Instrumenting distributed systems for operational visibility

David Yanacek

Diving headfirst into the logs

When I joined Amazon after college, one of my first onboarding exercises was to get the amazon.com web server up and running on my developer desktop. I didn't get it right on the first try, and I wasn't sure what I did wrong. A helpful coworker suggested that I look at the logs to see what was wrong. To do so, he said that I should "cat the log file." I was convinced they were playing some sort of prank on me or making a joke about cats that I didn't understand. I only used Linux in college to compile, use source control, and use a text editor. So I didn't know that "cat" is actually a command to print out a file to the terminal that I could feed into another program to look for patterns.

My coworkers pointed me to tools like cat, grep, sed, and awk. Equipped with this new toolset, I dove into the amazon.com web server logs on my developer desktop. The web server application was already instrumented to emit all kinds of useful information into its logs. This allowed me to see what configuration I was missing that prevented the web server from starting up, that showed where it might have been crashing, or that indicated where it was failing to talk to a downstream service. The website is made up of a lot of moving pieces, and it was essentially a black box to me at the beginning. However, after diving headfirst into the system, I learned how to figure out how the server worked and how to interact with its dependencies just by looking at the output of the instrumentation.

Why instrumentation

As I've moved from team to team over the years at Amazon, I've found instrumentation to be an invaluable lens that I and others at Amazon look through to learn how a system works. However, instrumentation is useful for more than just learning about systems. It is the core of operational culture at Amazon. Great instrumentation helps us see the experience we are giving our customers.

This focus on operational performance spans the company. Within services associated with amazon.com, increased latency results in a poor shopping experience and thereby lowers conversion rates. With customers who use AWS, they depend on the high availability and low latency of AWS services.

At Amazon, we don't just consider average latency. We focus even more closely on latency outliers, like the 99.9th and 99.99th percentile. This is because if one request out of 1,000 or 10,000 is slow, that is still a poor experience. We find that when we reduce high percentile latency in a system, our efforts have the side effect of reducing median latency. By contrast, we find that when we reduce median latency, this reduces high percentile latency less often.

The other reason we focus on high percentile latency is that high latency in one service can have a multiplier effect across other services. Amazon is built on a service-oriented architecture. Many services collaborate with each other to get something done, such as rendering a web page on amazon.com. As a result, an increase in the latency of a service deep in the call chain—even if the increase is in a high percentile—has a big ripple effect on the latency experienced by the end user.

Large systems at Amazon are made up of many cooperating services. Each service is developed and operated by a single team (large "services" are composed of multiple services or components behind the scenes). The team who owns a service is known as the *service owner*. Every member of that team thinks like an owner and operator of the service, whether that member is a software developer, network engineer, manager, or in any other role. As the owners, teams set goals on the operational

performance of all of the associated services. We also ensure we have the visibility into service operations to ensure that we will meet those goals, to deal with any issues that arise, and hold ourselves to even higher goals the following year. To set goals and get that visibility, teams must instrument systems.

Instrumentation also enables us to detect and respond to operational events tactically. Instrumentation feeds data into operational dashboards, so that operators can view real-time metrics. It also feeds data into alarms, which trigger and engage operators when the system is behaving in an unexpected way. The operators use the detailed output of the instrumentation to quickly diagnose why things went wrong. From there, we can mitigate the problem, and come back later to prevent the problem from happening again. Without good instrumentation throughout the code, we sink precious time into diagnosing problems.

What to measure

To operate services according to our high standards on availability and latency, we as service owners need to measure how our systems behave.

To get the necessary telemetry, service owners measure the operational performance from multiple places to get multiple perspectives on how things are behaving end-to-end. This is complicated even in a simple architecture. Consider a service that customers call through a load balancer: the service talks to a remote cache and remote database. We want each component to emit metrics about its behavior. We also want metrics on how each component perceives the behavior of other components. When metrics from all of these perspectives are brought together, a service owner can track down the source of problems quickly, and dig in to find the cause.

Many AWS services automatically provide operational insights about your resources. For example, Amazon DynamoDB provides Amazon CloudWatch metrics on success and error rates and latency, as measured by the service. However, when we build systems that use these services, we need a lot more visibility into how our systems are behaving. Instrumentation requires explicit code that records how long tasks take, how often certain code paths are exercised, metadata about what the task was working on, and what parts of the tasks succeeded or failed. If a team doesn't add explicit instrumentation, it would be forced to operate its own service as a black box.

For example, if we implemented a service API operation that retrieved product information by product ID, the code might look like the following example. This code looks up product info in a local cache, followed by a remote cache, followed by a database:

```
public GetProductInfoResponse getProductInfo(GetProductInfoRequest request) {

  // check our local cache
  ProductInfo info = localCache.get(request.getProductId());

  // check the remote cache if we didn't find it in the local cache
  if (info == null) {
    info = remoteCache.get(request.getProductId());

          localCache.put(info);
  }
```

```
  // finally check the database if we didn't have it in either cache
  if (info == null) {
    info = db.query(request.getProductId());

          localCache.put(info);
          remoteCache.put(info);
  }

  return info;
}
```

If I were operating this service, I'd need a lot of instrumentation in this code to be able to understand its behavior in production. I'd need the ability to troubleshoot failed or slow requests, and monitor for trends and signs that different dependencies are underscaled or misbehaving. Here's that same code, annotated with a few of the questions I'd need to be able to answer about the production system as a whole, or for a particular request:

```
public GetProductInfoResponse getProductInfo(GetProductInfoRequest request) {

  // Which product are we looking up?
  // Who called the API? What product category is this in?

  // Did we find the item in the local cache?
  ProductInfo info = localCache.get(request.getProductId());

  if (info == null) {
    // Was the item in the remote cache?
    // How long did it take to read from the remote cache?
    // How long did it take to deserialize the object from the cache?
    info = remoteCache.get(request.getProductId());

    // How full is the local cache?
    localCache.put(info);
  }

  // finally check the database if we didn't have it in either cache
  if (info == null) {
    // How long did the database query take?
    // Did the query succeed?
    // If it failed, is it because it timed out? Or was it an invalid query? Did we lose our database connection?
    // If it timed out, was our connection pool full? Did we fail to connect to the database? Or was it just
slow to respond?
    info = db.query(request.getProductId());

    // How long did populating the caches take?
    // Were they full and did they evict other items?
    localCache.put(info);
    remoteCache.put(info);
  }
```

```
  // How big was this product info object?
  return info;
}
```

The code for answering all of those questions (and more) is quite a bit longer than the actual business logic. Some libraries can help reduce the amount of instrumentation code, but the developer still must ask the questions about the visibility that the libraries will need, and then the developer must be intentional about wiring in the instrumentation.

When you troubleshoot a request that flows through a distributed system, it can be difficult to understand what happened if you only look at that request based on one interaction. To piece together the puzzle, we find it helpful to pull together in one place all of the measurements about all of these systems. Before we can do that, each service must be instrumented to record a trace ID for each task, and to propagate that trace ID to each other service that collaborates on that task. Collecting the instrumentation across systems for a given trace ID can be done either after the fact as needed, or in near real-time using a service like AWS X-Ray.

Drilling down

Instrumentation enables troubleshooting at multiple levels, from glancing at metrics to see if there are any anomalies that are too subtle to trigger alarms, to performing an investigation to find out the cause of those anomalies.

At the highest level, the instrumentation is aggregated into metrics that can trigger alarms and display on dashboards. These aggregate metrics let operators monitor the overall request rate, the latency of the service calls, and the error rates. These alarms and metrics make us aware of anomalies or changes that should be investigated.

After we see an anomaly, we need to figure out why that anomaly is happening. To answer that question, we rely on metrics made possible by even more instrumentation. By instrumenting the time that it takes to perform various parts of serving a request, we can see which part of the processing is slower than normal, or triggers errors more often.

While aggregate timers and metrics might help us rule out causes or highlight an area of investigation, they don't always provide a complete explanation. For example, we might be able to tell from metrics that errors are coming from a particular API operation, but the metrics might not reveal enough specifics about why that operation is failing. At this point, we look at the raw, detailed log data emitted by the service for that time window. The raw logs then show the source of problem—either the particular error that is happening, or particular aspects of the request that are triggering some edge case.

How we instrument

Instrumentation requires coding. It means that when we are implementing new functionality, we need to take the time to add extra code to indicate what happened, whether it succeeded or failed, and how long it took. Because instrumentation is such a common coding task, practices emerged at Amazon over the years to address common patterns: standardization for common instrumentation libraries, and standardization for structured log-based metric reporting.

Standardization of metrics instrumentation libraries helps library authors give consumers of their libraries visibility into how the library is operating. For example, commonly used HTTP clients integrate with these common libraries, so if a service team implements a remote call to another service they get instrumentation about those calls automatically.

When an instrumented application runs and performs work, the resulting telemetry data is written to a structured log file. Generally, it is emitted as one log entry per "unit of work," whether that is a request to an HTTP service, or a message pulled from a queue.

At Amazon, measurements in the application aren't aggregated and occasionally flushed to a metrics aggregation system. All of the timers and counters for every piece of work are written in a log file. From there, the logs are processed and aggregate metrics are computed after the fact by some other system. This way, we end up with everything from high-level aggregate operational metrics to detailed troubleshooting data at the request level, all with a single approach to instrumenting code. At Amazon we log first, and produce aggregate metrics later.

Instrumentation through logging

We most commonly instrument our services to emit two types of log data: request data and debugging data. Request log data is typically represented as a single structured log entry for each unit of work. This data contains properties about the request and who made the request, what the request was for, counters of how often things happened, and timers of how long things took. The request log serves as an audit log and a trace for everything that happened in the service. Debugging data includes unstructured or loosely structured data of whatever debugging lines the application emits. Typically these are unstructured log entries like Log4j error or warn log lines. At Amazon these two types of data are typically emitted into separate log files, partially for historical reasons, but also because it can be convenient to do log analysis on a homogenous log entry format.

Agents like the CloudWatch Logs Agent process both types of log data in real time and ship the logs to CloudWatch Logs. In turn, CloudWatch Logs produces aggregate metrics about the service in near real time. Amazon CloudWatch Alarms reads these aggregate metrics and triggers alarms.

Although it can be expensive to log so much detail about every request, at Amazon we find that it is incredibly important to do. After all, we need to investigate availability blips, latency spikes, and customer-reported problems. Without detailed logs, we can't give customers answers, and we won't be able to improve their service.

Getting into the details

The topic of monitoring and alarming is vast. In this article we won't cover topics like setting and tuning alarm thresholds, organizing operational dashboards, measuring performance from both the server-side and client-side, continuously running "canary" applications, and choosing the right system to use to aggregate metrics and analyze logs.

This article focuses on the need to instrument our applications to produce the right raw measurement data. We will describe the things that teams at Amazon make an effort to include (or avoid) when instrumenting their applications.

Request log best practices

In this section I'll describe good habits that we've learned over time at Amazon about logging our structured "per unit of work" data. A log that meets these criteria contains counters representing how often things happen, timers containing how long things took, and properties that include metadata about each unit of work.

How we log

- **Emit one request log entry for every unit of work.** A unit of work is typically a request our service received or a message it pulls from a queue. We write one service log entry for each request that our service receives. We don't combine multiple units of work together. This way, when we troubleshoot a failed request, we have a single log entry to look at. This entry contains the relevant input parameters about the request to see what it was trying to do, information about who the caller was, and all of the timing and counter information in one place.
- **Emit no more than one request log entry for a given request.** In a non-blocking service implementation, it might seem convenient to emit a separate log entry for each stage in a processing pipeline. Instead, we have more success in troubleshooting these systems by plumbing a handle to a single "metrics object" around between stages in the pipeline, and then serializing the metrics out as a unit after all of the stages have completed. Having multiple log entries per unit of work makes log analysis more difficult, and increases the already expensive logging overhead by a multiplier. If we are writing a new non-blocking service, we try to plan the metrics logging lifecycle up front, because it becomes very difficult to refactor and fix later on.
- **Break long-running tasks into multiple log entries.** In contrast to the previous recommendation, if we have a long-running, multiple-minute or multiple-hour workflow-like task, we may decide to emit a separate log entry periodically so that we can determine if progress is being made, or where it is slowing down.
- **Record details about the request before doing stuff like validation.** We find it important for troubleshooting and audit logging to log enough information about the request so that we know what it was trying to accomplish. We also have found it important to log this information as early as possible, before the request has the chance to be rejected by validation, authentication, or throttling logic. If we are logging information from the incoming request, we make sure to sanitize input (encode, escape, and truncate) before we log it. For example, we do not want to include 1 MB long strings in our service log entry if the caller passed one in. Doing so would risk filling up our disks and costing us more than expected in log storage. Another example of sanitization is to filter out ASCII control characters or escape sequences relevant to the log format. It could be confusing if the caller passed in a service log entry of their own, and was able to inject it into our logs! See also: https://xkcd.com/327/
- **Plan for a way to log at increased verbosity.** For troubleshooting some kinds of problems, the log will not have enough detail about problematic requests to figure out why they failed. That information might be available in the service, but the volume of information might be

too great to justify logging all the time. It can be helpful to have a configuration knob that you can dial to increase the log's verbosity temporarily while you investigate an issue. You might turn the knob on individual hosts, or for individual clients, or at a sampling rate across the fleet. It's important to remember to turn the knob back down when done.

- **Keep metric names short (but not too short).** Amazon has used the same service log serialization for over 15 years. In this serialization, each counter and timer name is repeated in plain text in every service log entry. To help minimize the logging overhead, we use timer names that are short, yet descriptive. Amazon is beginning to adopt new serialization formats based on a binary serialization protocol known as Amazon Ion. Ultimately, it's important to choose a format that log analysis tools can understand that is also as efficient to serialize, deserialize, and store as possible.
- **Ensure log volumes are big enough to handle logging at max throughput.** We load test our services at maximum sustained load (or even overload) for hours. We need to make sure that when our service is handling excess traffic, the service still has the resources to ship logs off-box at the rate that they produce new log entries, or the disks will eventually fill up. You can also configure logging to happen on a different file system partition than the root partition, so that the system doesn't break down in the face of excessive logging. We discuss other mitigations for this later, like using dynamic sampling that is proportional to throughput, but no matter the strategy, it is critical to test.
- **Consider the behavior of the system when disks fill up.** When a server's disk fills up, it can't log to disk. When that happens, should a service stop accepting requests, or drop logs and continue operating without monitoring? Operating without logging is risky, so we test systems to ensure that servers with nearly full disks are detected.
- **Synchronize clocks.** The notion of "time" in distributed systems is notoriously complicated. We do not rely on clock synchronization in distributed algorithms, but it is necessary for making sense of logs. We run daemons like Chrony or ntpd for clock synchronization, and we monitor servers for clock drift. To make this easier, see the Amazon Time Sync Service.
- **Emit zero counts for availability metrics.** Error counts are useful, but error percentages can be useful too. To instrument for an "availability percentage" metric, a strategy we've found useful is to emit a 1 when the request succeeds and a 0 when the request fails. Then the "average" statistic of the resulting metric is the availability rate. Intentionally emitting a 0 data point can be helpful in other situations too. For example, if an application performs leader election, emitting a 1 periodically for when one process is the leader, and a 0 when the process is not the leader can be helpful for monitoring the health of the followers. This way if a process stops emitting a 0, it's easier to know that something in it has broken, and it won't be able to take over if something happens to the leader.

What we log

- **Log the availability and latency of all of dependencies.** We have found this particularly helpful in answering questions of "why was the request slow?" or "why did the request fail?" Without this log, we can only compare graphs of dependencies with graphs of a service, and guess whether one spike in a dependent service's latency led to the failure of a request we are

investigating. Many service and client frameworks plumb metrics automatically, but other frameworks (like the AWS SDK, for example), require manual instrumentation.

- **Break out dependency metrics per call, per resource, per status code, etc.** If we interact with the same dependency multiple times in the same unit of work, we include metrics about each call separately, and make it clear which resource each request was interacting with. For example, when calling Amazon DynamoDB, some teams have found it helpful to include timing and latency metrics per table, as well as per error code, and even per the number of retries. This make it easier to troubleshoot cases where the service was slow from retries due to conditional check failures. These metrics also revealed cases where client-perceived latency increases were actually due to throttling retries or paginating through a result set, and not from packet loss or network latency.
- **Record memory queue depths when accessing them.** If a request interacts with a queue, and we're pulling an object out of it, or putting something into it, we record the current queue depth into the metrics object while we're at it. For in-memory queues, this information is very cheap to obtain. For distributed queues, this metadata may be available for free in responses to API calls. This logging will help find backlogs and sources of latency in the future. In addition, when we take things out of a queue, we measure the duration that things were in the queue. This means that we need to add our own "enqueue time" metric to the message before we enqueue it in the first place.
- **Add an additional counter for every error reason.** Consider adding code that counts the specific error reason for every failed request. The application log will include information that led up to the failure, and a detailed exception message. However, we've also found it helpful to see trends in error reasons in metrics over time without having to mine for that information in the application log. It's handy to start with a separate metric for each failure exception class.
- **Organize errors by category of cause.** If all errors are lumped into the same metric, the metric becomes noisy and unhelpful. At a minimum, we've found it important to separate errors that were the "client's fault" from errors that were the "server's fault." Beyond that, further breakdown might be helpful. For example in DynamoDB, clients can make conditional write requests that return an error if the item they are modifying does not match the preconditions in the request. These errors are deliberate, and we expect them to happen occasionally. Whereas "invalid request" errors from clients are most likely bugs that we need to fix.
- **Log important metadata about the unit of work.** In a structured metric log, we also include enough metadata about the request so we can later determine whom the request was from and what the request was attempting to do. This includes metadata that a customer would expect us to have in our log when they reach out with problems. For example, DynamoDB logs the name of the table a request is interacting with, and metadata like whether the read operation was a consistent read or not. However, it does not log the data being stored to or retrieved from the database.
- **Protect logs with access control and encryption.** Because logs contain some degree of sensitive information, we take measures to protect and secure that data. These measures include encrypting logs, restricting access to operators who are troubleshooting problems, and regularly baselining that access.

- **Avoid putting overly sensitive information in logs.** Logs need to contain some sensitive information to be useful. At Amazon, we find it important for logs to include enough information to know whom a given request came from, but we leave out overly sensitive information, such as request parameters that do not affect the routing or behavior of processing the request. For example, if the code is parsing a customer message, and that parse fails, it's important to not log the payload to protect customer privacy, as difficult as that may make troubleshooting later. We use tooling to make decisions about what can be logged in an opt-in way instead of an opt-out way, to prevent the logging of a new sensitive parameter added later. Services like Amazon API Gateway allow configuring which data to be included in its access log, which acts as a good opt-in mechanism.
- **Log a trace ID and propagate it in backend calls.** A given customer request will likely involve many services working in cooperation. This can be as few as two or three services for many AWS requests, to far more services for amazon.com requests. To make sense of what happened when we troubleshoot a distributed system, we propagate the same trace ID between these systems so that we can line up logs from various systems to see where failures happened. A trace ID is a sort of meta request ID that is stamped to a distributed unit of work by the "front door" service that was the starting point for the unit of work. AWS X-Ray is one service that helps by providing some of this propagation. We have found it important to pass the trace to our dependency. In a multi-threaded environment, it is very difficult and error-prone for the framework to do this propagation on our behalf, so we have gotten in the habit of passing trace IDs, and other request content (like a metrics object!) in our method signatures. We have also found it handy to pass around a Context object in our method signatures, so that we don't have to refactor when we find a similar pattern to pass around in the future. For AWS teams, it's not just about troubleshooting our systems, it's also about customers troubleshooting theirs. Customers rely on AWS X-Ray traces being passed between AWS services when they interact with each other on the customer's behalf. It requires us to propagate customer AWS X-Ray trace IDs between services so that they can get complete trace data.
- **Log different latency metrics depending on status code and size.** Errors are often fast—like access denied, throttling, and validation error responses. If clients start getting throttled at a high rate, it might make latency look deceptively good. To avoid this metric pollution, we log a separate timer for successful responses, and focus on that metric in our dashboards and alarms instead of using a generic Time metric. Similarly, if there is an operation that can be slower depending on the input size or response size, we consider emitting a latency metric that is categorized, like SmallRequestLatency and LargeRequestLatency. In addition, we ensure our request and responses are appropriately bounded to avoid complex brownout and failure modes, but even in a carefully designed service, this metric-bucketing technique can isolate customer behavior and keep distracting noise out of dashboards.

Application log best practices

This section describes good habits we've learned at Amazon about logging unstructured debug log data.

- **Keep the application log free of spam.** Although we might have INFO and DEBUG level log statements on the request path to help with development and debugging in test environments, we consider disabling these log levels in production. Instead of relying on the application log for request tracing information, we think of the service log as a location for the trace information that we can easily produce metrics and see aggregate trends over time. However, there is no black-and-white rule here. Our approach is to continuously review our logs to see if they are too noisy (or not noisy enough), and adjust log levels over time. For example, when we're log diving we often find log statements that are too noisy, or metrics that we wish we had. Fortunately, these improvements are often easy to make, so we've gotten into the habit of filing quick follow-up backlog items to keep our logs clean.
- **Include the corresponding request ID.** When we're troubleshooting an error in the application log, we often want to see details about the request or the caller that triggered the error. If both logs contain the same request ID, then we can easily jump from one log to the other. Application logging libraries will write out the corresponding request ID if properly configured, and the request ID is set as a ThreadLocal. If an application is multithreaded, consider taking special care to set the correct request ID when a thread starts working on a new request.
- **Rate-limit an application log's error spam.** Typically, a service won't emit much to the application log, but if it suddenly starts exhibiting a large volume of errors, it might suddenly start writing out a high rate of very large log entries with stack traces. One way we've found to guard against this is to rate-limit how often a given logger will log.
- **Prefer format strings over String#format or string concatenation.** Older application log API operations accept a single string message instead of log4j2's varargs format string api. If code is instrumented with DEBUG statements, but production is configured at ERROR level, it's possible to waste work formatting DEBUG message strings that are ignored. Some logging API operations support passing in arbitrary objects that will have their toString() methods called only if the log entry will be written out.
- **Log request IDs from failed service calls.** If a service is called and returns an error, the service likely returned a request ID. We've found it useful to include the request ID in our log so that if we need to follow up with that service owner, we have a way for them to easily find their own corresponding service log entries. Timeout errors make this tricky because the service might not have returned a request ID yet, or a client library might not have parsed it. Nonetheless, if we have a request ID back from the service, we log it.

High throughput services best practices

For the vast majority of services at Amazon, logging on every request doesn't impose an unreasonable cost overhead. Higher throughput services enter a grayer area, but we often still log on every request. For example, it's natural to assume that DynamoDB, serving at peak over 20 million requests per second of Amazon-internal traffic alone, would not log very much, but in fact it logs every request for troubleshooting and for audit and compliance reasons. Here are some high-level tips we use at Amazon to make logging more efficient at higher per-host throughput:

- **Log sampling.** Instead of writing every entry, consider writing out every N entries. Each entry also includes how many entries were skipped so that metric aggregation systems can estimate the true log volume in the metrics it computes. Other sampling algorithms like reservoir sampling provide more representative samples. Other algorithms prioritize logging errors or slow requests over successful, fast requests. However with sampling, the ability is lost to help customers and troubleshoot specific failures. Some compliance requirements disallow it altogether.
- **Offload serialization and log flushing to a separate thread.** This is an easy change and is commonly used.
- **Frequent log rotation.** Rotating log files logs every hour might seem convenient so you have fewer files to deal with, but by rotating every minute, several things improve. For example, the agent that reads and compresses the log file will be reading the file from page cache instead of disk, and the CPU and IO from compressing and shipping logs will be spread out over the hour instead of always triggering at the end of the hour.
- **Write logs pre-compressed.** If a log-shipping agent compresses logs before sending them to an archival service, the system CPU and disk will spike on a periodic basis. It's possible to amortize this cost, and reduce the disk IO by half, by streaming compressed logs out to disk. This comes with some risks though. We've found it helpful to use a compression algorithm that can handle truncated files in the case of an application crash.
- **Write to a ramdisk / tmpfs.** It might be easier for a service to write logs to memory until they are shipped off the server instead of writing the logs to disk. In our experience, this works best with log rotation every minute versus log rotation every hour.
- **In-memory aggregates.** If it's necessary to handle hundreds of thousands of transactions per second on a single machine, it might be too expensive to write a single log entry per request. However, you lose a lot of observability by skipping this, so we've found it helpful to not prematurely optimize.
- **Monitor resource utilization.** We pay attention to how close we are to reaching some scaling limit. We measure our per-server IO and CPU, and how much of those resources are being consumed by logging agents. When we perform load tests, we run them long enough so that we can prove that our log shipping agents can keep up with our throughput.

Have the right log analysis tools in place

At Amazon, we operate the services we write, so we all need to become experts at troubleshooting them. This includes being able to do log analysis effortlessly. There are many tools at our disposal, from local log analysis for looking at a relatively small number of logs, to distributed log analysis for sifting through and aggregating results from an enormous volume of logs.

We've found it important to invest in teams' tools and runbooks for log analysis. If logs are small now, but a service expects to grow over time, we pay attention to when our current tools stop scaling, so that we can to invest in adopting a distributed log analysis solution.

Local log analysis

The process of log analysis might require experience in various Linux command line utilities. For example, a common “find the top talker IP addresses in the log” is simply:

```
cat log | grep -P "^RemoteIp=" | cut -d= -f2 | sort | uniq -c | sort -nr | head -n20
```

However, there are a bunch of other tools that are useful for answering more complex questions with our logs, including:

- jq: https://stedolan.github.io/jq/
- RecordStream: https://github.com/benbernard/RecordStream

Distributed log analysis

Any big data analytics service can be used to do distributed log analysis (for example, Amazon EMR, Amazon Athena, Amazon Aurora, and Amazon Redshift). However, some services come equipped with logging systems, for example Amazon CloudWatch Logs.

- CloudWatch Logs Insights
- AWS X-Ray: https://aws.amazon.com/xray/
- Amazon Athena: https://aws.amazon.com/athena/

Conclusion

As a service owner and a software developer, I spend an enormous amount of my time looking at the outputs of instrumentation—graphs on dashboards, individual log files—and using distributed log analysis tools like CloudWatch Logs Insights. These are some of my favorite things to do. When I need a break after wrapping up some challenging task, I recharge my batteries and reward myself with some log diving. I start with questions like “why did this metric spike here?” or “could the latency of this operation be lower?” When my questions lead to a dead end, I often figure out some measurement that would be helpful in the code, so I add the instrumentation, test, and send a code review over to my teammates.

Despite the fact that many metrics come with the managed services that we use, we need to spend a lot of thought on instrumenting our own services so that we have the visibility we need to operate them effectively. During operational events, we need to quickly determine why we’re having a problem and what we can do to mitigate that problem. Having the right metrics on our dashboards is crucial so that we can do that diagnosis quickly. In addition, because we are always changing our services, adding new features, and changing the way they interact with their dependencies, the exercise of updating and adding the right instrumentation is forever ongoing.

Links

- “Look at your data,” by former Amazonian John Rauser: https://www.youtube.com/watch?v=coNDCIMH8bk (including at 13:22 where he literally prints out logs to get a better look at them)
- “Investigating anomalies” by former Amazonian John Rauser: https://www.youtube.com/watch?v=-3dw09N5_Aw
- “How humans see data” by former Amazonian John Rauser: https://www.youtube.com/watch?v=fSgEeI2Xpdc
- https://www.akamai.com/uk/en/about/news/press/2017-press/akamai-releases-spring-2017-state-of-online-retail-performance-report.jsp