The #1 Productivity Challenge:

Optimize Debugging: From Monolith to Distributed Environments

Computer bugs, contrary to popular belief, are not part of modern made up technological terminology. The term actually refers to an event that occurred on September 9, 1947, when Grace Murray Hopper went looking for the source of the computer's issue and found a moth stuck between relay contacts in the computer. Thus 'computer bugs' and then just 'bugs' came into the technical terminology of the software engineering world, and an important one at that.

60% of companies find that their engineers' time is spent finding and fixing errors. According to the Cambridge Judge Business School MBA report, 620 million developer hours a year are wasted on debugging software failures, at a cost of roughly $61 billion annually. The report also notes software engineers spend on average 13 hours to fix a single software failure.

KEY TAKEAWAYS:

When choosing your software's architecture, it's important to consider the ease- or the challenges- of debugging that comes with it.

This paper will show that while those at the ends of the monolith-distributed spectrum pose very different challenges debugging, all the challenges ultimately stem from the same source: **access to data that is simple, quick, and safe to capture.**

68% of organizations report that they face a tradeoff between shipping software faster, but without the data that they need to ensure optimal performance, or to delay releases while trying to fetch the data.

No matter which **architecture you work with, the key point remains: no matter what you do with your code, fast, secure, and simple data access is an absolute must.**

The paper will cover all three criteria and how to get them right

With that in mind, choosing your software's architecture is no simple task. One important consideration when doing so is the ease, or the challenges, of debugging that come with it. Whether it is a monolithic architecture or a distributed system, they both present individual challenges to the debugging process. As we analyze this architectural spectrum, this paper will show that while they pose very different challenges debugging, all the challenges ultimately stem from the same source- access to data that is simple, quick, and safe to capture. We'll discuss this common source and find techniques to overcome it.

In order to debug your chosen architecture successfully, data is an absolute necessity. The access to the correct data affords an understanding into your software, of which no debugging process could be successful without. To optimize your debugging, let's delve into each architecture and the challenges encountered when debugging them.

> ***Within distributed systems, when code leaves your machine, it is near impossible to track.***

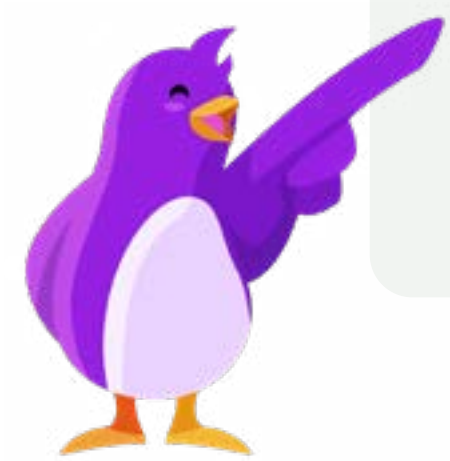
Problems of Debugging Monoliths

Monoliths, while they are inherently simpler structures, are still difficult to debug when experiencing a lack of data. 52% of software companies report developer productivity is hindered by legacy systems, which are difficult to understand, thus necessitating a need for more data in order to do so, and complicating debugging the system even further.

When compared to distributed environments, monoliths have some advantages when it comes to debugging. The fact that a monolith has a single code base, that often runs as a single process, makes it easy and straightforward to run in debug mode, attach a debugger, and move step-step-step until you find the problem. However, monoliths tend to justify their namesake by becoming robust, heavy monsters that are too resource intensive to run on a single machine. And step-step-stepping through a huge legacy code base tends to become a gigantic and time consuming task that only the most seasoned and patient developers are willing to endure. Even a log based debugging approach hits the wall with monoliths, as adding a single line of log will require a long and expensive rebuild and redeploy cycle.

When attempting data capture in monolith architectures, it is important to note that since they are typically created earlier, they often hold the infrastructural elements of your software. Thus debugging them can be a daunting task, as any attempt to capture data by changing and redeploying the code may potentially hit interdependencies, leading to a chain of malfunctions and crashes.

The key to debugging monoliths, then, is not to debug them in the now old fashioned way. Instead, it should be about having the correct data to understand what is happening in the system. For example, when working on an inherited system, it is challenging to understand the old code that was written. The correct data, for instance, will prevent the scenario in which the developer will spend countless hours attempting to get into the head of the person who wrote that old code, often to no success.



> *52% of software companies report developer productivity is hindered by legacy systems*

Problems of Debugging Distributed Architectures

Distributed systems have their own set of challenges when it comes down to the basics of debugging them, due to their inherent structure. Their architecture makes them complex entities and their many moving parts cause tracking down the root cause of the issue or the bug to be extremely tricky and complicated. Attaching to a debugger and setting a breakpoint is just not something you can do, and debugging with log lines has its own set of challenges.

When code leaves your machine, it is near impossible to track. Distributed architectures, and specifically microservices based applications or serverless applications, are often ephemeral. Servers or functions or pods will spin up and down dynamically, and there is no host you can remotely attach to or SSH to fetch log files. Code versions will change dynamically, and you will spend a significant amount of time building a logging and observability pipeline to make sure the data reported by these applications reaches the right data sink.

When attempting data capture in distributed architectures, it is important to find a set of tools that can address these challenges and allow you to fetch data in real time, with a minimal amount of effort.

The Common Thread

The need to access data

The lack of data, and therefore the need to get that data, is the underlying problem in all debugging challenges. This lack of data and the way to resolve it is comprised of three core criteria:

1. Speed- how fast it is to get the data you need to move forward?

Organizationally speaking, data capture via logging and redeployments often involve many levels of controls and approvals. The longer it takes, the more elements and metrics change and - in case of a bug -- the more damage gets accrued, all while slowing down developer velocity and accelerating risk exposure.

From the developer point of view, the longer data capture becomes, the less they are willing to wait for it. 68% of organizations in Digital Enterprise Journal's upcoming study on Enabling Top Performing Engineering Teams reported that they are experiencing a 'flying blind or flying slow' type of challenge. In other words, they are facing a tradeoff to either ship software faster, but without the data that they need to ensure optimal performance, or to delay releases while trying to get the data. The longer those delays, the more developers are prone to choose flying blind.

> ***620 million developer hours a year are wasted on debugging software failures, at a cost of roughly $61 billion annually.***

2. Simplicity- how simple is the process to data capture?

The most straightforward, though sometimes most difficult criterion to get right is avoiding complexity. When developing a new application, it's imperative to have a clear understanding of the problem domain and the issue you are trying to solve. The simpler and more focused the business requirements are, the less complex the solution is going to be.

Data capture complexity can also stem from technical debt. Tech debt needs to be minimized by keeping the code as organized, clear, concise and well documented as possible, so different stakeholders who need to interact with it can understand it with minimal data capture actions.

Which brings us to the process the organization goes through to get to a new release. Quite intuitively, the simpler and shorter that process is, the easier it will be to get the data you need (see Figure 1 below).

3. Safety- how safe is it to retrieve your data?

This element is made up of two specific components: generating new releases and the need for secure access to data.

Generating new releases over and over again always poses a risk. No matter what method you use to generate a release, whether it's writing new code and thus writing in new bugs, or triggering code, it's a process fraught with challenges. Minimizing build-test-deploy cycles contributes significantly to narrowing the exposure.

The need for secure access to data means that you need metrics and tracing data, but you obtain it in a controlled way as to who sees the data, particularly in industries such as finance and healthcare where dealing with data is a governance and compliance mega challenge.

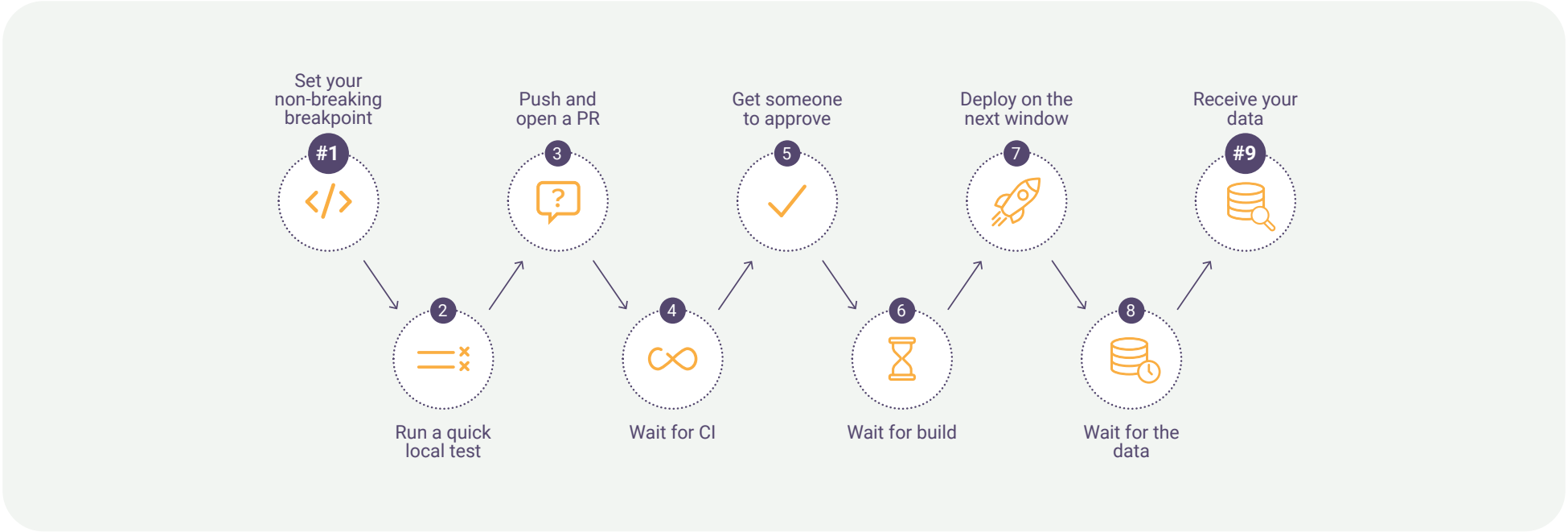


Figure 1: Typical Data Retrieval Process

Logging isn't enough

Unfortunately, logging won't solve your debugging problems, no matter if you set more log lines or haven't set enough. When over-logging, or what is known as Logging FOMO, a developer writes loglines in every possible place that they can in order to ensure they get the most amount of data they need. While this may seem like a good idea data-wise, it actually gives you an overload of data that is mostly irrelevant, creating too much noise in your code and skyrocketing the costs to store and analyze that log data. Being overly-careful with your log line placement, or having forgotten to set a log line due to ill-preparation, doesn't help either. Unfortunately, no matter which type of logging you choose, it won't necessarily get you the data you need to help you debug in a sustainable way.

> ***No matter what you do with your code, secure and simple data access is an absolute must.***

So what do we do about it?

No matter which architecture you work with, the key point remains: no matter what you do with your code, secure and simple data access is an absolute must. In order to debug, you need data and in order for it to be a smooth and easy process, you need to be able to get the data you need, as soon as you need it.

Following the above principles, Rookout's mission is just that - to provide secure, fast, and simple data access, no matter what architecture or where the code is running. By providing the developer with instant, laser-focused data, developer teams can get data to debug any architecture in a millisecond. As a result, our clients saw significant increases in dev velocity, code quality and even stress factor reductions.

The use of Non-Breaking Breakpoints empowers engineers to find the data they need on the fly, and deliver it anywhere, enabling them to better understand and advance their software. This allows them to maintain it much more easily and efficiently, which gives them more time to create other features and keep up with their workloads. This whitepaper allows access to the Rookout sandbox, to further illustrate how data can be captured in a click, while in compliance with SOC 2 Type 2, ISO 27001, GDPR and HIPAA certifications.

Happy debugging starts today.

TRY FREE