CHAPTER 17
Reliability Reporting

Alex Hidalgo

At their absolute core, SLOs are a means of providing you with data you can use to have better discussions and therefore (hopefully!) make better decisions. The entire process is fundamentally about thinking about the data you have in ways that you might not have before. The data you collect via an SLO-based approach should be meaningful in terms of being representative of what your users and customers feel and experience every day using your service. Because of this, your SLO data becomes a prime candidate for reporting on your service status to other people.

Who these "other people" are will depend heavily upon your exact service and who its users and stakeholders are, but reporting doesn't have to just be about presenting quarterly status updates to management and shareholders. Using SLOs to report on the general health of a service to the team or teams that support it is also an incredibly powerful communication method.

Reporting on SLOs is valuable for you, your team, your organization, the teams you depend on, the teams you're dependent on, external customers, external businesses, and even more. By measuring the performance and reliability of your service from the perspective of your users, you're also compiling data that can be meaningful directly to those users. And remember: when we say *user*, we mean literally anyone—or anything—that depends on your service. If your SLIs and SLOs capture their experience and requirements well, then you also immediately have data you can use to have conversations with those users about how reliable you've been, how reliable you currently are, and how reliable you're aiming to be in the future.

This chapter explains the ways that service reliability is often reported on today; how SLO-based approaches are different, and why these approaches are much more meaningful than alternative systems; how to use these new numbers to make your

project decisions clear to others; and the most valuable SLO-based metrics you can use to build useful dashboards and other tracking mechanisms.

Basic Reporting

Your service will have many stakeholders, including some or all of the following:

- Software engineers who work on the service
- Operational engineers who keep it running
- Product teams who plot the purpose of it
- Customer operations teams who have to answer questions about it
- Leadership and business people who need it to operate well so that your organization or company can perform well

It is also often the case that your service has stakeholders who don't even realize they are stakeholders. The stakeholders in a high-level service are stakeholders in all of the services that it depends on, whether they are aware of this or not. Just because the owners of the frontend of a website don't often think about the reliability of the network hardware that delivers data to their service doesn't mean that they aren't also stakeholders.

The stakeholders will likely expect to get regular updates on how your service is doing, and/or to have a discoverable way of checking on its status in the event of a problem. When things go bad, people need to know how badly. If you are able to get buy-in across the company about an SLO-based approach to reliability, stakeholders will especially want to know how the service is performing against its SLO.

Luckily, SLOs are basically built to be self-reporting. They are directly based on metrics related to the service behaviors that the users—and therefore the business, and therefore also the stakeholders—care about. Monitoring performance against SLOs is therefore much more straightforward than trying to extract signals users or stakeholders care about out of low-level signals not tailored to match user experience.

Without SLOs, it is not easy to report on the reliability of a service. You could try to count pages fired or tickets opened, but these are prone to being inaccurate due to false positives or the whims of the people opening the tickets. You could try to report on something like uptime, availability, or error rates, but as we've discussed, those things rarely tell the whole story.

Because meaningful SLIs give you better insight into what users need, SLOs give you an easy way to say something along the lines of: "We have been reliable 99.76% of the time this quarter" or "We exceeded our error budget by 1 hour and 6 minutes this month." These are simple sentences that can be digested quickly. Not only will they give your engineering teams a better idea of how users were impacted by any

incidents that occurred, but leadership will be better able to infer potential revenue or customer confidence loss. You can use these numbers to build dashboards or populate documents that can then be discovered by anyone that wants to know how your service is performing.

But before delving into how you can meaningfully report on service and team health via SLOs, let's discuss how organizations today do this in an ineffective manner.

Counting Incidents

The most common way I've seen people try to quantify the reliability of their services is to count how many incidents have occurred over time. This is a completely reasonable starting point: it's not difficult to follow the logic that says that, since incidents cause moments of unreliability, many incidents implies more unreliability, while few incidents implies better reliability. However, once we start to look at the math surrounding this sort of approach, things unravel quickly.

The Importance of CVEs

by Isobel Redelmeier

There's a special type of incident that's worth calling out separately: the security incident. Besides letting the public and our customers know about attacks on our systems that may have affected their data, it's important to also let them know about vulnerabilities as they are discovered (and, hopefully, patched). This data lets them determine the appropriate mitigations, such as updating once a fix is available or monitoring for active exploitation.

As with other types of incidents, traditional metrics for tracking common vulnerabilities and exposures (CVEs) within a company may mislead or even lead to perverse incentives. We want our systems to both *be* secure and *seem* secure—and being and seeming secure are not always the same!

Consider counting CVEs with the goal of reducing their number in the future. On the surface, this is sensible. Alas, the easiest way to reduce the number of CVEs is not to stop having them in our systems, but instead to stop *finding* them; and attackers, unlike customers affected by outages, don't generally let us know first before exploiting whatever weaknesses they may discover.

There are other ways to better align the reality and appearance of your security posture. Publicly reporting a larger number of CVEs may, perhaps unintuitively, actually signal that you take security seriously and are therefore actively trying to detect vulnerabilities before they get abused. Especially when coupled with a proven commitment to patching them quickly, this sort of transparency is invaluable for protecting your users.

The first problem is that it's difficult to define what an *incident* even is. Is it any time a team is paged based on metric thresholds? In that case, you have to be sure that your alerting is incredibly accurate, that false positives never trigger, and that you're constantly updating your thresholds as various aspects of your system change. Chapter 8 discusses the problems surrounding threshold-based alerts in greater detail.

Alternatively, you might say that an event only qualifies as an incident when things are severe enough that you have to post a public status page update to your paying customers. Or perhaps you wait until customers are filing tickets and calling your support queue—except now your approach is entirely subjective and plenty of real problems could be hidden. Not everyone is going to make the same decision about when something needs to be communicated externally, just like not every customer that notices a problem is going to have the same threshold for when they decide to complain.

Due to the ambiguity surrounding exactly what an incident is, people often try to resolve the problem by introducing the concept of *severity levels*.

Severity Levels

The basic premise behind severity levels is to establish some number of buckets with strict definitions that allow you to categorize your incidents. Most often these levels follow some kind of numbering system, where the lowest number equals the most severe type of incident. Table 17-1 shows an example of a traditional set of severity levels.

Table 17-1. An example severity level list

Severity	Description
S5	A feature is less convenient to use or there is a minor cosmetic problem. Unnoticed by most users.
S4	A minor feature is not operating correctly, but there are obvious workarounds available. Likely unnoticed by most users.
S3	A major feature is experiencing sporadic unreliability, but there are workarounds and many users won't notice.
S2	A major feature has significant problems. If there are workarounds, they are difficult to discover. Most users will be aware.
S1	A major feature is completely down. There are no workarounds. All users are aware.
S0	The entire service is down. All users are aware.

The intended outcome of defining such a list of severities is that you can now classify any incident or outage that may occur within one of these severity levels. With this data, you can better report on the reliability of your service! Except you probably can't.

The problem with severity levels is that, while they provide you with some guidance in determining exactly what a reliability incident is, and they're certainly better than

just having a binary "Was this an incident or not?" dichotomy, there is still far too much ambiguity at play. No matter how extensive you make your descriptions for each level, you're never going to be able to completely accurately classify every problem you encounter.

For example, when trying to classify a problem you're encountering as S3 or S2, how do you determine where the dividing line is between *sporadic unreliability* and *significant problems*? You could develop a strong rubric for this by adding more qualifying criteria. But how would you classify a problem that's flapping between the two? It's not often the case that incidents remain in the same state as things evolve.

Suppose that for the last hour 5% of requests have been failing, but every 10 minutes the failure rate increases to 95% for 30 seconds or so. Is this incident an S3 (sporadic unavailability) or S2 (significant problems)? You may have decided that S3 corresponds to availability between 50% and 99% and S2 to availability between 0% and 50% for a particular feature, but the service is alternating between the two.

Additionally, we could easily imagine a problem that starts by exhibiting *sporadic unreliability*, moves on to exhibit *significant problems*, and finally is just *completely down*. How do you classify such an incident?

You could say something like, "The most severe level reached during an incident is how that incident should be classified." This seems fairly reasonable in some ways, but again I can come up with some very simple counterexamples that muddy this view very quickly. For example, do you really want to have to classify an incident that *at one point* and *for 15 seconds* was entirely down as an S1 incident when it spent a full hour being sporadically unreliable before and after that brief moment of complete unavailability?

We can go even deeper. What does it mean if you're down at 03:00, when you basically don't have any traffic, versus during your normal daily peak, versus on your most important business day of the year? How do you accommodate these differences with severity levels?

Despite these ambiguities, I normally see organizations just kind of accept them as necessary.

A much better way to measure things is to use error budgets. If you have a meaningful SLI and a reasonable SLO, the data an error budget gives you will be much more representative of your reliability from a user's point of view than severity levels/buckets.

The next step most organizations that are using severity levels take to try to measure reliability is introducing some math in order to figure out how often they experience incidents of a certain severity level. This is when *mean time to X* is introduced.

The Problem with Mean Time to X

Once organizations have established their severity levels, they often realize that they still don't have a way of thinking about reliability over time. This is when the *MTTX* acronyms tend to be introduced. MTTX stands for *mean time to <something>*, with the *X* taking many possible values. I've seen all of the following used out in the real world:

- Mean time to acknowledgment
- Mean time to detection
- Mean time to engagement
- Mean time to escalation
- Mean time to failure
- Mean time to first action
- Mean time to fix
- Mean time to know
- Mean time to mitigation
- Mean time to notification
- Mean time to recovery
- Mean time to remediation
- Mean time to repair
- Mean time to resolution
- Mean time to response

There are probably more examples of this concept out there. We all might be better served by tracking *mean time to new mean time measurement*. This list alone should be enough to start sowing some seeds of doubt about the entire concept.

The idea behind measuring any type of average (or mean) time is that these numbers can help you understand incidents of level S0 or S1 that impact the reliability of your service. That is, if you measure the mean time to failure in the event of S0 and S1 (and S2? or even S3?) incidents, you can better understand how often a particular service fails to work as intended. Furthermore, if you're measuring the mean time to resolution, you have a data point about how long it takes, on average, for your team to fix problems of a certain severity.

This chapter is about reliability reporting, and the arguments here against using MTTX methods must be viewed via that lens. The argument is not that MTTX is always misleading (in fact, we demonstrate some useful applications of these measurements in Chapter 9!) but that they become very nebulous for anything involving incidents. Measuring the time-to-failure of hardware components, for example, is useful information and these sorts of measurements are used by engineers of all disciplines; however as soon as there is a direct human element involved, things change a bit. A major focus of this book is that you need math, but also that you need to use that math in the most meaningful manner.

These sorts of measurements are very common in reporting on the health of a service, and making decisions about allocating resources. Has your *mean time to X* been too high for the previous quarter? Better allocate resources to bring that number down!

The problem here is threefold:

1. Incidents are unique.
2. Using averages for your math is fallible.
3. As with counting incidents or defining severity levels, defining what any of this even means in the first place can be very subjective.

Let's dive into why incidents are unique and why means aren't always meaningful.

Incidents are unique

Complex systems generally fail in ways that are unique and have different contributing causes and factors each time. When you want to measure the mean of the time it took for something to happen or be resolved or acknowledged, you're gathering dirty data. MTTX measurements can only work if you have strict severity buckets to ensure you're not comparing apples to oranges. We don't have the space here to go into the details of exploring and learning from complex system failures, but the core idea is that due to the many factors at play—including, not least, the humans involved—you can't reliably categorize system failures into the same bucket over and over again. (John Allspaw has written a great article on this.)

For example, let's talk about an incident type almost any web-facing service will have to endure at some point or another: a distributed denial of service (DDoS) attack. Even if the same type of attack is aimed at the same website more than once, there are still going to be differences. For example, the attackers will likely have different motivations and use different techniques, targeting different endpoints, sending different traffic patterns, and so on. These factors alone make these sorts of incidents unique.

Even if you're willing to bite those bullets, DDoS incidents can manifest in countless ways, making data you've collected about previous DDoS attacks you've endured murky in terms of being able to predict anything at all. You might think a DDoS is just a DDoS, but in reality there are a vast number of techniques and approaches that can be used to bring a system down due to load.

The most common type of DDoS attack is known as a *volumetric attack*. Attacks of this nature seek primarily to overwhelm the network bandwidth available to a service, making it difficult or impossible to respond to legitimate requests for data. One of the most common types of volumetric attack is the SYN flood, where tons and tons of incoming requests are sent to a single service. However, even this one type of attack can be split into many categories.

For example, you can have flood attacks that originate from a single source with lots of bandwidth, or you might be dealing with one operating from a botnet with thousands or even millions of source IPs. Additionally, spoofing the source IP is easier than most people realize, so chances are many of the source IPs involved in the attack aren't even the actual originating addresses.

And volumetric attacks are just one example. Some attacks seek to starve the resources of your network equipment and services instead of overwhelming them; others try to "bust the cache" by introducing request parameters that are just slightly unique each time, exhausting the space you have in your cache and preventing you from responding quickly enough to legitimate requests; still others target known weaknesses in applications rather than the server itself.

We've barely scratched the surface here, but it should be clear it doesn't make a lot of sense to consider all DDoS attacks as a single incident type. Even if you're willing to start counting these sorts of incidents into different buckets, though, how do you deal with an attack that switches tactics? Since DDoS attacks are orchestrated by humans, they'll often change up or diversify their strategy as time goes on. If they see your teams successfully mitigate one approach, they might switch to another. Or several others. Or distract you with several others while also turning the first approach back on.

It goes on and on. We could conduct this same thought experiment with almost any sort of incident. All incidents are unique, regardless of how homogeneous they might appear at a high level.

And even if you want to believe that the incident vectors you face are similar enough that you can throw them all into the same buckets in order to perform some sort of MTTX analysis, you need to consider that the humans responding will be in different states. This could be anything from a different engineer being the primary on-call to the same human being busy with a different thing when they get paged. There is a very real difference between your most experienced veteran engineer responding to

an incident, and one that has just joined the rotation. Additionally, there is a very real difference in response time if your on-call engineer is sitting at their desk during working hours, or out on their lunch break, or being paged at 03:00 on a Sunday. Even if all of the other details of the attack are similar, the human factors will always be unique.

These might seem like nit-picky details, but they're important to consider when attempting to establish equivalence between incidents. Equivalence requires all factors to be the same, but they never truly are. This almost immediately throws the validity of an MTTX approach out the window. If you're not convinced yet, though, there are also purely mathematical reasons for why these aren't good approaches to measuring reliability.

Means aren't always meaningful

The other problem with using a means- or average-based approach to analyzing your unreliability incidents is that they can cover up how your users actually feel about your service. This is most provable by using a simple counterexample.

Let's say your organization uses mean time to resolution (MTTR) or something similar to measure how well teams are able to respond to incidents of unreliability. Let's also imagine that you have an objective for the quarter to improve your MTTR by lowering it. The idea here is that lowering your MTTR will result in your users having a better experience interacting with your service. This isn't the most absurd thought: if you are able to bring your system back into the correct operational parameters when it experiences a failure faster in this quarter than you did in the last one, users should be happier about your reliability, right?

This isn't how the math actually works, however. We can use an extreme counterexample to prove this, although more minor versions will result in the same outcome.

Imagine that the first quarter of the year has just ended. During this time your service had 20 total incidents. Even though they each directly impacted the reliability of your service from your users' point of view, they were all relatively minor. You were able to detect them quickly, and in most instances you simply had to roll back to a previous release to get the system running again. In this example your organization has been able to provision a strong CI/CD pipeline with an easy rollback system. The average time it took for you to resolve these sorts of problems (your mean time to resolution, or recovery, or remediation) was about 20 minutes.

During this quarter, your team has learned a lot about the failure modes and reliability of your service, and you'd like to improve on your MTTR for the next quarter. So, you set an OKR that aims for the team to improve its MTTR to 10 minutes. This all seems very reasonable from a high level.

But then math happens in ways you might not expect.

The following quarter your service is much more reliable. You learned a lot from the 20 incidents you had in the first quarter, and you've been able to fix a lot of the problems your system experienced previously. Things run smoothly for almost the full quarter—until you have a disastrous outage. It isn't even directly your service's fault: the underlying database you depend upon was down for three hours when a human accidentally dropped the primary tables and a restore had to take place. (This isn't just a dramatic device; this sort of thing happens all the time!)

When it comes time to perform your end-of-quarter math, you look at your MTTR. It now reads 3 hours, while in the previous quarter it was 20 minutes. This very much goes against your goal of improving your MTTR from 20 to 10 minutes! And more importantly, it obfuscates how much your users actually suffered.

Twenty incidents at a length of 20 minutes each equals 400 minutes, or 6 hours and 40 minutes of time that you were unreliable for your users during the first quarter. The single 3-hour incident you endured in the second quarter is well under half of that amount of time, yet a means-based approach to measuring things would tell you that you had been less reliable in that quarter than the one before.

This is, of course, an extreme example, but it acts as a good starting point for how to think about these sorts of quandaries. You don't need to be comparing 20 short incidents versus 1 long one to understand that perhaps your more frequent and short-lived reliability issues are actually more of a problem for your users than isolated longer periods of downtime are.

It turns out that neither counting incidents nor measuring mean timings around them really answers the questions your users need you to answer. The questions you need to be asking are "During which of these two quarters did our users experience more unreliability?" and "How can we quantify this better?"

The answer, as you should suspect, is SLOs.

SLOs for Basic Reporting

In contrast to counting a number of incidents or MTTX approaches to measuring the performance of your systems over time, SLOs can provide you with a much more holistic sense of your reliability. Not only can they ensure you don't fall into the math-based traps described in the previous section, but they can also give you more concrete goals to work toward.

Let's start with a reexamination of the situation we just looked at, where two quarters were being reported on: one with a single large period of unreliability and the other with many smaller periods.

An SLO-based approach to reliability using error budgets would have correctly told you that your users experienced 6 hours and 40 minutes of unreliability in the quarter

with 20 incidents of 20 minutes each—far worse than the quarter with the single 3-hour incident. Purely from a math standpoint, error budgets do a better job of exposing what your users experienced than any kind of MTTX calculation ever could.

Additionally, SLO-based approaches can help you address the concerns we identified about using severity levels and counting incidents. Think back to our discussion about how severity levels are inherently flawed because it's impossible to fit all incidents into strict buckets. Remember the example of the incident that was flapping between the definitions outlined for S3 and S2, and even briefly entered S1 territory? Using severity levels and thinking about incidents in terms of the number of times they happen, we run into a problem of definition. Ambiguity and subjectivity abound. If you have well-defined and meaningful SLIs, using SLOs can help you avoid this.

Let's tie all the examples laid out so far into one cohesive story.

A worked reporting example

Imagine that you're responsible for some kind of web-facing service. For our purposes here it doesn't have to be something incredibly out there or complicated; let's say your primary function is to serve journalistic and opinion-based content about the New York City restaurant scene. You cover things like new openings and changes in dining trends, and have a forum where people can express their opinions—and you need to be able to report to people about how your services function in terms of those user journeys.

One day, one of your restaurant critics writes a post that is particularly scathing about a restaurant that has been considered sacred by NYC foodies for years. Some people agree with the review; perhaps this once-esteemed restaurant really has been going downhill of late. But it also has legions of fans, and one of them is so upset with this new review that they decide they want to take you down. They launch a DDoS attack.

At first, their attack is primitive and doesn't accomplish much. Your traffic is routed through a CDN that offers automated DDoS protection. However, it does take a few minutes for this protection to kick in, and during this time your site serves errors to some people requesting content due to the load. Of the approximately 9,000 requests you normally serve per hour, you end up serving about 630 errors because of this attack, consolidated within a 6-minute period where 50% of your responses were errors.

If you're counting incidents, would that have even been one? Does that six-minute period of partial unavailability count as an incident? Perhaps it does. Perhaps it goes on your incident register as an S1 incident with an MTTR of six minutes.

The upset restaurant fan gave up when they realized their attempt to bring your site down had failed, but they return an hour later with a more sophisticated attack.

They've scoured the web and found a botnet they can lease for some amount of bitcoin, so they try again with a more complex approach. This time they're hitting you from hundreds of thousands of IPs, and your CDN cannot comprehend this as an attack at all. Suddenly it's all hands on deck on your end, as you have to figure out how to block this traffic. This time around, your entire site goes down for a full 30 minutes until you're able to resolve things.

Are these the same incident? Are they separate ones? How do you quantify the impact of 50% of responses being errors versus 100% of them timing out? Does an error count differently than timing out? What severity level do you apply if this is all treated as one incident?

These are all important questions in terms of learning from the incident and improving your resilience to these sorts of attacks in the future—but they tell you almost nothing about how your users perceived the reliability of your service during the incident. This is especially true if you're just counting incidents or applying MTTX math.

When did the incident start? Was it when you first got an alert? When the attack was first triggered? When the first user received an error response? When the first person complained on Twitter?

How can you resolve all of this?

If you have a meaningful SLI measuring your users' interactions with your website, you don't have to worry about things like what severity level to lump things into or if this situation counts as two incidents or one. You have an error budget, and that budget tells you how bad things have actually been.

During the first part of the attack, where 50% of responses resulted in errors over 6 minutes, you will have burned 3 minutes of your error budget. During the second wave of the attack, you will have burned 30. This equals 33 minutes of total unreliability. Since you've picked a reasonable and well-studied target of 99.9% reliability every 30 days, this means you have 13 minutes of error budget remaining before the bulk of your users are likely to decide your site is too unreliable and go elsewhere.

We could extend this example infinitely in terms of how hard you're hit at any one point in time. Perhaps you had 2 minutes of 90% errors and then 7 minutes of 10% errors followed by 4 minutes of 40% errors—an SLO-based approach would allow you to actually capture how users experienced interacting with your service during this time, whereas a counting incidents/severity level approach or one relying on MTTX would have left you with extremely incomplete data.

However, SLOs provide you with much more than just a better alternative to the basic reliability reporting you might be used to.

Advanced Reporting

As we've discussed relentlessly throughout this book, SLO-based approaches to reliability are at their base almost entirely about providing you with new telemetry that's better than what you may have had before. It doesn't matter if you're dealing with metrics that only report in once per minute or using an advanced observability vendor that allows you to analyze every single real event that occurs within your system: SLOs allow for you to think about your service in better ways than raw data ever will. Because of this, you can also report on the state of your services in new and better ways.

The first and hopefully most obvious difference is that SLO-based approaches are entirely focused on your users or customers. If you've taken the time to ensure that your SLI is as close to your users' experience as possible, you no longer have to worry about things like whether the errors in your logs or metrics actually correlate to your users' experience. You might recall that Chapter 3 made the case that developing meaningful SLIs is the most important part of this entire story. At the end of this book, that still holds true. Measure the things your users actually care about, and you're already in a much better position than you were before.

However, beyond SLIs, we've also discovered together how SLO targets and error budgets can help you better understand your users' experiences. Let's talk now about how to use those to better report on the reliability of your service.

SLO Status

One of the most confusing parts about moving to an SLO-based approach to reliability is talking about your current status. You can say things like "Our service is currently meeting SLO" or "Our service is currently not meeting SLO," but what does that actually mean? If your service is expected and allowed to fail a certain amount of the time, what does it mean to say that it's *currently* doing so?

The short answer is that it's an indicator to yourself and others that something is currently wrong. This can be incredibly powerful and useful data, even if you aren't exceeding your error budget.

A major focus of this book has been to communicate that you're allowed to fail, so long as you're still meeting your users' availability expectations. Therefore, it can be tempting to not care about anything until you've exceeded your error budget, which is what should be representing how much failure your users can absorb. However, you should also be paying attention to your current and recent status in terms of your target. The goal is not to only react when your users are extremely unhappy with you—it's to have better data to discuss where work regarding your service should be moving next.

One of the easiest ways to do this is to report on your *reliability burndown*. Burndown is a number that is very closely related to your error budget status, but focuses more closely upon the percentage of recent events that have reported a good versus bad status. While an error budget is necessarily tied to a specific time window, your reliability burndown can be used to look at how you've performed over any window of time. The math is no different than what we explored in Chapter 5, just with a flexible number in terms of the total time units you care about at any point in time. Remember our discussion about building dashboards for discoverability and understandability in Chapter 12? Figure 17-1 shows an example dashboard of a service that has burned through its budget and is continuing to do so.

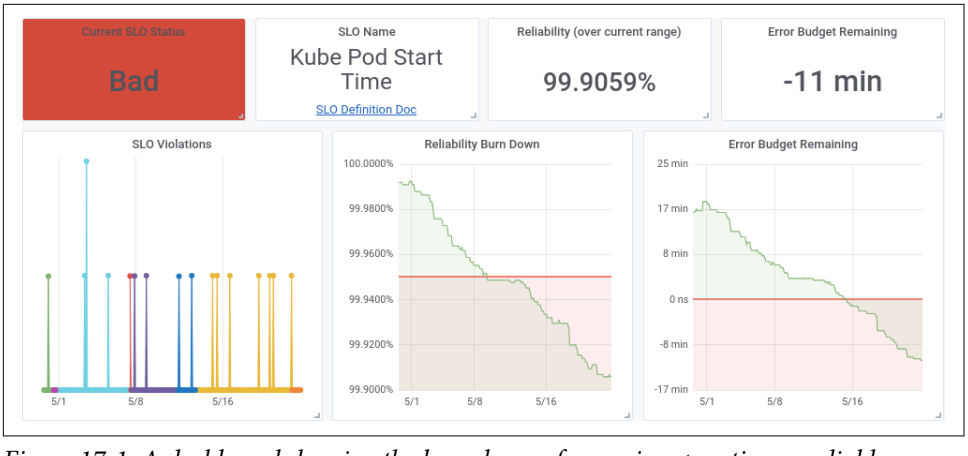

Figure 17-1. A dashboard showing the burndown of a service operating unreliably

Reliability burndown is also closely related to alerting on burn rates, as discussed in Chapter 8. It's not uncommon for people to declare that dashboards are obsolete if you have the best, near-perfect observability tooling—but in practical terms very few of us do. Therefore, dashboards that include things like graphs showing reliability burndown rate can be instrumental in allowing humans to spot times where they should take action before computers can. Humans are exceptionally good at spotting patterns in visual data.

Build dashboards like the ones discussed in Chapter 12, but use them as a reporting feature and not necessarily as a thing your engineers should be actively monitoring. Burndown graphs are a great thing to review during daily or weekly syncs, not something someone should have to have their eye on at all times. They should be flexible in a way error budgets aren't—that is to say, their time windows should always be malleable and not defined and fixed. Always trust your humans to spot important changes before a computer can. An SLO burndown graph is a great way to enable this.

Another important aspect of dashboards is that they can provide a simple indicator that reports on whether your service is currently meeting its targets or not. This can be a purely Boolean "yes or no" kind of indicator. It can help inform users of the status of your service, and is especially useful to other internal teams that might depend on your service. If their own service is currently experiencing problems and they suspect it is due to a dependency, having a Boolean yes or no, good or bad status that is discoverable on a dashboard or something similar can help them determine what action to take, if any.

Error Budget Status

Error budgets are meant to be the ultimate informer of how a service has performed over time. It should be no surprise that accurately reporting on them is one of the most important aspects of using SLOs to report on your services. We've already discussed at length how you can use error budgets to drive discussions, but let's now talk about how they can also drive communications and reporting.

Error budget status is a bit more complex than SLO status is. Error budgets have to have a defined window of time, making their reporting more absolute, regardless of whether you've chosen a rolling or fixed window. Whereas you can use an SLO burndown chart to figure out how you've performed over the last hour, six hours, or day, error budgets have a defined window that cannot change from a reporting perspective.[1]

Error budgets can be tied to individual events or time-based events, and there is a reason why I prefer the latter. You already have SLO burndown charts to help you calculate and visualize what things look like in totality; I love time-based error budgets because they communicate to other humans in a way that humans immediately understand. A sentence like "We have 17 minutes of error budget remaining" is immediately understandable and actionable by another human.

An argument could be made that it's actually more accurate to say something like "We project to be able to have 17 more minutes of unreliability before we start losing users" instead of talking about your remaining budget. This is a bit of a semantic argument, but it's worth calling out here. Choices about the language you use are important and worthy of thought! Use the terms that best help you have the discussions you need to have.

1 This is not to say that error budgets cannot change over time, and they absolutely should when needed, but they cannot be static values in terms of reporting.

In addition to providing point-in-time information, the error budget remaining is a great metric to plot on a time series graph. It's one thing for people to see that you're "negative 2h and 35m" and an entirely different thing for people to see a time-based chart showing how you got there.

Reporting on error budget lets you do many things that you can't otherwise. The first is that you can report to leadership how you've actually performed against your users' expectations. As we discussed earlier, counting incidents, using severity levels, or doing any kind of MTTX calculation doesn't really cover the user experience. SLOs, however, do cover those bases. By having meaningful SLIs that fuel proper SLO targets that inform error budgets, you'll be well equipped to let leadership know how component services—or your entire product—have looked over time. This can be a watershed moment for your organization or company.

Summary

An SLO-based approach to reliability gives you many benefits. One of the most important of these is how you can report on the status of your services to others, whether that be via better dashboards, better reports, or just better numbers.

SLOs are all about providing you and your business with data that is more suited to having discussions about the future of the focus of your work. You can use them for many things. You can use them to have better discussions within your own team, and you can use them to have better conversations with other teams. You can use them to set SLAs and have better conversations with your paying customers.

SLOs are about learning and making the best decisions you can. You can do many things with them, but all of them revolve around the idea that you can have better discussions in order to ensure the humans involved—be they external customers or your internal coworkers—are happier. That's the entire point of all of this. Measure things from your users' perspective in a way that can also make your coworkers happier. Being able to report properly on your status is perhaps the single most important part of being able to have these discussions.

Remember the lessons Molly taught us in the Preface. You can't be perfect, no one needs you to be perfect anyway, it's too expensive to try to be perfect, and everyone is happier at the end of the day if you embrace this. You have all the tools you need; you're going to be amazing.