13
Instrumentation

> Everything that happens happens as it should, and if you observe carefully, you will find this to be so
> — *Marcus Aurelius*

Why you care: *Before you can run any experiments, you must have instrumentation in place to log what is happening to the users and the system (e.g., website, application). Moreover, every business should have a baseline understanding of how the system is performing and how users interact with it, which requires instrumentation. When running experiments, having rich data about what users saw, their interactions (e.g., clicks, hovers, and time-to-click), and system performance (e.g., latencies) is critical.*

A detailed discussion of how to instrument a system is beyond the scope of this book and highly dependent on system architecture (Wikipedia contributors, List of .NET libraries and frameworks 2019, Wikipedia contributors, Logging as a Service 2019). This chapter discusses key points of instrumentation in the context of experimentation. Privacy is also a crucial consideration when it comes to instrumentation, which we discuss in Chapter 9. In the context of this book, we use the terms "instrument," "track," and "log" interchangeably.

Client-Side vs. Server-Side Instrumentation

When implementing instrumentation, understanding what happens client side vs. server side is important (Edmons et al. 2007, Zhang, Joseph and Rickabaugh 2018). The focus of client-side instrumentation is what the user experiences, including what they see and do, for example:

- **User actions:** What activities does the user do, such as clicks, hovers, scrolls? At what times are these done? What actions are done on the client without a server roundtrip? For example, there could be hovers generating help text or form field errors. Slideshows allow users to click and flip through slides, so capturing the times of those events is important.
- **Performance:** How long does it take the page (webpage or app page) to display or become interactive? In Chapter 5, we discuss the complexities around measuring the time from a search query request to displaying a full page.
- **Errors and crashes:** JavaScript errors are common, and may be browser dependent, and it is critical to track errors and crashes in client software.

System-side instrumentation focuses on what the system does, including:

- **Performance:** How long does it take for the server to generate the response, and which component takes the longest? What is the performance at the 99th percentile?
- **System response rate:** How many requests has the server received from the user? How many pages has the server served? How are retries handled?
- **System information:** How many exceptions or errors does the system throw? What is the cache hit rate?

Client-side instrumentation is useful as it offers a view of what the user sees and does. For example, client-side malware can overwrite what the server sends and this is only discoverable using client-side instrumentation (Kohavi et al. 2014). However, client-side instrumentation has drawbacks in terms of data accuracy and cost to the user. Here are specific concerns for JavaScript-based clients (for mobile-related concerns, see Chapter 12):

1. Client-side instrumentation can utilize significant CPU cycles and network bandwidth and deplete device batteries, impacting the user experience. Large JavaScript snippets will impact load time. This increased latency not only impacts user interaction on that visit, but also how likely those users will return (see Chapter 5).
2. The JavaScript instrumentation can be lossy (Kohavi, Longbotham and Walker 2010): web beacons are often used to track user interactions, such as when users click a link to go to a new site; however, these beacons may be lost when:
   a. A new site loads before the web beacon is successfully sent, meaning that the beacons can be cancelled and lost. The loss rate due to this race condition varies by browser.

b. We force the web beacon to be sent before the new site loads, such as via a synchronous redirect. While the beacon lossiness decreases, latency increases, resulting in a worse user experience and an increased likelihood of the user abandoning the click.
c. You can choose to implement either scenario depending on the application. For example, because ad clicks must be reliably tracked as they relate to payments and compliance requirements, b) is the preferred scenario even though there is added latency.
d. Client clock can be changed, manually or automatically. This means that the actual timing from the client may not be fully synchronized with server time, which must be considered in downstream processing. For example, never subtract client and server times, as they could be significantly off even after adjusting for time zones.

Server-side instrumentation suffers less from these concerns. It offers a less clear view of what the user is actually doing but can provide more granularity of what is happening inside your system and why. For example, you can log the time to generate the HTML for a page; because it is not impacted by the network, the data tend to have lower variance, allowing for more sensitive metrics. In search engine results, there are internal scores indicating why specific search results were returned and their ranking. Instrumenting these scores is useful for debugging and tuning the search algorithm. Another example is logging the actual servers or data center the request is served from, which allows for debugging of bad equipment or finding data centers under stress. It is important to remember that servers also need to be synchronized often. There can be scenarios where the request is served by one server while the beacon is logged by another, creating a mismatch in timestamp.

Processing Logs from Multiple Sources

It is likely you will have multiple logs from different instrumentation streams (Google 2019), such as:

- Logs from different client types (e.g., browser, mobile)
- Logs from servers
- Per-user state (e.g., opt-ins and opt-outs)

It is important that you ensure that the relevant logs can be easily utilized and combined by downstream processing. First, there must be a way to join logs. The ideal case is to have a common identifier in all logs to serve as a join key.

The join key must indicate which events are for the same user, or randomization unit (see Chapter 14). You may also need a join key for specific events. For instance, there can be a client-side event that indicates a user has seen a particular screen and a corresponding server-side event that explains why the user saw that particular screen and its elements. This join key would let you know that those events were two views of the same event shown to the same user.

Next, have some shared format to make downstream processing easier. This shared format can be common fields (e.g., timestamp, country, language, platform) and customized fields. Common fields are often the basis for segments used for analysis and targeting.

Culture of Instrumentation

Instrumentation should be treated as critical for the live site. Imagine flying a plane with broken instruments in the panel. It is clearly unsafe, yet teams may claim that there is no user impact to having broken instrumentation. How can they know? Those teams do not have the information to know whether this is a correct assumption because, without proper instrumentation, they are flying blind. Indeed, the most difficult part of instrumentation is getting engineers to instrument in the first place. This difficulty stems from both a time lag (time from when the code is written to when the results are examined), as well as a functional difference (i.e., the engineer creating the feature is often not the one analyzing the logs to see how it performs). Here are a few tips on how to improve this functional dissociation:

- Establish a cultural norm: nothing ships without instrumentation. Include instrumentation as part of the specification. Ensure that broken instrumentation has the same priority as a broken feature. It is too risky to fly a plane if the gas gauge or altimeter is broken, even if it can still fly.
- Invest in testing instrumentation during development. Engineers creating features can add any necessary instrumentation and can see the resulting instrumentation in tests prior to submitting their code (and code reviewers check!).
- Monitor the raw logs for quality. This includes things such as the number of events by key dimensions or invariants that should be true (i.e., timestamps fall within a particular range). Ensure that there are tools to detect outliers on key observations and metrics. When a problem is detected in instrumentation, developers across your organization should fix it right away.