



Avoiding Customer Pain



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

Why Monitor?

Critical Measures

Four Golden Signals

SLIs, SLOs, SLAs

Choosing a Good SLI

Specifying SLIs

Developing SLOs and SLIs



Specifically, you will learn how 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.
- Achieve developer and operation harmony with SLO-based error budgets.

Agenda

Why Monitor?

Critical Measures

Four Golden Signals

- SLIs, SLOs, SLAs

- Choosing a Good SLI

- Specifying SLIs

- Developing SLOs and SLIs

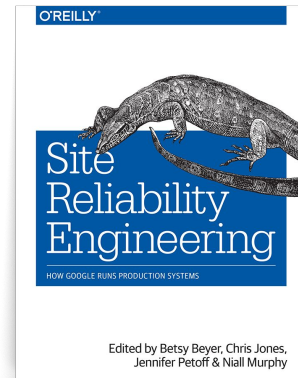


Why should we monitor?

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

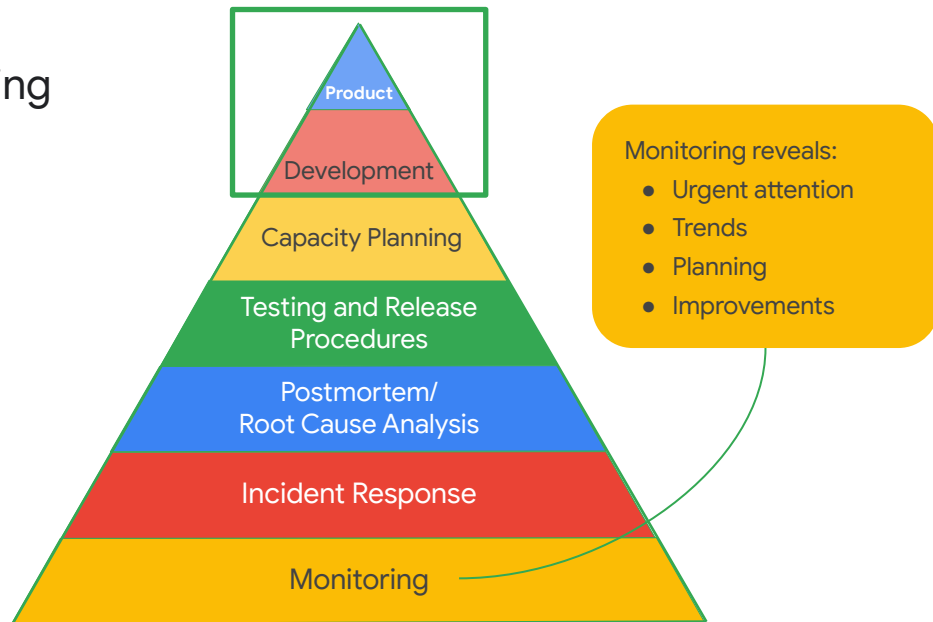


Let's start by defining our terms. In Google's Site Reliability Engineering book, Monitoring is defined as "Collecting, processing, aggregating, and displaying real-time quantitative data about a system, such as query counts and types, error counts and types, processing times, and server lifetimes."

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/.

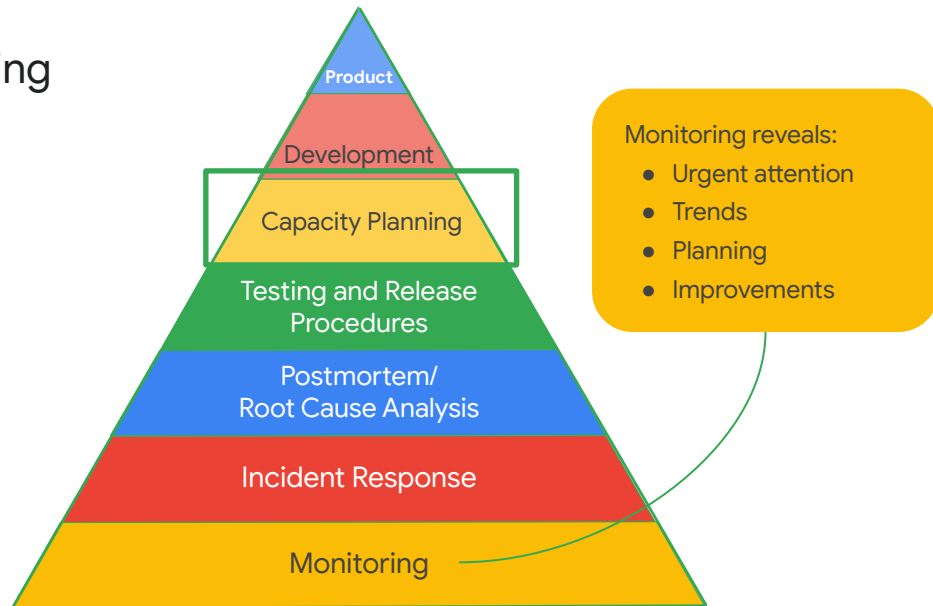
Monitoring



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.

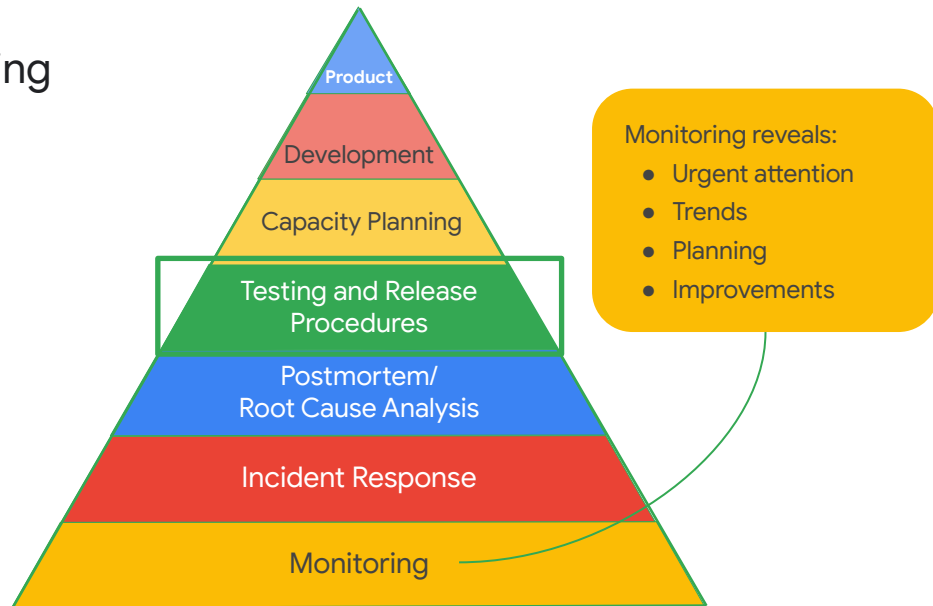
But developing the product is just the tip of the pyramid.

Monitoring



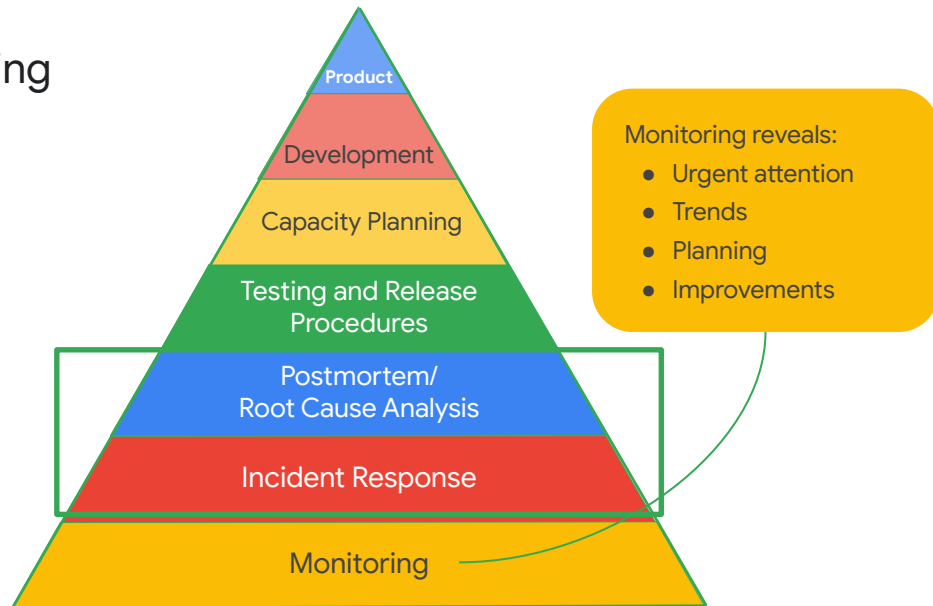
Great products need to be deployed into environments with enough capacity to handle client load, so someone needs the correct information to make those capacity planning decisions.

Monitoring



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.

Monitoring



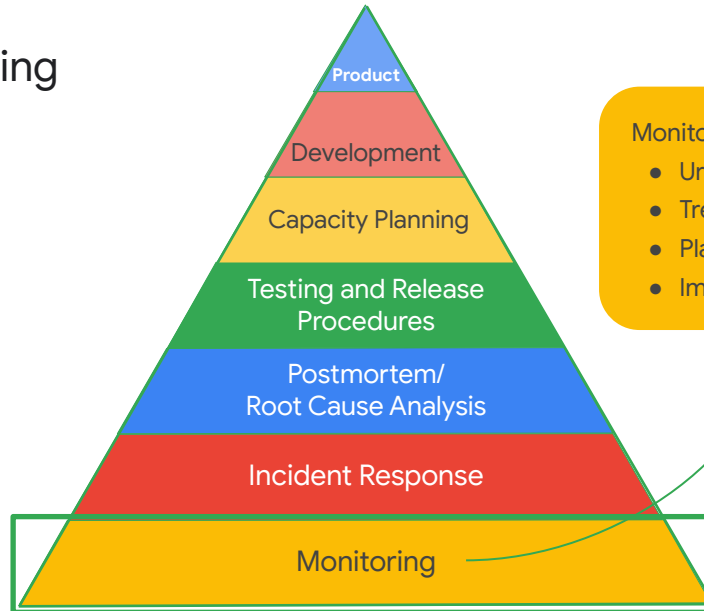
Past that, we also have to keep clients happy by building trust.

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.

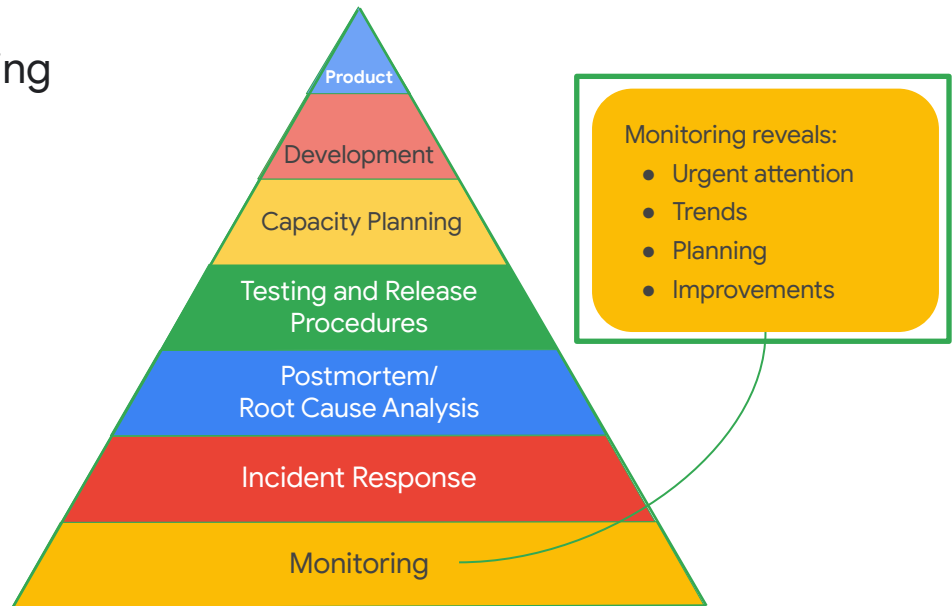
Monitoring



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

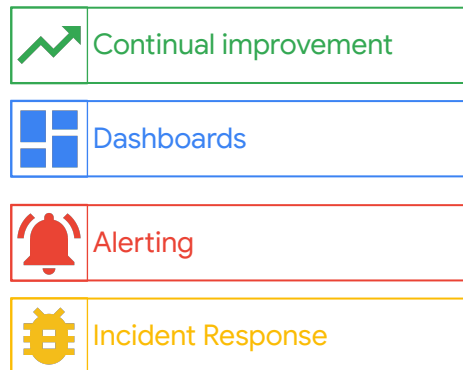
All of this sits squarely on a solid base of monitoring systems and data.

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?



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, which they always show in the movies, is a fallacy. Humans look when there's something important for them to look at, even better, construct automated systems to handle as many alerts as possible.

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



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 it affecting performance?
- How would change X impact capacity?



Continual improvement



Data for business, application, and security analysts

Spot evolving customer needs

How (and where) could software be
improved?

Is someone trying to hack the system?

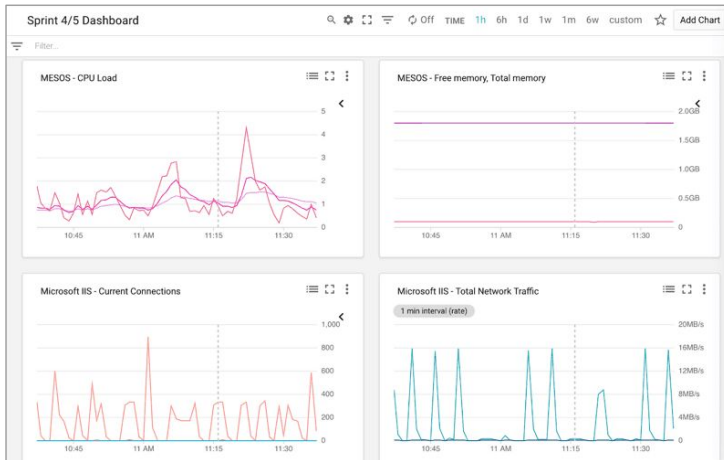


Continual improvement-related monitoring also provides data crucial for business, application, and security analysts, helping answer questions like:

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



Dashboards provide



- Core metrics in a summary view.
- Views targeted to user.
- Metrics in usable amounts.
- Options for non-human monitoring.

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!**



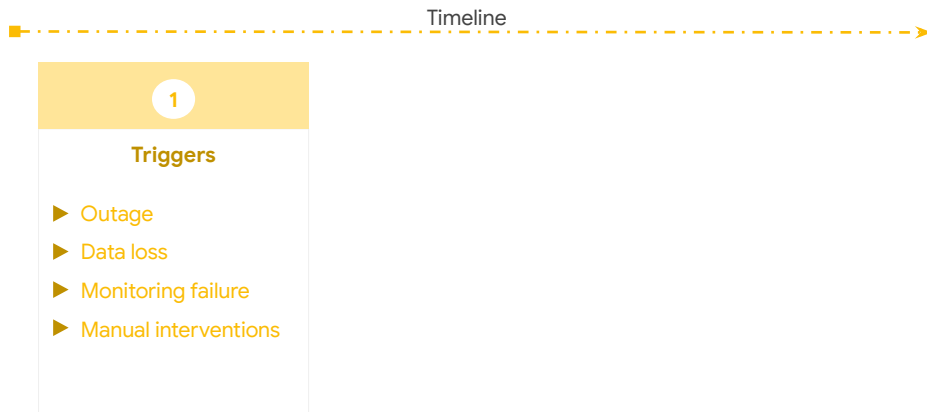
Configure alerting policies to notify you when events occur, or a particular system or custom metrics violate rules that you define.

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.



Incident response

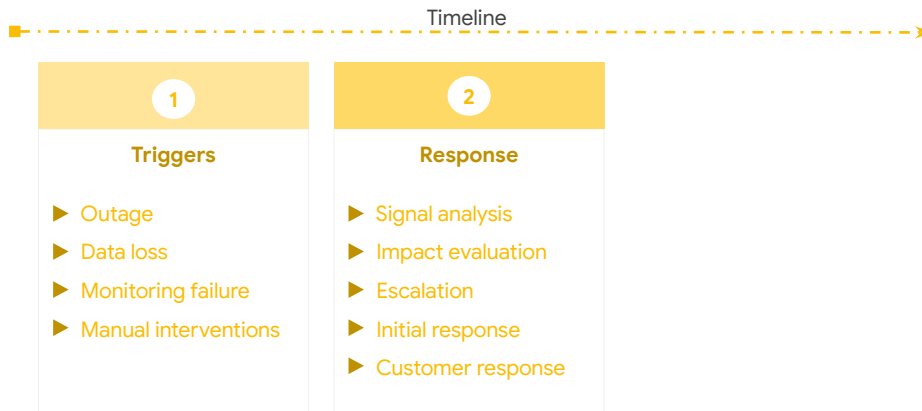


In any system, bad things can happen, and when they do, there needs to be an organized response.

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



Incident response

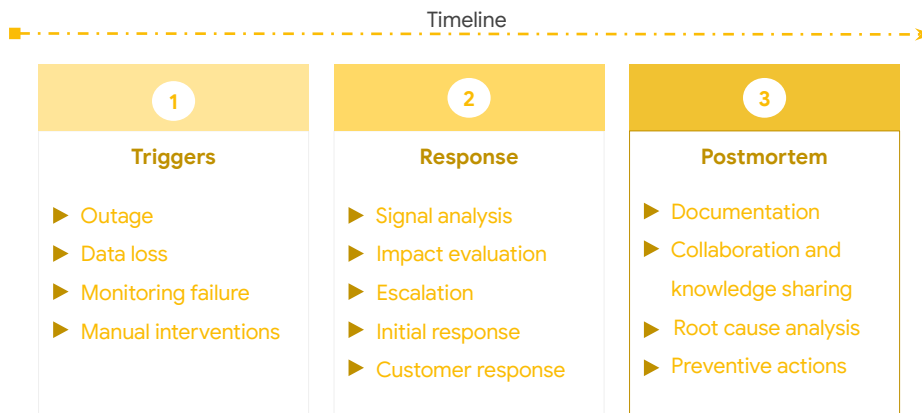


The trigger leads to a response by both automated systems and DevOps personnel.

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.



Incident response



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 infallible.
 - 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.



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 is, this class covers multiple Google Cloud products.

There is no such thing as an infallible system. You're never going to meet and keep meeting a 100% SLO, but there can be SLOs that makes sense, and you can build trust through transparency and quick fixes.

Never forget the KISS, Keep it Simple Stupid, principle when you're planning and implementing monitoring.

Avoid overly complex "Magic" monitoring solutions, KISS. Start with the four golden signals.

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 overly myopic. Monitor from multiple vantage points. External monitoring is helpful because it represents the client's view, but it's also limited in what it can access data-wise. 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 Google Cloud 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.



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.



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
<ul style="list-style-type: none">• The site is serving “500” errors.• Service latency has spiked.• A private file is publicly accessible.• The service is returning errors to UK clients.

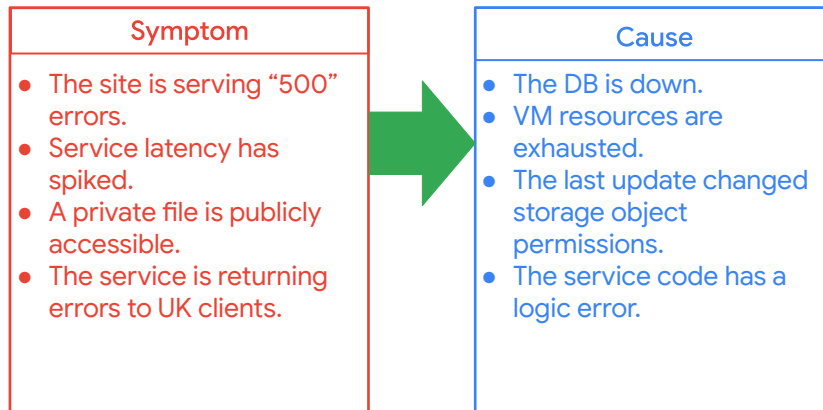


Your monitoring system should address two questions: what exactly is broken, and why?

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

- A website 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)

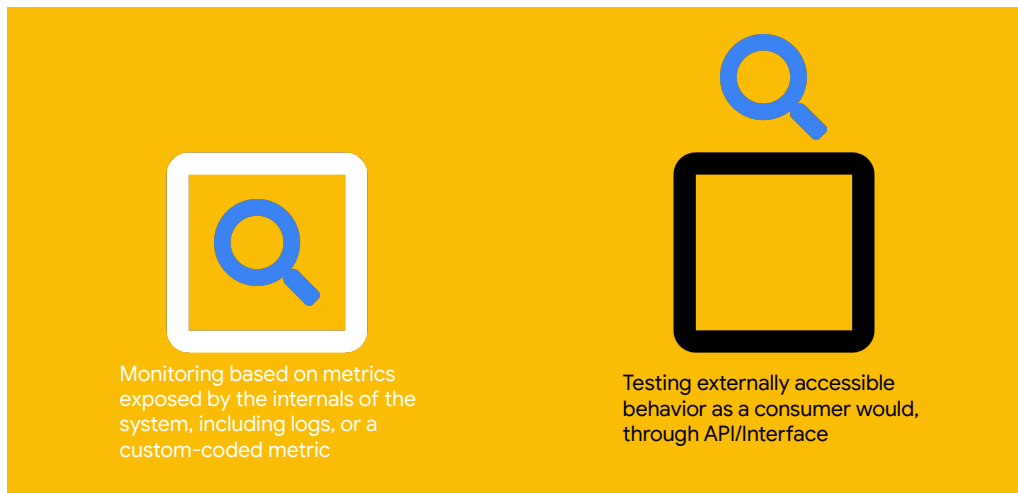


The "why" from our two questions that monitoring systems should address indicates a possible cause. We hope our monitoring path ultimately leads us to the cause, because that's what needs to be fixed or at least mitigated.

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.

White vs. black box



Best practice monitoring should combine the heavy use of white-box monitoring with modest but critical uses of black-box monitoring.

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.

White-boxed 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, white-box better allows detection of imminent problems, failures masked by retries, and so forth.

Note that in a multi-layer, 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



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



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.

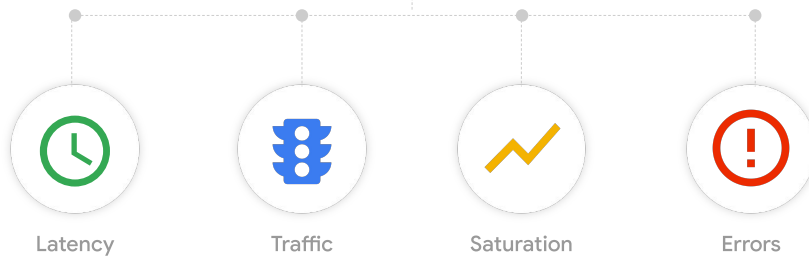
In business, success metrics might include:

- 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

The four golden signals



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.



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 related to **capacity demands**.

May be used to show **improvements**.



Sample metrics

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

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



Important because

Indicates current
system demand.

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

Core to calculating
**infrastructure
spend.**



Traffic measures how many requests are hitting 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



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

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?**



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.

Errors



Important because

Indicates that **something is failing**.

May indicate **configuration or capacity issues**.

Can indicate **SLO violation**.

Time to **alert**?



Sample metrics

- Wrong answer/content
- # 400/500 HTTP codes
- # failed requests
- # exceptions
- # stack traces
- Server fails liveness check
- # dropped connections



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

SLIs, SLOs, SLAs

Choosing a Good SLI

Specifying SLIs

Developing SLOs and SLIs



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**



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



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



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 its **reliability**.



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.

Developers



Agility

Operators



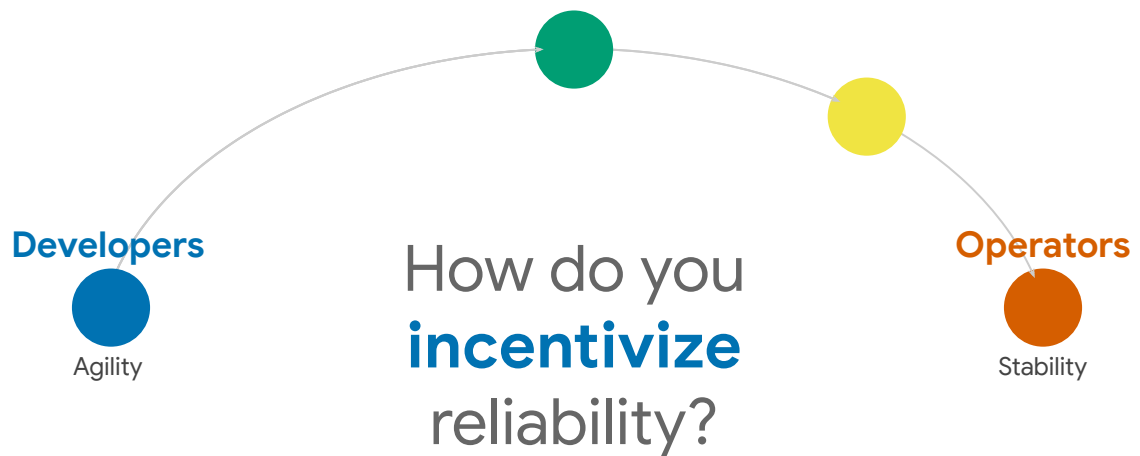
Stability



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



But you can't spend *all* of your engineering time on reliability, because reliability is not a stand-alone 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?

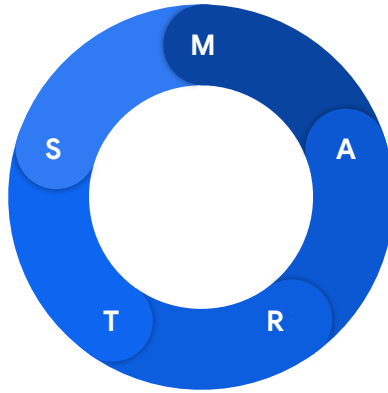
Think about Netflix, Google Search, Gmail, Twitter...
How do you tell whether they are ‘working’?



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?

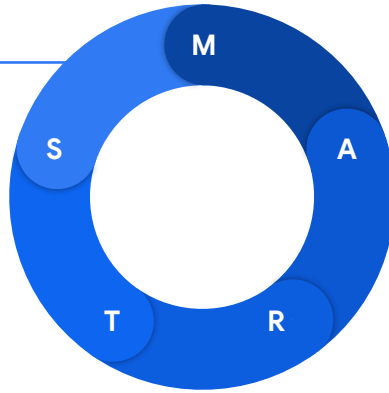
To be effective, SLOs must be SMART



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

To be effective, SLOs must be SMART

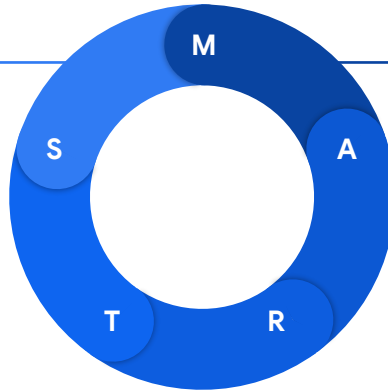
Specific
"Fast" is not as specific as
"Results in 100ms."



SLOs should be **Specific**. "Hey everyone, is the site fast enough for you?" That's not specific; it's subjective. "The 95th percentile of results are returned in under 100ms." That's specific.

To be effective, SLOs must be SMART

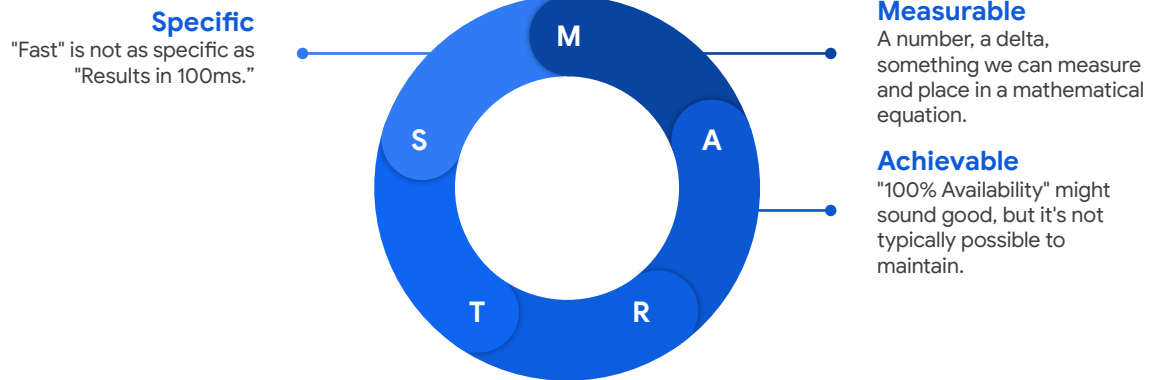
Specific
"Fast" is not as specific as
"Results in 100ms."



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

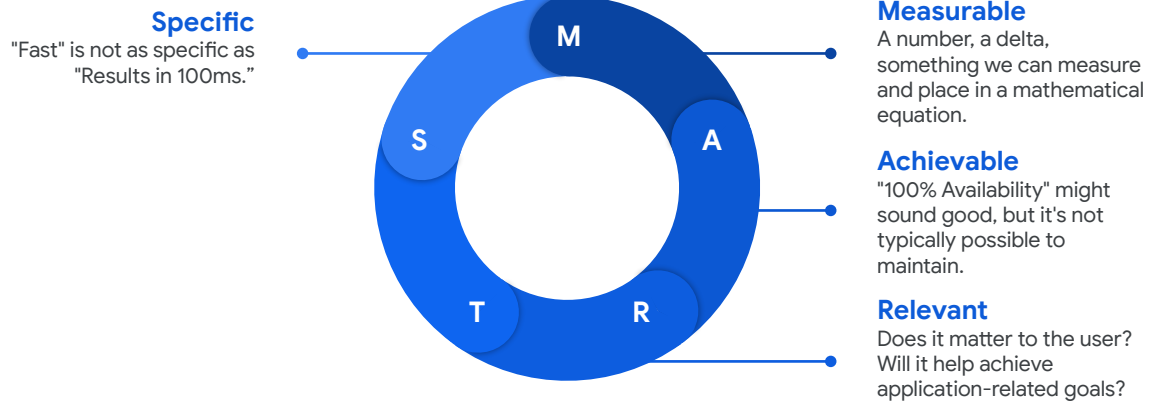
They need to be based on indicators that are **Measurable**. A lot of monitoring is numbers, grouped over time, with math applied. An SLI needs to be a number or a delta, something we can measure and place in a mathematical equation.

To be effective, SLOs must be SMART



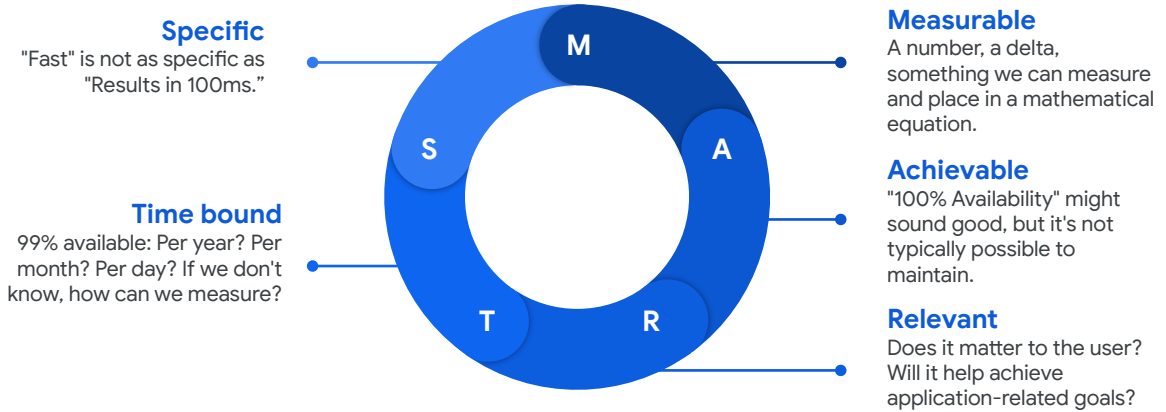
SLOs goals should be **Achievable**. "100% Availability" might sound good, but it's not possible to obtain, let alone maintain, over an extended window of time.

To be effective, SLOs must be SMART

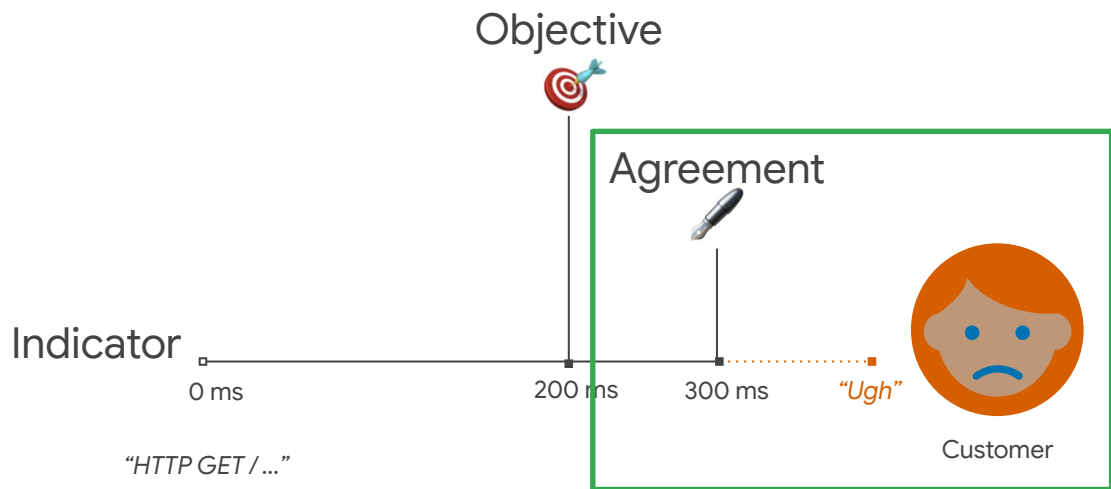


SLOs should be **Relevant**. Does it matter to the user? Will it help achieve application-related goals? If not, then it's a poor metric.

To be effective, SLOs must be SMART



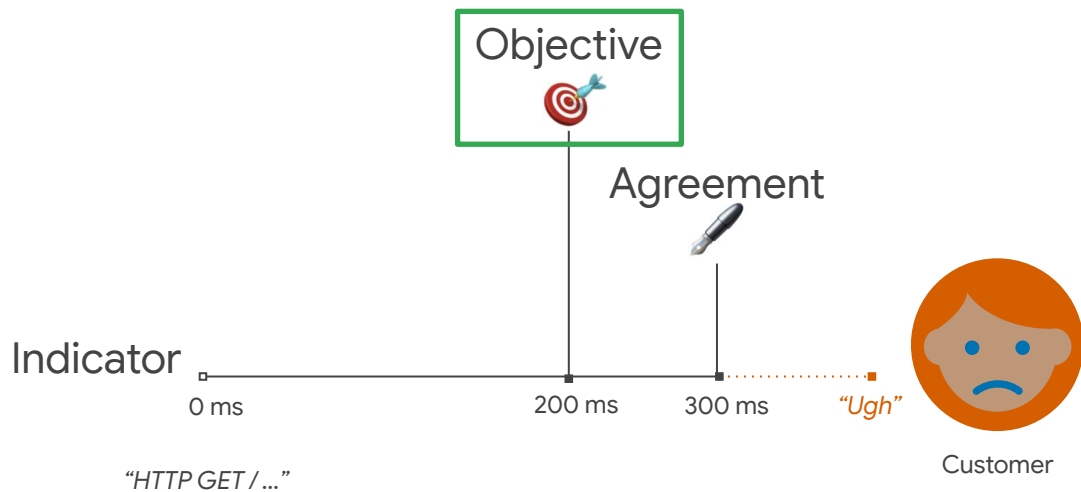
And SLOs should be **Time-bound**. You want a service to be 99% available? That's fine. Is that per year? Per month? Per day? Does the calculation look at specific windows of set time, from Sunday to Sunday for example, or is it a rolling period of the last seven days? If we don't know, how can we measure it accurately?



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.

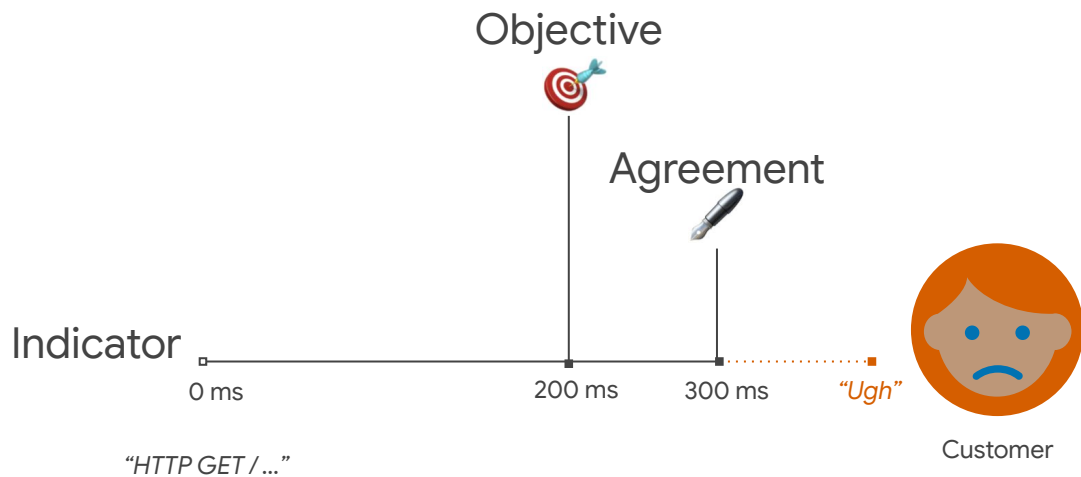


Compensating your customers all the time can get expensive, so what targets do you hold yourself to internally?

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.



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.

How reliable is **reliable enough**?



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%~~

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

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



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 trade-off 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



For example sake, on the chart you can find a given reliability level in the left-most 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 are cases where the total downtime for the given period is less than a single hour.

The smaller chart further examines the 99.95% reliability target over a 28-day window. This time, the 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 rule of thumb? 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”



SLOs should capture the performance and availability levels that, if barely met, would keep the typical customer of a service happy.

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



Typically there will be a point at which customer satisfaction rapidly drops, and complaints sharply rise. That's the point we want to find.

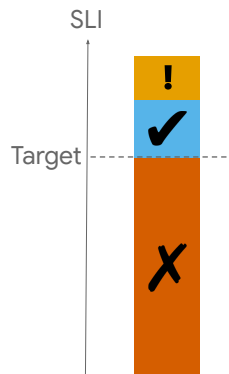
There's no reason to exceed it drastically because once your users are happy with the reliability of your service, additional reliability has little value.

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.

"Workflow", Randall Munroe, XKCD
Source: <https://xkcd.com/1172/>



EVERY CHANGE BREAKS SOMEONE'S WORKFLOW.



...but don't be
too much better,
or users will
depend on it.

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**.*

$100\% - \text{SLO} = \text{Error Budget}$



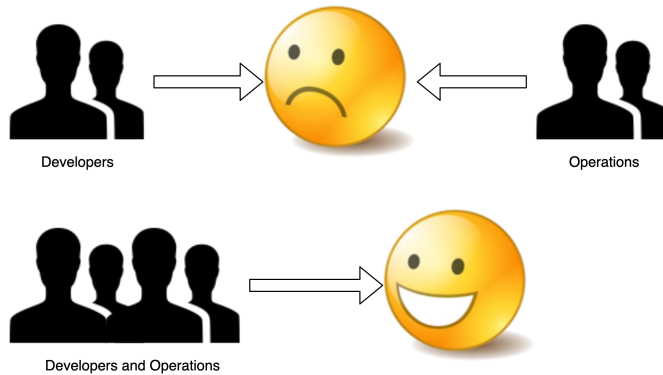
If we're not targeting 100% reliability, then whatever target we do set implicitly allows for a small number of errors to be served to users.

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.

Error budgets and unifying goals



Developers make changes to gain features. Operations folk like stability because it's easier to keep things working. Error budgets provide wiggle room. Developers test new features and the error budget provides them room for failure. Ops no longer has to choose uptime over improvements. Everyone wins.

Lecture notes:

Error budget allows developers to try new things. If developer team is planning to try a new feature but not sure how it impacts the reliability - look at whether you have the error budget to support it, or should you wait for the next window of time to check whether we have enough error budget to support taking that risk?

Implementation mechanics

Evaluate SLO **performance** over a set **window**, e.g., 28 days.
Remaining budget (100%-SLO) **drives prioritization**
of engineering effort.



Goal: Explain the role of measurement windows in turning SLOs into error budgets.

Points:

- Converting SLO target → error budget requires a time window
- Calendar vs. rolling windows
- Recommend 28 days, exactly 4 weeks ⇒ no variation from weekly release cycles or weekends

Words:

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 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?

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!

Once you've decided on the time window, you 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.

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**



Goal: Enumerate various ways to spend error budgets.

Words:

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, though if the downtime is communicated enough in advance, your users should be expecting it and, therefore, be 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

Why Monitor?

Critical Measures

Four Golden Signals

SLIs, SLOs, SLAs

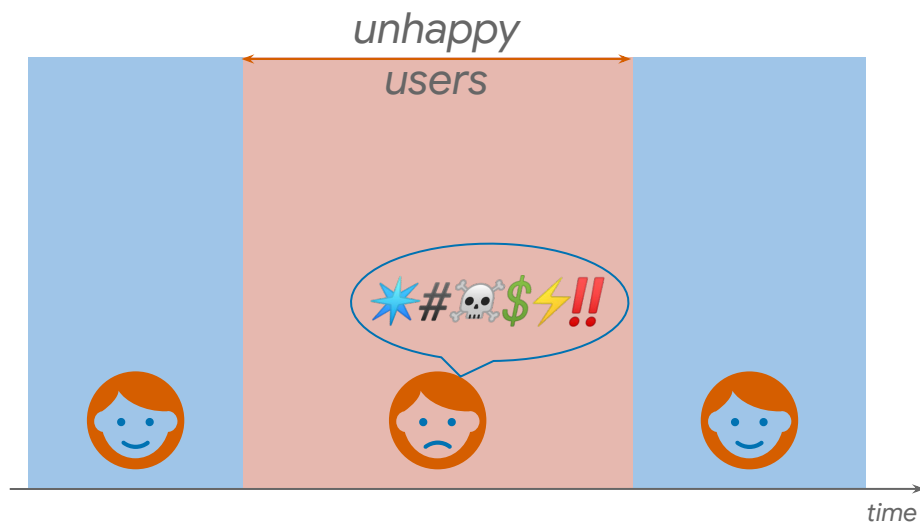
Choosing a Good SLI

Specifying SLIs

Developing SLOs and SLIs



So how exactly do you choose a good SLI?



Goal: Distinguish bad vs. good metrics.

Points:

- Red area \Rightarrow increased support requests \Rightarrow more unhappy users
- Maybe some already existing metrics correlate with outage?
- Bad ideas: load average, CPU or memory utilization, thread-pool fullness, queue length

Words:

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.

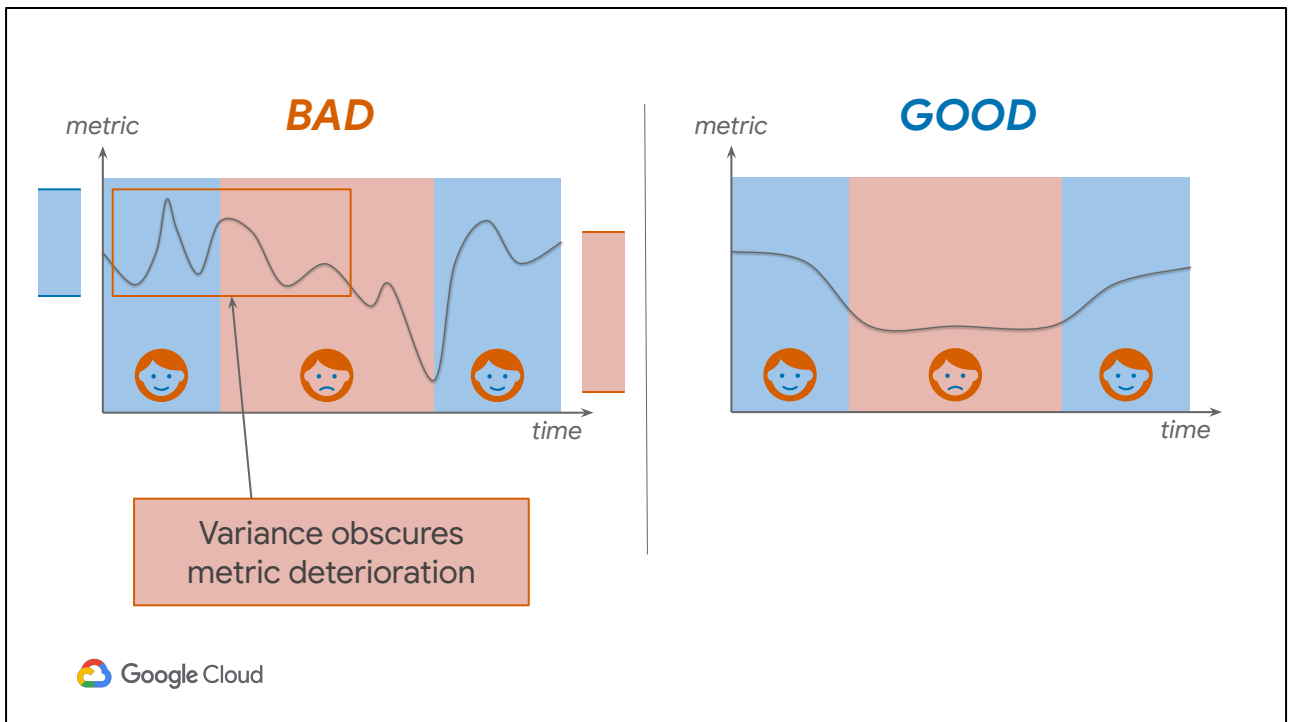


So, what properties of a metric make it good for use as an SLI?

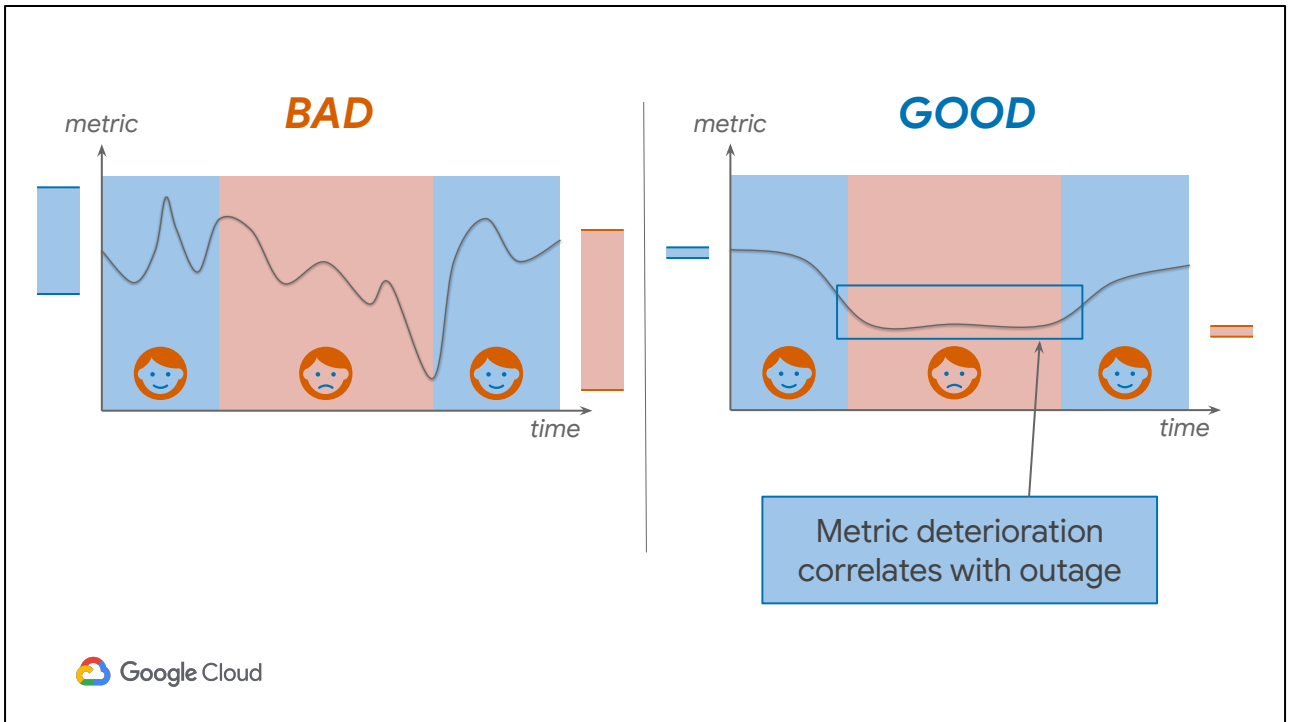
Let's assume we've trawled through our metrics and come up with these two candidates. If we had to choose between them, why would we say that the metric on the left is bad for use as an SLI, while the metric on the right is better?

Lecture Notes:

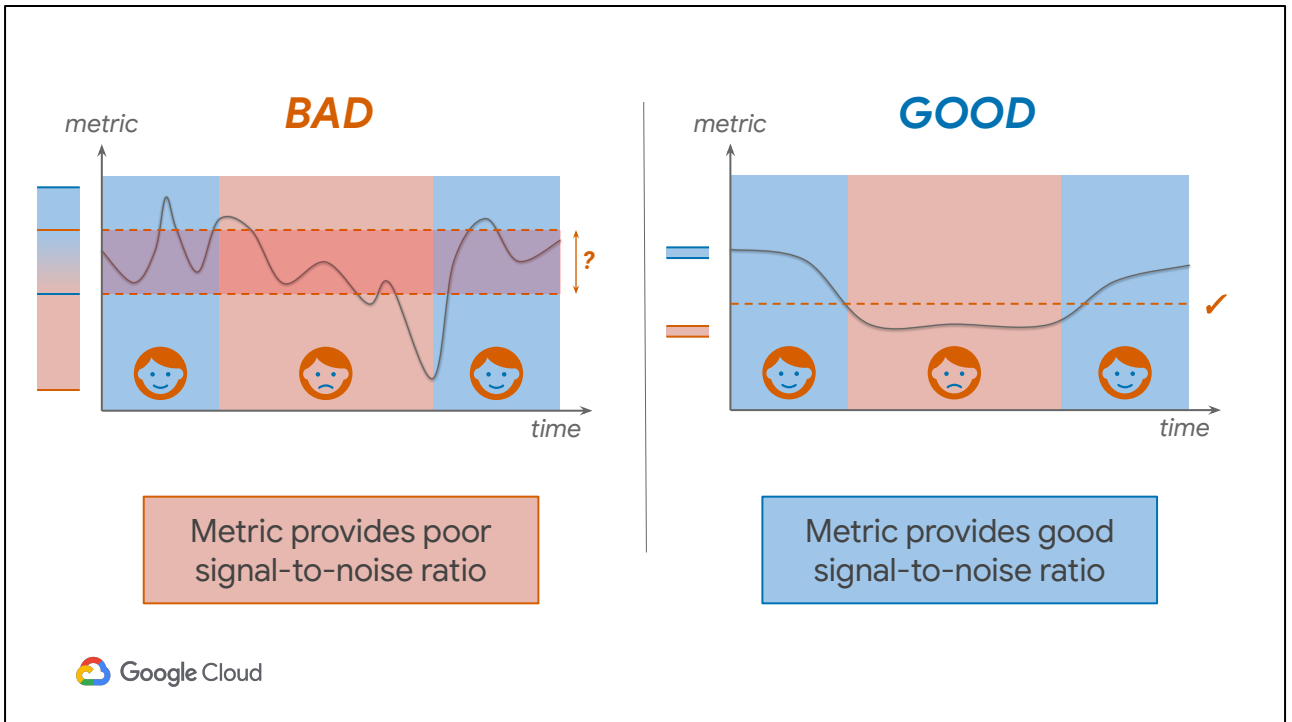
You want to identify only the metrics which has direct correlation with user experience. There are a lot of metrics available - but you have to choose your metrics to monitor very carefully and those metrics should have a direct correlation to your use-case/user experience. In the "BAD" example, it has a really bad signal to noise ratio.



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.

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



Goal: Express SLIs like this!

Points:

- SLI equation has useful properties
 - Intuitive
 - Translates to percentage reliability SLO targets
 - Consistent format makes building tooling easy
- The term "valid" implies that known bad events are excluded from SLI

Words:

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.

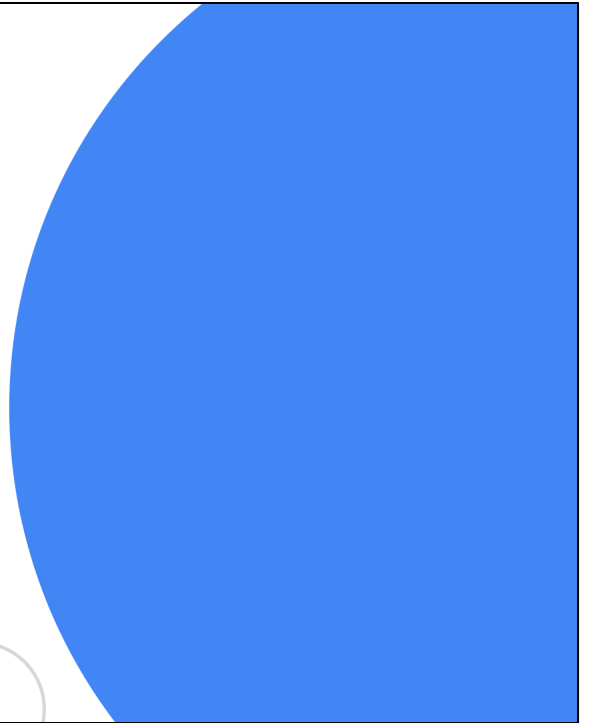
But what do we mean by "valid" and why don't we use "all events" in the SLI equation? Sometimes you may need to exclude some events recorded by your underlying metrics from being included in your SLI, so they don't consume your error budget. A good example here might be ignoring 300 and 400 HTTP response 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".

Lecture Notes:

One example where customer considered HTTP 404 as invalid event, i.e. if the user tries a URL that does not exists, it will not be considered a valid request. However, what happens if there is a change that has gone in that has a broken link. Now, since you are no longer tracking 404, you will not know it at all.

Lab Intro

Postmortem Review



Agenda

Why Monitor?

Critical Measures

Four Golden Signals

SLIs, SLOs, SLAs

Choosing a Good SLI

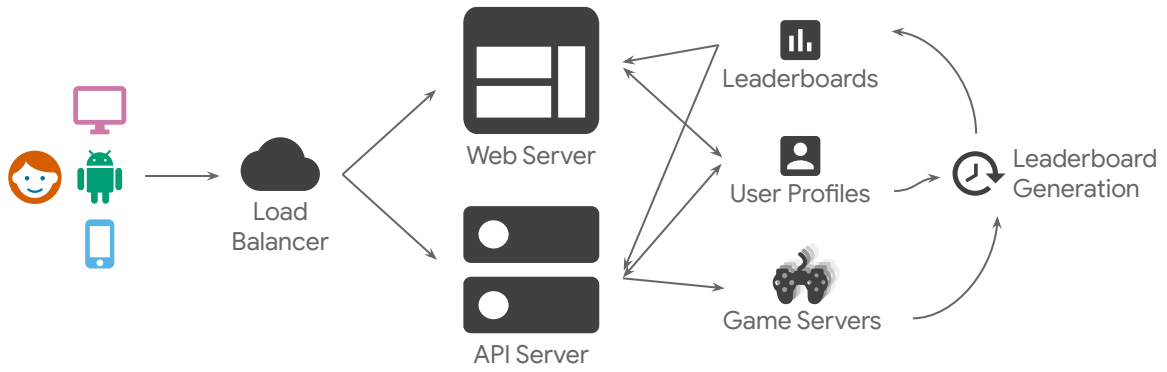
[Specifying SLIs](#)

Developing SLOs and SLIs



Now let's talk about specifying SLIs.

Our game: Fang Faction



 Google Cloud

Goal: Introduce the game.

Points:

- Game background and technical details:
 - Two data stores, user profiles, and leaderboards
 - Pool of game servers with their own databases
 - API servers and web servers behind L7 load balancer
 - JSON-RPC, REST, web socket connection for game-state updates

Words:

Each of those goals involves one or more interactions with our game's serving infrastructure.

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.


Notes for presenter:

This example and a lot of the accompanying advice comes from the [chapter on SLOs](#) from the Site Reliability Workbook. There's a lot of good background reading there!

We will return to this diagram a couple of times in the deck. It is strongly suggested that you see these as natural pause points, where you ask the audience questions instead of talking to them.

Loading a profile page

https://fangfactiongame.com/profile/someuser



SomeUser's Profile

Faction Name:

Leader Name:

Email Address:

SomeUser
Tribe of Frog
Faction Score: **31337**
[Midwest Canyon](#)

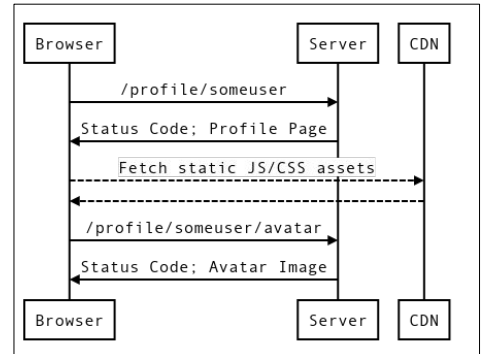
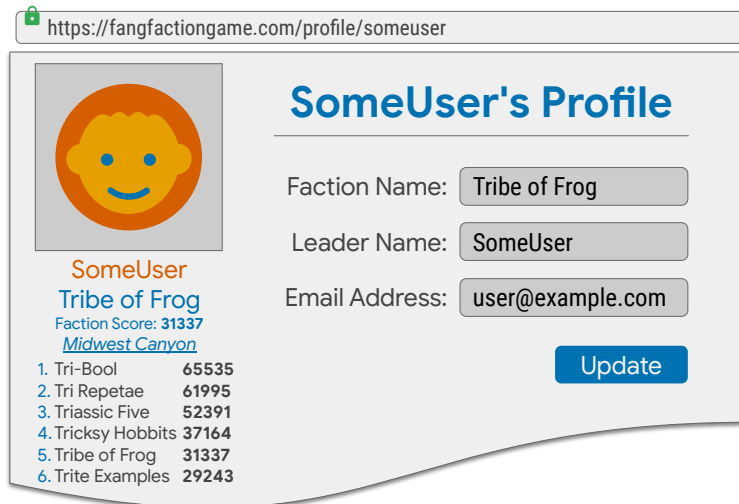
1. Tri-Bool	65535
2. Tri Repetae	61995
3. Triassic Five	52391
4. Tricky Hobbits	37164
5. Tribe of Frog	31337
6. Trite Examples	29243



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

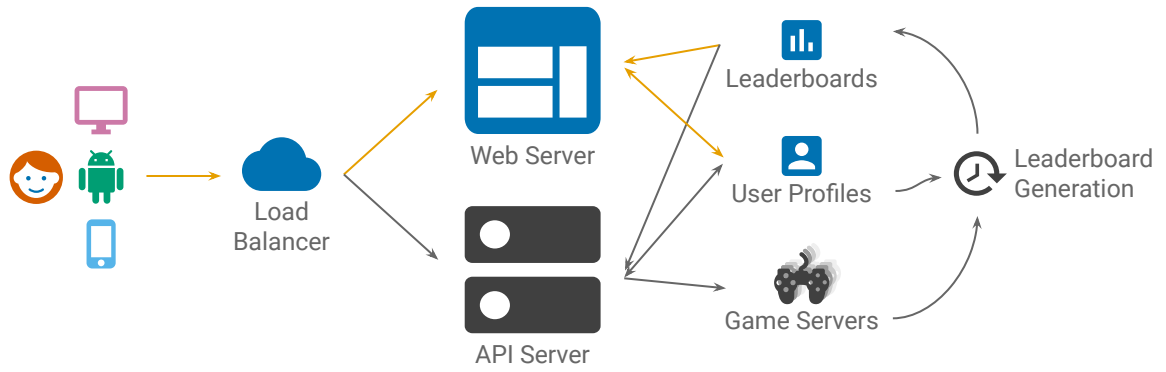
I'm sure you can imagine a number of user journeys for this game. We're going to keep things simple for now, so we can step through the process without too many complicating factors, but you'll find some more complex journeys in your handout.

Loading a profile page



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

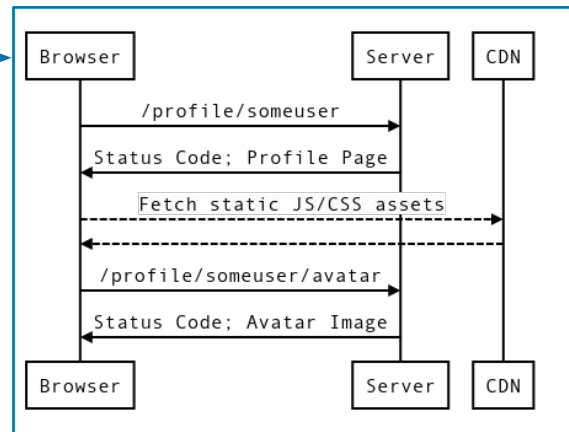


Here's how our infrastructure serves a profile page.

- 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



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. For instance, "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.

For comparison, a list of user journeys in Gmail may include:

- Reading an email and opening an attachment.
- Composing and sending an email.
- Organizing a message by applying tags.

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 learn how to identify these metrics and use them to ensure your user experience remains within acceptable bounds.

3–5 SLIs*

*per user journey



Goal: 3-5 SLIs per user journey. (Obvious but included for consistency.)

Points:

- SLIs tell you *something* is wrong—you need other monitoring systems to tell you *what*.

Words:

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. We're going to call a set of interactions to achieve a single goal a "user journey"—a term we've borrowed from the field of user experience research. You may also have heard people talk about CUIs, or Critical User Interactions. 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 you ditch all your other monitoring and observability systems once you've started measuring SLIs. The way we think about it is that a deterioration in your SLIs is an indication that something is wrong. Once 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



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. You can find more information in the SLI/SLO handout linked at the end of this module.

Agenda

Why Monitor?

Critical Measures

Four Golden Signals

SLIs, SLOs, SLAs

Choosing a Good SLI

Specifying SLIs

Developing SLOs and SLIs



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?



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.

Lecture Notes:

In a lot of exam questions, the business need specified in the question decides which is the right answer. Identify them when you are answering those



User expectations
are *strongly* tied to
past performance.



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.

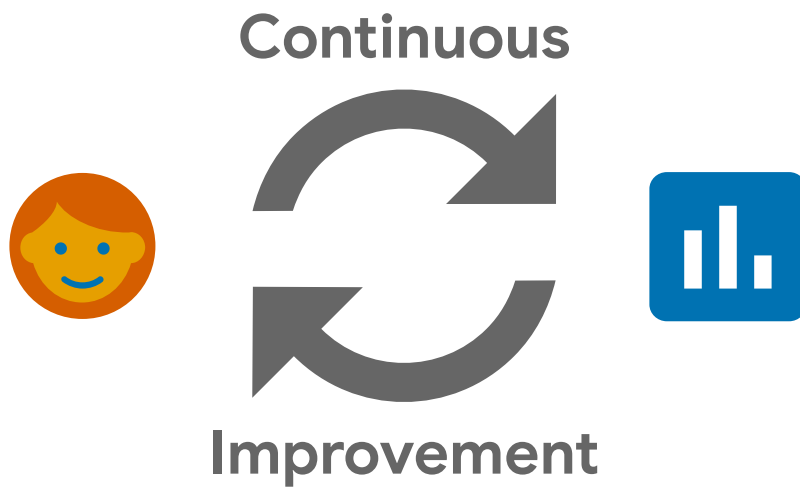
Continuous



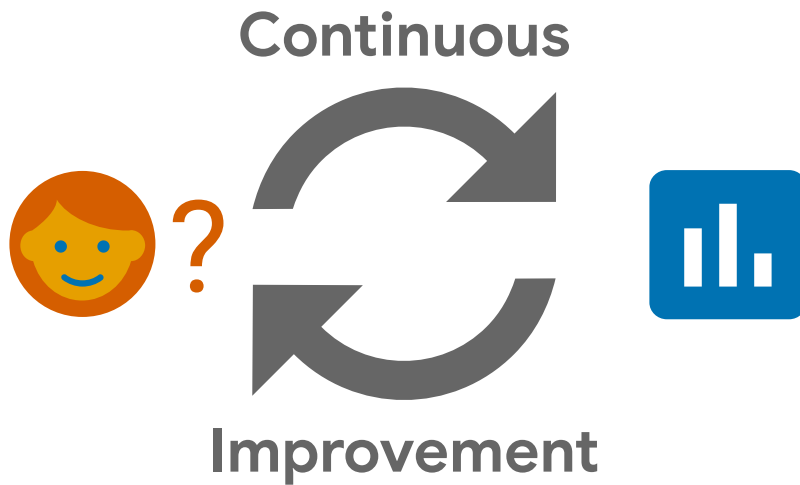
Improvement



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**.

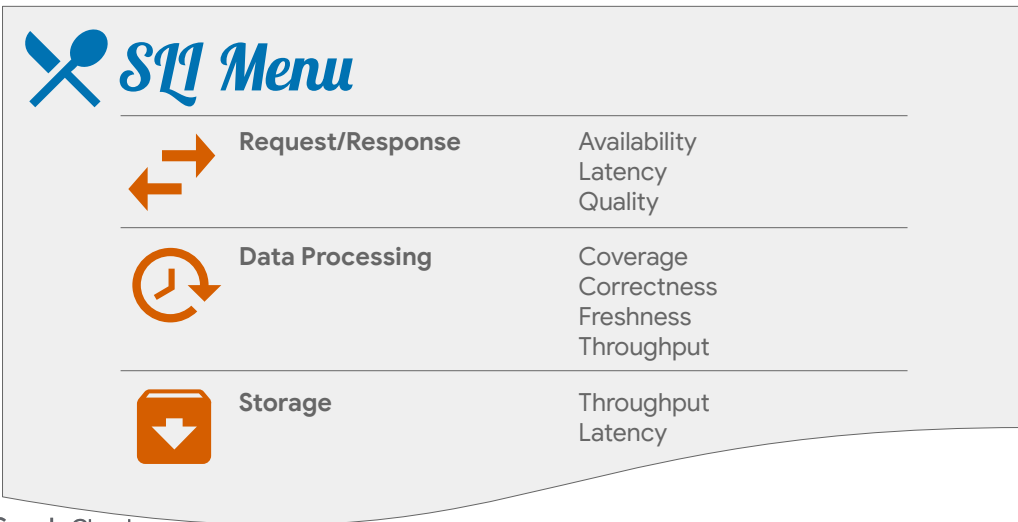


In general, people use your service to achieve some set of goals, so the SLIs for that service must measure the interactions they have with the service in pursuit of those goals.

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.”

1 Choose an SLI specification from the menu



 Google Cloud

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**.

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 SLO 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.

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

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

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.

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

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 **within X ms**



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

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**

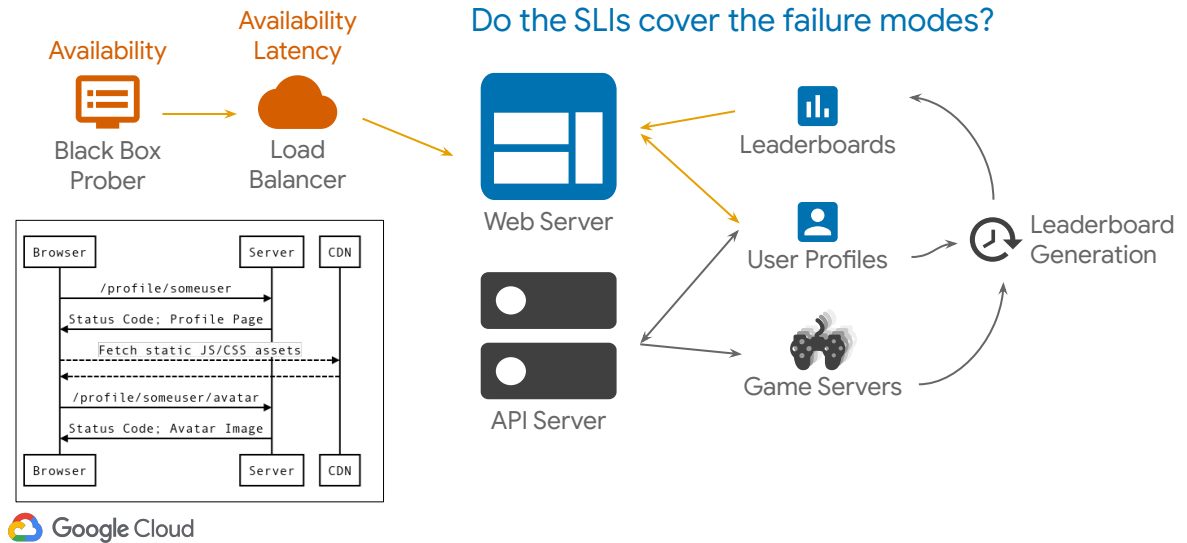


In this case, the company decided that the metrics provided by their load balancer were good enough to establish some initial SLIs without investing too much engineering effort.

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



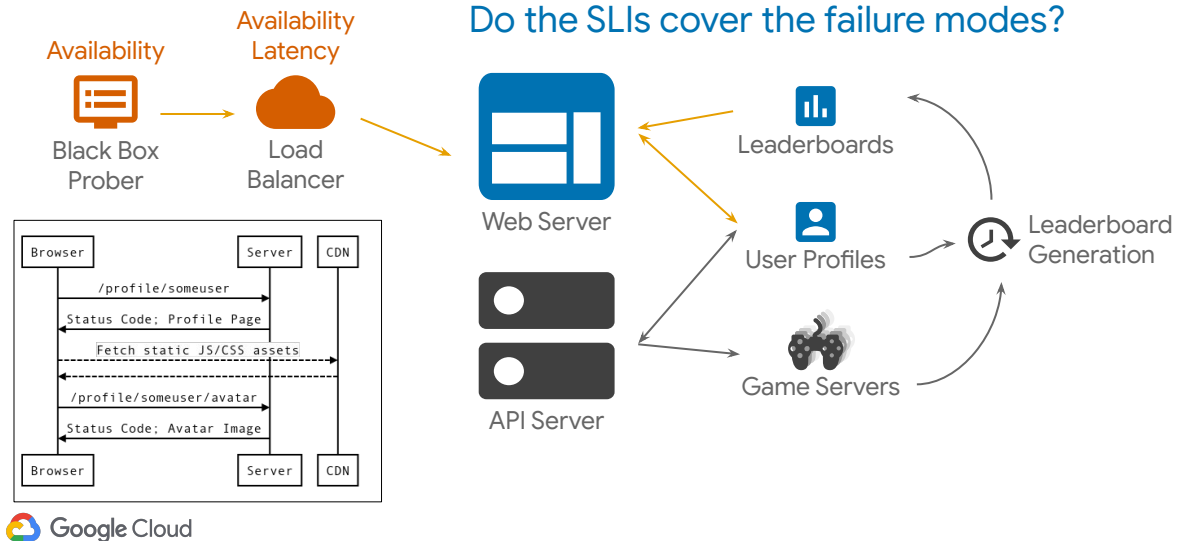
This example outage illustrates why you should take a critical look at your infrastructure when creating your SLIs.

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

Do the SLIs cover the failure modes?



We say "adequately capture" because reaching 100% coverage of all possible failure modes is again the wrong target.

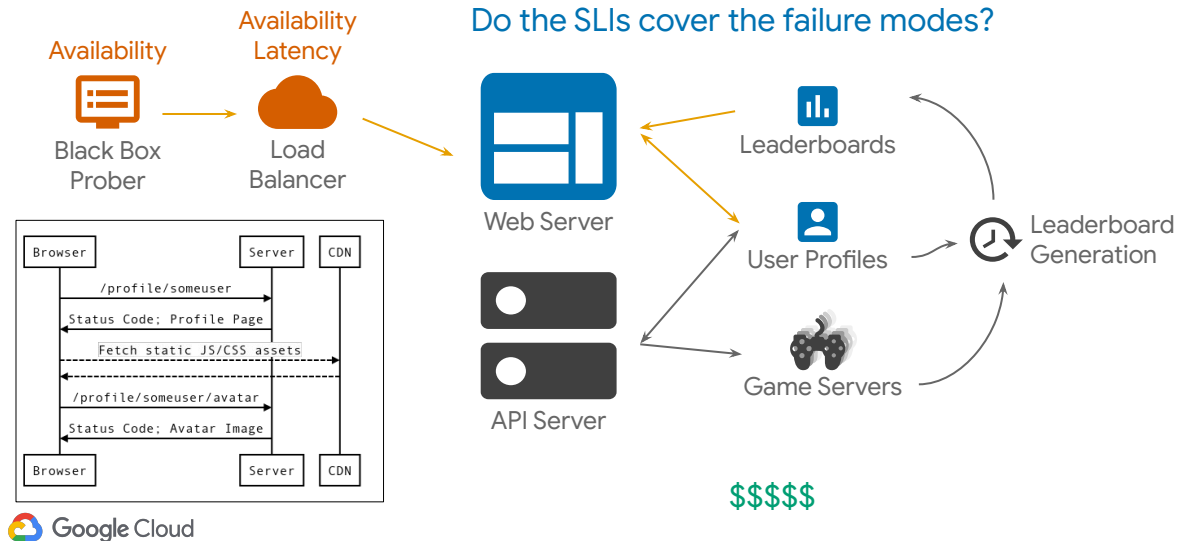
Distributed systems fail in complex and surprising ways, so 100% is an unrealistic goal.

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**.

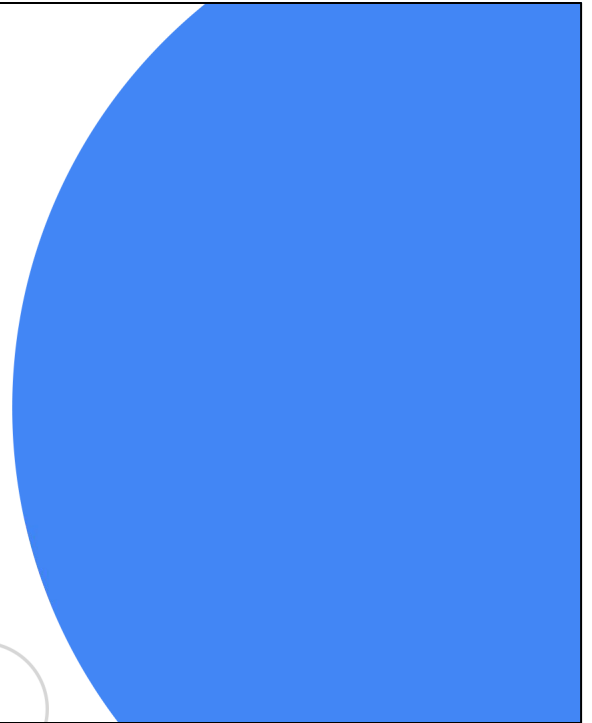
Service	SLO Type	Objective
Web: User Profile	Availability	99.95% successful in previous 28d
Web: User Profile	Latency	90% of requests < 500ms in previous 28d
...	...	



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

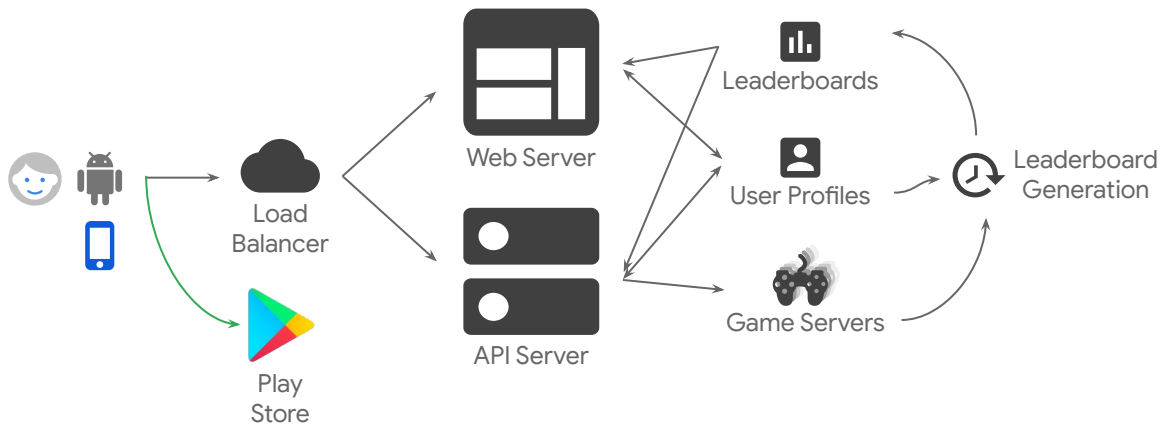
Lab Intro

Develop SLIs and SLOs



In this exercise, you 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.

Our game: Fang Faction



 Google Cloud

Instructors: Leave this slide up while people work, for reference.

Buy In-Game Currency

Model Answer



Instructors: This part of the deck switches slides at a faster pace, it should only take 10-15 minutes to get through it. It's a cut-down version of the Train-the-Trainers deck. You can also leave it for student self-study if you so desire.

Words:

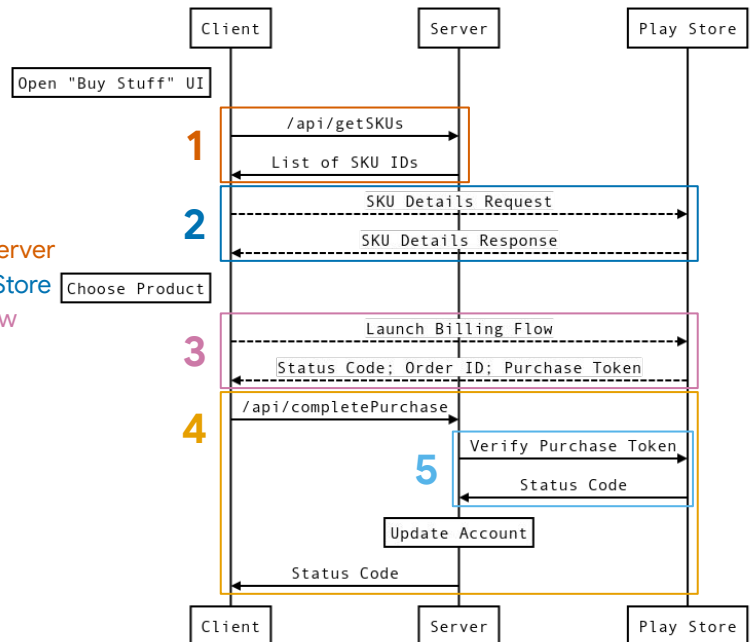
We're now going to go through one possible set of SLIs and SLOs for the Buy In-Game Currency user journey.

I want to stress that this isn't the only answer! I expect you all will have come up with something different before the break. This is just to provide a frame of reference and show how the process we went through in the slides can be applied to more complex journeys too.

Break down the Journey

Five request/response pairs

1. Fetch list of SKUs from API server
2. Fetch SKU details from Play Store
3. User launches Play billing flow
4. Send token to API server
5. Verify token with Play Store

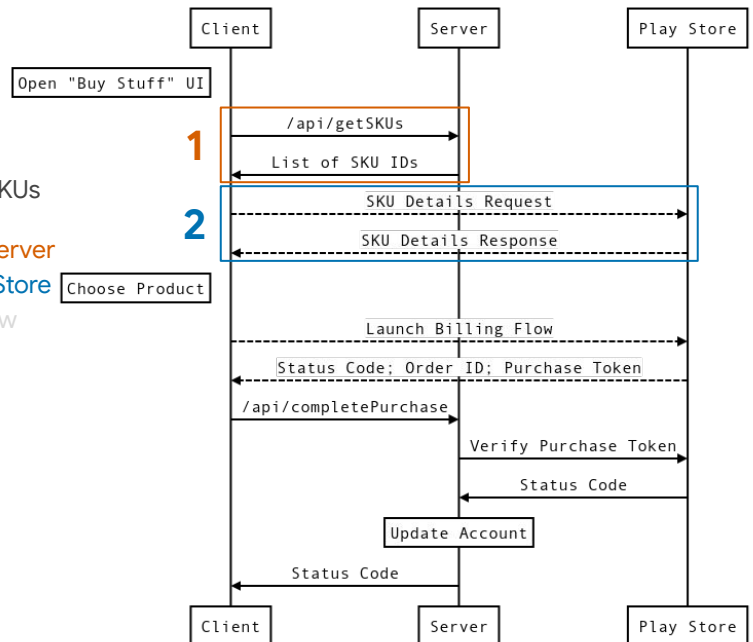


Let's take a look at the journey, so we can figure out what SLIs we're going to want for it. There are five request-response pairs involved in someone buying in-game currency.

Break down the Journey

Journey has **two** parts. **A:** Fetch SKUs

1. Fetch list of SKUs from API server
2. Fetch SKU details from Play Store
3. User launches Play billing flow
4. Send token to API server
5. Verify token with Play Store



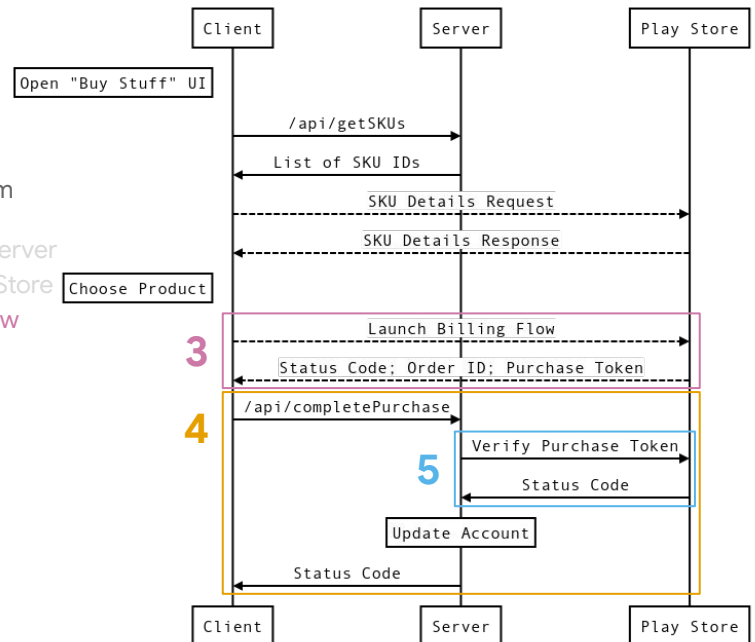
But they're not all equivalent—there are two distinct parts to the journey. First, when the user opens up the in-game "store", the game client has to ask our servers what items are available to purchase, then it has to retrieve details like pricing information from the Play store. Both of these things have to happen successfully for the client to be able to display the store UI to the user.

Note: SKUs are [stock keeping units](#).

Break down the Journey

Journey has **two** parts. **B**: Buy Item

1. Fetch list of SKUs from API server
2. Fetch SKU details from Play Store
3. User launches Play billing flow
4. Send token to API server
5. Verify token with Play Store



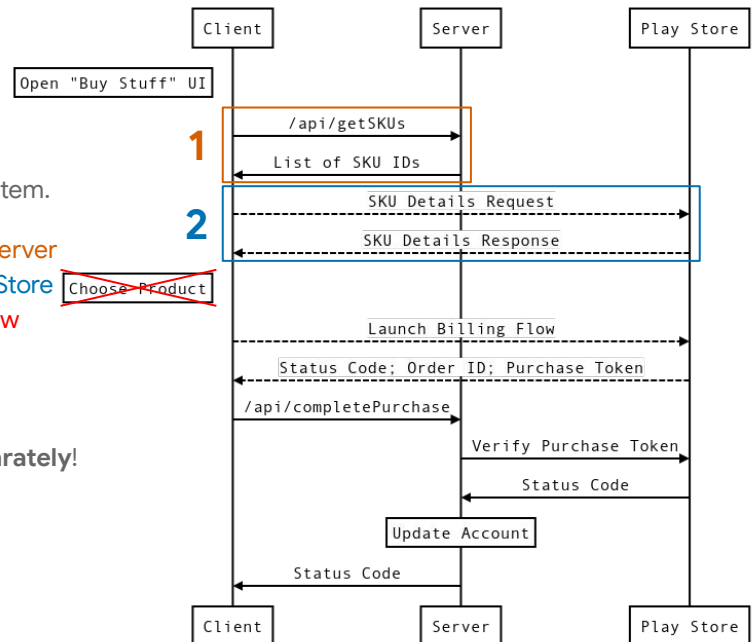
The second part of the journey occurs after the user chooses a particular item to buy. This triggers the Play Store's ["billing flow"](#), which takes control of the device away from the app and gives it to the Play Services framework. The Play Services framework handles things like taking credit card details and talking to the Play Store to charge the user for the item they've chosen. The billing flow returns a [status code](#), and if the purchase was successful, there will also be an order ID and purchase token. Our servers need to know about successful purchases so we can grant the item the user has chosen to them. Therefore, the device sends the order ID and purchase token to our servers via the completePurchase API call. We have to validate the purchase token with the Play Store before giving the user their item, to prevent double-spend attacks.

Break down the Journey

User might choose **not** to buy an item.

1. Fetch list of SKUs from API server
2. Fetch SKU details from Play Store
3. User launches Play billing flow
4. Send token to API server
5. Verify token with Play Store

We have to treat these parts **separately!**



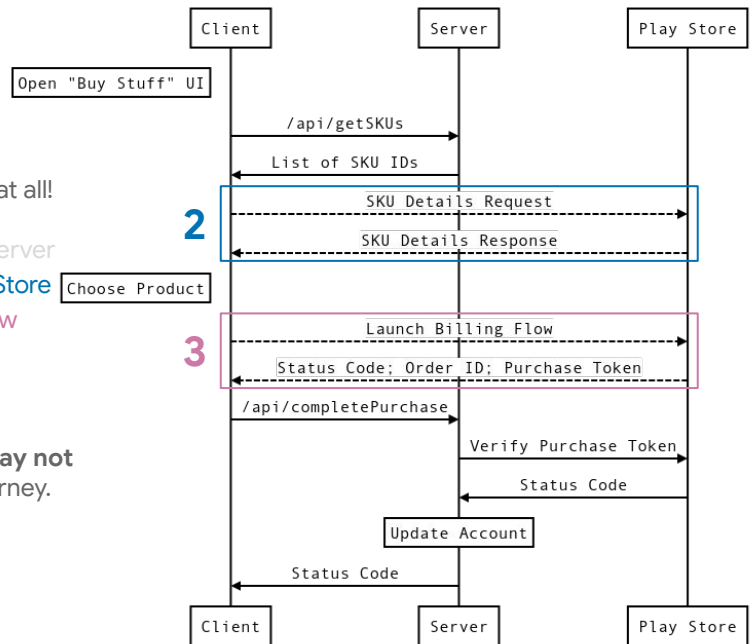
Breaking the journey down like this is important, because we are not going to get anything like a 100% conversion rate when people open up our in-game store. Anecdotally, numbers in the 1-2% range are much more realistic. If we were to create SLOs for the entire journey as a whole, the difference in traffic between the two parts would cause any signal from the second part of the journey to be swamped by the first. And arguably the second part is more important because it's the source of all our company's revenue.

Break down the Journey

Two requests don't hit API server at all!

1. Fetch list of SKUs from API server
2. Fetch SKU details from Play Store
3. User launches Play billing flow
4. Send token to API server
5. Verify token with Play Store

Server or load balancer metrics **may not give enough coverage** of the journey.



Another thing to consider is that there are two request-response interactions that occur between the user's device and the Play Store. Our serving infrastructure will not see these requests, so we will not be able to measure them except via client-side telemetry, or at a pinch via a synthetic client.

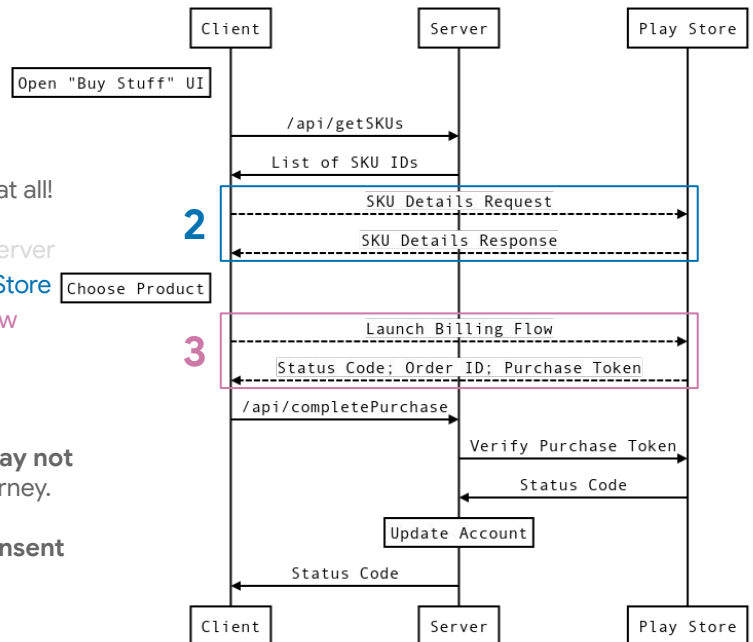
Break down the Journey

Two requests don't hit API server at all!

1. Fetch list of SKUs from API server
2. Fetch SKU details from Play Store
3. User launches Play billing flow
4. Send token to API server
5. Verify token with Play Store

Server or load balancer metrics **may not give enough coverage** of the journey.

... we'll have to ask our users to **consent** to client-side telemetry

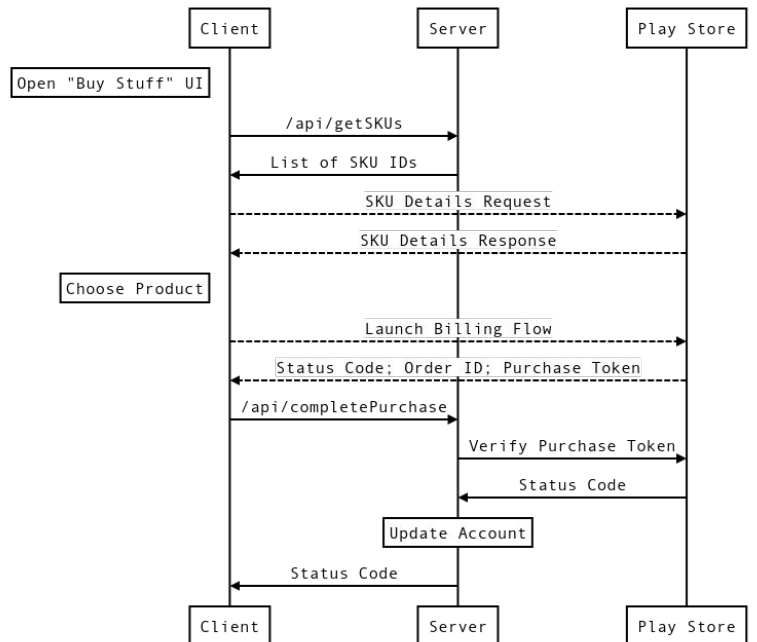


And if we're going to have our game client report statistics back to us, it's very important that we get the consent of our users beforehand. This is commonly done with a one-time pop-up dialog asking users whether they want to report crashes and other statistics back to our company for the purposes of product improvement, along with a description of the sort of data being collected. Contact your friendly neighborhood lawyer for more information!

Buy Flow What SLIs?

Buy Flow journey is:
Request/Response

SLI menu suggests we use:
Availability and Latency SLIs



Now that we've considered the sequence of interactions that make up the user journey, we can decide on the SLIs we want to measure for it. The SLI menu suggests Availability and Latency SLIs, because all of our interactions are HTTP requests.

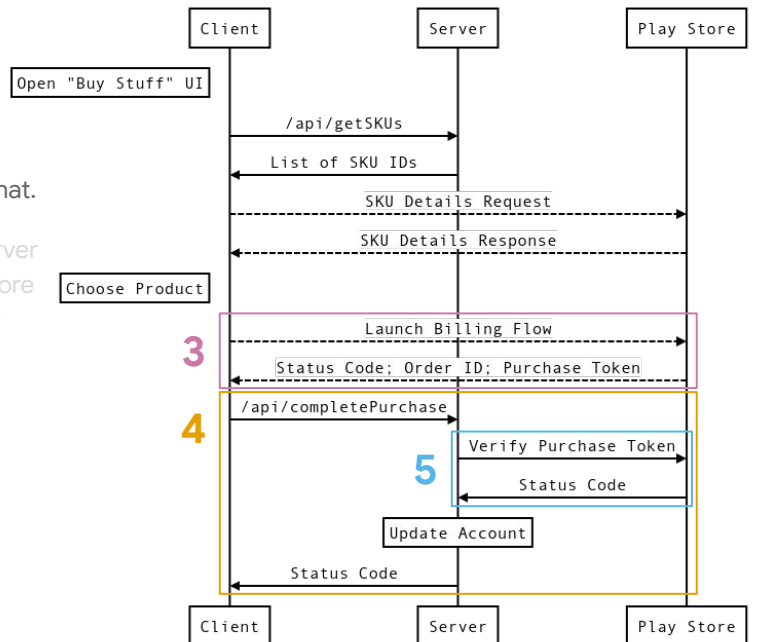
Buy Flow Availability: Specification

B makes money, so let's start with that.

1. Fetch list of SKUs from API server
2. Fetch SKU details from Play Store
3. User launches Play billing flow
4. Send token to API server
5. Verify token with Play Store

Availability SLI Specification

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



We'll want availability and latency SLIs for both parts of the journey, but because the second part is where we actually take money off our users and give them things, that's what we'll focus on first. Just like the user profile journey, we can start from the generic availability SLI specification, which is simply "the proportion of valid requests served successfully". Figuring out good definitions for "valid" and "successful" will turn this into something that we could implement in a monitoring system.

Buy Flow Availability: Valid Requests

Availability SLI

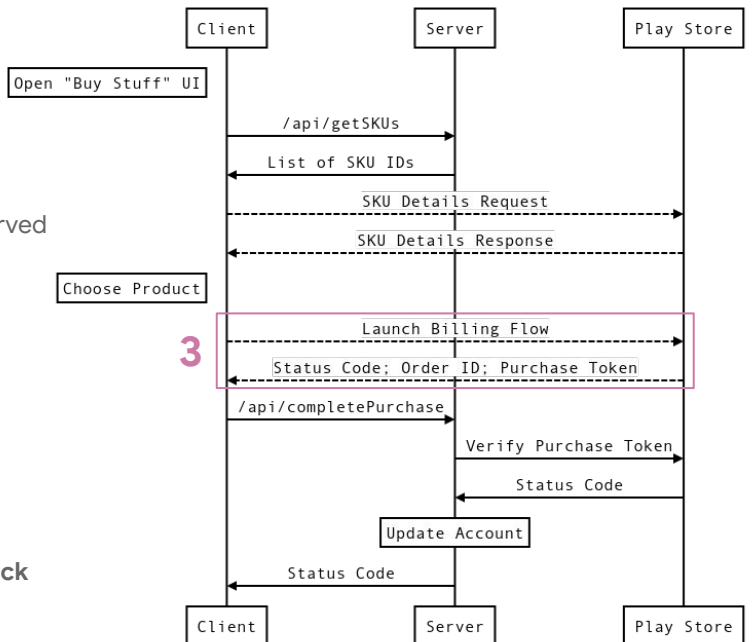
The proportion of **valid** requests served **successfully**.

... but which requests are **valid**?

3. User launches Play billing flow
4. Send token to API server?
5. Verify token with Play Store?

Launching the billing flow indicates a user's **intent** to buy a product.

Users **consenting** to client-side telemetry collection allows us to **track** this intent.



When deciding which requests to include or exclude from our SLI, it's important to think about the user and their expectations. In this case, when the user clicks one of our products, they are expressing a clear intent to buy that product, and they expect that we will sell it to them. If they get an error instead, they will not be happy and we will have forfeited some potential revenue. The first useful measurable event that occurs once that button has been pressed is the launch of the Play Store's billing flow, so we'll consider any launches of this flow to be requests we should include into our SLI. We can only track this on our users devices, so we will have to ask for the user's permission to track and record this data.

Buy Flow Availability: Success Criteria

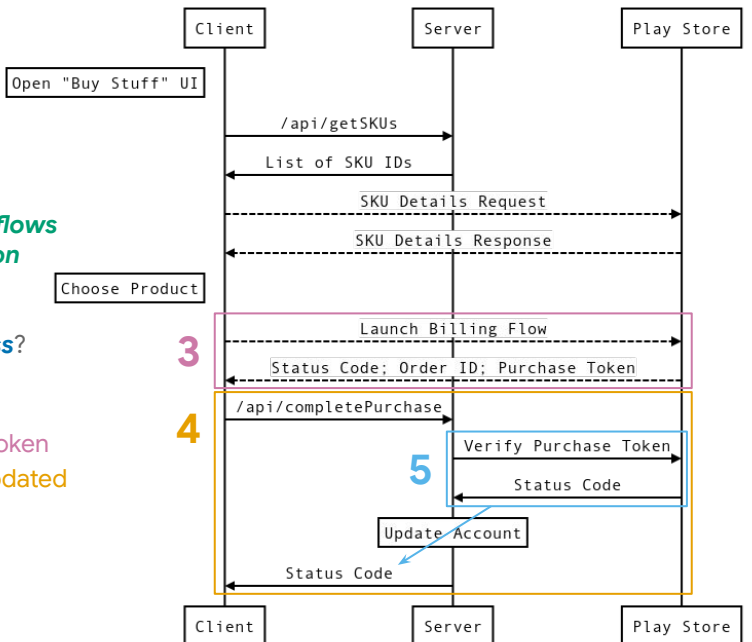
Availability SLI

The proportion of **launched billing flows from users consenting to collection** served **successfully**.

... and how do we determine **success**?

All interactions must be successful!

3. Good status code; purchase token
4. Good status code; account updated
5. Good status code; valid token
 - o Return 402 to API call when token is invalid



Oh dear. Our SLI is already quite a mouthful when we substitute in that definition of validity, and we still need to define "successfully" in this context. For the user to get the product they intended to purchase, all of the interactions in the journey must complete successfully. Success for the billing flow means that we get a "good" status code back from the Play Store and a purchase token if the purchase was successful. Similarly, for the completePurchase call, the server needs to respond with a "good" status code, and it needs to update the user's account to give them the product they purchased.

The interaction between our servers and the Play Store to validate the purchase token isn't visible from the client. It's useful to measure both valid requests and successful requests in the same place so that it's harder for discrepancies to creep in. We'll report failures in token validation back to the client with specific HTTP status codes to bridge this gap. Specifically, when the Play Store tells us the purchase token is invalid, we'll send a 402 "Payment Required" code back to the client. This won't be counted as a failure by the SLI—chances are that invalid tokens are people attempting replay attacks—but the client will be able to display a meaningful error to the user.

Buy Flow Availability: Measurement

Availability SLI

The proportion of **launched billing flows from users consenting to collection.**

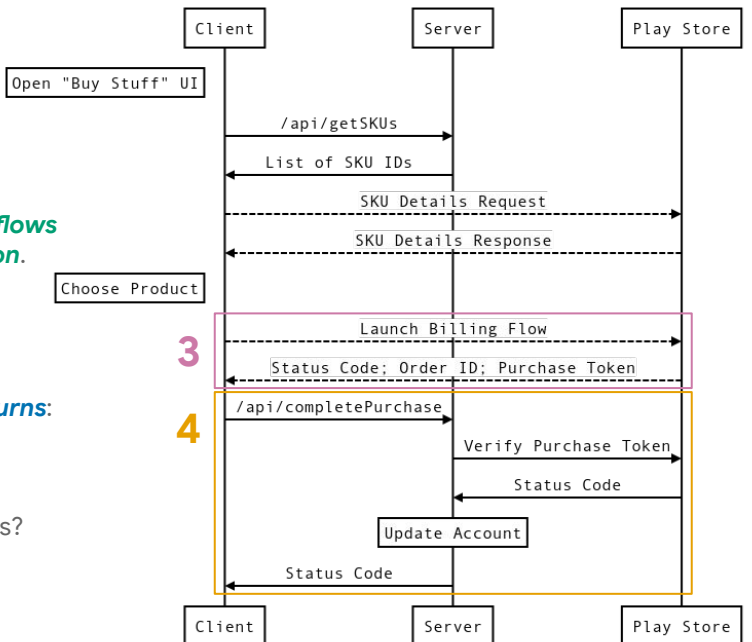
where **the billing flow returns:**

- OK and a purchase token
- or FEATURE_NOT_SUPPORTED
- or ITEM_UNAVAILABLE
- or USER_CANCELED

and **/api/completePurchase returns:**

- 200 OK and Parseable JSON
- or 402 Payment Required

... but where are we **measuring** this?



When we substitute in our definition of success, the SLI is almost complete, we just need to describe where its components will be measured. As we noted earlier, it's important to be specific and detailed about exact status codes. The constants in CAPITAL_LETTERS are response codes from the Play Store's [documentation](#). Hopefully "OK" is obvious, but the others are worth a brief explanation.

FEATURE_NOT_SUPPORTED means that the user's device is running a version of Android or the Play Store app that is too old. The server-side parts of the Play Store are no longer able to support purchases from the device.

ITEM_UNAVAILABLE is basically a race condition: between the user listing the SKU details from the Play Store and trying to buy a particular product, we took down the product from the Play Store so it was no longer available to purchase.

USER_CANCELED is the user giving up and no longer wanting to buy the product in question. All of the various permutations of credit card expiry or other payment failures roll up into this, because the Play Store's billing flow will simply keep prompting users for a valid payment method until either it manages to charge them for the product or they give up and cancel the purchase.

Buy Flow Availability: Measurement

Availability SLI

The proportion of **launched billing flows from users consenting to collection.**

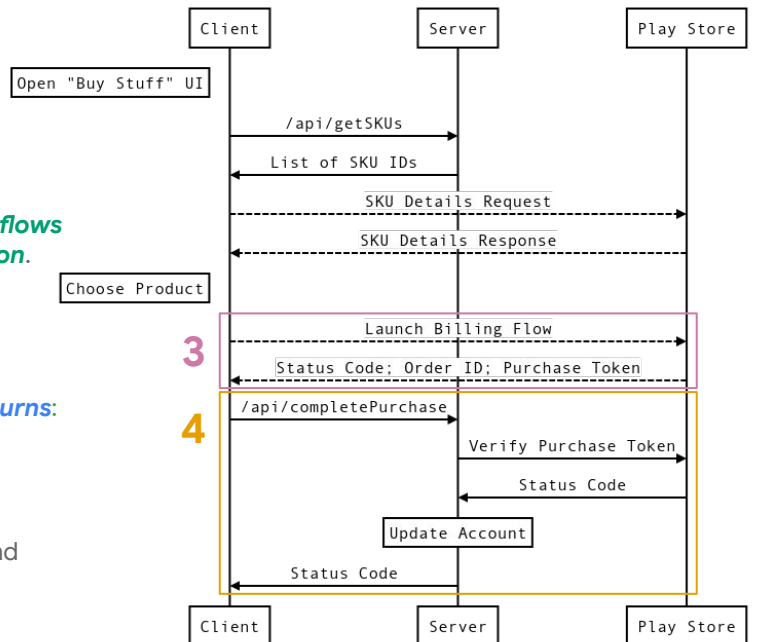
where **the billing flow returns:**

- OK
- or FEATURE_NOT_SUPPORTED
- or ITEM_UNAVAILABLE
- or USER_CANCELED

and **/api/completePurchase returns:**

- 200 OK
- or 402 Payment Required
- and Parseable JSON

Measured by the **game client** and reported back asynchronously.



The constraints we've already discussed mean we have to measure this SLI on the client. Adding this on the end is a formality, but an important one, because all SLI implementations should be clear about where and how they are measured.

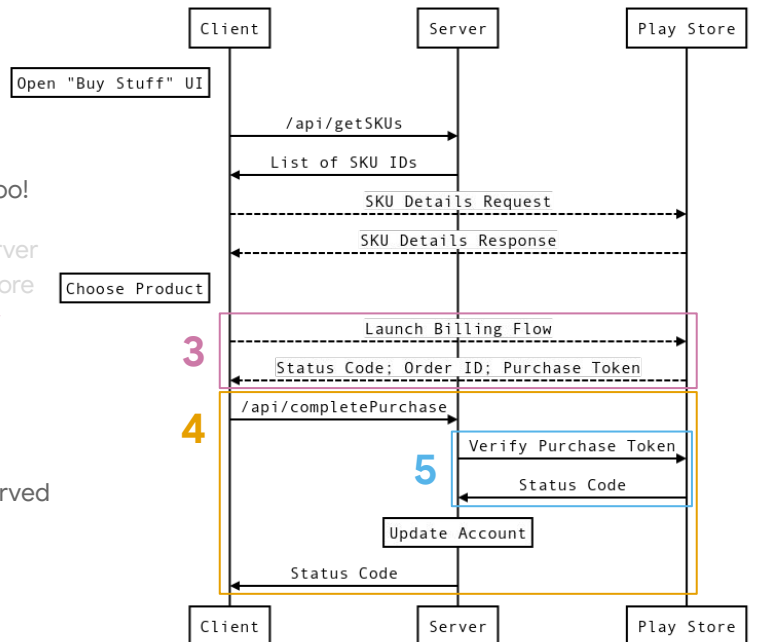
Buy Flow Latency: Specification

We want to measure latency for **B** too!

1. Fetch list of SKUs from API server
2. Fetch SKU details from Play Store
3. User launches Play billing flow
4. Send token to API server
5. Verify token with Play Store

Latency SLI Specification

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



We can use the same process of mechanical substitution to develop a latency SLI for our purchases too, starting from the generic latency SLI specification "the proportion of valid requests served faster than a threshold".

Buy Flow Latency: Valid Requests

Latency SLI

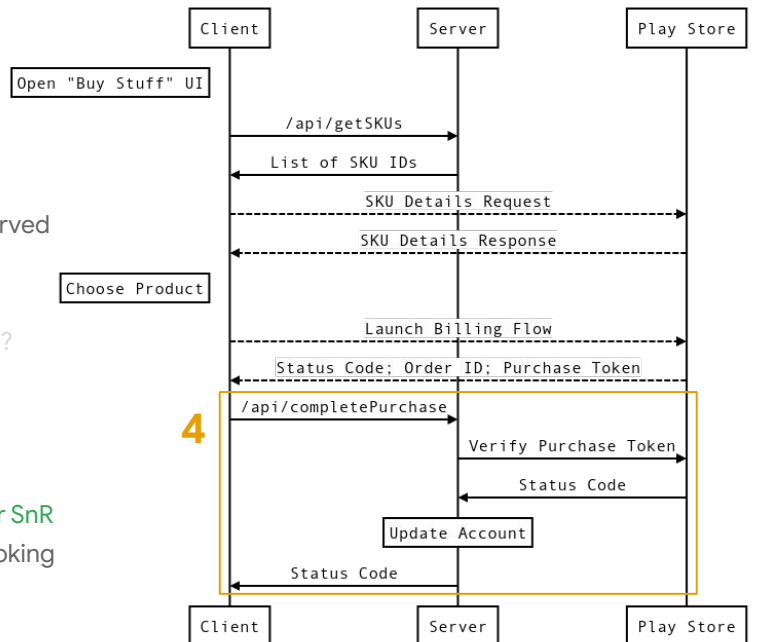
The proportion of **valid** requests served **faster** than a threshold.

... but which requests are **valid**?

3. User launches Play billing flow?
4. **Send token to API server**
5. Verify token with Play Store?

Why not 3?

- Too variable, SLI will have poor SnR
- Billing flow contains lots of “poking device with a finger” time



Measuring the latency of the entire billing flow won't work for us, because while the user's device is running, the billing flow in our app has to relinquish control. If we're measuring the entire flow, but, say, the user needs to add a new credit card because the one they had used previously expired, we'll include that time where the user is poking at their screen in our measurements. This makes our metric highly variable; as we discovered earlier, this results in an SLI with a poor signal-to-noise ratio.

Instead, we'll measure the latency of the requests to /api/completePurchase. These are served by systems we control and should have a consistent latency profile, so our SLI will provide a strong signal and we can take concrete action if it deteriorates.

Buy Flow Latency: “Too Slow” Threshold

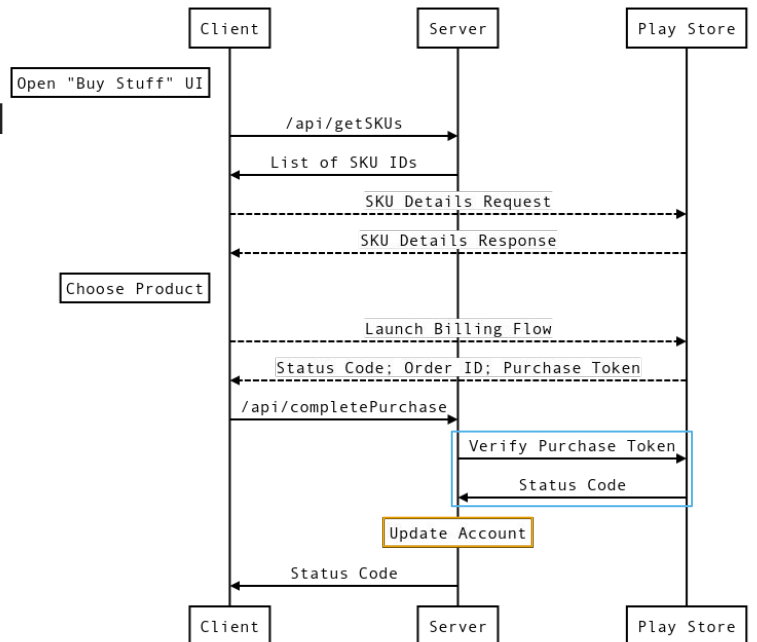
Latency SLI

The proportion of */api/completePurchase requests* served **faster** than a threshold.

... and what is **fast enough**?

Rough estimate time!

- Verify Token $\leq 500ms$
- Database Write $\leq 200ms$
- Round up a bit...



That's easy to substitute in. But what is "fast enough" for these requests?

Because this is a fictional example, we don't have any historical data on the performance we can achieve in practice, nor users who might have expectations about how fast we can serve responses. Instead we'll have to make an educated guess about what our systems might be able to achieve, and try to justify those estimates. If we're only setting a single latency SLI, we should target the long tail—most requests should be served substantially faster than these thresholds.

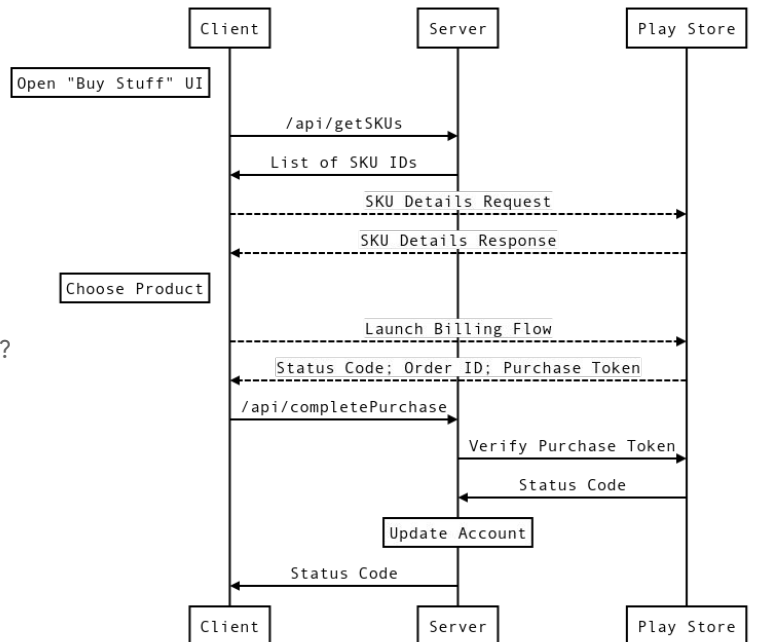
Buy Flow Latency: Measurement

Latency SLI

The proportion of `/api/completePurchase` requests served **within 1000ms**

... but where are we **measuring** this?

Where does the timer start/stop?



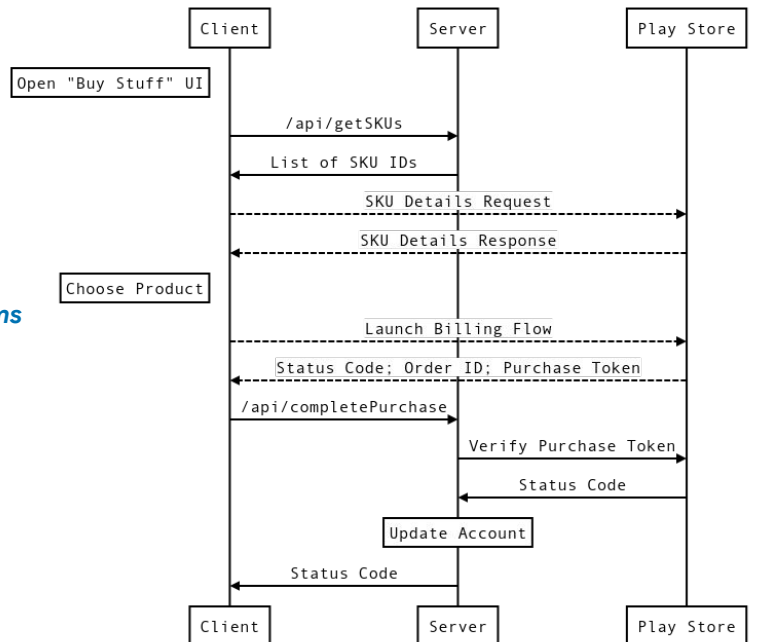
An important facet of engineering is carefully padding your estimates to give yourself a safety margin. In this case, we've opted to round our latency threshold up to a second, and we'd probably pair this with a 95% or 99% SLO target.

We have a lot more flexibility around where we can measure this SLI, because we're not constrained by the need to measure requests that are only observable from the client.

Buy Flow Latency: Measurement

Latency SLI

The proportion of */api/completePurchase requests* where the **complete response** is returned to the client **within 1000ms** measured at the **load balancer**



A good rule of thumb for latency SLIs is to measure at the closest point to the user which is still under your direct control, which in practice usually means at your load balancers. Measuring closer to the user than this usually means incorporating the round-trip time from wherever that user is on the internet to wherever your service is hosted. This is undesirable because it introduces a lot of variability into the measured latency, decreasing the signal-to-noise ratio of the resulting SLI.

<https://cre.page.link/art-of-slos>



First, all of the content for the SRE part of this module is freely available online under the Creative Commons CC-BY-4.0 license. So if you've found this section interesting and you'd like to be able to run an Art of SLOs workshop for your business or at a conference, head to <https://cre.page.link/art-of-slos> to get hold of the slides and handbooks.

You'll also find the student handout on that page:

<https://docs.google.com/document/d/11qMVVdn95tyGvYiVA5HwjIIV750-gYiT-dJCNS0ZPE0/edit#heading=h.hy16vkhmorks>.

Quiz

What are the four golden signals?

- A. Availability, durability, scalability, resiliency
- B. Latency, traffic, saturation, errors
- C. Get, post, put, delete
- D. KPIs, SLIs, SLOs, SLAs

Quiz

What are the four golden signals?

- A. Availability, durability, scalability, resiliency
- B. Latency, traffic, saturation, errors
- C. Get, post, put, delete
- D. KPIs, SLIs, SLOs, SLAs

Quiz

Which definition below best describes an SLI?

- A. It is a time-bound measurable attribute of a service
- B. It is a percentage goal of a measure you intend your service to achieve
- C. It represents a contract with your customers regarding service performance
- D. It is a key performance indicator, for example, clicks per session or customer signups

Quiz

Which definition below best describes an SLI?

- A. It is a time-bound measurable attribute of a service
- B. It is a percentage goal of a measure you intend your service to achieve
- C. It represents a contract with your customers regarding service performance
- D. It is a key performance indicator, for example, clicks per session or customer signups

Quiz

Describe the difference between white-box and black-box monitoring.

Quiz

Describe the difference between white-box and black-box monitoring.

White-box monitoring is based on metrics exposed by the internals of the system, including logs, interfaces like the Java Virtual Machine Profiling Interface, or an HTTP handler that emits internal statistics.

Black-box monitoring tests externally visible behavior as a user would see it. Latency for example.

Quiz

Which is true about error budgets?

- A. Developers can work on new features as long as they are within their error budget
- B. The error budget is calculated as 100% minus the SLO for a given SLI
- C. You should devise an alerting strategy that notifies developers of an event that is consuming an unusually high percentage to the error budget
- D. All of the above

Quiz

Which is true about error budgets?

- A. Developers can work on new features as long as they are within their error budget
- B. The error budget is calculated as 100% minus the SLO for a given SLI
- C. You should devise an alerting strategy that notifies developers of an event that is consuming an unusually high percentage to the error budget
- D. All of the above

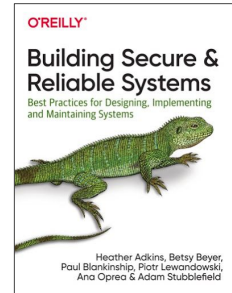
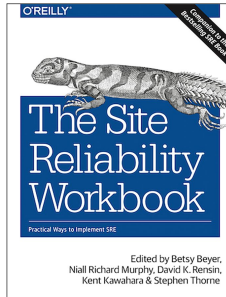
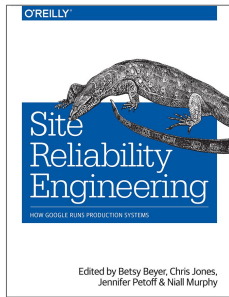
Learned how to...

- 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
- Achieve developer and operation harmony with SLO based error budgets



Excellent job! In this module, you learned how 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.
- Achieve developer and operation harmony with SLO-based error budgets.



Both of these are now available in HTML format for free!

<https://landing.google.com/sre/books/>



Finally, both the SRE book and the Site Reliability Workbook are available on google.com/sre for you to read for free!

