Avoiding Customer Pain

Google Cloud

In our next module, we discuss several Site Reliability Engineering (SRE) concepts and how we can use them to help avoid customer pain. In this context, a customer is any consumer of a cloud-based system.

Agenda

Google Cloud

Specifically, you will learn to...

- Construct a monitoring base from the four golden signals: latency, traffic, errors, and saturation.
- Define critical system measures with Service Level Indicators (SLIs).
- Use Service Level Objectives (SLOs) and Service Level Agreements (SLAs) to measure, and avoid, customer pain, and
- Achieve developer and operation harmony with SLO-based error budgets.
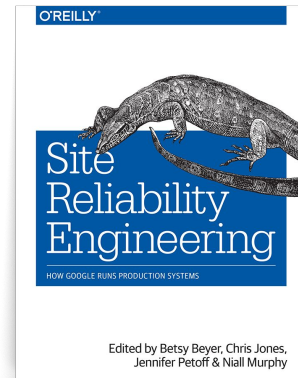
Agenda

Google Cloud

Why should we monitor?

Perhaps this question has an obvious answer, but let's ask it anyway.

Monitoring

Collecting, processing, aggregating, and displaying real-time quantitative data about a system, such as query counts and types, error counts and types, processing times, and server lifetimes

O'REILLY®

Site Reliability Engineering

HOW GOOGLE RUNS PRODUCTION SYSTEMS

Edited by Betsy Beyer, Chris Jones, Jennifer Petoff & Niall Murphy
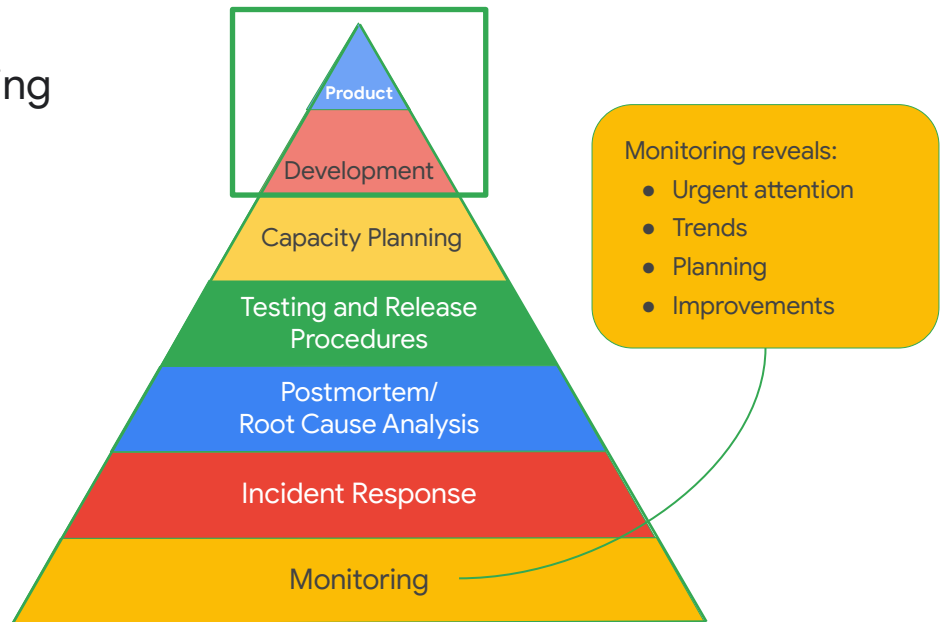
Google Cloud

Let's start by defining our terms. In Google's Site Reliability Engineering book, Monitoring is defined as "Collecting, processing, aggregating, and displaying real-time quantitative data about a system, such as query counts and types, error counts and types, processing times, and server lifetimes."

There are many answers to the "Why monitor?" question. Continued system operations, trend analysis over time, building dashboards, alerting personnel when systems violate predefined SLOs, comparing systems and systems changed, and providing data for improved incident response, are just a few.

For reference, Google has published several books on SRE. They are available to read for free at: landing.google.com/sre/books/
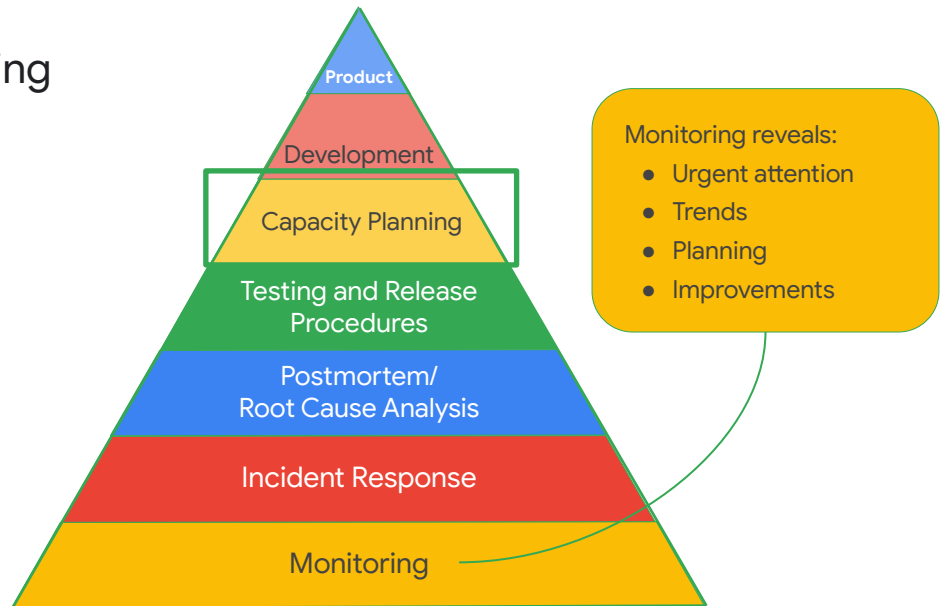
Monitoring

Pyramid from bottom to top:
- Monitoring
- Incident Response
- Postmortem/Root Cause Analysis
- Testing and Release Procedures
- Capacity Planning
- Development
- Product

Monitoring reveals:
- Urgent attention
- Trends
- Planning
- Improvements

Google Cloud

An application client sees the product, and as a result, developers and business stakeholders both tend to think that the most crucial way to make the client happy is by developing a great product.
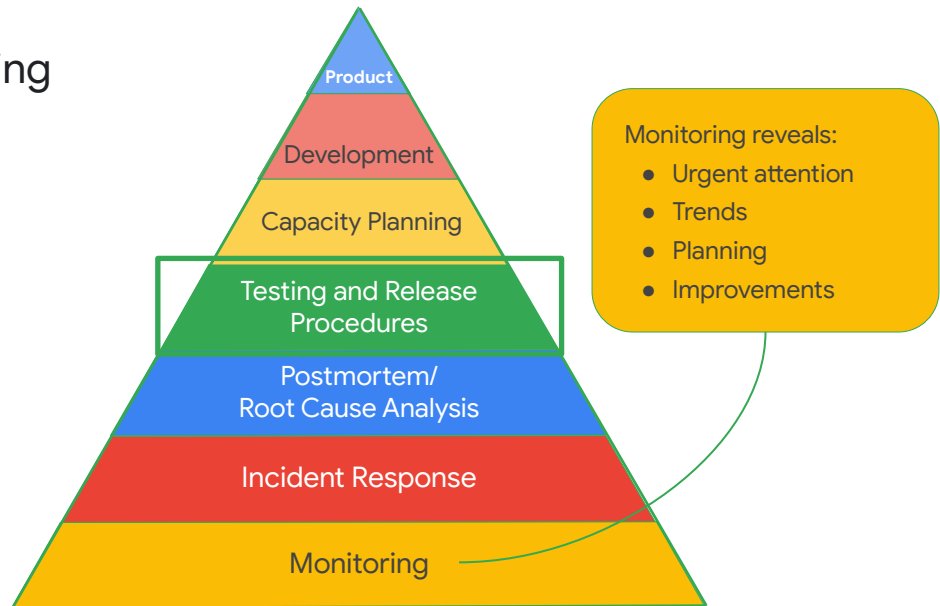
But developing the product is just the tip of the pyramid.

Great products need to be deployed into environments with enough capacity to handle client load, so someone needs the correct information to make those capacity planning decisions.

Great products need thorough testing, preferably at least partially automated, and also preferably as part of a refined Continuous Integration/Continuous Development (CI/CD) release pipeline.
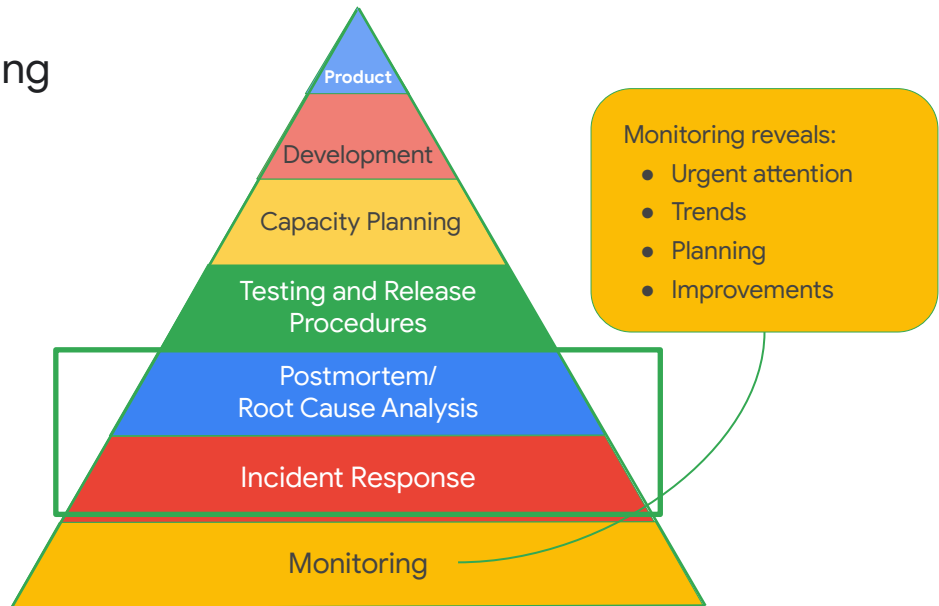
Past that, we also have to keep clients happy by building trust.

Trust is gained by creating good products with good uptime and SLO compliance, but it also requires transparency when things go wrong.

Postmortems and root cause analyses are the DevOps team's way of letting the client know what happened and why it is unlikely to happen again

These are integral parts of robust incident response.
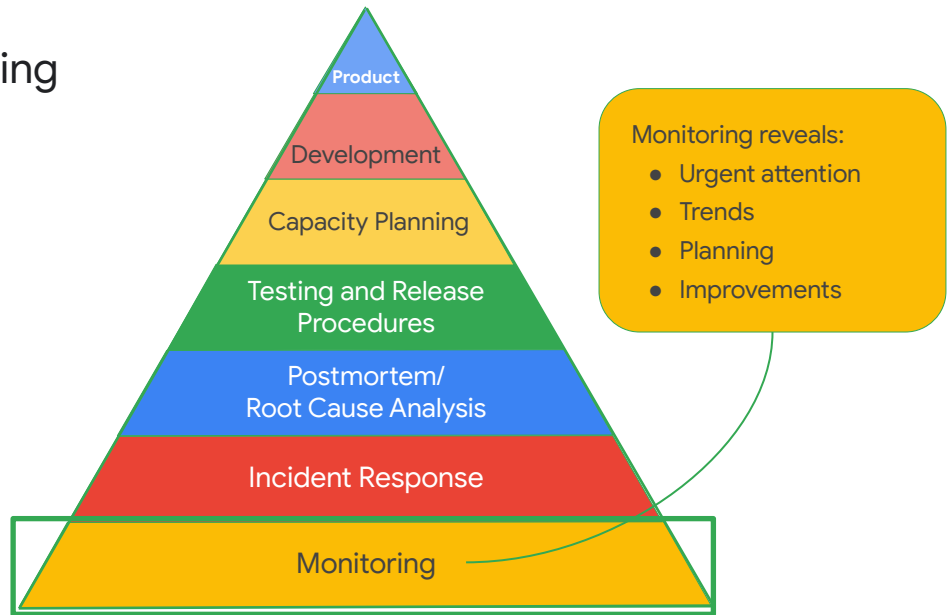
All of this sits squarely on a solid base of monitoring systems and data.

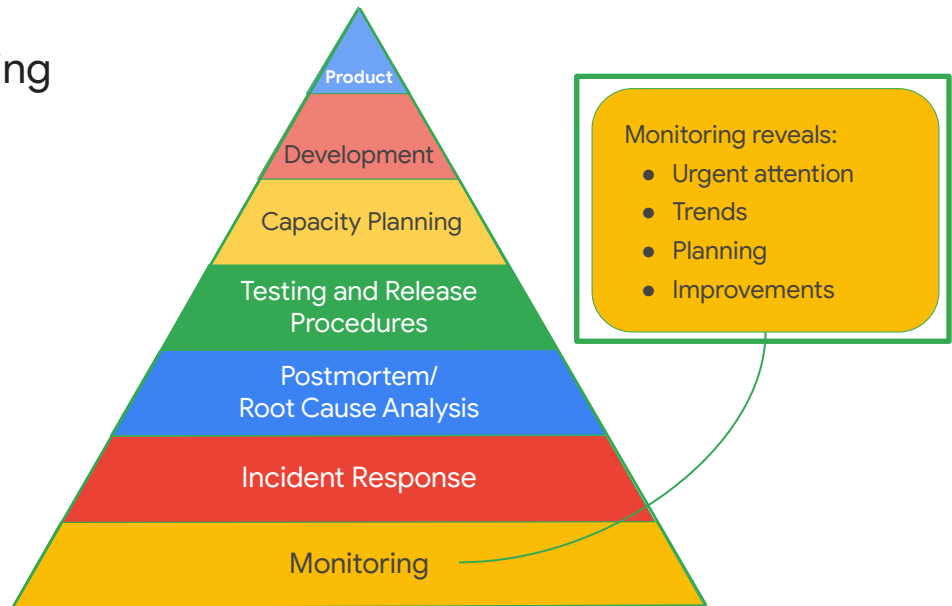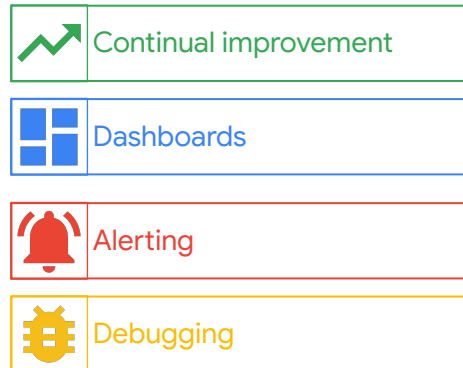# Monitoring

Monitoring reveals things that need urgent attention and trends in application usage patterns, which can yield better capacity planning, and generally help improve the client's experience and lessen their pain.

Why monitor?

Continual improvement

Dashboards

Alerting

Debugging

Google Cloud

Why do we monitor? Because we want that good, reliable, and trusted product we saw on the top of our pyramid.

We want our product to improve continually, and we need the monitoring data to make sure that it does.

We want dashboards to provide business intelligence, so our DevOps personal have the data they need to do their jobs.

We want automated alerts because the huge wall of monitors with someone staring at it (as is often shown in the movies) is a fallacy.

Humans tend to only look when there's something important for them to look at, so it is better to construct automated systems to prioritize and handle as many alerts as possible.

Finally, we want monitoring systems that help provide data crucial to debugging the functional and performance issues of applications.

Continual improvement

Forecasting and trend analysis

DB capacity and growth rate?

Able to handle the holiday rush?

Handling new international client load?

Continual improvement starts with robust forecasting and trend analysis. It answers questions like:
- What's our current database capacity and growth rate?
- Can current systems and infrastructure handle the holiday rush?
- What should we change if we want to expand and start handling new international client load?

Continual improvement

Testing changes and improvements

Memorystore improves site performance?

Latest software update affects performance?

How would change X impact capacity?

When paths for improvement are discovered, monitoring helps with testing changes and proposing improvements such as:

- Would adding Memcache improve site performance?
- How about the latest software update; is if affecting performance?
- How would change X impact capacity?

Continual improvement related monitoring also provides data crucial for business, application, and security analysts, helping answer questions like:

- How can we spot evolving customer needs?
- How and where could software be improved?
- Is someone trying to hack the system?

Dashboards provide

Sprint 4/5 Dashboard   🔍 ⚙ ▣ ▽ ⟳ Off   TIME   1h   6h   1d   1w   1m   6w   custom   ☆   Add Chart
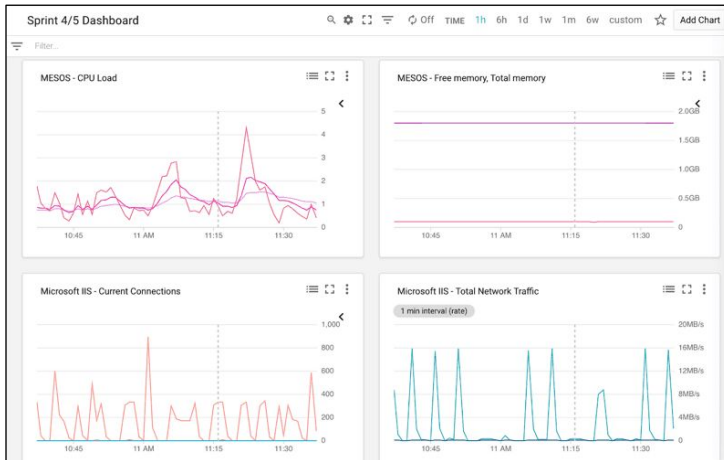
- Core metrics in a summary view.
- Views targeted to user.
- Metrics in usable amounts.
- Options for non-human monitoring.

Google Cloud

Dashboards provide core metric visibility into your cloud resources and services with no configuration.

Define user- or job-targeted custom dashboards and take advantage of Google's powerful data visualization tools.

Dashboards can be combined with automated alerts to help avoid the need to have personnel stare at a huge wall of displays.

Even better, many alert signals could be handled by non-human, automated systems.

Alerting

- A notification to a human that something needs attention
  - Email
    - FYI, something happened...
    - Be careful, alert spam, need to avoid.
  - Ticket
    - A human should handle this "soon!"
  - Page
    - A human must do something **now**!

Google Cloud

Configure alerting policies to notify you when events occur, or a particular system or custom metrics violate rules that you define.
Use multiple conditions to define complex alerting rules, and receive notifications through one of several alert channels, including:

- **Email.** Good for informational messages, but be careful that alerts don't become just another type of spam.
- **Tickets** from a ticketing system. Good when an alert needs to be handled by a human "soon," but not now.
- One of the several **pager**-type options. Good when human interaction needs to happen sooner than later.

Debugging (ad hoc retrospective analysis)

Timeline

1

Triggers

▶ Outage
▶ Data loss
▶ Monitoring failure
▶ Manual interventions

Google Cloud
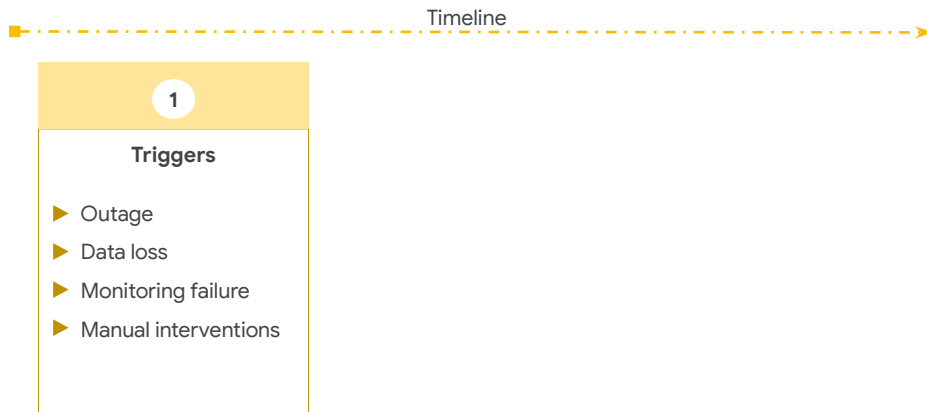
In any system, bad things can happen, and when they do, they need to be debugged and fixed.

Typically, there's some triggering event: a system outage, data loss, a monitoring failure, or some form of manual intervention.

Debugging (ad hoc retrospective analysis)

Timeline

1

Triggers

▶ Outage
▶ Data loss
▶ Monitoring failure
▶ Manual interventions

2

Response

▶ Signal analysis
▶ Impact evaluation
▶ Escalation
▶ Initial response
▶ Customer response

Google Cloud
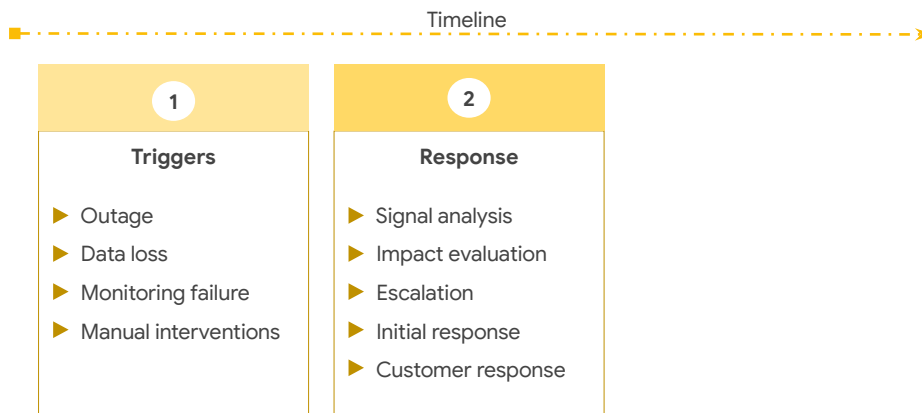
The trigger leads to a response by both automated systems and DevOps personnel.

Many times the response starts by examining signal data coming in through monitoring. The impact of the issue is evaluated and escalated when needed, and an initial response is formulated. Throughout, good SREs will strive to keep the customer informed and respond when appropriate.

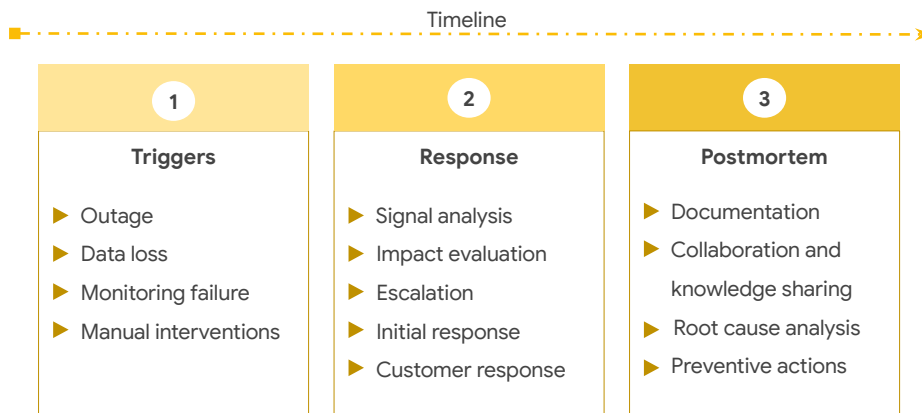# Debugging (ad hoc retrospective analysis)

Timeline

1	2	3
Triggers	**Response**	**Postmortem**
▶ Outage	▶ Signal analysis	▶ Documentation
▶ Data loss	▶ Impact evaluation	▶ Collaboration and knowledge sharing
▶ Monitoring failure	▶ Escalation	▶ Root cause analysis
▶ Manual interventions	▶ Initial response	▶ Preventive actions
	▶ Customer response	

Google Cloud

After the issue has been fixed or mitigated, a postmortem is created. It should contain documentation for both future DevOps personnel and the client. Included in the documentation, where appropriate, should be a thorough root cause analysis and a list of preventive actions.

Setting expectations

- Monitoring is a skill, not a job.
  - Multiple team members need this skill.
- No single tool can do it all.
- No system is infallable.
  - Build trust through transparency and quick fixes.

- KISS (Keep it Simple, Stupid)
  - Forget "magic" (overly complex) monitoring.
- Focus on one system at a time.
- Monitor from multiple vantage points.
  - Remember the importance of the consumer.

Google Cloud

A word on expectations. Monitoring isn't a job so much as it's a skill that multiple team members need.

- There is no single do-it-all tool; that's why this class covers multiple Google Cloud products.
- There is no such thing as an infallible system, but there can be SLOs that makes sense, and you can use these to build trust through transparency and quick fixes.
- Never forget the KISS—Keep it Simple, Stupid—principle when you're planning and implementing monitoring.
- How do you eat an elephant? One bite at a time. The same is true with monitoring. Focus on one system, one service, at a time, and "divide and conquer" your way to success.

- It's easy for monitoring to become too focused..
- It's better to monitor from multiple vantage points.
- External monitoring is helpful because it represents the client's view, but the data it can access is limited.
- Testing from inside a service has more access to data and detail, but lacks that client perspective. Use them both!

Misunderstanding "single pane of glass"

Single pane of glass does not mean:

- All GCP projects are in one dashboard.
- All metrics are in a single view.
- One view rules the world.
- There is a wall of monitors to watch.

Google Cloud

In discussions related to monitoring in Google Cloud, the phrase "Single pane of glass" is often used, and just as often, is misunderstood.

A single pane of glass does not mean that all Google Cloud projects can be monitored from one enormous monitoring project.

At most, a monitoring workspace can support 100 sub-projects, so unless your cloud footprint is smaller than that, you will need multiple monitoring workspaces.

A single pane of glass also does not mean that all metrics get displayed in a single view such that there's one view to rule the monitoring world.

And, a single pane of glass certainly doesn't mean that someone gets to watch a wall of monitors all day - that's boring, impractical, and ineffective.

Misunderstanding "single pane of glass"

Single pane of glass does deliver:

- A logical grouping of multiple projects into a single workspace.
- Multiple charts in one dashboard.
- A signal collection across multiple apps, platforms, and services.

Google Cloud

What single pane of glass does deliver is a logical grouping of multiple projects into a single workspace. If 15 teams create 15 services that all work together to form a contiguous whole, then yes, I could monitor those 15 production projects from a single monitoring workspace created in a 16th project.

Single pane of glass can also mean that it's possible to display multiple charts in one dashboard.

And it's possible to collect - and show correlation - between monitoring signals across multiple apps, platforms, and services.

Symptom vs. cause (what vs. why)

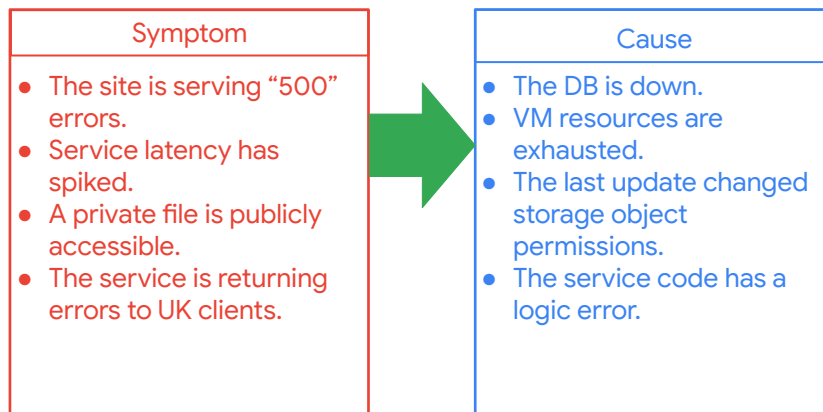| Symptom |
| --- |
| ● The site is serving "500" errors. |
| ● Service latency has spiked. |
| ● A private file is publicly accessible. |
| ● The service is returning errors to UK clients. |

Your monitoring system should address two questions: what exactly is broken, and why?

The "what's broken" indicates the symptom. Examples could be:

- A web site that suddenly starts serving 500 HTTP status errors.
- A service that is showing latency spikes.
- A file that should be private but is publically accessible.
- A service that returns errors, but only to UK-based clients.

Symptom vs. cause (what vs. why)

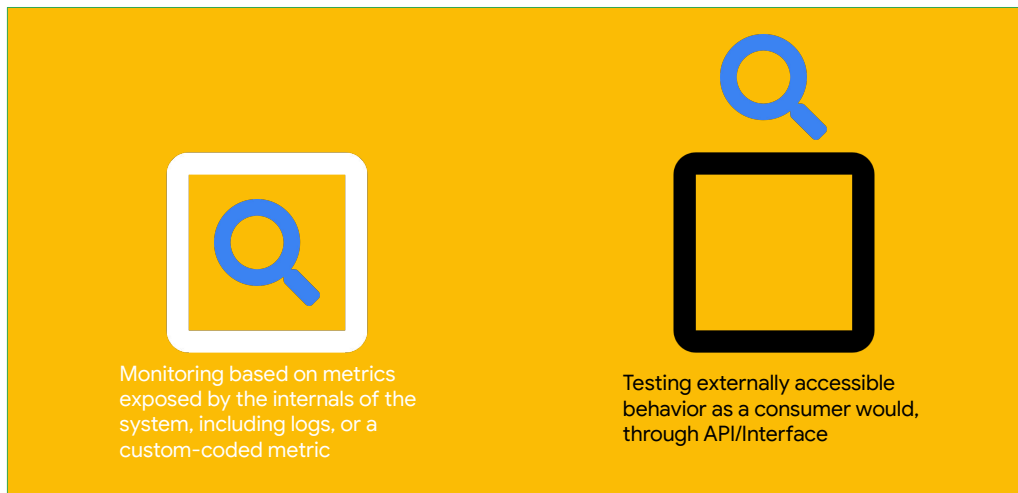| Symptom	Cause
● The site is serving "500" errors. ● Service latency has spiked. ● A private file is publicly accessible. ● The service is returning errors to UK clients.	● The DB is down. ● VM resources are exhausted. ● The last update changed storage object permissions. ● The service code has a logic error.

The "why" from our two questions that monitoring systems should address indicates a possible cause. We hope our monitoring path ultimately leads us to the cause, because that's what needs to be fixed or at least mitigated.

The cause could be:

- that a database is down,
- that our Compute Engine VM resources pool is exhausted,
- that the last code update inadvertently changed a storage object's permission,
- or that a piece of service code has a logic error.

Clear box versus black box

Monitoring based on metrics exposed by the internals of the system, including logs, or a custom-coded metric

Testing externally accessible behavior as a consumer would, through API/Interface

Google Cloud

When you're monitoring, there are two crucial perspectives. Black-box monitoring is symptom-oriented and represents active problems: "The system is returning 500 errors right now." It tests externally accessible behavior as a consumer would, through the API/Interface.

Clear box monitoring is based on metrics exposed and collected by the internals of a system. It might include code-created logs or metrics.

With its added visibility from inside the system being tested, clear box better allows detection of imminent problems, failures masked by retries, and so forth. Best practice monitoring should combine the heavy use of clear box monitoring with modest but critical uses of black-box monitoring.

Note that in a multilayer, microservice system, one service's symptom is another's cause.

Agenda

Why Monitor?

Critical Measures

Four Golden Signals

SLIs, SLOs, SLAs

Choosing a Good SLI

Specifying SLIs

Developing SLOs and SLIs

Google Cloud

Monitoring starts with metrics.

Metrics help measure success

In business, common metrics include:

- Return on investment (ROI)
- Earnings before interest and taxes (EBIT)
- Employee turnover
- Customer churn

In software, common metrics include:

- Pageviews
- User registrations
- Click-throughs
- Checkouts

Google Cloud

Business decision-makers want to measure the value of projects so they can better support the most valuable projects, while not wasting resources on those that are not beneficial. A common way to measure success is to use metrics. Metrics can be categorized as business metrics and technical metrics.
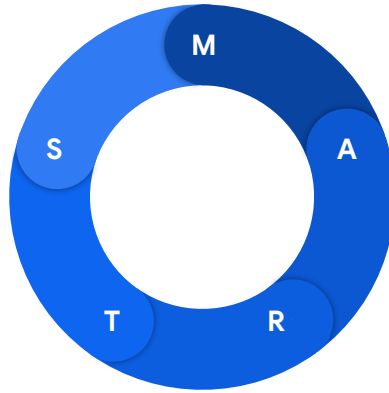
In business, success metrics might include:
- Return on Investment (ROI)
- Earnings before Interest and Taxes (EBIT)
- Employee turnover
- Customer churn

In software, success metrics could be:
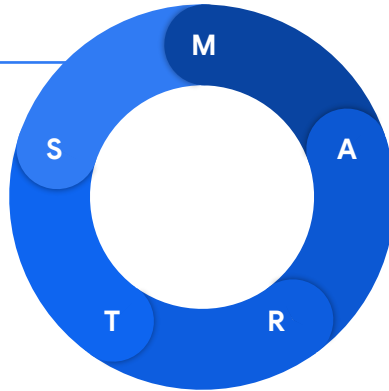- Pageviews
- User registrations
- Click-throughs
- Checkouts

To be effective, metrics must be SMART

But you can't measure everything, so when possible, you want choose metrics that are S.M.A.R.T.

To be effective, metrics must be SMART

Specific
"Fast" is not as specific as
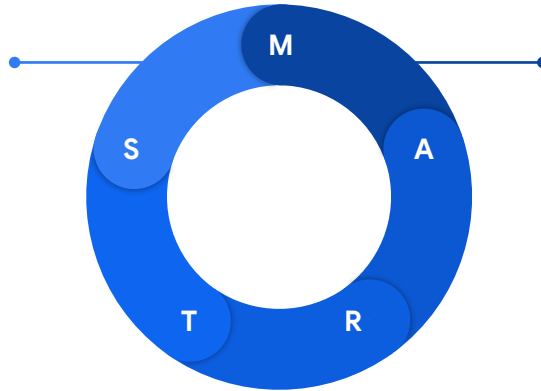"Results in 100ms."

S

M

A

T

R

Google Cloud

Metrics should be **Specific**. "Hey everyone, is the site fast enough for you?" is not specific; it's subjective.

"The 95th percentile of results are returned in under 100ms." Now, that's specific.

To be effective, metrics must be SMART

Specific
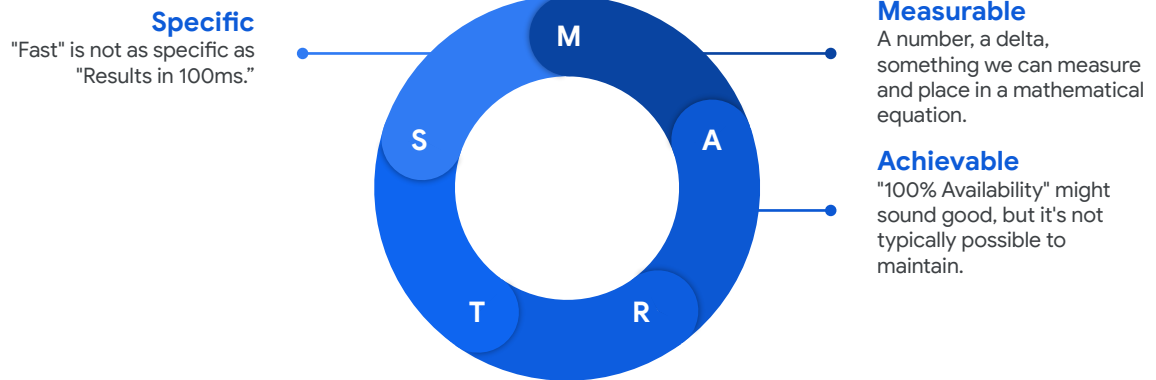"Fast" is not as specific as "Results in 100ms."

M

S

A

T

R

Measurable
A number, a delta, something we can measure and place in a mathematical equation.

Google Cloud

Metrics should be **Measurable**. A lot of monitoring is numbers, grouped over time, with math applied.

A metric needs to be a number or a delta, something we can measure and place in a mathematical equation.
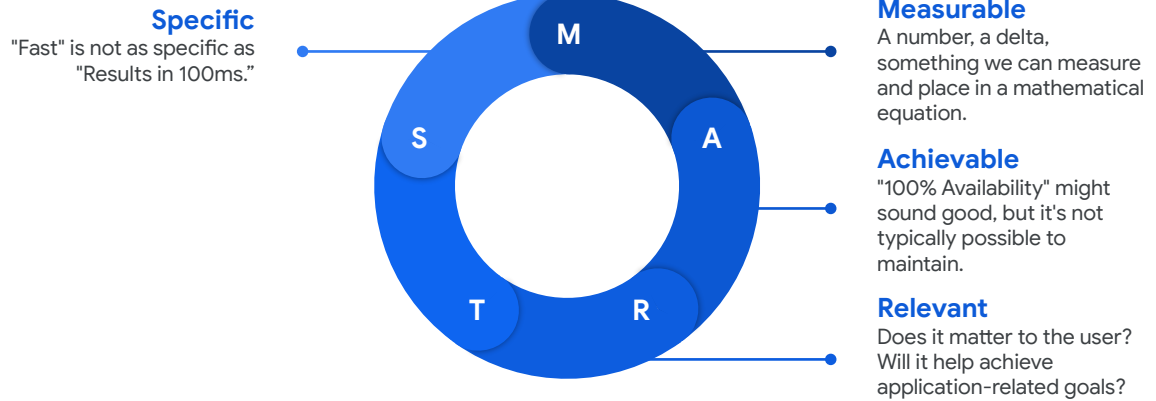
To be effective, metrics must be SMART

Specific
"Fast" is not as specific as "Results in 100ms."

Measurable
A number, a delta, something we can measure and place in a mathematical equation.

Achievable
"100% Availability" might sound good, but it's not typically possible to maintain.

Google Cloud

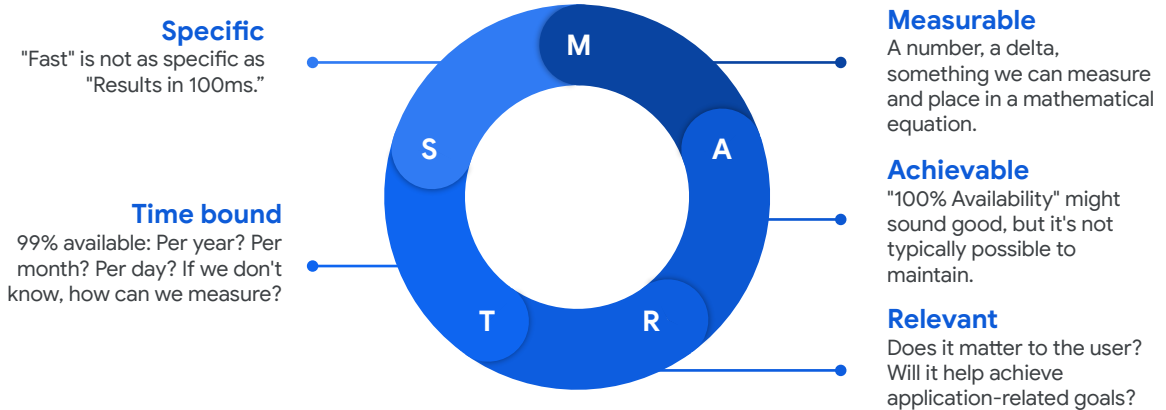Metrics goals should be **Achievable**.

"100% Availability" might sound good, but it's not possible to obtain, let alone maintain, over an extended window of time.

To be effective, metrics must be SMART

Specific
"Fast" is not as specific as "Results in 100ms."

Measurable
A number, a delta, something we can measure and place in a mathematical equation.

Achievable
"100% Availability" might sound good, but it's not typically possible to maintain.

Relevant
Does it matter to the user? Will it help achieve application-related goals?

Google Cloud

Metrics should be **Relevant**. Does it matter to the user? Will it help achieve application-related goals? If not, then it's a poor metric.

To be effective, metrics must be SMART

Specific
"Fast" is not as specific as "Results in 100ms."

Time bound
99% available: Per year? Per month? Per day? If we don't know, how can we measure?

Measurable
A number, a delta, something we can measure and place in a mathematical equation.

Achievable
"100% Availability" might sound good, but it's not typically possible to maintain.

Relevant
Does it matter to the user? Will it help achieve application-related goals?

Google Cloud

And, finally, metrics should be **Time-bound**.

You want a service to be 99% available? That fine. Is that per year? Per month? Per day?

Does the calculation look at specific windows of set time, from Sunday to Sunday for example, or is it a rolling period of the last seven days?

If we don't know, how can we measure it accurately?

The four golden signals

| Latency | Traffic | Saturation | Errors |

Google Cloud
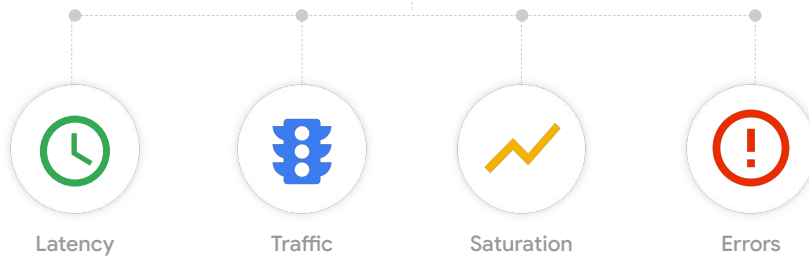
A good place to start metric selection is with one or more of the four golden signals: latency, traffic, saturation, and errors.

Latency

Important because

Impacts user experience.

Could indicate **emerging issues.**

May be tied to **capacity demands.**

May be used to show **improvements.**

Google Cloud

Latency measures how long it takes a particular part of a system to return a result.

It's important because it directly impacts the user experience, because changes in latency could indicate emerging issues, because its values may be tied to capacity demands, and because it may be used to show or measure system improvements.

Latency

Important because

Impacts user experience.

Could indicate **emerging issues.**

May be tied to **capacity demands.**

May be used to show **improvements.**

Sample metrics

- Page load latency
- # requests waiting for a thread
- Query duration
- Service response time
- Transaction duration
- Time until first response
- Time until complete data return

Google Cloud

Sample latency metrics include:

- Page load latency
- Number of requests waiting for a thread
- Query duration
- Service response time
- Transaction duration
- Time till first response
- Time till complete data return

Traffic

Traffic measures how many requests are reaching your system.

Traffic is important because it is an indicator of current system demand, its historical trends are used for capacity planning, and it is a core measure when calculating infrastructure spend.

Traffic

Important because

Indicates current **system demand.**

Historical trends are used for **capacity planning.**

Core to calculating **infrastructure spend.**

Sample metrics

- **# HTTP requests per second**
- **# requests for static vs. dynamic content**
- **Network I/O**
- **# concurrent sessions**
- **# transactions per second**
- **# retrievals per second**
- **# active requests**
- **# write ops**
- **# read ops**
- **# active connections**

Google Cloud

Sample traffic metrics include:

HTTP requests per second
requests for static vs. dynamic content
Network I/O
concurrent sessions
transactions per second
retrievals per second
active requests
write ops
read ops
and
active connections

📈 Saturation

⭐ **Important because**

Indicates **how full the service is.**

Focuses on **most constrained resources.**

Frequently tied to **degrading performance.**

Saturation measures how close to capacity a system is, with capacity being a subjective measure depending on the underlying service or application.

It's important because it's an indicator of how full the service is, it focuses on the most constrained resources, and it is frequently tied to degrading performance as capacity is reached.

Saturation

Important because

Indicates **how full the service is.**

Focuses on **most constrained resources.**

Frequently tied to **degrading performance.**

Sample metrics

- **% memory utilization**
- **% thread pool utilization**
- **% cache utilization**
- **% disk utilization**
- **% CPU utilization**
- **Disk quota**
- **Memory quota**
- **# available connections**
- **# users on the system**

Google Cloud

Sample capacity metrics include:

% memory utilization
% thread pool utilization
% cache utilization
% disk utilization
% CPU utilization
Disk quota
Memory quota
# of available connections
and
# of users on the system

Errors

Important because

Indicates that **something is failing.**

May indicate **configuration or capacity issues.**

Can indicate **SLO violation.**

Time to **alert**?

Google Cloud

Errors measure system failures or issues.

They are important because they may indicate that something is failing, they may indicate configuration or capacity issues, they can indicate SLO violations, and an error might mean it's time to send out an alert.

Sample error metrics include:

Wrong answers or incorrect content
400/500 HTTP codes
failed requests
exceptions
stack traces
Servers that fail liveness checks
and
dropped connections

Agenda

Why Monitor?

Critical Measures

Four Golden Signals

Google Cloud

Now that we know some general monitoring concepts and since we've discussed the four golden signals, let's take some time to lay a conceptual foundation in SLIs, SLOs, and SLAs.

Service level indicator

A **quantifiable** measure of service **reliability**

Google Cloud

You're going to hear two terms a lot through the rest of this module: SLIs and SLOs. I'm sure many of you will have run into them before, but we're going to briefly discuss the concepts so we can be sure everyone here shares the same understanding.

Service level indicators (SLIs) are carefully chosen monitoring metrics that measure one aspect of a service's reliability. Ideally, SLIs should have a close linear relationship with your users' experience of that reliability, and we recommend expressing them as the ratio of two numbers: the number of good events divided by the count of all valid events.

Service level objective

A **reliability target** for an SLI

Google Cloud

A Service level objective (SLO) combines that SLI with a target reliability. If you express your SLIs as we've recommended, your SLOs will generally be somewhere just short of 100%, for example, 99.9%, or "three nines."

Users? Customers?

Customers are users who **directly pay** for a service.

We talk about users and customers interchangeably during the workshop. We have an expansive definition of who your "users" are—they can be internal or external to your company, humans, other companies, or automated systems.

Customers are the subset of your users who are paying directly for a service, with real money.

Services *need* SLOs

Services need SLOs. It's quite a bold assertion, isn't it?

But this is why we're here … we want you to leave this module convinced that you should set SLOs for the services you are responsible for, with a solid understanding of how to do this in practice.

Don't believe us?

"Since introducing SLOs, the **relationship** between our operations and development teams has **subtly but markedly improved**."

— *Ben McCormack, Evernote;* The Site Reliability Workbook, Chapter 3

"... it is difficult to *do your job well* without clearly defining *well*. SLOs **provide the language** we need to **define *well***."

— *Theo Schlossnagle, Circonus;* Seeking SRE, Chapter 21

Google Cloud

You don't have to believe us about the value of SLOs.

In Chapter 3 of the Site Reliability Workbook, you can read two case studies from large businesses that successfully introduced SLOs throughout their organizations.

The book is now freely available, so you won't even have to pay to do so!

SLOs also feature heavily in David Blank-Edelman's excellent book "Seeking SRE."

We've obviously "cherry-picked" the quotes here, but these case studies demonstrate that SLOs bring long-lasting improvements to your business, and they also provide blueprints and strategies for rolling out SLOs in your own company.

The most important feature of any system is reliability.

Note that reliability and availability, while often related, do not mean the same thing.

Availability measures the ability of an application to run when needed, while **reliability** measures the ability of that application to perform its intended function for a specific time without failure.

Why do we want you to believe that your services need SLOs? It stems from this driving principle: that the most important feature of any system is its reliability.

You can easily lose the trust and respect of your users if your service becomes too unreliable—or is simply *perceived* as being too unreliable. Those countless hours you've spent designing and building the most amazing user experiences are all for nothing if your users can't access them.

Unreliability can have disastrous consequences for your business. Competitors will snap up your current, dissatisfied users and your poor reputation will slow—or even halt—the acquisition of new users.

It's not enough to just declare that your services must be more reliable. If that worked, you wouldn't be here listening to me.

Who's experienced a story like this?

The development team for a service decides to focus on feature iteration speed, adopting practices like Agile development and push-on-green to accelerate planning and release cycles. This increases the burden on service operators, who are left scrambling to catch up with a steady stream of "beta" features making their way into production. Inevitably, something gets pushed before it's quite production-ready, leading to a very visible outage and lots of customers shouting on social media. The operations team reacts by setting up procedures to protect the production environment from all this terrible, outage-inducing change, slowing developers down and making them sad.
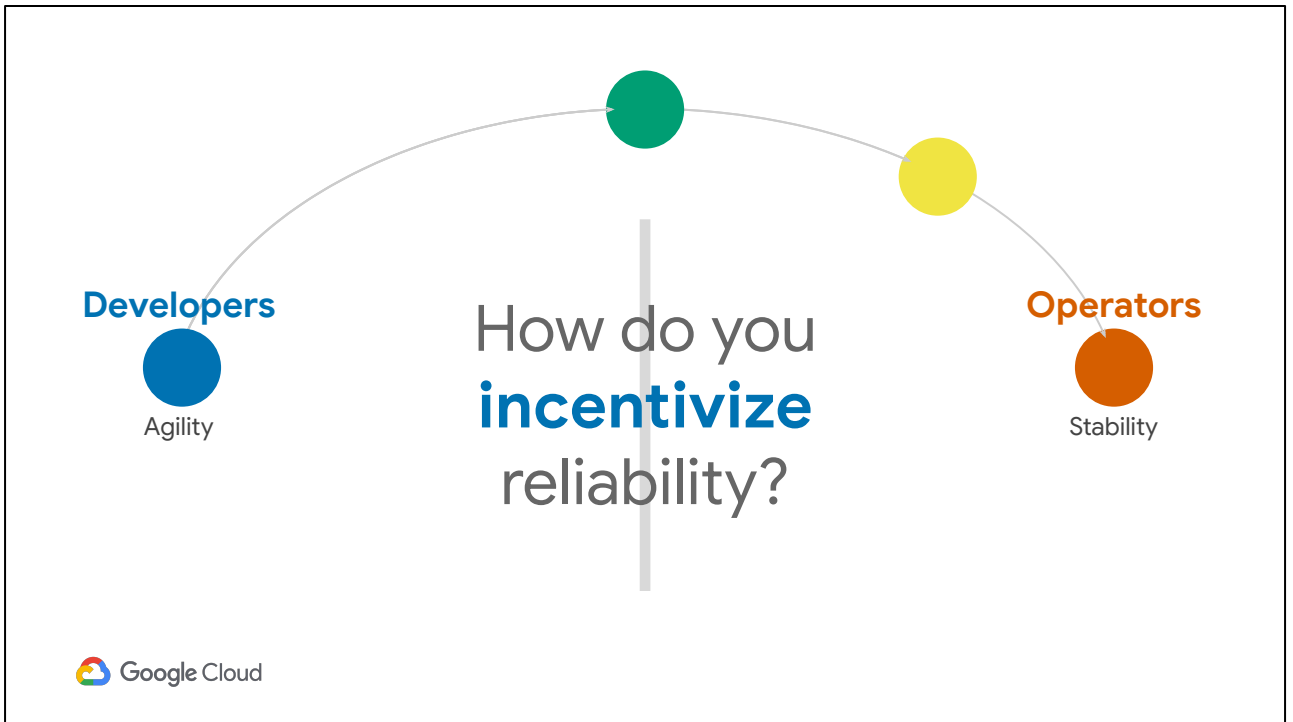
Reliability is a shared goal that many different parts of your organization must work toward *together*. In many companies, it's common to separate out the responsibility for operating a service from that of building new features for it. This allows people in each role to specialize, but it alters their priorities in a way that can often cause conflict.

To avoid this scenario, your organization needs to find a way to incentivize the developers and operators of a service to cooperate, so that you can build new features *without* regularly suffering serious, reputation-damaging outages.

SLO
A **principled** way
to agree on the
desired reliability
of a service

Google Cloud

But you can't spend *all* of your engineering time on reliability, because reliability is not a standalone product. So the question you end up asking is "What is reliable *enough*?", shortly followed by "How can I objectively measure the reliability of my service?"

That's why we think your services need SLOs. We think of SLOs as a principled way to agree on the desired reliability of a service.

SLOs provide different value to different parts of an organization. We've claimed that reliability is the most important feature, but for many product-oriented folks, this brings up a difficult question: When should engineering for increased reliability take priority over that shiny new feature you've been designing for months? If you have an agreed-upon and widely communicated target for service reliability, you can answer that question with objective data.

So, SLOs provide a common language and shared understanding, anchoring your conversation about service reliability on concrete data. It's tempting to consider SLOs to be a purely operational concern, but for them to function correctly as a signal for prioritization of engineering work, your reliability targets must be set in conjunction with engineering and product teams. Everyone must agree that the target accurately represents the desired experience of your users.

What does "**reliable**" mean?

Think about Netflix, Google Search, Gmail, Twitter...
How do you tell whether they are 'working'?

What do we mean by "reliable" in the context of an internet service? Let's start by talking about what it means for a service to be "working well"—or indeed, just "working".

- What should you be able to do?
- What characteristics are important to you?
- What are your expectations of how the service should respond?
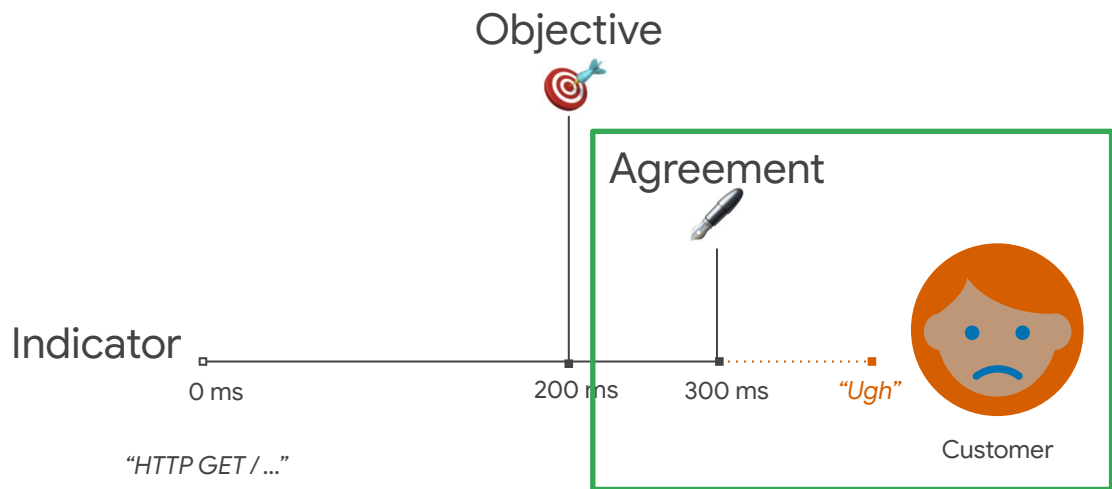
If your service has paying customers, you probably have some way of compensating them with refunds or credits when that service has an outage.

Your criteria for compensation are usually written into a service level agreement, which describes the minimum levels of service that you promise to provide and what happens when you break that promise.

The problem with SLAs is that you're only incentivized to promise the minimum level of service and compensation that will stop your customers from jumping ship to a competitor. Customers often feel the impact of reliability problems before these promises are breached, when reliability falls far short of the levels of service that keep your customers happy and contributes to a perception that your service is unreliable.
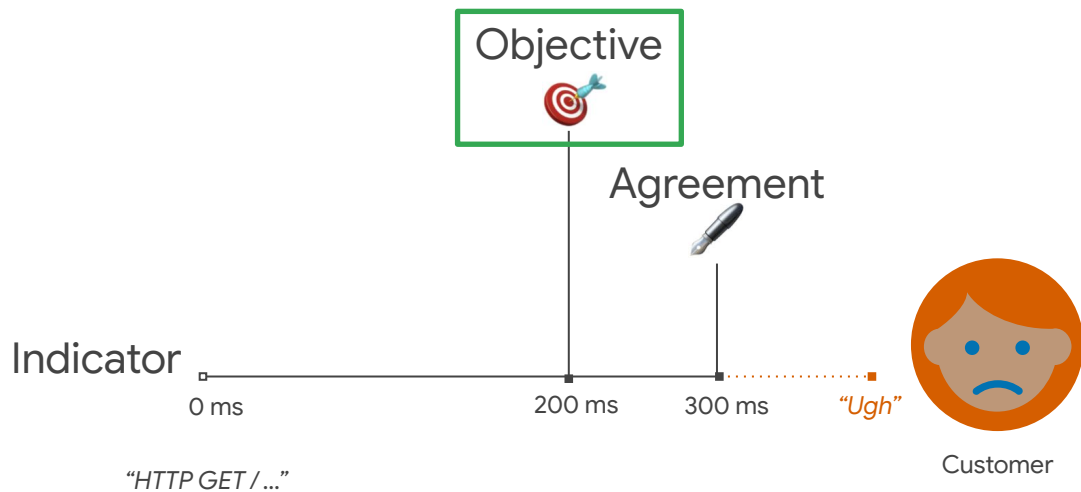
Compensating your customers all the time can get expensive, so what targets do you hold yourself to internally?

When does your monitoring system trigger an operational response?

To give you the breathing room to detect problems and take remedial action before your reputation is damaged, your alerting thresholds are often substantially higher than the minimum levels of service documented in your SLA.
SLOs provide another way of expressing these internal reliability targets.

Objective

Agreement

Indicator

0 ms   200 ms   300 ms   *"Ugh"*

"HTTP GET / ..."

Customer

Google Cloud
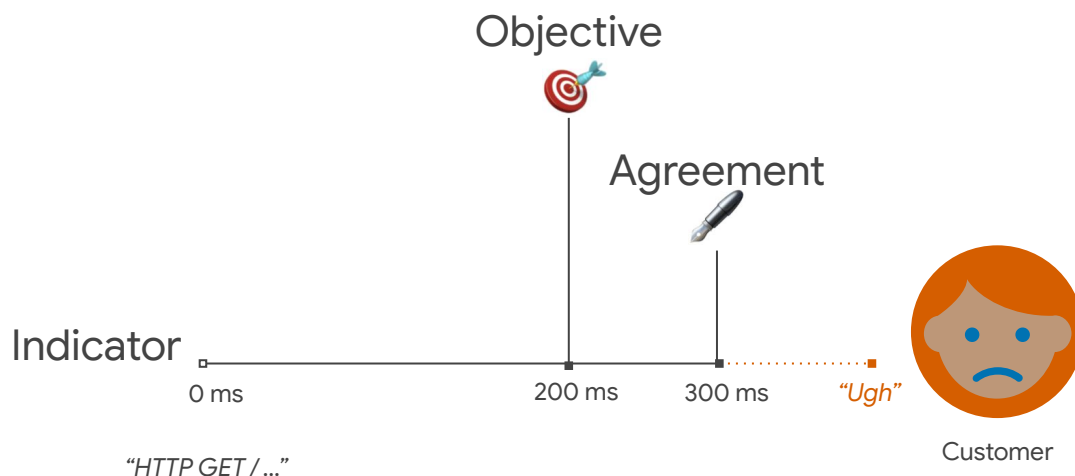
For SLOs to help improve service reliability, all parts of the business must agree that they are an accurate measure of user experience and must also agree to use them as a primary driver for decision making.

Your customers probably don't need to be aware of them, but they should have a measurable impact on the priorities of your organization.

Being out of SLO must have concrete, well-documented consequences, just like there are consequences for breaching SLAs.

For example, slowing down the rate of change and directing more engineering effort towards eliminating risks and improving reliability will get you back into SLO faster.

Operations teams need strong executive support to enforce these consequences and effect change in your development practice.

When do you need to make a service **more reliable**?

How reliable is "reliable enough"?

One way of looking at an SLO is that it helps you answer this question, *When do we need to make a service more reliable?*

But really, the question is, "Where do we draw the line?" Thus far, all we've established is that your SLO targets should be substantially higher than the reliability your SLAs promise. This doesn't help the many "free" services, funded by advertising, that don't have user-facing SLAs at all.

So, how reliable is "reliable enough"? How do you measure how far off that target you are? And what do you do if you decide that you're missing that target?

100% is the **wrong** reliability target for basically **everything**.

— *Benjamin Treynor Sloss*, *VP 24x7, Google;* Site Reliability Engineering, Introduction

Google Cloud

You might think that you want to set your targets as high as possible, to serve your users as best you can, but this is not the correct way to set targets.

The key realization that Ben Treynor Sloss, the founder of SRE at Google, had was that 100% is the wrong reliability target for almost everything[1].

This is even true for supposedly reliable things like pacemakers, which, according to a paper[2] published in 2005, had an average reliability of around 99.6% between 1990 and 2002! Making a service more reliable requires increasing commitments of both engineering time and operational support, for ever-decreasing improvements to overall reliability.

At some point before you reach 100% reliability, that tradeoff is no longer worth making because the costs outweigh the benefits.

But how do you figure out where the optimal target is?

Reliability Level	Allowed 100% outage duration		
	per year	per quarter	per 28 days
90%	36d 12h	9d	2d 19h 12m
95%	18d 6h	4d 12h	1d 9h 36m
99%	3d 15h 36m	21h 36m	6h 43m 12s
99.5%	1d 19h 48m	10h 48m	3h 21m 36s
99.9%	8h 45m 36s	2h 9m 36s	40m 19s
99.95%	4h 22m 48s	1h 4m 48s	20m 10s
99.99%	52m 33.6s	12m 57.6s	4m 1.9s
99.999%	5m 15.4s	1m 17.8s	24.2s

Allowed consistent error% outage duration per 28 days, at 99.95% reliability			
100%	10%	1%	0.1%
20m 10s	3h 21m 36s	1d 9h 36m	14d

Google Cloud

As an example, on the chart are given reliability levels in the left column - take 99.95% as an example - and to the right are the allowed 100% outage durations.

In column two, you'll see that annually, a system with a reliability level of 99.95% could be down completely for a total of 4 hours 22 minutes and 48 seconds before it violated its targeted reliability level.

For any given 28-day period, the same system with the same reliability target could be down 20 minutes and 10 seconds.

The red sections show cases where the total downtime for the given period is less than a single hour.

The smaller chart on the right further examines the 99.95% reliability target over a 28-day window.

This chart compares the error percentage and how it affects the time before the SLO error budget is depleted.

If an error is only generated for 10% of the requests, the system could run for 3 hours, 21 minutes, and 36 seconds before depleting its error budget.

A general guideline? Adding a 9 of reliability typically raises the cost by a factor of 10, while providing 1/10 the benefit.
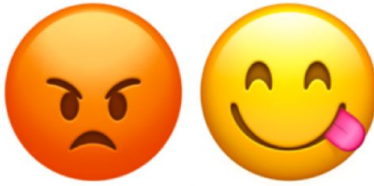
SLOs should capture the performance and availability levels that, if **barely met**, would keep the **typical customer** of a service happy.

"meets SLO targets" ⇒ "happy customers"
"sad customers" ⇒ "missed SLO targets"

We have a simple rule of thumb for where SLO targets should be, that generalizes the case: a typical user of your service should be just about happy with the service, when it operates at those targets.

If the service was any less reliable, you'd no longer be meeting their expectations and they would start to become unhappy.

We call this the "happiness test."

The challenge is quantifying and measuring that happiness, because measuring it directly tends to require the kind of invasive medical procedures that customers are unwilling to consent to.

Measure SLO achieved and try to be *slightly* over target...

SLI

Target

✔

✗

Google Cloud

When we talk about the happiness test, we say "barely met" because there are downsides to exceeding your reliability target, too.
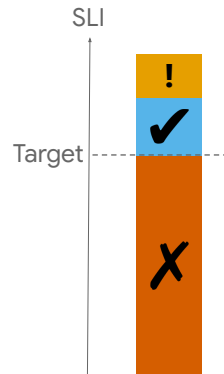
Once your users are happy with the reliability of your service, additional reliability has little value.

Maybe your users would be happier if you built new features or made other service improvements with the resources you're instead dedicating to far exceeding your reliability targets.

The worst part is that if you're consistently well over-target for your SLO, your users will eventually come to expect the service to be that reliable all the time.

A common way for this to occur stems from factoring the risk of rare-but-damaging events into your SLO targets.

If these events don't happen for a long time, through the combination of good operational response and better luck, you'll be in for a bad time from your users when they do finally occur.

Your users' expectations form a kind of implicit SLO that you won't even know exists until they unexpectedly start complaining.

Error budgets

An SLO implies an **acceptable level** of unreliability.
*This is a **budget** that can be **allocated**.*

If we're not targeting 100% reliability, then whatever target we do set implicitly allows for a small number of errors to be served to users.

If everyone agrees that the SLO target represents the point at which users start to become unhappy with the reliability of the service, we can theoretically serve errors beyond that point without impacting user happiness.

For example, if you set a three nines target, that means you can serve one error in every 1000 requests to your users.

Or, in terms of complete downtime, your service can be unavailable for a little over 40 minutes in a four-week period. Instead of just passively measuring the SLO—and potentially exceeding it substantially—you can treat the acceptable unreliability as a budget that you can spend on various development and operational activities.

Implementation mechanics

Evaluate SLO **performance** over a set **window**, e.g., 28 days.

Figuring out how much budget you've got available from a percentage reliability target means specifying a time window for the events your SLI is measuring.

Within Google, we often use fixed, quarterly SLO windows for reporting purposes, and we've seen this pattern at many of our customers' businesses too.

The problem with fixed, calendar-oriented windows is they refill your error budget in one large step as the window rolls over.

This makes them unsuitable for operational purposes: your customers are still going to remember yesterday's massive outage on the first day of the new quarter, but your budget has been reset, so everything's fine, right?

Implementation mechanics

Evaluate SLO **performance** over a set **window**, e.g., 28 days.

Google Cloud

Instead, we use shorter rolling windows to drive decisions about operational priorities.

Cloud services use a 30-day rolling window, but we generally recommend 28-day rolling windows for new SLOs.

Since it's exactly 4 weeks, there's no variation introduced by having five "release days" within the window when you are doing weekly software releases, or five weekends within the window.

We also use very short-term rolling windows like 1 or 12 hours to drive alerting—if you burn multiple hours of error budget within an hour, you can be relatively certain that an operational response is necessary!

Implementation mechanics

Evaluate SLO **performance** over a set **window**, e.g., 28 days.
Remaining budget **drives prioritization** of engineering effort.

Google Cloud

When you've decided on the time window, count the number of events within that window.

Multiplying this by the SLO target yields the absolute minimum number of good events that must be served within the window to meet the SLO, which in turn tells you the allowable number of bad events—the error budget.

When you subtract the actual number of bad events, you know how much error budget you have remaining.

This is a key signal to feed into your project planning cycle: if you have plenty of budget to spare, you can take more risks and move faster, but if you've already used most of your budget, that's a signal that you need to start prioritizing and fixing those pesky bugs instead of building that cool new stuff. If you are way over budget, your users are unhappy, and you urgently need to start trading feature velocity for reliability work.

ITIL approximation

Service *in SLO* → most operational work is a **standard change**.

Service **close** to being *out of SLO* → revert to **normal change**.

(No, I don't understand the difference between "standard" and "normal" either...)

Google Cloud

For those of you coming from more structured environments, we've found this approximation works well when trying to fit SLOs into the world of ITIL change management. (ITIL, by the way, is a set of detailed and documented practices for managing IT services)

When a service is in SLO, the developers and operators have much more leeway to make changes to the service without going through potentially lengthy review and approval processes, something ITIL calls a "standard change."

If the service is running low on error budget and looks like it may exhaust it in the near future, those processes kick back in again to catch any changes that might result in burning the rest of the budget.

Confusingly, ITIL calls this "normal change."

What should we **spend** our error budget on?

This all leads us to the question: What should we spend our error budget on?

For large services, we generally expect that some of it will be burned by a background rate of failures, but if we're making a conscious decision to allocate this budget to development or operational activity, what kind of activity are we talking about?

Error budgets can accommodate:

✓ New **feature releases**
✓ Expected system **changes**
✓ Inevitable **failure** in hardware, networks, etc.
✓ Planned **downtime**
✓ Risky **experiments**

Google Cloud

At Google, we've observed that the two biggest factors that burn error budget are software releases and configuration changes. This is why we often recommend that a reasonable consequence of burning all your error budget is to slow down or even completely stop making non-essential changes to a system until the service is back within SLO.

Of course, sometimes things just break, and it's a good idea to make sure you keep some slack in your budget allocations to account for emergencies and unforeseen circumstances.

If you have to take a service down for essential maintenance, you can pay for it with downtime from your error budget, although if the downtime is communicated far enough in advance, your users should be expecting it and therefore less unhappy about it.

Lastly, one useful thing to spend small amounts of error budget on is experimentation. The ideal is that your SLO target represents the dividing line between happiness and unhappiness, but how do you know you've drawn the line in the right place? If you have other ways of gauging the happiness of your users, you can design experiments to test this. For example, you can set things up so a small group of users are served exactly at SLO, and observe whether their happiness metrics decline.

Agenda

Google Cloud

So how exactly do you choose a good SLI?
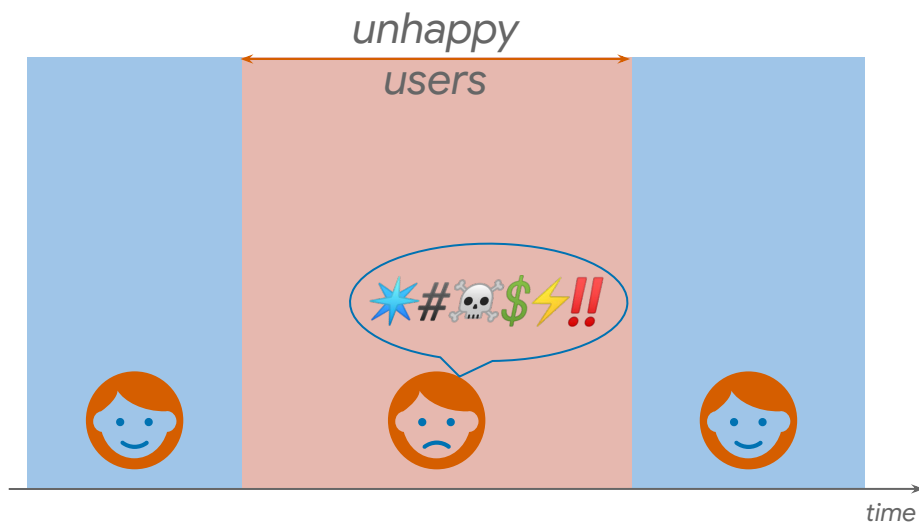
To show how this works in practice, let's consider the interval highlighted in red, where we know our users were unhappy, because we saw many more support requests during that time.

You might be wondering why we can't use "requests for support" as an SLI directly. There are a few reasons:

- First, it's a trailing indicator: while it does accurately measure "unhappy users," it's of no use if we want to *proactively* prevent those users from becoming unhappy.
- Second, many users who are unhappy with a service will silently stop using it because the burden of filing a support ticket is too high. We need to capture these users in any SLI we create.
- Third, users don't need support when the service is working correctly. This means we get no signal *at all* from this SLI most of the time, so we can't base an error budget on it.

But we've got plenty of monitoring metrics we're already collecting about our services! Maybe some of those changed during this time period, and we can use them as SLIs?

Like most of you, we've got system metrics like load average, CPU utilization, memory usage, and bandwidth graphed and visible on our monitoring dashboards. Sharp changes in these are often associated with outages, but they don't make for good SLIs. If you think about it from the perspective of your users, they don't directly

see your CPUs pegged at 100%, they see your service responding slowly. The same goes for internal service metrics like threadpool fullness or request queue length.
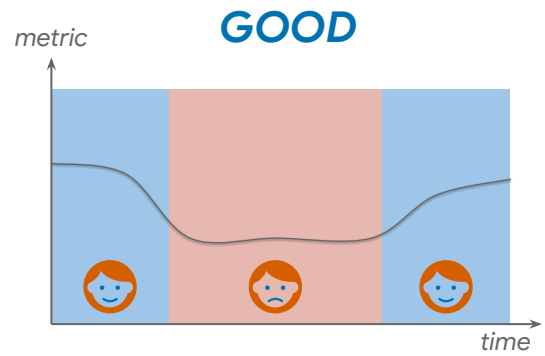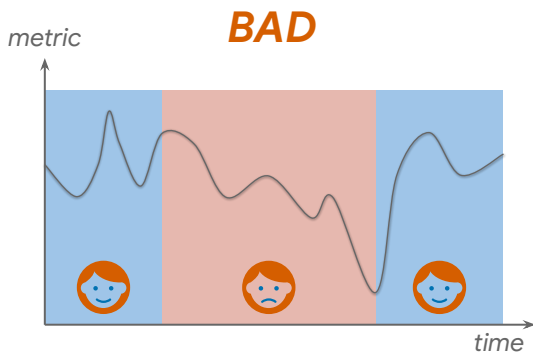
So, what properties of a metric make it good for use as an SLI?

Let's assume we've trawled through our metrics and come up with these two candidates. If we had to choose between them, why would we say that the metric on the left is bad for use as an SLI, while the metric on the right is better?
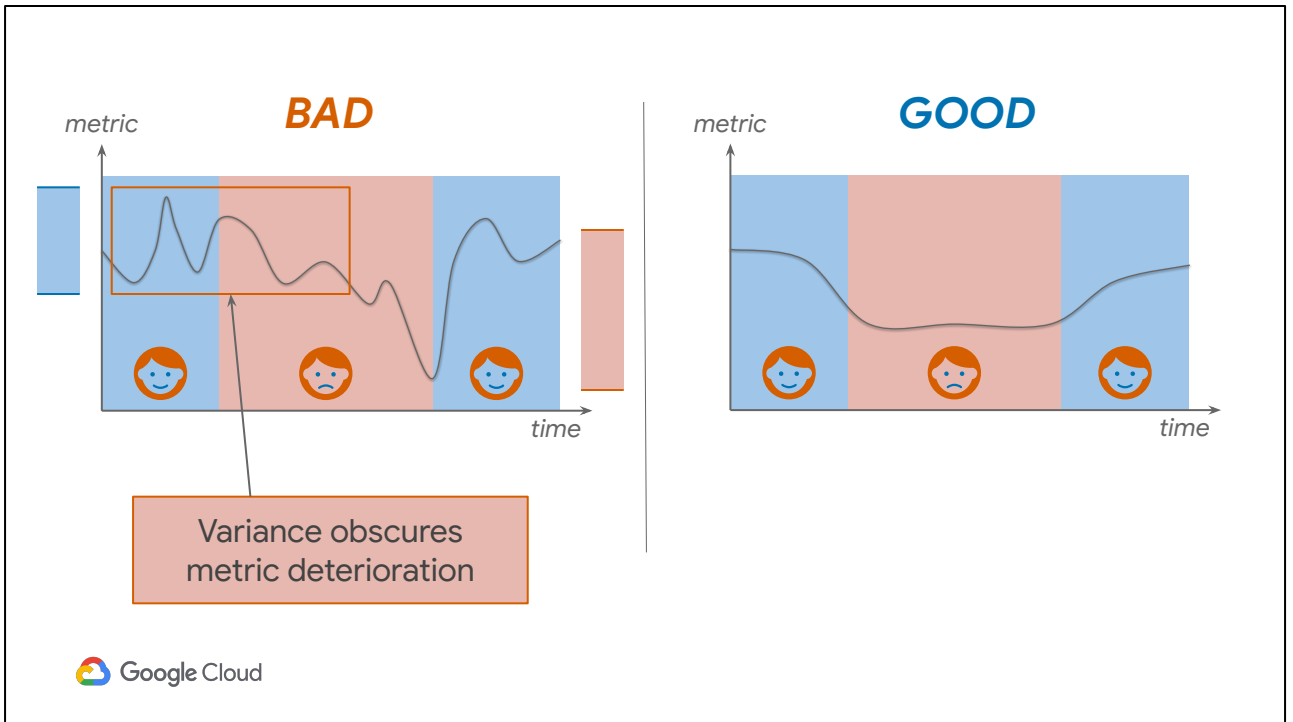
While our "bad" metric does show an obvious downward slope during the outage period, there is a large amount of variance, and the expected range of values seen during normal operation—shown in blue on the left—has a lot of overlap with the expected range of values seen during an outage—in red on the right. This makes the metric a poor indicator of user happiness.
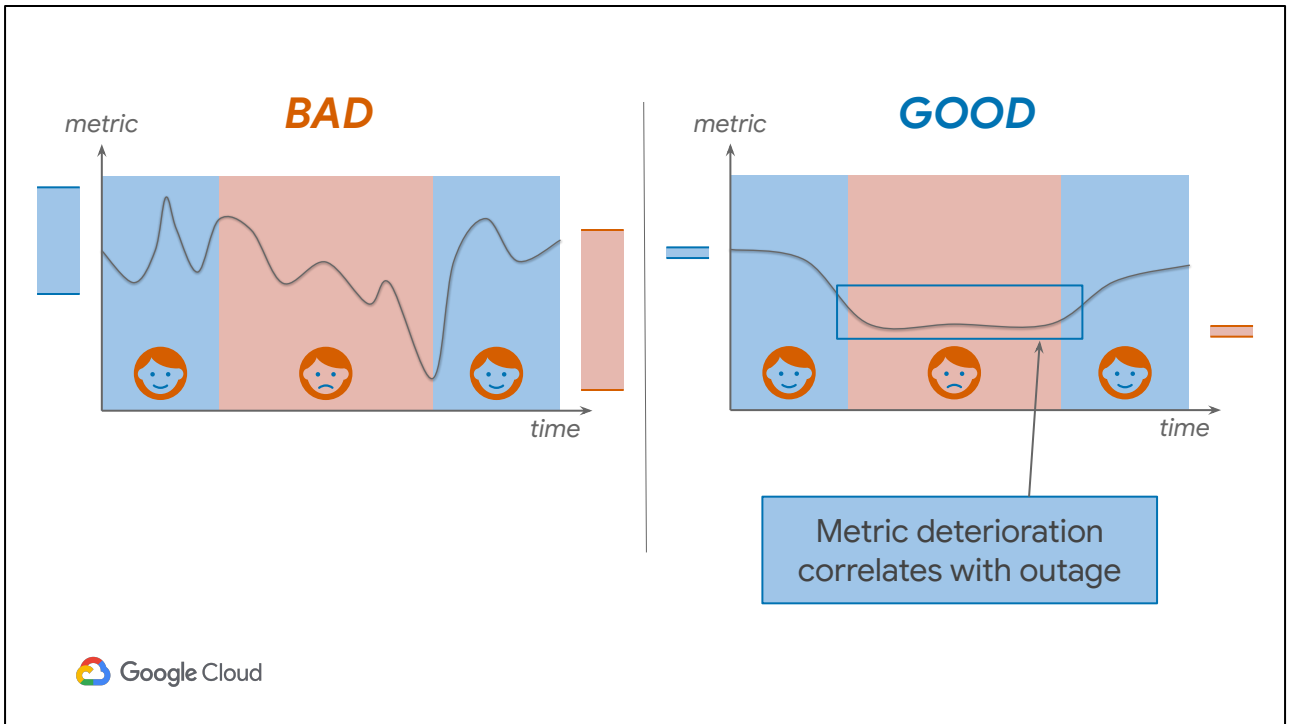
In contrast, our "good" metric has a noticeable dip that matches closely to the outage period. During normal operation, it has a narrow range of values that are quite different from the narrow range of values observed during an outage. The stability of the signal makes overall trends more visible and meaningful. This makes the metric a much better indicator of user happiness.
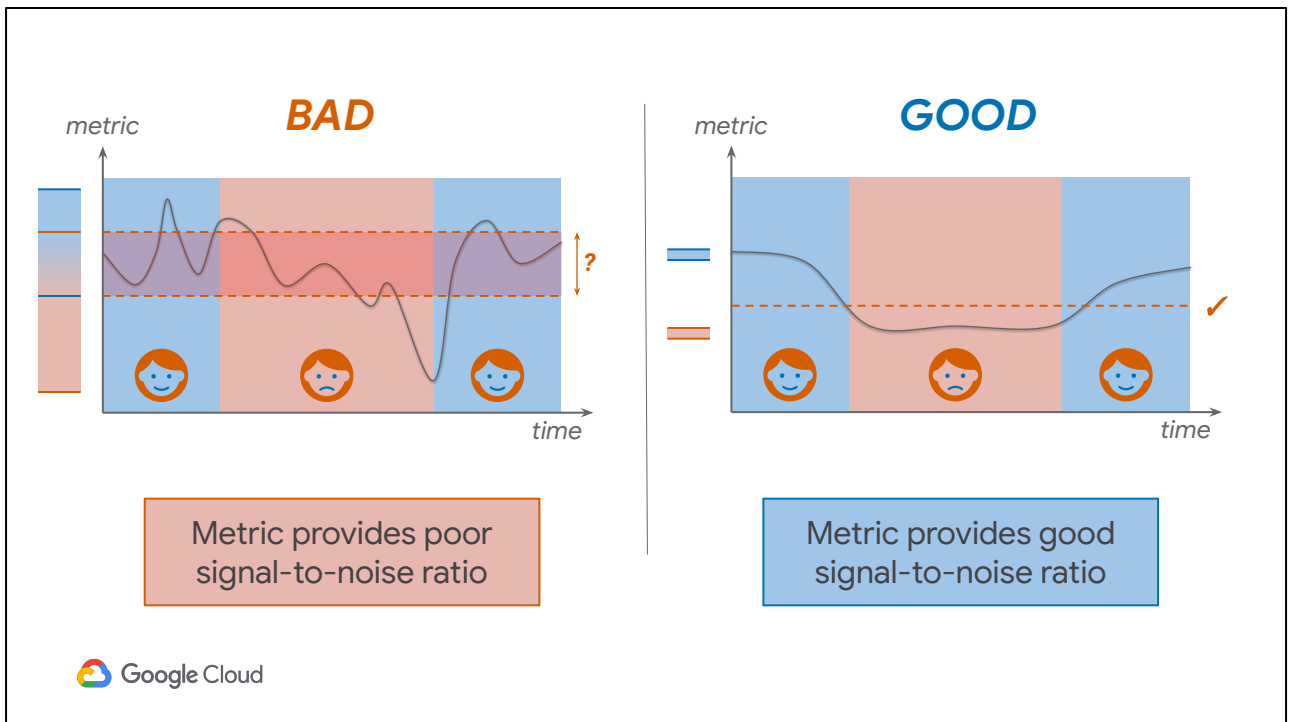
This matters because SLIs need to provide a clear definition of good and bad events, and a metric with lots of variance and poor correlation with user experience is much harder to set a meaningful threshold for. Because the "bad" metric has a large overlap in the range of values, our choices are to set a tight threshold and run the risk of false-positives, or to set a loose threshold and risk false-negatives. Worse, choosing the middle ground means accepting both risks. The "good" metric is much easier to set a threshold for because there is no overlap at all. The biggest risk we have to contend with is that perhaps the SLI doesn't recover as quickly as we might have hoped for after the outage ends.
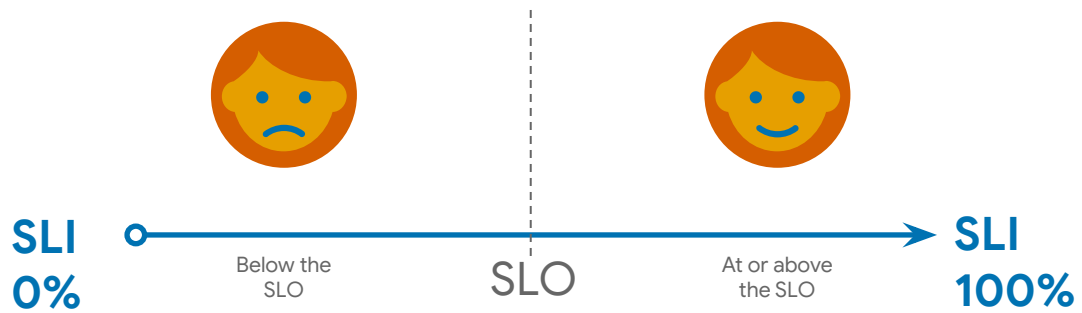
Summing up: we want our SLIs to be things we can measure about our system that also correlate well with the happiness of our users.

$$\text{SLI} : \left(\frac{\textcolor{blue}{\text{good}} \text{ events}}{\textcolor{red}{\text{valid}} \text{ events}} \right) \times 100\%$$

Google Cloud

We generally recommend that you express your SLIs as the proportion of a set of events that were considered to be "good."

Expressing all your SLIs in this form has a couple of useful properties. First, your SLIs fall between 0% and 100%, where 0% means nothing works and 100% means nothing is broken. This scale is intuitive and directly translates to percentage-reliability SLO targets and error budgets.

Second, your SLIs have a consistent format which can serve as an interface for abstraction. Consistency allows common tooling to be built around your SLIs. Alerting logic, error budget calculations, and SLO analysis and reporting tools can all be written to expect the same inputs: good events, valid events, and your SLO threshold.

But what do we mean by "valid," and why don't we use "all events" in the SLI equation? Sometimes you may need to exclude some events recorded by your underlying metrics from being included in your SLI, so they don't consume your error budget. A good example here might be ignoring 300 and 400 HTTP response codes as irrelevant to your service's SLO performance. This phrasing allows you to make this exclusion explicit and specific by declaring those events to be "invalid."

Lab intro

Postmortem Review
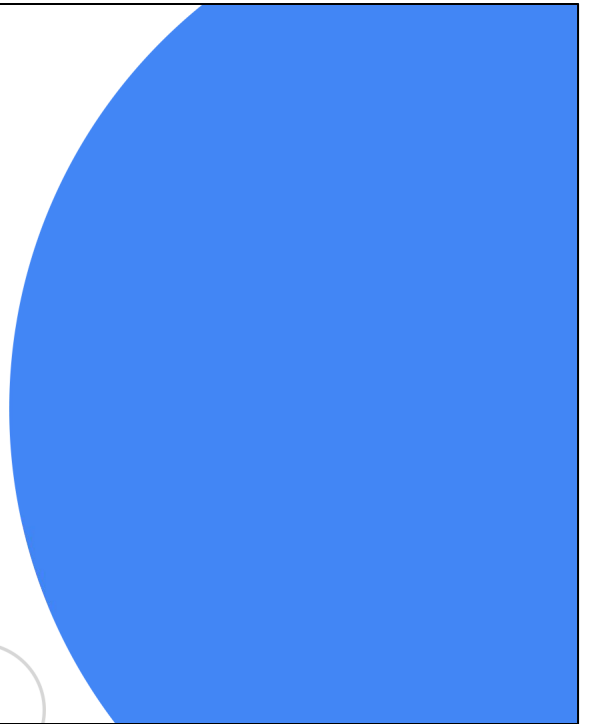
Google Cloud

Agenda

Why Monitor?

Critical Measures

Four Golden Signals

Google Cloud

Now, let's talk about specifying SLIs

Our game: Fang Faction

Web Server

Load Balancer

API Server

Leaderboards

User Profiles

Game Servers

Leaderboard Generation
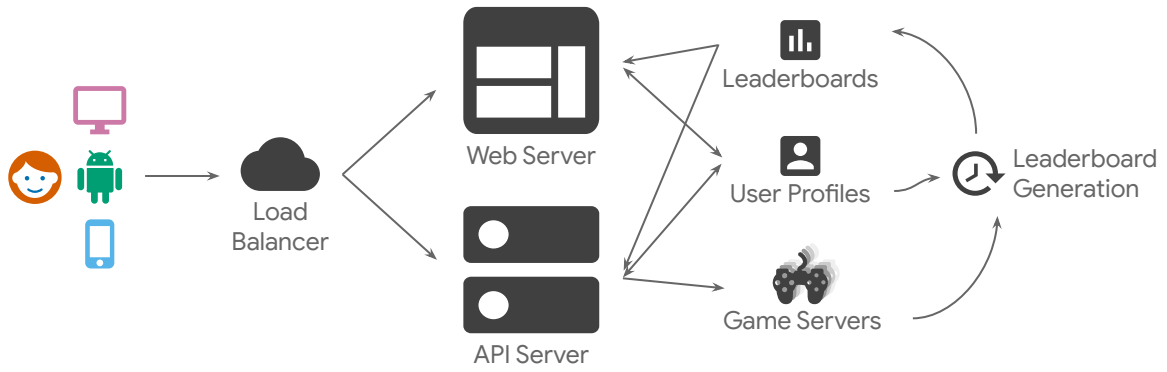
Google Cloud

Let's start with an example application: Fang Faction.

In the game, players build and manage a post-apocalyptic settlement, recruit survivors to their faction, and wage war on other players. Like many games, it makes revenue by selling in-game currency for real-world money, which players can then use to skip time gates, upgrade buildings, and get more recruits.

The serving infrastructure is relatively simple. There are two data stores, a set of API servers and web servers, and a set of game servers with their own databases all sitting behind a layer 7 HTTP load balancer. The game has both a mobile client and a web UI. The mobile client makes requests to our serving infrastructure via JSON RPC messages transmitted over RESTful HTTP. It also maintains a web socket connection to receive game-state updates. Browsers talk to the web servers via HTTPS. Leaderboards are updated every 5 minutes.

Loading a profile page

https://fangfactiongame.com/profile/someuser

SomeUser's Profile

SomeUser
Tribe of Frog
Faction Score: **31337**
Midwest Canyon
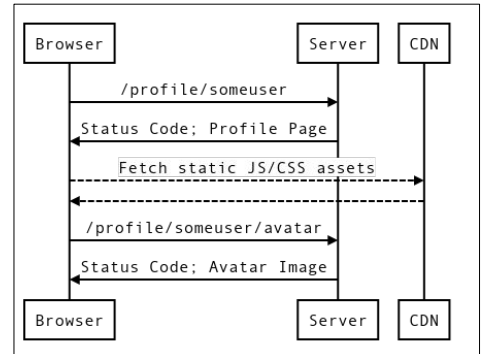
1. Tri-Bool   **65535**
2. Tri Repetae   **61995**
3. Triassic Five   **52391**
4. Tricksy Hobbits   **37164**
5. Tribe of Frog   **31337**
6. Trite Examples   **29243**

Faction Name: Tribe of Frog

Leader Name: SomeUser

Email Address: user@example.com

Update

Google Cloud

What are users' expectations when they try to load this page?

I'm sure you can imagine a number of user journeys for this game. We're going to keep things simple for now, so we can step through the process without too many complicating factors, but you'll find some more complex journeys in your handout.

Loading a profile page

https://fangfactiongame.com/profile/someuser

SomeUser's Profile

SomeUser
Tribe of Frog
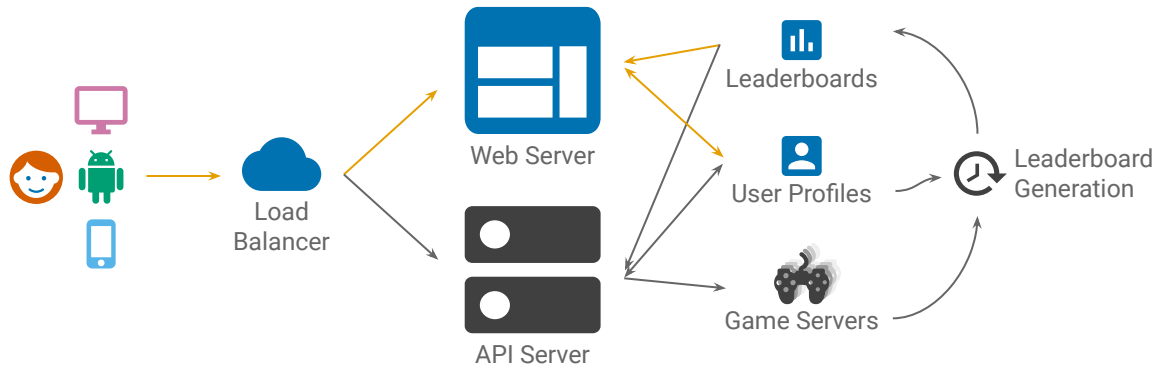Faction Score: **31337**
Midwest Canyon
1. Tri-Bool   **65535**
2. Tri Repetae   **61995**
3. Triassic Five   **52391**
4. Tricksy Hobbits **37164**
5. Tribe of Frog   **31337**
6. Trite Examples **29243**

Faction Name: Tribe of Frog

Leader Name: SomeUser

Email Address: user@example.com

Update

Google Cloud
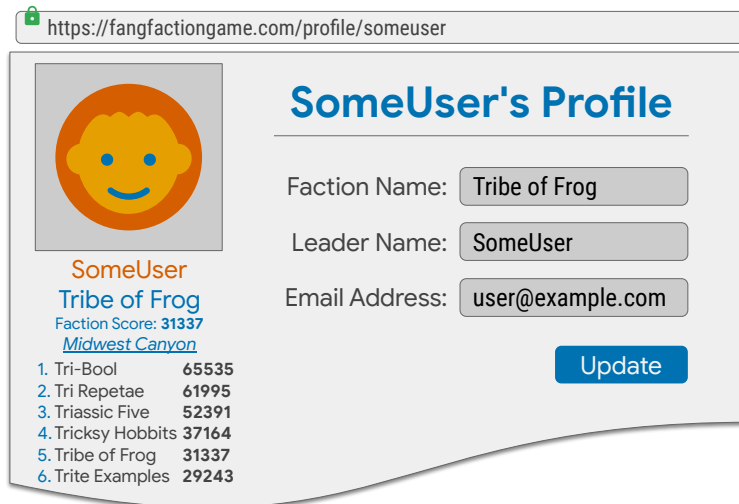
Our game allows players to log in to their account via a web browser and keep track of their settlement while they aren't actively playing on their device. Here, you can see a mock-up of what a player sees when they log in to the game's website and open their profile page. On the right is a sequence diagram showing the requests and responses involved in serving this page to them. ~~You can also find this on page 18 of your handout.~~ The dotted lines are requests that are not observable from our serving infrastructure. What do you think their expectations are for this page? What would disappoint them, and make them think the service is not reliable?

How the profile page is served

Web Server

Load Balancer

API Server

Leaderboards

User Profiles

Game Servers
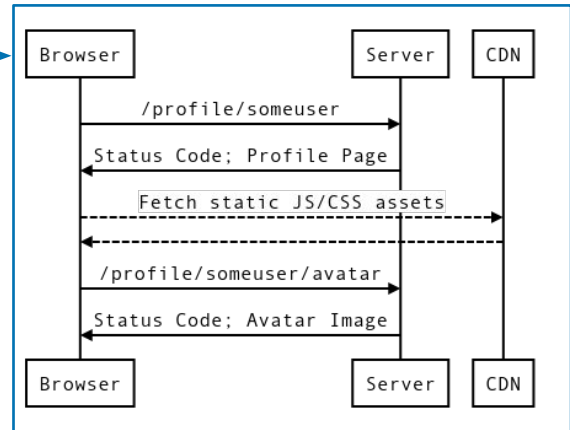
Leaderboard Generation

Google Cloud

Here's how our infrastructure serves a profile page.

- The player's HTTPS request is terminated at our load balancers, which forwards it to a pool of web servers.
- That server looks up the player's profile in the user profile data store.
- The profile contains most of the information displayed on the profile page, including the player's score and current location of their settlement.
- The web server requests the leaderboard for that location from the leaderboard data store, then builds the HTML response for the player and sends it back.
- When the player's browser subsequently requests their avatar, this request is served in much the same way.

User journeys within our game

- **View Profile Page** →
- Buy In-Game Currency
- App Launch
- Manage Settlement
- Battle Another Player
- Generate Leaderboards

Browser — Server — CDN

/profile/someuser

Status Code; Profile Page

Fetch static JS/CSS assets

/profile/someuser/avatar

Status Code; Avatar Image

Browser — Server — CDN

A "user journey" is the set of interactions a user has with a service while achieving a single end result.
- Similar to Critical User Interactions (CUIs) from user experience (UX) research

In our game, each user journey represents a discrete set of actions the user could perform, such as "View Profile Page." This user journey starts with the user's web browser initiating a connection to the web servers. It fetches content from a Content Delivery Network (CDN), and may allow the user to upload a new avatar photo.

Applications have many user journeys, and each journey will have separate measurable metrics that users likely already use subconsciously to rate the reliability of your application or service. In this lesson, you will learn how to identify these metrics and use them to ensure that your user experience remains within acceptable bounds.

SLIs

It may be tempting to rush off and start looking at which metrics might make good SLIs for your services, but we've got one more recommendation for you first.

In general, your users are using your service to achieve some set of goals, so your SLIs must measure their interactions with your service in the pursuit of those goals.

3–5 SLIs[*]

[*]per user journey

Google Cloud

You should aim to have around 3-5 SLIs covering each user journey, even if your system and your user journeys are relatively complex. A service with too many SLIs places an additional cognitive burden on the people operating that service, especially if those SLIs start to contradict each other.

We're not recommending that you discard all your other monitoring and observability systems after you start measuring SLIs. The way we think about it is that a deterioration in your SLIs is an indication that something is wrong. When that problem has become bad enough to provoke some form of incident response, you'll need those other systems to debug and help you figure out what that is.

SLI Menu

	Request/Response	Availability Latency Quality
	Data Processing	Coverage Correctness Freshness Throughput
	Storage	Throughput Latency

Google Cloud

To begin figuring out what SLIs you should have for each journey, ask yourself:

- What are the user's expectations for the reliability of this service?
- How can we measure the user's experience versus those expectations with our monitoring systems?
- How does the user interact with the service?

The SLI menu we're showing you here is a good place to start if you're not sure what kind of SLIs you should measure for a particular user journey. It's also on page 6 of your handout, so you can refer to it this afternoon.

Agenda

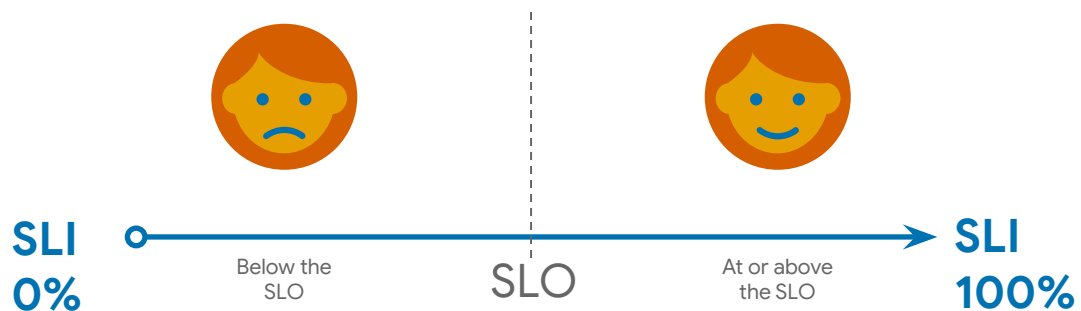Why Monitor?

Critical Measures

Four Golden Signals

Google Cloud

Now let's take our SLI measures and develop them into SLOs and SLIs

An SLO is supposed to represent the line at which users become unhappy with a service.

Once you're sure you have a good SLI—one that has a close, predictable relationship with the happiness of your users—what reliability target should you set?

How do you figure out where to draw the line?

What **performance** does the **business** need?

Google Cloud

The ideal is that your reliability targets reflect the needs of your business. As we said earlier, being too reliable has a cost—if your service is amazingly reliable but your users would be happy with two nines because its failure modes are acceptable, maybe that means you can take more risks and ship features faster. Not being reliable enough is often more costly: what if the only thing keeping your users with you is that your competitors reliability is even worse than yours—for now!

We call SLOs based on a business need "aspirational SLOs," because it's entirely reasonable for you to not be able to meet them initially. Over time, with some engineering effort, this is the level of reliability that your operations, development, product, and executive functions want to reach, that represents your best guesses at what level of reliability is right for your services, customers, and business over the long term.

User expectations are *strongly* tied to **past performance**.

Google Cloud

But if you have or can derive historical data for your SLIs, we recommend that you look at this data and base your *first* SLOs on the past performance of your service. Starting from this point means you can be reasonably sure of your ability to meet the SLO over the short-to-medium term. If your users are happy enough with your service at the moment, you can be reasonably sure that this SLO is not set too low.

We call SLOs based on past performance "achievable SLOs"; since you should set the SLO threshold so that you can expect to meet it most of the time. Just remember that past performance does not indicate future reliability; the data you're basing your judgments on may not be representative of the long-term reliability achievable by your service.

The difference between these two targets is a useful signal. If the business needs better performance than your service is currently capable of achieving, that's a problem. But it's common for there to be some divergence when you're just setting out on your SLO journey.

What's needed is some way to drive convergence between the two targets over time. Aspirational SLOs are best guesses at what the business thinks makes the user happy, and achievable SLOs are making the assumption that current performance makes the user happy. To validate these assumptions, you need to find some external signals that indicate the happiness or discontent of your user base. Good examples of these are support requests, forum complaints, and Twitter.

Understanding any discrepancies between these signals of user happiness and your SLOs, particularly instances where your users were sad but you were still in SLO, will help you refine your SLO targets. Significantly overperforming your SLO means that you have error budget to spend—maybe you could safely take more risks. Significantly underperforming your SLO may mean that you're taking too many risks, or—if your users are happy—you should relax your SLO targets.
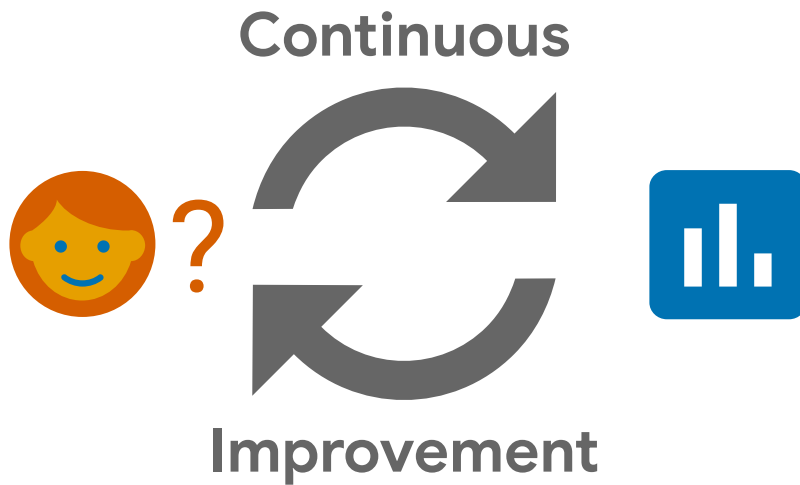
When you're just starting out, checking your SLOs against these signals more frequently is a good idea. You don't want to wait a whole year to find out that the first shot at setting reliability targets was way off! As you gain more confidence that your targets are in the right place, you can revisit them less frequently, but we recommend doing so at least once a year. A lot can happen in a year: your user base may grow dramatically, or your business might pivot to new markets with different requirements.

For each critical user journey:

1   Choose an **SLI specification** from the menu.

2   Refine the specification into a detailed **SLI implementation**.

3   Walk through the user journey and look for **coverage gaps**.

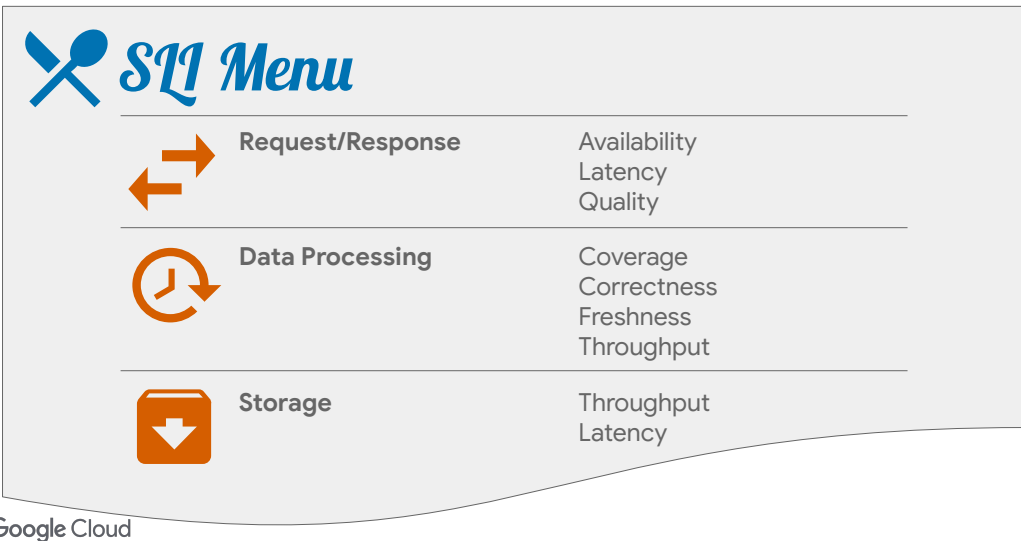**4**   Set SLOs based on **past performance** or **business needs**.

In general, people use your service to achieve some set of goals, so the SLIs for that service must measure the interactions they have with the service in pursuit of those goals.

An SLI specification is a formal statement of your users' expectations about one particular dimension of reliability for your service, like latency or availability. The SLI menu gives you guidelines for what dimensions of reliability you probably want to measure for a given user journey.

When you have SLIs specified for a system, the next step is to refine them into implementations by making decisions around measurement, validity, and how to classify events as "good."

To begin figuring out what SLIs we should have for this journey, we ask ourselves:

- What are the user's expectations for the reliability of this service?
- How can we measure the user's experience versus those expectations with our monitoring systems?
- How does the user interact with the service?

The SLI menu we're showing you here is a good place to start if you're not sure what kind of SLIs you should measure for a particular user journey.

2 Refine the specification into a detailed SLI implementation

Availability

The **profile page** should load **successfully**.

Latency

The **profile page** should load **quickly**.

Google Cloud

This is a request/response interaction, so we almost certainly want to measure the availability and latency experienced by players when they load the profile page. Put another way, they expect the profile page to load *successfully* and *quickly*.

2 Refine the specification into a detailed SLI implementation

Availability

The **profile page** should load **successfully**.

- How do we define **success**?
- Where is the success/failure **recorded**?

Latency

The **profile page** should load **quickly**.

- How do we define **quickly**?
- When does the timer **start/stop**?

But these expectations raise questions: what does "successfully" and "quickly" really mean? It's important to be precise about the SLI definition. Your SLI should be clear about where it is measured, what is being measured (including any units), and what attributes of the underlying monitoring metrics are included or excluded. How can we define an SLI that answers these questions?

2 Refine the specification into a detailed SLI implementation

Availability

The **profile page** should load **successfully**

- How do we define **success**?
- Where is the success/failure **recorded**?

The proportion of **valid** requests served **successfully**

Latency

The **profile page** should load **quickly**

- How do we define **quickly**?
- When does the timer **start/stop**?

The proportion of **valid** requests served **faster** than a threshold

Our first problem is that our web servers serve way more than just profile page requests, but it's only these requests that are part of this particular user journey.

Refine the specification into a detailed SLI implementation

Availability

The **profile page** should load **successfully**

- How do we define **success**?
- Where is the success/failure **recorded**?

The proportion of **valid** requests served **successfully**

Latency

The **profile page** should load **quickly**

- How do we define **quickly**?
- When does the timer **start/stop**?

The proportion of **valid** requests served **faster** than a threshold

Google Cloud

So let's define what attributes of a request make it valid for inclusion in our SLIs.

We identify profile page requests from the HTTP request path, which we know from our sequence diagram is either /profile/{user} or /profile/{user}/avatar. We also know the user's browser sends HTTP GET requests to these paths. So let's substitute these definitions in!

2 Refine the specification into a detailed SLI implementation

Availability

The **profile page** should load **successfully**

- How do we define **success**?
- Where is the success/failure **recorded**?

The proportion of **HTTP GET** requests for **/profile/{user}** or **/profile/{user}/avatar** served **successfully**

Latency

The **profile page** should load **quickly**

- How do we define **quickly**?
- When does the timer **start/stop**?

The proportion of **HTTP GET** requests for **/profile/{user}** served **faster** than a threshold

Here I've chosen to include avatar images in the availability SLI, but exclude them from the latency SLI.

Avatar images are uploaded by users and we don't know how large they'll be. We can realistically assume that images take longer to load than HTML. This means that including avatar image latency into our SLI would make the underlying latency distribution bimodal and introduce a lot of variance.

If you remember what we talked about earlier, this could cause problems when we're trying to set a latency SLO target, because we can't accurately target just one of the two loading times. Our users might be unhappy with a 500ms increase in page loading time, but that could still be less than the average time it takes to serve an avatar image, which they're still happy with in aggregate. Since the page is still broadly usable even before the avatar image loads, page loading time is the metric we care about more, so that's what we'll base our SLI on.

2 Refine the specification into a detailed SLI implementation

Availability

The **profile page** should load **successfully**

- How do we define **success**?
- Where is the success/failure **recorded**?

The proportion of **HTTP GET** requests for **/profile/{user}** or **/profile/{user}/avatar** served **successfully**

Latency

The **profile page** should load **quickly**

- How do we define **quickly**?
- When does the timer **start/stop**?

The proportion of **HTTP GET** requests for **/profile/{user}** served **faster** than a threshold

Google Cloud

Next, we need to be more specific about what "successfully" and "quickly" really mean. Our company just wants to get something simple measured quickly, so we're going to use the HTTP status code as an indicator of success.

The latency threshold is a bit harder. If you have historical data, it's common to choose the cutoff point where requests are too slow in tandem with the SLO target that specifies what percentage of requests are allowed to be slower than the cutoff.

The company has historical data for this particular endpoint, so that's what we're going to do here with this "X milliseconds" construct. That makes this SLI implementation incomplete—you need a threshold to convert the latency distribution to a binary good/bad signal. We'll be asking you to figure out what thresholds are appropriate in a bit!

Instead of enumerating every possible code for our availability SLI, we'll assume that any 500-class codes are bad, and exclude 429s from our "good requests" too because it's what our servers return to users when they're overloaded.

But there's still one piece of the puzzle missing. We haven't said how or where we're going to measure any of this!

2 Refine the specification into a detailed SLI implementation

SLI Menu

Measurement Strategies
Application-Level Metrics
Logs Processing
Front-End Infra Metrics
Synthetic Clients/Data
Client-Side Instrumentation

Google Cloud

Broadly speaking, there are five ways to measure an SLI, and none of them are perfect. Making an informed engineering decision on which measurement strategy to use, based on the current needs of your service, is an important part of refining your SLI specification into concrete implementation.

It's entirely reasonable to use more than one strategy for each specification. For example, you may choose to use real-time metrics from your load-balancing infrastructure to drive short-term operational response, but base your long-term SLOs on reconstructing user journeys from session IDs in server-side logs.

2 Refine the specification into a detailed SLI implementation

Availability

The **profile page** should load **successfully**.

- How do we define **success**?
- Where is the success/failure **recorded**?

The proportion of **HTTP GET** requests for **/profile/{user}** or **/profile/{user}/avatar** that have *2XX*, *3XX* or *4XX* (excl. 429) status measured at the **load balancer**

Latency

The **profile page** should load **quickly**.

- How do we define **quickly**?
- When does the timer **start/stop**?

The proportion of **HTTP GET** requests for **/profile/{user}** that send their **entire response within *X* ms** measured at the **load balancer**

Google Cloud

In this case, the company decided that the metrics provided by their load balancer were good enough to establish some initial SLIs without investing too much engineering effort.
This is an entirely reasonable trade-off to make: your first SLIs are almost certainly going to need some improvement, so the sooner you start gathering data and begin the cycles of iteration, the better.

These final SLI implementations provide a lot of explicit detail about what exactly is being measured and how it is measured. If you give them to someone, they could actually build something that evaluates these metrics and gives you performance data!

Prior example: User profile journey
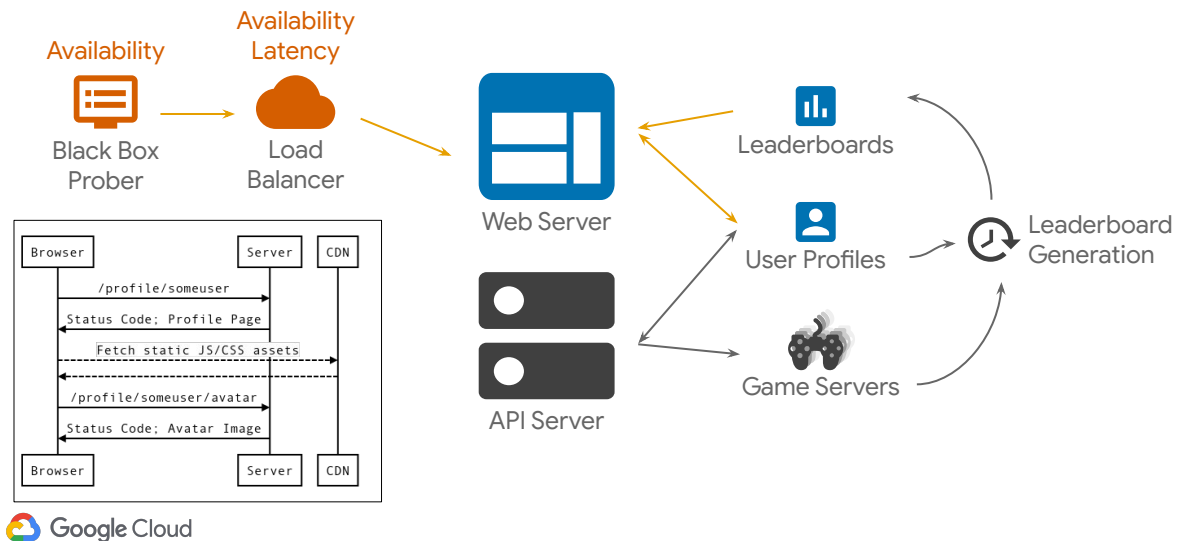
This example outage illustrates why you should take a critical look at your infrastructure when creating your SLIs.

When we're considering the failure modes of our infrastructure, we're trying to answer four questions:
- Can we measure the SLIs we've created where we want to measure them, given this infrastructure?
- Do the SLIs adequately capture the user journey and its failure modes?
- Are there any exceptions or edge cases to consider?
- Do the SLIs capture multiple journeys with differing requirements?

Prior example: User profile journey
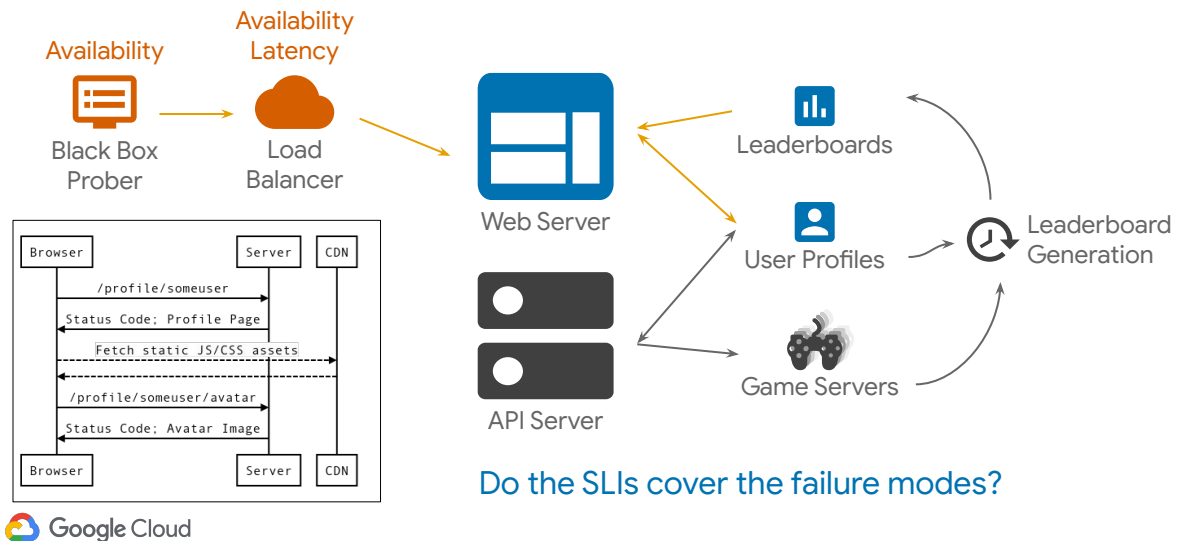
We say "adequately capture" because reaching 100% coverage of all possible failure modes is again the wrong target.

Distributed systems fail in complex and surprising ways, so 100% is an unrealistic goal.

If the failure mode is sufficiently rare that it won't eat more than a fraction of your error budget, it's reasonable to argue that it's safe for your SLIs not to capture it.

Failure modes that are outside of your control, and your users are unlikely to blame you for, are best left out of your SLIs.

Things like failures at consumer ISPs or mobile phone carriers would fall into this bucket: you can't do much about them, but in some cases, it can be useful for your support teams to know they have happened.

Prior Example: User Profile Journey
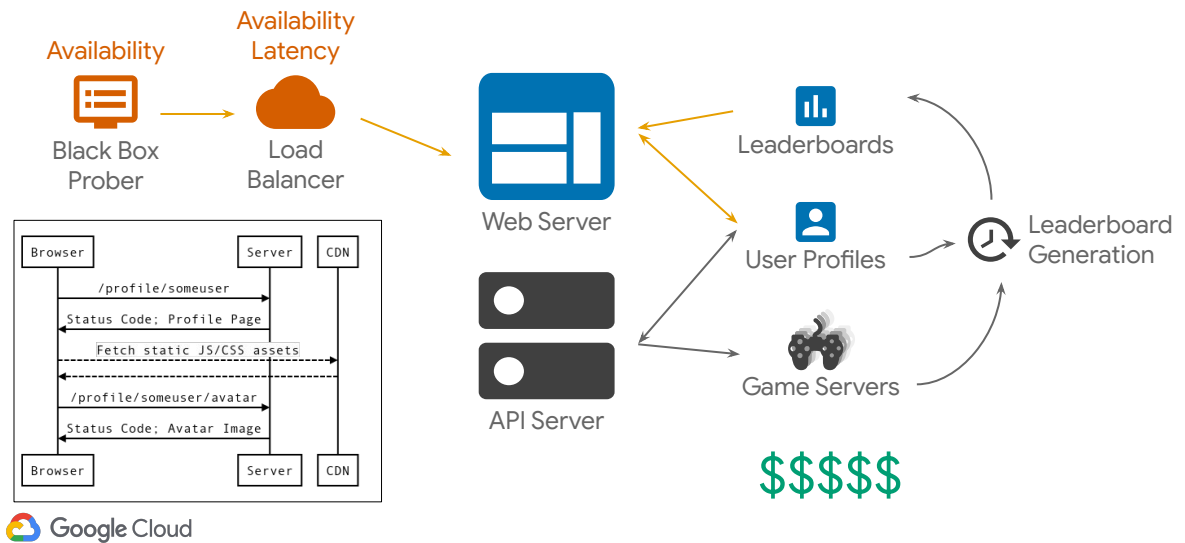
You also have to perform some form of cost-benefit analysis. In some cases, extending your SLI to cover a particular failure mode could cost months of engineering effort and require re-architecting your service, your monitoring systems, or both—when building detailed client-side instrumentation from scratch.

Prior Example: User Profile Journey

What goals should we set for the reliability of our journey?

Your objectives should have both a **target** and a **measurement window**.

Service	SLO Type	Objective
Web: User Profile	Availability	**99.95% successful** in **previous 28d**
Web: User Profile	Latency	**90%** of requests **< 500ms** in **previous 28d**
...	...	

Google Cloud

What targets do you think are justifiable given the previous six weeks of data? What measurement window would you measure this target over?

Lab Intro

Develop SLIs and SLOs

Google Cloud

In this exercise, you will follow the process we learned on the user profile journey to develop service level indicators (SLIs) and service level objectives (SLOs) for the Buy In-Game Currency journey.