{ eBook }

loggly

The Pragmatic Logging Handbook

Proven Best Practices for Instrumenting Your Applications for Maximum Insight

by Jon Gifford

CHIEF SEARCH OFFICER AND CO-FOUNDER, LOGGLY

Introduction

If you're responsible for keeping a net-centric application operational, you probably know that even the best code can fail when it meets the real world. In order to keep ahead of potential problems, you need to be able to understand why something is happening when your code fails.

Instrumentation, nicely defined on **Wikipedia,** refers to an ability to monitor or measure the level of an application's performance, to diagnose errors, and to capture informative messages about the execution of the application at run time. The output of your instrumentation goes to your application logs.

Of course, **how much to log** is an age-old question for developers. Logging everything that happens in your application can be great because you have plenty of data to work from when you have a problem. But it's not so great if you have to grep and inspect it all yourself. **In my mind, developers should instead be thinking about logging the right events in the right format for the right consumer. This eBook will tell you how to do that.**

Application Logging Best Practices

1. Treat application logging as an ongoing, iterative process. Log at a high level and then add deeper instrumentation.
2. Always instrument anything that goes out of process because distributed system problems are not well behaved.
3. Always log unacceptable performance. Log anything outside the range in which you expect your system to perform.
4. If possible, always log enough context for a complete picture of what happened from a single log event.
5. View machines as your end consumer, not humans. Create logs that your log management solution can interpret.
6. Trends tell the story better than data points.
7. Instrumentation is NOT a substitute for profiling, and vice versa.
8. Flying more slowly is better than flying blind. So the debate is not whether to instrument, just how much.

Why Log Instrumentation Data from Your Applications?

Developers pride themselves on writing code that's solid and reliable—and passes QA with flying colors. This is no surprise: Smart designs and great code are what tend to drive recognition—and rewards—within development teams and across the industry. But in big, distributed systems like the one we run at Loggly, a lot of factors are out of the control of developers. Running tens or hundreds of machines increases the risk that one of them will start misbehaving. The most elegant design in the world can still have edge cases that throw them for a loop. Customers have an uncanny ability to do things in ways no sane developer would anticipate. And those factors can—and do—wreak havoc on even the best code.

WHEN YOU'RE DEALING WITH BILLIONS OF EVENTS PER DAY, THE ABILITY TO QUICKLY IDENTIFY HOT SPOTS IS INVALUABLE.

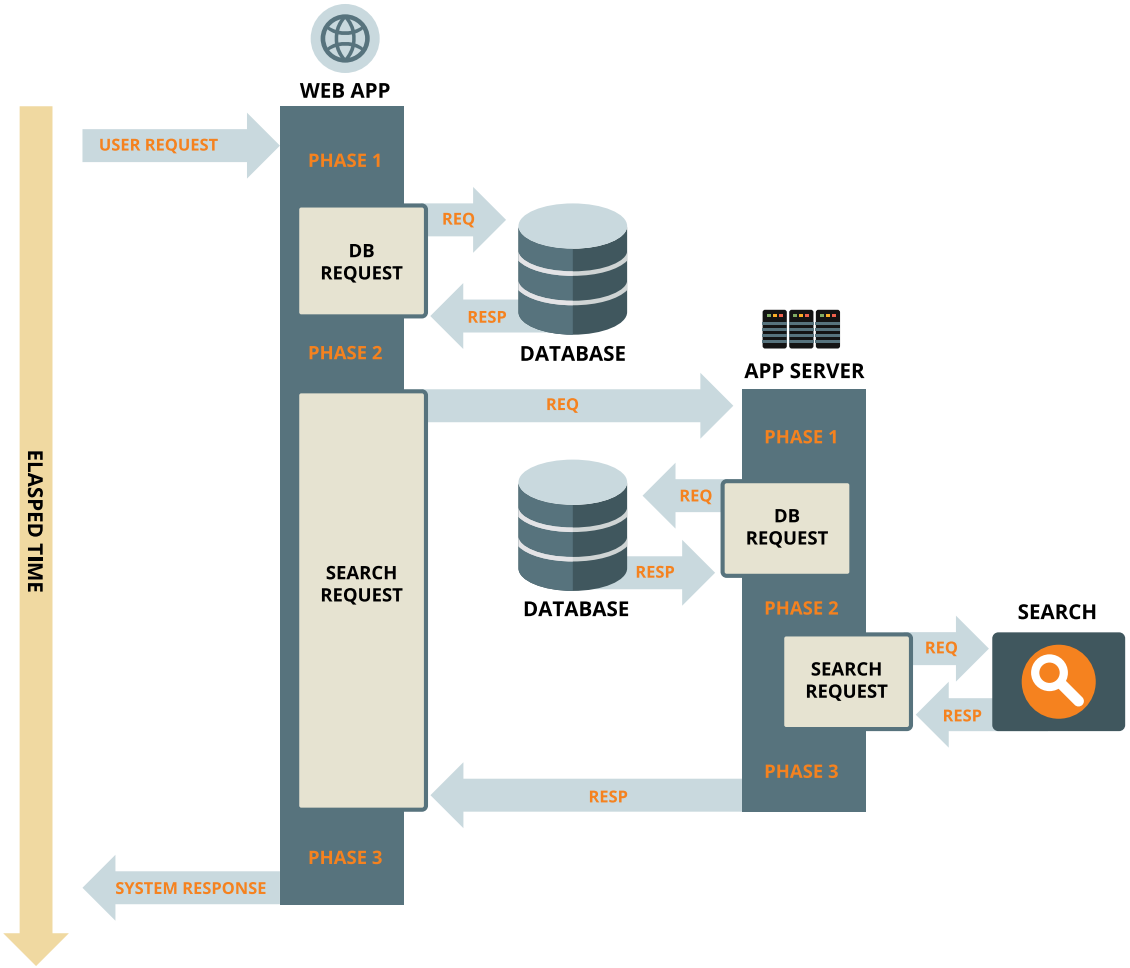
When code starts its "real life" in production, you need to be ahead of the curve. That's why instrumentation is one of the most valuable sources of data you can log. At Loggly, we instrument like crazy in our code because we need to understand what our system is doing and where we can improve it.

When you're dealing with billions of events per day, the ability to quickly identify hot spots is invaluable. However, there are lots of different opinions about what you should instrument and concerns about the performance impact of that instrumentation. What follows are the guidelines and best practices we've developed to help ourselves do a better job of instrumentation. We hope they'll help you, too.

Our Example System

In order to make all of this advice a little more concrete, we're going to use a simple example system that looks like many web-based systems out there. It's a simple multi-tier architecture, with the following components:

- A web application (e.g. Django, RoR, etc.) that receives user requests and services them using other services (a database and a separate application server)
- An application server (e.g. Tomcat, Jetty, Jboss) that receives requests from the web application and services them using yet more services (in our example, a database and a search system)



Our Example Application Functionality, Explained in Less Than 1,000 Words

In less than 1,000 words...

- The web app receives a user request.
- It does some initial processing (we'll call this Phase 1), then makes a database request and receives a response.
- It does some more processing (Phase 2), then makes an app server request.
- The app server receives a request from the web app.
- It does some processing (Phase 1), then makes a database request and receives a response.
- It does some more processing (Phase 2), then makes a search request and receives a response.
- It does some final processing (Phase 3), then responds to the web app.
- The web app receives the response from the app server, does some final processing (Phase 3), and responds to the user.

Below, we'll refer to this example system and use some example logging that demonstrates this path. You may think it's confusing to have two sets of Phase 1, Phase 2, and Phase 3 processing steps, but because they happen in different applications (and we can always qualify them with that application name), it should always be clear what piece of code each refers to.

Since there are so many different ways to implement web apps these days, we're going to ignore the logs that might be generated by apache, tomcat, jetty, node, or any of the other similar services/frameworks in our examples below. Those logs should also be available (and are incredibly useful), but are not relevant to this discussion of instrumentation.

So let's get on to the best practices

#1 Best Practice

Treat Instrumentation as an Ongoing, Iterative Process

As you first develop your application, start by logging everything at a high level and then add deeper instrumentation as you find yourself asking "What happened here?" You can start this process during the development phase and continue to refine it during beta, general availability, and growth stages.

For every application, you'll probably find that you're logging too much data in some areas and not enough in others. If you find that you're measuring something that consistently behaves well, you're a very lucky person and might be tempted to remove the instrumentation—but read #3 in this list before you do.

In areas that are more variable or more expensive than you thought, instrument deeper. Keep going down this path until you understand where the time is going. If you're lucky, you have a single component that is causing you grief. If not, at least you will have a deeper understanding of where the problem is coming from.

In our example, we have split each app into three "phases." After running the system in production for a while, you may decide that you need to split one of those phases further. For example, let's say phase 2 of the app server is clearly the slowest part of the system. At that point, you can just split it into phase 2.1, 2.2, 2.3, 2.4, and so on until you have isolated the code that is causing the performance issue. Don't be afraid to do this, since you're very unlikely to be able to predict the instrumentation you're going to need six months from now. But you should start somewhere!

Best Practice

Always Instrument Anything That Goes out of Process

It's very tempting (and very, very common) to make assumptions about how your code performs, but those assumptions can blind you to some very common performance issues. For example, a call to your database "should always be fast" so you might be tempted not to instrument it. The problem is that distributed system problems are, by definition, not well behaved. In fact, these things that you thought would never cause a problem can actually be indicative of a larger problem in your system, like the canary in the coal mine.

If you follow this best practice religiously, you'll have a much better understanding of where the problems are in your system. If your web app occasionally goes slow and your app server has a corresponding hiccup, then you can focus on the app server. If there isn't a hiccup deeper in your stack, then dive deeper into your web app.

Here is a simple example of an event that logs latencies at the web-app level.

This event is a summary of what is happening in our example web app. Specifically, the lat.* fields tell us the latency for each phase of the request:

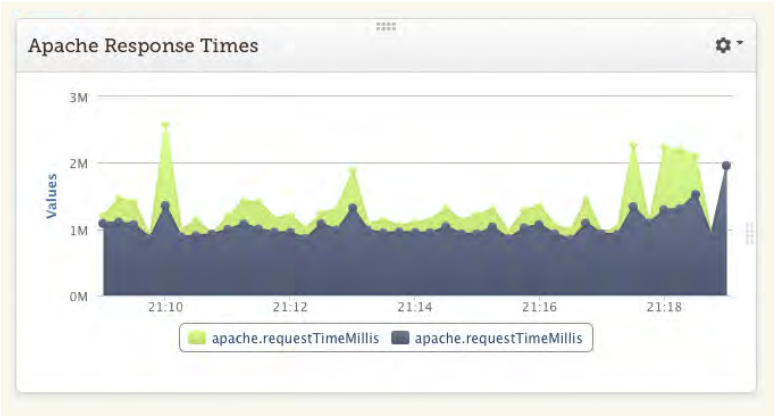
- p1 = phase 1
- db = database request
- p2 = phase 2
- java = application server request
- p3 = phase 3

Both the db and java (app server) requests are out of process as far as the web process is concerned. Without this level of instrumentation, we'd have no idea where to start looking for performance improvements. In this example, it's pretty clear that the database is the slowest part of the entire request.

#3 Best Practice

Always Log Unacceptable Performance

If you expect some part of your system to perform at a certain level, you should make sure you log when it falls outside that range. For example, if your database "should always be fast," then you can log any DB access that takes more than, say, 100ms. When the inevitable happens and the DB does slow down, you'll see that this has happened immediately thanks to this logging.



This is "trust but verify": If you don't think you need to log every request, that's fine, but at the very least make sure you're logging the bad ones.

Think of it as logging "soft exceptions." (My colleague Jason Skowronski wrote a useful **blog post** about using this type of performance information for production system monitoring.)

In subsequent sections, I'm going to be logging everything, but that is not always possible. If you do have to reduce your logging, this is the litmus test you should apply to decide whether any specific log message makes the cut. It should be easy enough to pick a threshold that will let you find the most egregious problems, while still reducing your logging volume.

4

Best Practice

If Possible, Always Log Enough Context so That One Log Event Contains the Complete Picture

For example, instead of logging the start and end of a process, log the end plus the elapsed time. This is not always possible, but when it is, it will save you a huge amount of time when things go wrong.

In our example system, the logs we generated for a single request look like this:

You can see that we log each phase of the request, the latency added by that specific phase, and the total latency for the app (the first 10 lines, then lines 12 and 13). This is useful because we're not trying to find two log lines that might look like:

```
Start Phase 1
…
End Phase 1
```

Effectively, all we're doing is adding performance data to the "End Phase n" log lines. Take a look at the first two lines again. Now imagine trying to work out what the latency for p1 was using just the time stamps. It's easy enough in this case, but what about when you're crossing hour, day, or month boundaries? Simply adding the data to the log line removes this problem entirely.

Taking this approach to the next level, we also log two summary lines that capture all of the data for each tier in a single event (the 11th and 14th lines). Arguably, this negates the need for the first type of log (the "End Phase n" logging), but there are times when it may still be useful to log at that level. For example, if you don't have good exception handling, then seeing a trail go dead will help you narrow problems. Imagine, for example, if the last event we saw was this:

That would tell us that something bad happened in phase 3 of the web app.

One final thing to point out about this type of logging is that we're not even trying to calculate the total elapsed time for the web app while we're logging from the app server. That would require that we pass in the current elapsed time as part of the app server request.

This is easy enough to do in theory, but as the system becomes more complex, it quickly falls apart. It's easier to just pass the requestID, because that gives us a global "handle" on all logging for each request throughout the entire stack. We can change the order in which we process parts of the request, add and remove new applications and/or services, reuse parts of the processing chain for different (offline, prototype, experimental) purposes… All of the things that happen in normal system evolution can happen without concern for "breaking your logging."

```
2015-01-27 03:34:43.308 UTC {"timestamp":"2015-01-27T03:34:43.308Z","requestId":"468E8437","tier":"web","phase":"search","latency":358,"elapsed":780}
```

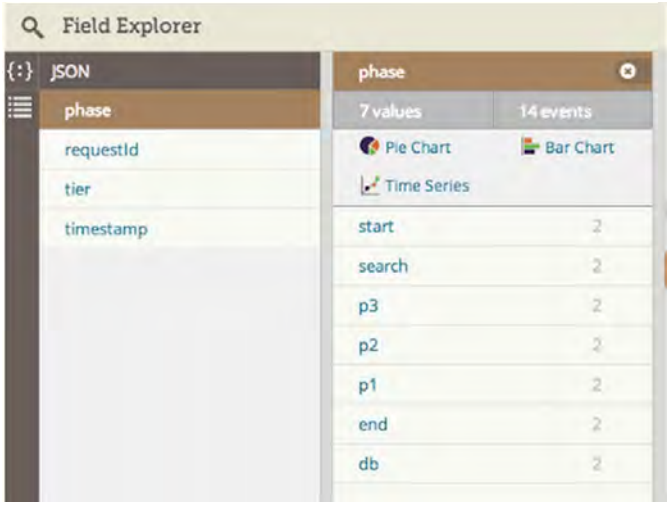
#5 Best Practice

View Machines as Your End Consumer

Rather than creating logs that humans can read, create logs that your log management solution can interpret. Developers are reluctant to dump copious amounts of diagnostic data into logs because they think they'll never be able to look at all of that data. And they're right. Machines, on the other hand, don't get tired eyes or carpal tunnel syndrome. In fact, they're better at combing through lots of data (they love to eat JSON structured data) and giving back much more consumable insights. So let a machine-based solution like Loggly crunch the data and save your brain power for the problem solving.

Our example log lines (above) are all JSON specifically for this reason. Yes, it is possible for you to read them (JSON is a pretty straightforward serialization format), but you're not going to read 100 million of them, are you?

If you generate your JSON with just a little thought about the structure and field names, you can use that to your advantage when you want to try and understand what metrics you're measuring. For example, **Loggly Dynamic Field Explorer™** actually displays all of the phases that are being measured. Here it is again:



Because we always use "phase" as the field name for the metrics we're logging, it's easy to see every available metric, then explore and graph any of these values. The ability to see all of your metrics in one place is incredibly valuable in itself.

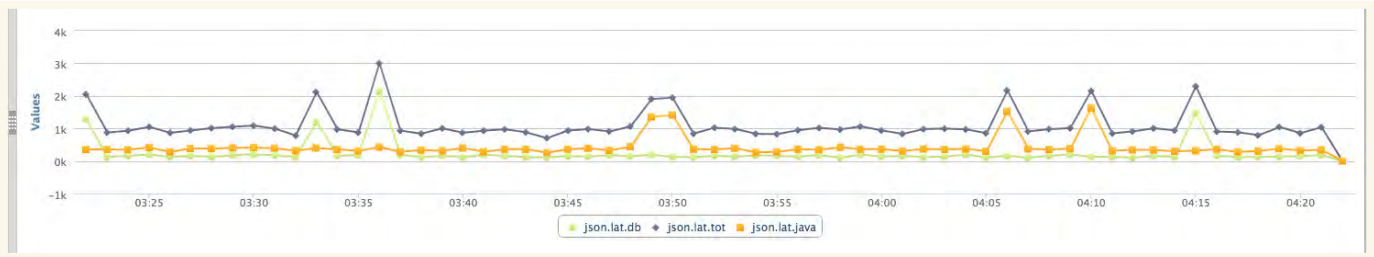
#6 Best Practice

Trends Tell the Story Better Than Data Points

Being able to see how your system performs over time (period over period) will tell you more about what is really going on than any single data point. If you have full instrumentation, you can graph it over time and see how your system performance fluctuates throughout the day. Sometimes, however, it is simply not practical to log every single event that is flowing through your system, so you may need to implement rollups in your instrumentation code. In our Java code, we use a metrics class that gives us counts and elapsed times every 10 seconds for every pipeline in the system. (You can learn more about it in a **blog post** by our CTO.) Graphing this data shows us exactly how we're doing and helps us find slowdowns before they start causing problems.

In our example app, we see the total latency for a set of requests to the web app along with the latency for the app server and db requests. We can see that:

1. When things are normal, the average latency is around 1 second.
2. When either the app server or DB slows down, the average increases to around 2 seconds.
3. We have problems in both tiers.



#7 Best Practice

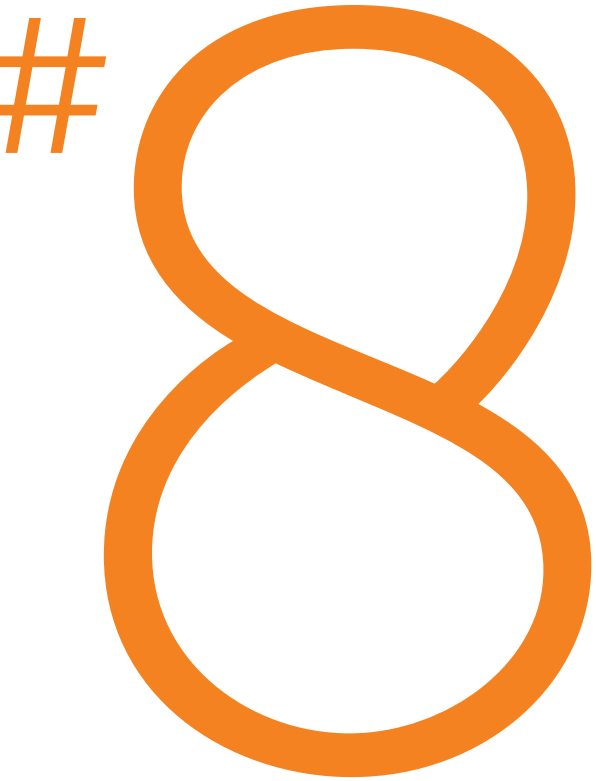
Instrumentation Is NOT a Substitute for Profiling, and Vice Versa

If you have a serious performance issue in your code, instrumentation is probably not going to take you deep enough to find it. You should be profiling your code to get to the root cause. Conversely, if your code is running live in production, you almost certainly don't want to attach a profiler to it. You should be using instrumentation to find the problem area.

Profiling is a critical part of the development cycle for any high-performance code, and it should be judiciously used to get your code to the performance level at which you need it to operate. It's a great way to figure out why a particular piece of code is slow. However, it's also an ***expensive*** way to figure out what's going on since it can slow down your application by an order of magnitude or more. And that's why you should almost ***never*** use it in production.

Instrumentation, on the other hand, tells you how your application is running in the real world, outside of your test bed. It should always be significantly more lightweight, with close to zero impact on the performance of your systems. It provides much less detailed data than profiling does—it can tell you that a particular part of your program is slow but not why.

Think of instrumentation as your eyes and profiling as your microscope. Both have their uses, and both are infinitely better than being blindfolded, which is what you are when you don't do either.



Best Practice

Flying More Slowly Is Better Than Flying Blind

We've had, and continue to have, some vigorous debates about "overhead" when it comes to instrumentation. Engineers (rightly) don't want to clutter up their code with things that make it run more slowly. But the benefits of instrumentation are so huge that we know we simply have to have it. Our debates these days are more about how much instrumentation to use, not whether we should use it. Without instrumentation, we wouldn't be able to understand our systems' performance, so the "speed" advantage of not instrumenting is somewhat illusory.

Of course, once you make the decision to take logging seriously, you owe it to yourself to make sure that the logging solution you use is as robust and performs as well as possible. You don't ever want to be scared away from logging because of performance concerns.

For example, when Loggly implemented metrics collection, we spent a LOT of time making sure that this code was as fast as it could possibly be because we knew we were going to use it throughout our system. So treat logging in the same way you'd treat any code that is used everywhere:

- **Make it fast.**
- **Make it easy to use.**
- **Make it robust.**

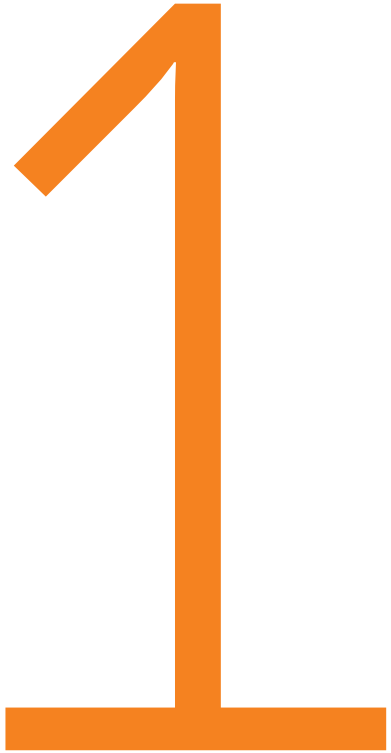
Our instrumentation adds less than 1 percent additional latency to our apps and repays that cost many times over. It helps us plan what to work on and when, and it helps us understand and deal with the inevitable production issues that arise when you run systems as complex as ours.

Your Next Task: Selling Appropriate Instrumentation to Your Development Team

SO HOW DO YOU CONVINCE YOUR DEVELOPMENT TEAM TO PUT THESE BEST PRACTICES TO WORK? IT'S ABOUT MINDSET, PROCESS, EDUCATION, AND ITERATION.

So here are 5 steps you can follow

Believing in Appropriate Instrumentation Requires a Mindset Change



The complexity and release cadence of today's net-centric applications are breaking down traditional boundaries between development and operations. In the past engineers were often shielded from the details of the production environment in which their code lived, either through choice or process. As a result, they viewed the code in production as yesterday's news. If it's a little difficult to manage, then that is "not my problem" or "next release." Today's engineers need to think differently:

PUT AS MUCH EFFORT INTO MAKING SURE THAT THE SYSTEM IS EASY TO MANAGE IN PRODUCTION AS YOU DO INTO MAKING SURE THAT IT IS NICELY DESIGNED AND PASSES ALL THE UNIT TESTS.

In the initial design, development, and QA phase, the most important question is "Is my code doing what the design says it should be doing?" Once it hits production, an equally important question must be asked: "Is the system handling the real world as well as we thought it would?" **And you need to be ready to answer that question very, very quickly.**

Since this is a very open-ended question, one of the best ways to answer it is to use instrumentation to give you visibility into the system. In other words, if you don't know what questions you're going to need to answer, make sure you at least have the answers to the questions you do know.

Let's Hope You Don't Make the Shift the Hard Way

Unfortunately, the most zealous converts to the new mindset are usually those developers who have paid the price, losing hours of sleep in wild goose chases after something has gone awry at 2 a.m. Once you:

- See your code fail in ways you never would have anticipated
- Have a pet belief destroyed
- Find that you simply have no idea why something is misbehaving

... you're much more likely to question your other assumptions about how things should behave. Here's an example.

At Loggly, we needed an industrial-strength service for handling some of our data that was easily accessible by a wide range of clients. Enter Amazon S3. We expected that it would be rock-solid and fast enough for our purposes. Every now and then the second of these expectations has proved to be an issue, because although the 99.9th percentile behavior is just fine, the 99.99th percentile proved a little harder to handle. How did we know that this was the issue? One in 10,000 requests is a pretty small needle in a pretty big haystack, after all. Truth be told, we didn't even suspect S3 would be problematic, but our general rule of instrumenting everything that went out of **process** gave us the data we needed to find and explain some very sporadic issues in our system. We still use S3, and we're very happy with it. But now that we know more about how it behaves in extremis, we can factor that into our design.

Use Instrumentation to Form the New Wall Between Development and Ops

There are probably as many definitions of DevOps as there are developers and ops people, but as far as I'm concerned, here's the bottom line:

- **Your developers** should want to know how their code is behaving in production—they should "know the shapes."
- **Your ops people** should want to know about the internal monitoring, and should be comfortable using it to dig a little deeper than they otherwise could.
- There should be **as few barriers as possible** between the two groups.
- They have different specializations, but the end goal for both should be **a smoothly running, high performance, well understood system.**

I've spent a long time building distributed systems, and I've never felt comfortable just handing them over to ops without first building a suite of internal monitoring tools. Those tools—and the instrumentation that feeds them data—are the key to closing the barrier between the development and operations teams and forming a new, more flexible wall.

Help Everyone Understand What Machine-Based Log Management Can Do

Developers may resist instrumentation because they are worried about becoming log watchers. You need to make sure that they are instead thinking about creating logs that are watchable by a machine:

- Problems become visible before they affect every user.
- The logs contain all of the data necessary to identify causes of failure.
- The logs contain performance data for easy analytics.

Logs as prose have a long tradition in software engineering. On the other hand, machine-readable logs may look "cluttered" to developers in their raw form. So here's how I suggest changing their minds: Show them a graph of your application's latency, generated with a couple of mouse clicks rather than a grep / sed / sort / awk / gnuplot pipeline that always goes wrong the first three times. I promise that they will quickly see the value. Add some alerts, and they'll realize they don't have to sit and tail files to see what is going on in the system.

Treat Visibility as the Mother of More Instrumentation

Once your development team starts down the path of putting the right logs in the right format for the right consumer, you'll find that positive feedback loops will form. Your team starts to see what's going on at a level of detail they never could before—and where you can't see enough. They'll save hours or even days in debugging and operational troubleshooting.

Eventually, you'll get to the point where your instrumentation is telling you what's really going on, as opposed to what your design tells you should be going on. We hope there's not a big divergence, but when there is it's nice to be able to measure it. And once you do, you'll never go back.

Use Trending to Look Forward, Not Just Back

The value of log data is not just in understanding what has already happened, but in guiding future action. By using log metrics to guide planning, you'll embed best-practice instrumentation more deeply into your organizational culture.

At Loggly, we use a wide range of metrics for every part of our system—data volumes, flow through each component, indexing rate, search latency, and more—to guide our growth planning. It's reassuring to make decisions based on real data, not idealized tests that don't always reflect the complexity of our production service.

Parting Thoughts

The real world will always throw you curve balls, no matter how hard you try. Some instances will perform worse than other "identical" instances. Some application will be configured differently than all the rest. Some disk drive will fail. Some customer will send 100 times more data than anyone else. The list is endless, and no one I've ever worked with, or heard about, has been smart enough to predict every possible failure mode and have tests for them.

Life as a developer of a cloud-based application can be like standing in front of a fire hose of pain. The only way to survive it is to do everything you can to measure what is actually happening in your system, and use that data to get a deeper understanding of what is actually going on, rather than guessing based on what you think should be going on.

I hope that this eBook can convince your team without the painful experience of a 2 a.m. disaster.

About Loggly

Loggly is the world's most popular cloud-based, enterprise-class log management solution, used by more than 5,000 happy customers to effortlessly spot problems in real-time, easily pinpoint root causes, and resolve issues faster to ensure application success. Founded in 2009 and based in San Francisco, the company is backed by Harmony Partners, Trinity Ventures, True Ventures, Matrix Partners, Cisco, Data Collective Venture Capital, and others.

Visit the Loggly website: loggly.com.

{ eBook }

loggly

TRY LOGGLY FOR FREE:
loggly.com/trial

The Pragmatic Logging Handbook