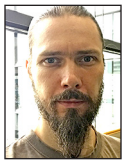
Structured Logging
Crafting Useful Message Content

VLADIMIR LEGEZA AND ANTON GOLUBTSOV WITH BETSY BEYER



Vladimir Legeza is a Technical Solutions Engineer at Google Cloud Japan. For the last few decades, he has worked for various companies in a variety of sizes and business spheres such as business consulting, Web portals development, online gaming, and TV broadcasting. Since 2010, Vladimir has primarily focused on large-scale, high-performance solutions. Before Google, he worked as an SRE on search services and platform infrastructure at Yandex and then in a similar position at Amazon Japan. lgz@google.com

Anton Golubtsov is a Software Development Engineer at Amazon Japan. Before Amazon, he worked at Yandex in a few roles: an SDE, a team leader, and a Technical Project Manager. zoomacode@zoomacode.ru

Betsy Beyer is a Technical Writer for Google Site Reliability Engineering in NYC and is the editor of *Site Reliability Engineering: How Google Runs Production Systems* and *Site Reliability Workbook*. She has previously written documentation for Google datacenters and hardware operations teams. She holds degrees from Stanford and Tulane. bbeyer@google.com

In the context of logging, the word "structured" typically refers to the way log records are represented in a machine-readable format, such as JSON or XML. In this article, we focus on another aspect of logging structure: the message content.

Computing today offers several automated ways of collecting, delivering, and processing log records from different types of systems. But modern technologies are not supportive if the information describing a specific event is insufficient or otherwise not helpful.

To approach this topic, it's useful to understand the most common logging issues, why they occur and possible solutions. By discussing some representative use cases, we aim to provide practical insights and approaches to improving the structure of your logs. As with most advice, our proposed solutions are just one way of approaching a problem space—feel free to either apply our suggestions wholesale or pick and choose the pieces that suit your needs.

Reasons to Invest in Well-Structured Logging

Before diving into specifics: why should you invest time and effort on designing and implementing a sound logging strategy?

Imagine a scenario in which you're trying to investigate event statements to determine why your main service isn't responding. Meanwhile, angry customers are reaching out to you via every possible communication channel, and upper management is shouting at you to resolve the situation quickly. Every corner of the office seems to be consumed with anxiety and pressure, but your only reasonable response is, "I couldn't find any useful information in our logs...I'll need to reproduce this entire event on a staging environment."

For many companies, every minute of downtime results in a certain amount of harm: outages entail both financial costs and damage to your reputation. When customers and investors experience a serious scare, the entire business may be at risk.

Well-structured logging can make a world of difference in the above scenario. After a few years in the industry, our experience has shown that investing time and effort in improving the logging process is worthwhile. When a crisis occurs, the alternative is too costly and painful.

Anatomy of a Log Entry

Logs can be split into three broad categories:

- **Operational logs:** contain information about service usage, such as user requests and transactions.
- **Telemetry logs:** contain application-internal metrics, expressed in the form of log records.
- **Behavioral logs:** show what is happening inside the application.

Operational and telemetry logs are typically generated from a pre-formatted template, while behavioral logs incorporate manually crafted components that are unique to each record. We'll focus on behavioral logs, the most widely used and complex of the three, but you can apply the solutions we discuss to operational and telemetry logs as well.

Message Type	Severity Level
Milestone	INFO
Alert	WARNING or ERROR
Data samples, additional details	DEBUG

Table 1: Message types mapped to severity levels

Types of Records/Messages

You might implement logging for a variety of purposes, most of which enable effective root cause analysis (RCA):

- **Tracking milestones:** To show an application's current state or a specified milestone. This type of information is useful in a couple scenarios:
  - When you want to understand what the application is doing right now—whether or not it reaches a particular milestone. **Example milestones:** *Operation A is completed; starting operation B. No more data to process.*
  - When you want to see a sequence of state changes in order to understand the application's end-to-end behavior. **Example implementation**: *The* `ssh` *binary expresses certain information into STDERR if executed with the* `-v` *flag.*
- **Alerting**: To emit alert notifications when something goes amiss.
- **Debugging/sampling**: To capture the data samples and statements with which a program is operating. These types of messages appear in the code in early development stages, when you simply want to see if an application is behaving as expected.

These message types map to a set of standard logging severity levels, as shown in Table 1.

Granular Decomposition

It's helpful to think of all log messages as a collection of answers about *why* an issue is happening. To answer *why*, we also need more information about *when* an event happened, *where* it happened, and *what* exactly happened. (Of course, not every type of log will answer all four "W" questions.)

Regardless of the type and category, every log record consists of two parts:

- **Metadata**: Information about the event statement
- **Content**: The statement itself

Metadata is generated automatically, whereas content is manually crafted. Each part answers its own set of "W" questions, as represented in Figure 1.

Complications

Problems with logging fall into two main buckets: inconsistency and missing data.

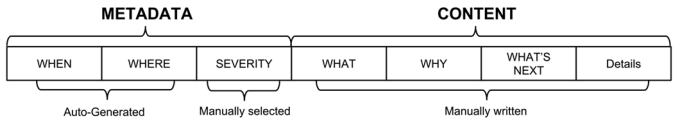


Figure 1: Anatomy of a log entry (simple)

Inconsistency

Modern systems that use microservice architectures typically aggregate logs from multiple applications. The inconsistency problem arises at the edge between apps when you attempt to correlate events across several services. Because these problems largely pertain to the auto-generated portion of logs, you can address them systematically by establishing a set of rules about metadata and content.

For example, while one engineer might assign a severity level of INFO to a given error notification, another might classify that notification as an ERROR. The same uncertainty might apply to WARNING versus ERROR classifications, ERROR versus CRITICAL classifications, and so on. You can overcome this inconsistency problem by establishing clear guidelines around appropriate severity levels.

It can also be difficult to track *where* events occur across microservices, as identity elements like process and thread IDs, host, service, executable names, and source code pointers tend to be tightly coupled to specific service instances. To overcome this inconsistency, consider introducing global variables, such as a unique Request ID that's randomly generated in a front line server and passed along all data paths.

Missing Data

Sometimes the content portion of a log record is missing a chunk of valuable information, which means the log entry is less meaningful or even meaningless. Overcoming this issue is the difference between implementing log messages and making sure that their recipients can read meaningful information.

In order to craft meaningful log messages, put yourself in the shoes of potential readers. A log entry's audience likely needs much more context about the application than the engineer who crafts the message. In concrete terms, consider two of the previously mentioned "W" questions: *what* and *why* (as *where* and *when* were automatically generated). After writing a baseline message, recursively iterate over each "W" question until the message has only one possible meaning.

Tips and Tricks

Now that you're familiar with the anatomy of a log entry and the broad categories of complications, we'll address some of the most common problems with the structure of logging contents.

Metadata Issues

Time Zone

Some timestamp formats may not include critical details, or may include non-essential information. For example, it's very important for log messages to include time zone information—particularly if your organization spans or will someday span more than one location—whereas including the weekday is simply a waste of space. Daylight saving time is another hidden issue: even if you use a single time zone for all systems and services, this quirk can translate into either an empty hour of data or two independent sets of records captured for the same hour.

To address these issues, the timestamp identifier should always include the time zone. We recommend basing your time zone on UTC. Unlike GMT, UTC is not impacted by daylight saving time. You likely also need to implement a time converter to align logs across time zones.

Severity Level

As previously mentioned, there's a lot of room for discordance in imprecise name-based severity-level definitions. Overcome this friction by establishing clear organization-wide guidelines.

INFO vs. DEBUG

You can define the boundary between INFO and DEBUG buckets by restricting the INFO level to represent milestone information. You also need to account for two known gray areas:

- Logs that describe decisions that an application made. For example:

```
Descale cluster 'abc' from 7 to 5 nodes. Reason: average node load < 30% for the last 10min.
```

Note: Here, and for most of the examples that follow, we've omitted the metadata from log entries since this metadata takes up space and isn't particularly relevant.

- Operational logs that require an attached severity level. For example, if user request logs pass through the same logging mechanism as other entries.

You might classify both of these cases as INFO messages: while they're not clear milestones, they roughly represent a logical point in the code, reached during execution.

If a given log record contains only statement information, but you also need to provide additional knowledge about that information, we recommend distributing the information across two log messages: the statement itself as an INFO message, and the additional information as a DEBUG message. This approach maintains clear information on every level, and the person who reads the logs can easily decide which level of detail they want to see. DEBUG messages are ideal for hosting information that doesn't fit into other level criteria or information that may be valuable in the future.

WARNING vs. ERROR Messages

We recommend differentiating between WARNING versus ERROR messages according to application behavior:

- **WARNING**: If the app can automatically recover from this state
- **ERROR:** If the app can't automatically recover from this state

You can implement these classifications on a more granular level—e.g., on an individual thread, branch, or transaction level.

For example, the following issue occurred in the middle of the request processing:

```
Request "/api/v1/get?obj_id=12345678"
Attempt to retrieve from cache.
Unable to resolve "cache-farm.example.com": Not found. Abort.
Cache miss.
Attempt to retrieve from origin.
Object obtained.
Response sent.
200 OK "/api/v1/get?obj_id=12345678"
```

If you're treating signals on a thread basis, you should treat the highlighted alert as a WARNING because processing isn't blocked. If you're treating signals from a branch perspective, you should mark the alert as an ERROR because the operational branch that retrieves objects from the cache was aborted and reached its logical end. By handling alerts based upon the branch depth, some ERRORs don't result as an error in the overall request processing. However, you can identify smaller anomalies faster.

In the following example, which attempts to reach an unavailable API, a set of WARNING messages precedes the final ERROR message. The service attempts to connect five times before giving up. The overall procedure is not yet aborted during retries, so the first four requests are marked with WARNING messages; only the last attempt is stated as an ERROR. If your logs only accounted for ERROR messages, you'd only see the final message, which doesn't tell the entire story. It's key here that the ERROR message explicitly references the five previous attempts and four WARNING messages—otherwise, the WARNING messages may be buried among hundreds of other unrelated messages, and the reader might not even realize that there were previous attempts to connect to the API.

```
...
WARNING "Unable to connect to Awesome API. Connection timed out. Attempt 3/5"
WARNING "Unable to connect to Awesome API. Connection timed out. Attempt 4/5"
ERROR "Unable to connect to Awesome API after 5 attempts. Connection timed out. Exiting."
```

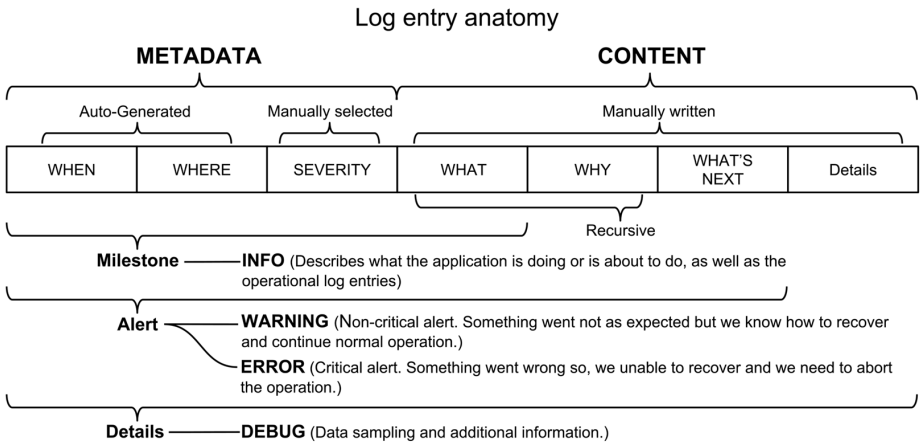


Figure 2: Anatomy of a log entry (complex)

ERROR vs. CRITICAL Severity

While a CRITICAL severity doesn't necessarily indicate that something bad happened, an ERROR severity unequivocally does.

We've personally found that CRITICAL severity isn't useful in most scenarios and have chosen to do away with that severity entirely. However, you may find useful scenarios for using the CRITICAL severity level. Be sure to determine a precise definition of what information is deemed CRITICAL and how to clearly distinguish that information.

Figure 2 consolidates a proposed schema for severity levels and their meanings.

Content Issues

INFO

INFO is a simple statement addressing *what* happened or is about to happen. Your aim in crafting this message should be to provide clarity.

Consider the following message, which, although accurate, isn't sufficiently descriptive:

```
Server has started.
```

This message fails to indicate why this information is important. Your *what* questions should only have one answer. In this case, is there more than one server that could have started? If so, which server started?

The following message is a marked improvement:

```
HTTP API Server has started.
```

You can improve further upon an INFO message by asking, *What is the most valuable information about the subject of this event?* In this case, *What is the most valuable information about the HTTP API server?* For any network communication HTTP server, the answer to this question is the entry point:

```
API Server start listening on http://0.0.0.0:80.
```

This event statement is three times more useful than the original.

ERROR and WARNING Messages

ERROR and WARNING messages are categorized as alerts, which describe the difference between the expected and actual behavior: *We expected A, but got B.* To craft meaningful ERROR and WARNING messages, ask yourself:

- What happened?
- Why is there a difference between the expected and actual conditions?

In the following example, we attempted to call an HTTP API and received an error. We'll iterate a couple of times on *What* and *Why* in order to demystify details.

What happened?

```
Unsuccessful API call.
```

What was the reason for this call?

```
Unable to retrieve the data object via API.
```

What data object?

```
Unable to retrieve a file's metadata via API.
```

What file?

```
Unable to retrieve metadata for the "abc123" file via API.
```

What API?

```
Unable to retrieve metadata for the "abc123" file from "https://
api.example.com/v1/get_meta?obj=abc123".
```

Why?

```
Unable to retrieve metadata for the "abc123" file from "https://
api.example.com/v1/get_meta?obj=abc123": Failed to parse
server response.
```

What further information do we know about the server response?

```
Unable to retrieve metadata for the "abc123" file from "https://
api.example.com/v1/get_meta?obj=abc123": Failed to parse JSON
response.
```

Why?

```
Unable to retrieve metadata for the "abc123" file from "https://
api.example.com/v1/get_meta?obj=abc123": Failed to parse JSON
response. No JSON object could be decoded.
```

What happens next?

```
Unable to retrieve metadata for the "abc123" file from "https://
api.example.com/v1/get_meta?obj=abc123": Failed to parse JSON
response. No JSON object could be decoded. Aborting.
```

Working from the original message text, gathering all this information from the live system would take minutes or even hours. By iterating through this series of questions, we gather information that the original message didn't provide:

- **Exact file name:** We can make a request for this object in a separate system to clarify the current object condition, including the state of its metadata.
- **Exact URL of a metadata request:** Now we can quickly request the metadata again for further inspection without having to craft this request from scratch. We can also immediately verify that the data was requested from the correct place and with the correct environment (API version, additional parameters, modifiers, etc.).
- **The data source service:** We know who to page in case we encounter a massive issue.
- **Metadata encoding format:** JSON.
- **Final action:** No additional retries were made, and the requester did not get the data they needed.

It's worth emphasizing the *What's next?* question here, which comes in handy when you encounter a vague statement. In this example, the reader was left with questions like, *Will the system retry requesting data from another source? Was this the only attempt?* and *Did the original request ever succeed?* The "Aborting" statement, which reports the application's next expected action, removes all these uncertainties.

Accounting for sensitive information in DEBUG messages

Take care in choosing how to represent the original alert as a DEBUG message. Whenever you log a working data sample, even partially, make sure that the sample doesn't contain sensitive information. Publishing such information accidently can pose a threat and security risk to people and systems!

Alternatively, when an error requires a long explanation (for example, with potential causes and suggestions for various methods of mitigation), instead of packing the entire text into the message itself, consider assigning a unique ID and providing a reference to a full explanation. The disadvantage of this method is that the log entry may not explain the situation. On the other hand, the description can be highly enriched with background information, solution playbooks, examples, and so on.

DEBUG

The debugging level serves two main purposes: increasing output verbosity and providing data samples.

When it comes to increasing output verbosity, DEBUG messages can better detail lower-level milestones and illustrate ongoing values, which are useful to real-time application tracing. For example, consider a well-known open-source "OpenSSH" utility. Both the server and the client support extended verbosity that displays the files being read, network connection establishment details, cryptography negotiation, and much more, thereby helping the reader understand how the utility works.

Data sampling is relatively straightforward. Consider it in the context of the previous file metadata retrieval example. The next logical step for a reader faced with the message "No JSON object could be decoded" is to examine the content of this object. We can make the reader's life easier by placing this data as a DEBUG message that follows the original alert.

You need to decide whether or not to include the original alert information in the DEBUG message—does it suit your purpose better to optimize for saving storage or to optimize for individual message clarity? Because you can link these messages using metadata, there's no strong need to include the original message. However, longer records can save valuable time during incident investigations, and they aren't filtered out from non-metadata searches.

Our final DEBUG entry perfectly aligns with the log entry anatomy scheme:

Complete entry:

```
2019.01.15 00:55:12.345012 UTC 43526 62837563 file_manager
[.../example_api/__init__.py:1024] DEBUG Unable to retrieve
metadata for the "abc123" file from "https://api.example.com/
v1/get_meta?obj=abc123": Failed to parse JSON response. No
JSON object could be decoded. Aborting. Response: 'Internal
Server Error.'
```

Breakdown:

2019.01.15 00:55:12.345012 UTC	WHEN	
43526	Process ID	WHERE
62837563	Thread ID	
file_manager	Binary Name	
[.../example_api/__init__.py:1024]	Code Pointer	
DEBUG	SEVERITY	
Unable to retrieve metadata for the "abc123" file from "https://api.example.com/v1/get_meta?obj=abc123"	WHAT	
Failed to parse JSON response. No JSON object could be decoded.	WHY	
Aborting.	WHAT'S NEXT	
Response: 'Internal Server Error.'	Details	

Message Formatting and Processing

Formatting

You can use formatting to coherently represent metadata. Aim to keep your formatting brief but sufficiently explicit.

Here's an example of formatting that's applied to Kubernetes log metadata (you can see the original format description at https://github.com/kubernetes/klog/blob/master/klog.go):

```
I0115 02:31:05.029108 1083 server.go:796] GET /stats/summary/:
(10.507359ms) 200 [[Go-http-client/1.1] 10.44.1.11:60556]
```

Note the following:

- The severity level is collapsed to a single capitalized character, *I*, which stands for INFO. You can represent the other severity levels with the letters *W (WARNING)*, *E (ERROR)*, and *D (DEBUG)*. Because the severity is the first character in the line, it can be easily expressed in a regular expression.
- The four digits concatenated with the severity level represent a date: `0115` refers to January 15, and the year is omitted, likely because records aren't stored for more than 12 months or because this information is added during entry processing.
- A single closing square bracket (*]*) separates the metadata from the content.

You could improve this formatting by accounting for time zones. For example, Google Cloud Platform collects the full timestamp, along with additional metadata.

By consolidating the date format, this record is readable, contains *almost* all the data we need, and saves about 10 bytes of space per record.

Output

Messages to the standard file descriptor should be delivered to STDERR, as opposed to STDOUT. Because users and various tools expect messages to appear on the STDERR, messages directed to STDOUT will be ignored or potentially cause harm by injecting data into a data flow pipeline.

You can save time and ensure log format consistency by creating a small set of libraries for various languages, which can coordinate all logging configuration out of the box.

Multi-line Messages

Multi-line entries can be problematic when log processing software treats messages as one entry per line. You can mitigate this problem by performing an additional layer of pre-processing on the application level: you can adjust every outgoing string by replacing the newline character with another unique string (`\n`, for example), so that the message can be restored by reverse conversion.

This solution's only drawback is message length. For example, a Java stack trace may run up against the maximum permitted message size in the processing or delivery stage. If you run into this problem, you can consider splitting one message into a sequence of several messages.

As a practical example, consider the shell script below. Shell scripts notoriously suffer from poor logging. We can improve the script by replacing the simple `echo` function with the more meaningful `log_info`. By logging INFO messages, we address the previously discussed time zone and output issues, thereby accommodating compact formatting and multi-line entries.

Here's an example implementation, distributed under the Apache 2.0 license:

```
$ cat standardized_bash_log.sh
#!/bin/bash
date_fmt='%m%d %H:%M:%S'
tz='UTC'

log_preproc(){
   echo "$@" | awk -v ORS='' '{if (NR!=1) $0 = "\\n" $0};{print}'
}
```

```
log(){
   metadata="${1}$(TZ=$tz date "+$date_fmt") $tz $$ \
$(basename $0)"
   content=$(log_preproc "$2")
   echo "${metadata}] $content" >&2
}

log_info(){
   log 'I' "$@"
}
```

Usage:

```
$ cat logging_example.sh
#!/bin/bash
source standardized_bash_log.sh
log_info 'The first line of text;
   The second;
   And finally, the third one.'
```

Execution with multi-line restoration:

```
$ echo -e $(./logging_example.sh 2>&1 | grep "second")
I0116 07:42:59 UTC 40844 logging_example.sh] The first line of
text;
 The second;
 And finally, the third one.
```

Storage

Root cause analysis (RCA) benefits from robust logging data. However, crafting and storing a comprehensive set of logging records requires a prohibitively large amount of storage. Does storing all logging data from all apps and environments in full really make sense?

When performing any kind of RCA, each investigation is initiated by an error. An investigator needs the information surrounding this problematic event. In reality, you need to store operational logs (user requests and decision-making information) in full, as these events are unique and unrelated to each other. The behavioral logs can be partially truncated because event sequences are repetitive.

To reduce storage volume, you can place a filter between an application and the delivery mechanism. For example, a filter can accumulate all messages for the last two to five minutes of operation in a buffer; when an error occurs, the filter dumps the entire buffer to remote storage. An engineer can then find all error-related records as well as all potentially correlated events that were in flight during the incident.

This approach has a couple of positive side effects. Because the price of log storage depends directly on the number of errors the service experiences, the fewer errors your service undergoes, the less storage you need. The filter can also prevent the system from flooding the logging system with identical errors.

If you're concerned about potentially problematic situations that don't produce errors and hence can't be easily detected, you can keep the entire logging set locally on the host with a shorter retention period.

If you want to minimize storage use when conducting other research and development, you can narrow the observation scope to a single application instance for a certain period of time. That way, you can easily reclaim the space occupied by locally stored data once the experiment is complete.

Conclusion

Many of the problems with logging in modern computing can be addressed by bridging the gap between the people writing and reading the logs. You can narrow this distance by clarifying and restricting the meanings of various terms and by using a question-based approach to ensure that you express all of the necessary data. We hope you find the recommendations in this article useful and that you adjust our approaches according to your preferences and experience, improve them, and share your further ideas and best practices with your team and beyond.