logdna.com

LOGDNA EBOOK

The Importance of Logging Across the SDLC

How to include logging in a way that maximizes its value not only in delivering the platform successfully but also in maintaining it throughout its lifetime.

INTRODUCTION

Traditionally, logging was most commonly associated with the post-deployment part of the software development lifecycle, or SDLC. Logs typically served first and foremost to help IT engineers find and troubleshoot problems that arose in production.

Today, however, logging can help teams optimize much more than just production-environment application management. And indeed, logging needs to be leveraged across all stages of the SDLC in order to ensure the reliable, continuous delivery of software. Developers, testing teams, and anyone else involved in software delivery must make use of logs and log analysis as one way to ensure the smooth flow of code across the entire SDLC.

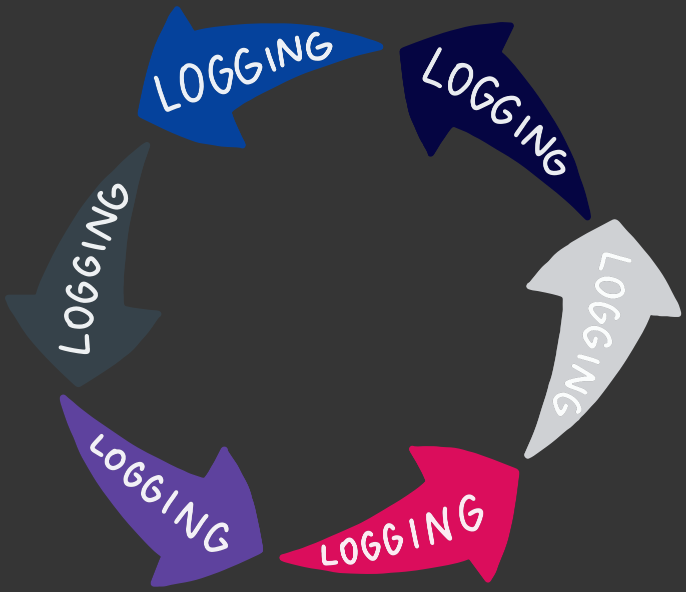


TABLE OF CONTENTS

THE IMPORTANCE OF LOGGING ACROSS THE SDLC

There are multiple phases in the software development process that need to be completed before the software can be released into production. Those phases, which are typically iterative, are part of what we call the software development life cycle, or SDLC. During this cycle, developers and software analysts also aim to satisfy nonfunctional requirements like reliability, maintainability, and performance.

One of the most critical services that developers can include in their applications is logging. Logging is a way to expose contextual information along with the main application runtime. When developers distribute those applications in a live environment, they will collect and store logs, either locally or in an external service.

There's usually not much debate about whether or not to include logging, because almost everyone expects it to be included by default.

The discussion that really matters revolves around how to include logging in a way that maximizes its value not only in delivering the platform successfully but also in maintaining it throughout its lifetime.

Keep reading for an explanation of some of the most compelling reasons why logging matters throughout the SDLC.

Logging Improves Software Maintainability

Developers use logs to perform several crucial tasks like debugging, load testing, and performance testing. They almost always capture log errors and fatal exceptions (because those are usually the most important) and then export them to external services like LogDNA for further processing.

However, if you only log errors or fatal exceptions, you can miss significant information. To enable more extensive logging, you can utilize code libraries, which offer a proper set of methods and filters to configure the log information based on a level of sensitivity. Developers can then preconfigure apps to collect or store logs with a specific format and log level.

Utilizing different log levels in different places throughout the codebase allows developers to handle two important things. First, it allows them to configure the sensitivity of the information that gets stored to match the parameters of what they aim to capture. For example, they can include stack traces or detailed context using debug or trace log levels. Second, it allows developers to configure the sensitivity of the log collectors or external services to respond to events that satisfy specific criteria and then forward them into relevant channels.

The aim is to make sure that there will be instances in the code where logging operations are not just distinct actions, but also ways to communicate intent throughout the SDLC. That way, future maintainers can refactor the code easier without breaking any assumptions about the logging operations. Let's take a look at an example of feature-driven logging in Rails in which we try to log user activity information per request. To use the LogDNA Logger instance directly, you would type:

```
logger.debug("SSL Purchase Created for
'#{domain.name}'")
```

Instead of doing that, you need to use a user activity logging module to wrap those events into methods, like this:

```
module ActivityLogging

  extend ActiveSupport::Concern

  def log_ssl_purchase(domain)

     logger.debug("SSL Purchase Created
     for '#{domain.name}'") end

  end

end
```

That way, if you try to change the log details or the level for the SSL Purchase event, you will only have to revise it in one place. Using logging facilities like this is a good way to logically scope them and configure them in a more granular way, thus making the application easier to maintain.

Logging Helps with Migration Phases

You can use logging to record how and when certain method calls are made and also to monitor parts of the code from which you want to migrate. You'll want to log these often and ascertain how much of the existing system is dependent on them. Let's walk through some typical use cases:

Logging Deprecated Features

Our first example is deprecation log messages, wherein we log the usage of functions that are deprecated and slated to be removed in future versions. One way to do this is to wrap the usage of deprecated functions with function decorators. When an attempt is made to call this function, we can log the call and record it into the stream of events. This is what a rough implementation looks like in Javascript using the class decorator syntax for conciseness:

```
@deprecated({useInstead:
'UserManagementService', since:
'v1.0', onCall: (params) =>
logDeprecatedEvent(params)})

class UserApiService {

    static getUsers() { }

}

function logDeprecatedEvent(params):
void {

   Logger.getInstance().info("Deprecated
   call of function with params: " +
   params.toString())

}
```

As far as the client knows, the **`UserApiService`** will still work as expected, but it will now log the call in **`logDeprecatedEvent`** to capture a depreciation event.

Logging Feature Flag Usage

You can log information whenever a feature flag is modified and used to expose new features to users. By doing so, you can gain a more thorough understanding of how the usage of new features affects the existing application performance or reliability. For example, once you've enabled a flag, you might log this information:

```
logger.info("Feature Flag", {"feature_
flag_update": {"name": '#{flag.name}',
"from": "'#{flag.isEnabled}'",

"to":"'#{!flag.isEnabled}'"}})
```

Then, using this as a guidepost, you can record any performance or related changes in the application and measure their success or failure.

Logging Helps with Managing New Work vs Technical Debt

Technical debt is a term that developers use to describe unfinished work that they have postponed due to time or technical limitations. It shouldn't be a frightening word, and it does not indicate poor work ethic. It's relatively common to produce working code with few extra dependencies, quality issues, coupling, or inflexible patterns. Trying to refactor those pieces into a more reusable component may prove to be counter-productive, since it might not be very valuable to the end users, and most of the time, it may not be required.

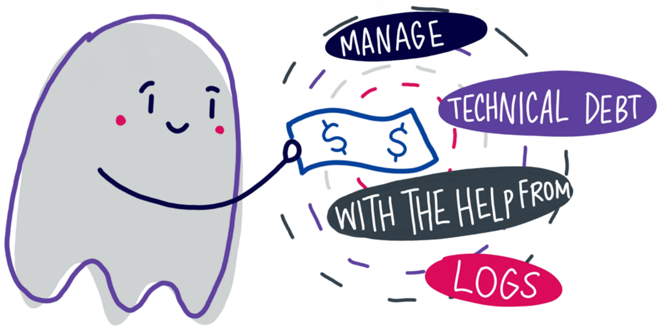
It's at this point in the SDLC that developers might choose to delay any such changes to the existing codebase, thereby managing the technical debt. This does not mean that the code will not change in the future; if it did, then the developers would have to repay this technical debt by spending more time refactoring.

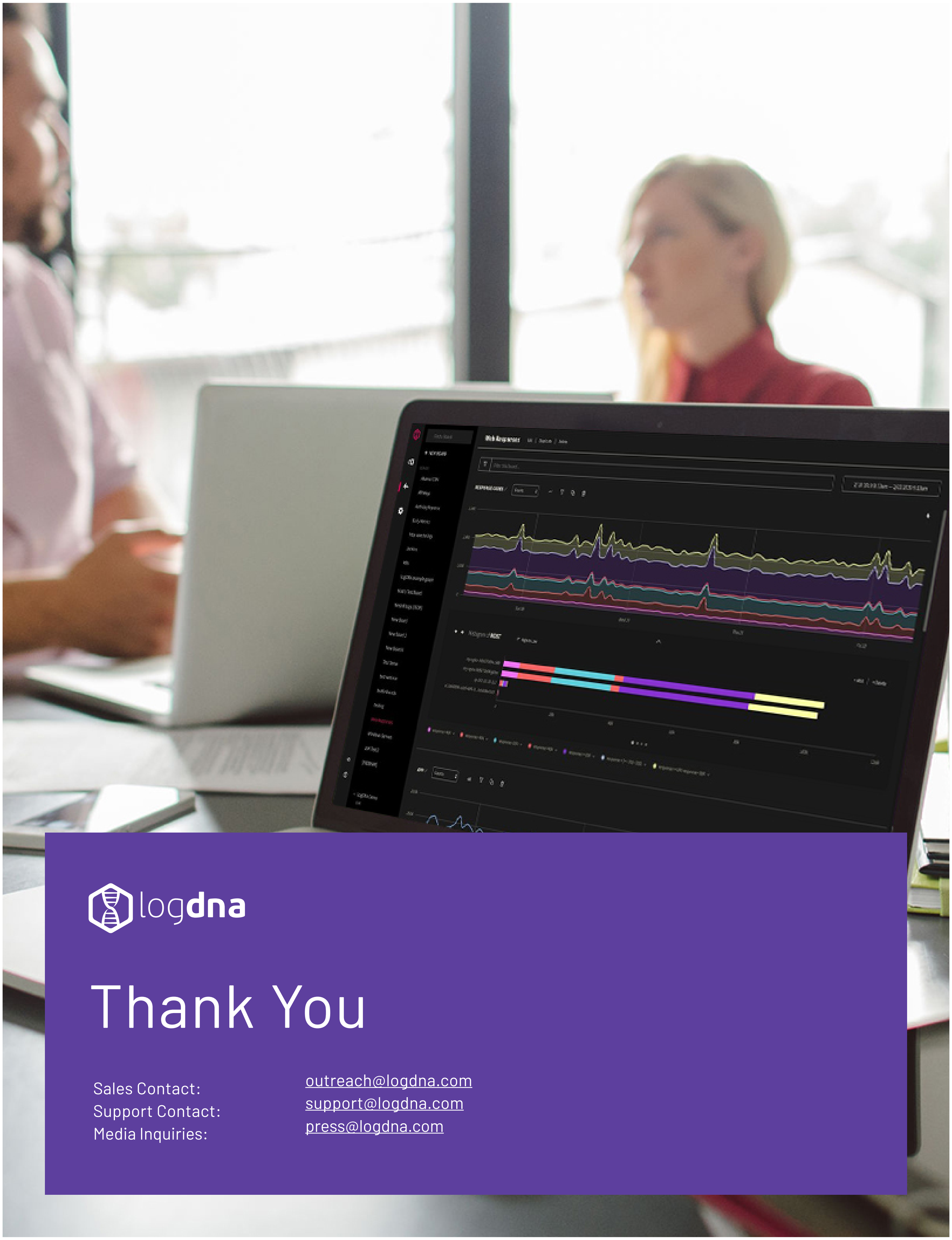
Ultimately, technical debt can become an issue if left unchecked. For example, it can become a problem if you write code without considering reusability, or if you add feature flags here and there without removing them in later stages.

Logs can help manage technical debt by enabling you to compare existing nonfunctional requirements and metrics like performance, error recovery, and availability metrics. For example, developers might touch a piece of code that already exists by including extra features and releasing it to production. If they had log monitors for nonfunctional requirements in place, they could compare the new behavior to the original and assess changes.
For example, did the new code trigger more error messages? Did it reduce build time? Did it reduce memory consumption, or did it increase it? How much time did it take to fix a particular error? All of these are important factors to consider when maintaining new and existing features with logging.

Next Steps with Logging

The aforementioned reasons for including logging are excellent illustrations of just how critical it is to deliver maintainable and reliable software. To realize the enormous potential of logging and improve the SDLC process, you can always rely on a dedicated logging platform that offers a variety of related services – like LogDNA. For example, you can receive alerts and notifications when preconfigured conditions are met using LogDNA Alerts. Their Boards and Graphs give you a completely customizable visualization dashboard that will allow you to present high-level, comprehensive metrics to stakeholders. By pairing those with Time-shifted Graphs, you can see how an app performs across different versions.





logdna

Thank You

Sales Contact:	outreach@logdna.com
Support Contact:	support@logdna.com
Media Inquiries:	press@logdna.com