contribute to the data we collect for each event as they allow OverOps to understand the nature of each error/exception. Some of the additional items OverOps derives include:

- New/Reintroduced Classification
  With a fingerprint of an event we can compare it against every other event and determine with high probability if it is new. If we combine this with the associated release number we can identify if it is a regression as well. Many use this to understand the overall quality trends of software over various releases.

- Frequency and failure rate
  The fingerprint allows us to count how many times this explicit error has happened which is the frequency. And with the code graph, we know the failure rate of the errors as well as we know how many times the error happens epr run through this section of code. This information helps us focus on issues that are causing the most pain..

- First seen/Last seen
  With a fingerprint of an error we can also determine when it was first introduced and when it was last seen. This has significant value when you use OverOps data to investigate the impact of the event or the overall quality of a release or an application in whole.

The digital fingerprint of every error and exception is only possible with OverOps and extremely useful to classify and define focus for your teams. It is a foundation on which you can ensure promotion of quality code and to create a culture of accountability within and beyond your organization.

Using OverOps Data

Over the past three sections, we outlined some of the key technical capabilities found in OverOps and the net new machine data we extract from your applications and services. This wealth of data provides deep contextual content around every error/exception and is ultimately structured and complete. In summary, the list of data includes:

- *Full Stack Variable State*
- *Execution Pipeline Source Code*
- *Memory State*
- *State of Garbage Collection*
- *Operating System, Environment Variables and System Properties*
- *Active Threads*
- *Deployment/Build/Release Information*
- *Previous 250 Log Statements (including DEBUG in prod)*

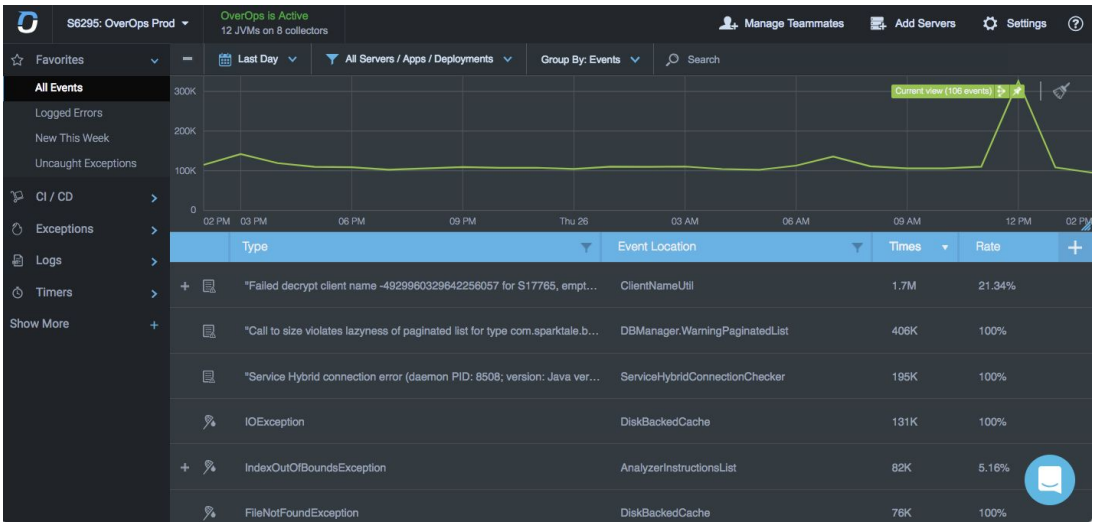
- *Entry Points*
- *Error Type*
- *New/Reintroduced Classification*
- *Frequency and failure rate*
- *First seen/Last seen*

No other tool is capable of capturing this data without code modification and without a significant performance impact, especially in production. The OverOps data is a big part of the story but equally important is how you access and use this data. From root cause automation to continuous reliability to AIOps, there are several business use cases that OverOps can fuel. Before we delve into these, let's review the technical capabilities found in OverOps that enables you to access this data.

The OverOps ARC (Automated Root Cause) Analysis UI

OverOps isn't just about the data, we also ship an intuitive and useful UI that allows you to gather high level insight but also allows you to classify and sift through all events and also to drill down into an explicit error to troubleshoot using this rich information.

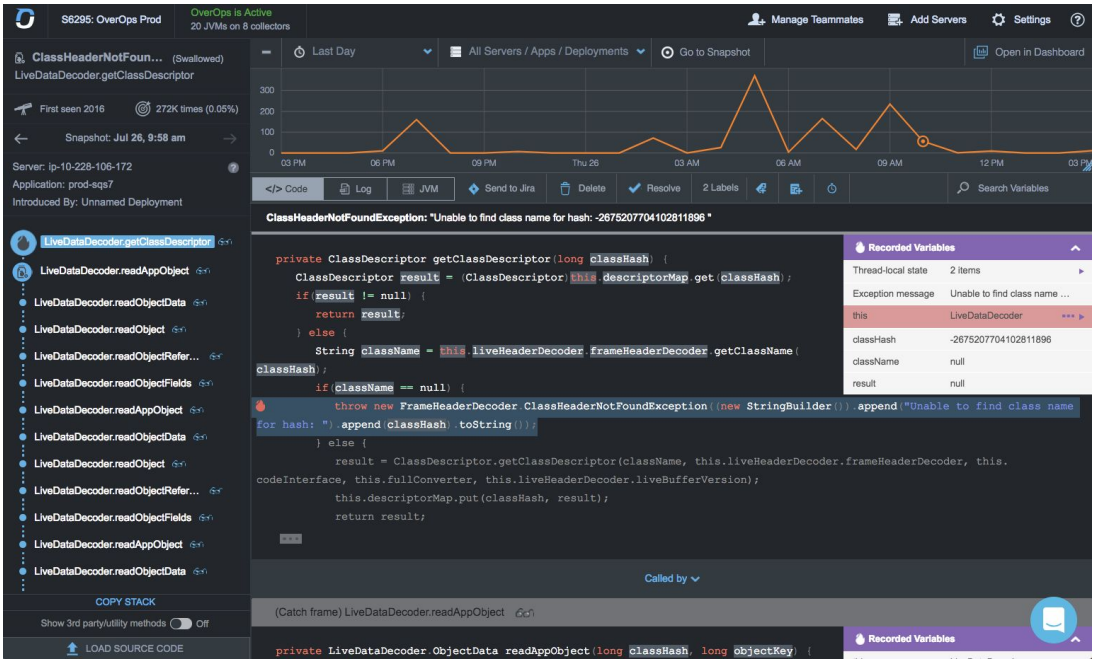
The Dashboard that comes standard allows you to sort all events based on a variety of parameters such as release, type, when they happened, where they happened and even by type. You can look at all your uncaught exceptions or view DB or HTTP errors. You can also sort by frequency and failure rate. With this powerful classification, you can use nearly every variable that is caught to sort through these errors and find exactly what you're looking for.



OverOps Dashboard and Classifications UI

From the dashboard you can investigate an error by double clicking on it and opening the Automated Root Cause (ARC) analysis screen for the event. Here you are presented with a

complete story for exactly what has happened. The ARC analysis screen is typically used by developers to investigate this information. You will see the entire stack with the moment in time of an error pointed out and as you mouse over the full source code, you will be able to investigate the state of every variable in the execution call stack. You can also check out al the log statements and the JVM state as well. It will show the complete set of data as collected for that event in an intuitive and useful format.

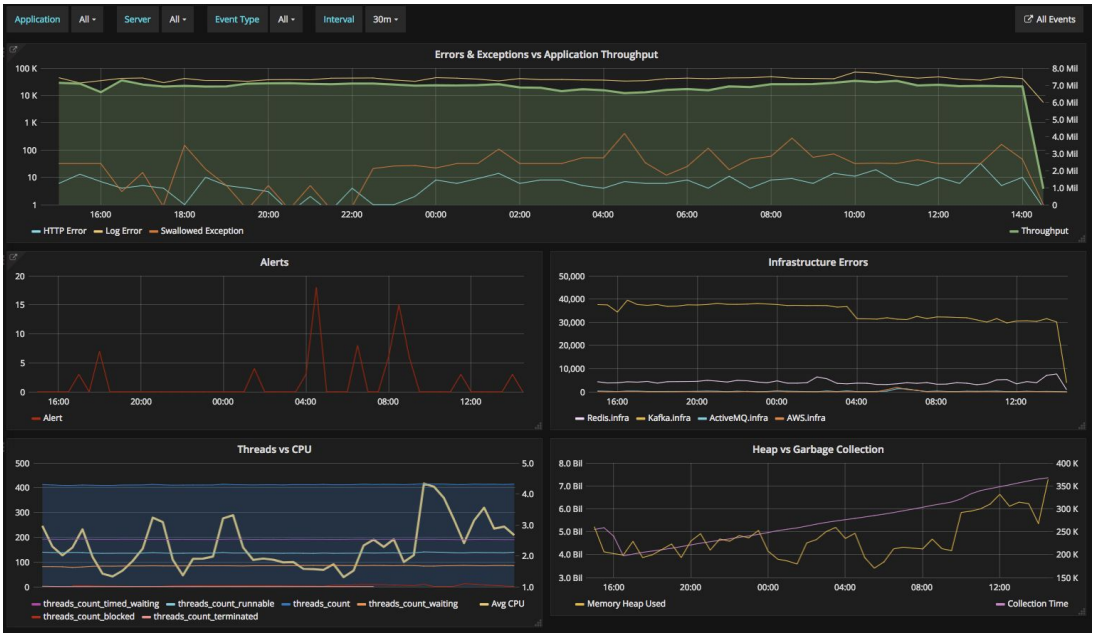


OverOps Automated Root Cause (ARC) Analysis UI

The OverOps API & Data Extracts

OverOps includes a powerful API that allows you to control the platform and also integrate and interact with the data in real time. The secure API is available and allows you to extract a single event or to look at all them in aggregate. This can be used for instance from a Jenkins plugin to query OverOps for quality metrics of a piece of code and regulate when it is safe to promote. OverOps also includes an ability to extract the data in aggregate via StatsD or directly to all your major tooling. Many use this interface to create their own metrics hubs or dashboards using Grafana, but this can also be used to feed metrics dashboards in Splunk, Datadog and others.

Reliability over time dashboard - provides insight into releases



Applications Reliability Dashboard in Grafana

OverOps Workflow Integration

The platform is designed to work with all the tools you already have and there are several explicit integration points beyond this API that allow you to insert it into your DevOps workflow. For instance, some of these integrations include can:

- *ARC Links for use in AppDynamics, Splunk, Dynatrace, etc…*
  As part of the processing of the snapshots, OverOps is present at the time a log file is written and can insert a link into the log file itself that will direct you to the OverOps ARC analysis screen that is associated to the error being logged. These links are present for both logs (where they may not even be an exception) or for exceptions (that may not even be logged - uncaught , swallowed). This is useful within any tool that bases their functionality off log files, but especially useful with tools like Splunk, AppDynamics and Dynatrace that help you know something has happened but only present limited information for what has actually occurred. OverOps is used in these situations to pick up where they leave off.

- *Ticketing with Jira*
  As part of deployment of OverOps, many organizations enable direct integration with Jira so that tickets can be created for certain events directly. With Jira integration, tickets are created with explicit details about each event and the links mentioned above can be placed into the description of the error. This approach greatly simplifies the research a developer would have to conduct to fix and close an issue. Further it is helpful to remedy the “cannot recreate” conversations between QA and dev.

- *Alerting & Email (and generic Webhook)*
  Another key point of integration is for alerting the right resources when certain events happen. OverOps allows you to configure direct integration with Slack, HipChat, PagerDuty, ServiceNow and your email system so that information is directed immediately to the right person. And if your alerting and workflow tool is not included in the standard integrations, there is a WebHook API to accommodate it.

- Custom workflow and extensions
  OverOps Extensions provides an AWS Lambda-based framework (and on-premises code as an option) for organizations to create their own custom functions and workflows based on the valuable OverOps data. So when an explicit event occurs, you can customize the actions that are taken. With open access to OverOps’ machine data and functional extensions, DevOps can enhance the entire software delivery supply chain to improve reliability of their applications and services, and avoid costly downtime.

Ultimately, OverOps takes a very open approach and philosophy so that this code-aware application data can be used within and with the enterprise tools that are already deployed.

SECTION THREE: REQUIREMENTS

Capturing this unique data across all environments, even production is not easy and has explicit requirements. The solution must be simple and not require manual manipulation. It must not have significant performance impact on the application it is monitoring. And finally, it must be secure. OverOps meets all three of these requirements.

- **No Code Modification or developer pre thought required**
  OverOps works at the software virtual machine (SVM) layer and uses the code graph to understand when an error is logged or an exception is thrown. There is no need for a developer to make any manual changes other than respecting the organizations best practices for logging and exceptions. There is no need to change the build process. This is the same design concept also allows OverOps to catch uncaught exceptions. It is also what allows Overops to capture this complete insight, not just the few variables the developer thought of.

- **No significant impact on application performance**
  As noted in a previous section, the code graph is a major innovation in the OverOps Platform. It not only provides contextual information for information gathering, but it also provides significant performance gains. Having a map of all execution paths allows OverOps to have the foresight into what is going to happen and eliminates the need for any "thought" during execution it knows when an error or exception will be thrown and captures it. Other approaches rely on native language calls that will slow the SVM. While this might suffice in dev, it is not acceptable in production environments. There are several other components in the overall architecture that work walong with the code graph to keep the application performance impact to always less than 3% of CPu and at steady state is typically under 2%.

- **Security first design mindset**
  Third and most importantly, security first is a mindset within the OverOps team. There are two main areas in which OverOps protects data. First, using static analysis via the code graph, the platform can proactively redact personal information when it is encountered in the code. For instance, if a social security number or password was to be captured in a snapshot, OverOps will automatically redact this information so that it is not disclosed. There are settings in the product to redact by a variable or field name or by using pattern recognition. OverOps also implements strict encryption policies of all the snapshot data and the code throughout the platform. This data is encrypted from the time it is captured until it is seen.

Finally, it is important to address **language coverage**. Currently, OverOps works with all JVMs and the .NET CLR, so it can cover Java, Scala, C# and other languages running on those SVMs. This is important to note in the context of these requirements because there are other

products that can capture *some* of the information OverOps can collect, but they do so at the cost of not meeting these basic requirements. It is the philosophy of OverOps to extend the capabilities to other languages but not before the platform meets these requirements and more importantly is extremely stable and a non-risk for your production applications.

EPILOGUE

For more information or to schedule a meeting with us, please visit overops.com.