The Art of SLOs

Facilitator Handbook

About the Art of SLOs

*Art (noun): A skill at doing a specified thing, typically one acquired through **practice**.*

The Art of SLOs is a workshop designed to teach the essential elements of developing *Service Level Objectives* (SLOs) to a diverse audience from across the realms of development, operations, product and business.

In the theoretical part of the workshop, participants learn how setting a target that describes the desired reliability of their services can resolve the organizational tension that so often arises between development and operations teams. They are shown how SLOs and Error Budgets can be used to measure and manage the reliability of a service in a data-driven, objective and user-focused manner. The workshop takes a technical turn as the participants are given a brief introduction to the qualities that make for good SLIs. Finally, the session wraps up with an application of the four-step process for developing SLIs to a simple interaction users have with the server-side infrastructure of a fictional mobile game.

The practical part of the workshop asks participants to apply what they've learned to more complex interactions between the users of the game and its infrastructure. Each interaction has a particular twist that challenges them to think hard about what the user is expecting and how to find a good proxy measure for how well the service is meeting those expectations. Finally, they're given an example answer to compare their own progress and reasoning to.

Target Audience

The workshop content is relatively technical and is primarily aimed at development and operations engineers and their immediate management. But you'll have the best results if you can include technically-minded product people and business leaders as well. SLO targets need to be set with your users in mind and error budgets can only resolve organizational tensions if the consequences for exceeding them have executive backing.

Workshop Structure

The recommended way of running this workshop is as a full-day event, though it is *possible* to compress it into as little as 2½ hours with some careful deck editing. The session timings we suggest below are just that: suggestions. We've found that we can hit these timings consistently with some time for audience questions throughout the workshop.

09:30–09:45	Introduction
09:45–10:45	SLIs, SLOs and Error Budgets
10:45–11:00	Break
11:00–12:00	Developing SLOs and SLIs
12:00–13:00	Lunch
13:00–14:30	Practical Exercise
14:30–14:45	Break
14:45–15:00	Example Answer
15:00–16:00	Q&A Session

No slides are included for the introduction. This is a good time to introduce the presenters, explain the workshop in the context of your own company, and cover administrative details like where the restrooms are and what to do in case of a fire, if these are necessary. We have found that spending a little time getting to know your audience can help judge how to pitch the message on the day. We do this by asking them to raise their hands if they self-identify as *e.g.* a software engineer, operations engineer, someone whose job title begins with "C" and ends with "O", and so on.

The Q&A session may not be relevant for your particular needs, but if you're running this workshop as part of an organizational journey towards using SLOs to manage the reliability of your services, this is a great opportunity to address people's concerns.

Running a Workshop

People

There's no point in running an Art of SLOs workshop without a reasonably large audience who will get some concrete value out of learning how to create SLIs and SLOs for their services. The practical exercise is designed for groups of 6-8 people working together. While you *could* just have one group, the organization overheads are fixed, so it makes sense to go large where possible. We've successfully conducted this workshop with 80+ people at a time!

You'll need to find some volunteers who understand the concepts well to help facilitate the workshop. For instance, they might be SREs or operations engineers with experience of using SLOs in their service. It's not strictly necessary to have one volunteer for each table or group—in fact, this can even be counter-productive, as the group looks to the volunteer to lead them rather than beginning to solve the problems on their own. But it's also possible for groups to find it hard to get started, or to get into the weeds of system design instead of focusing on the minimum assumptions necessary to create SLOs. Having a few free-floating experts who can step in and bring the discussions back on track serves to correct these failure modes and ensure a smooth learning experience for all participants.

These volunteers also make good candidates for panel members in the Q&A session after the workshop proper.

Location

You'll need a large room with presentation equipment, naturally. Most importantly, you'll want to arrange for the room to be set up "cafeteria style"—square or circular tables spread out across the room—so that people can work on the practical exercise together in groups. This can be hard in conference spaces, so make sure you get your requests in early! Having space for breaks and lunchtime co-located is very helpful: herding people back into the workshop space afterwards can consume a lot of valuable time, and the further they have to go the longer this takes.

Preparation

Pre-Workshop Timeline

T−2w:	Print handbooks
	Identify facilitators
T−1w:	Organise snacks
	Train facilitators
T−3d:	Feedback form
	Customize deck

A couple of weeks before the event you'll need to get a run of handbooks printed, so that every participant has one on the day. It's *possible* to do this yourself for small groups, especially if your office printers offer "short-edge flip" booklet printing, but manually collating, folding and stapling the 7 sheets of paper into a 28 page booklet quickly becomes tiresome. There are plenty of online printing services that will do short runs of 1-200 booklets for reasonable prices, if you're planning to reach this many people. You'll also want to print enough copies of the SLO worksheet that each table has at least 5 to work with.

You should also have identified your volunteers and Q&A panel members by this point. A week or so before the event, it's good to run through a "train the trainers" session (see later on in this handbook) with these people, even if they're relatively experienced at facilitating this workshop. This ensures all the user journeys—the sets of interactions users have with the fictional mobile game infrastructure in pursuit of a singular goal—are fresh in people's minds, so they're prepared to answer questions or provide suggestions that keep the table discussions moving forward.

If you're running a day-long event, you'll need lunchtime catering and snacks for the breaks. At the minimum participants will need something to keep their brains working through the workshop. These things often take some time to organize!

A few days before your workshop, you should make a copy of the original public deck and customize it. Simple things like adding your own logo and the names of the presenters to the intro slide are valuable. Create a feedback form and add any questions you want to ask your audience to it. We recommend the "Course Evaluation" Google Forms template as a starting point. Generate a QR code linking to the form using one of the many free online generators and place this QR code on the "Thanks!" slide in the spaces indicated. Most people will have a phone capable of scanning this link from where they are sitting, which means they can fill in the feedback form during the Q&A session. This dramatically increases the amount of feedback you get.

On the Day

It's worth arriving some time before your audience. Alongside the usual setup and A/V testing, this gives you time to put stacks of the printed handbooks and worksheets on each table. You can make the room more welcoming for attendees as they arrive by playing some classical music quietly. Don't forget your attendees will need pens to write with too!

Take a second before starting to make sure everyone has a handbook. Don't skip the welcome activity on the second slide: people are happier to ask questions and volunteer answers if they feel part of the group.

The deck is split into four approximately equal-sized parts with pauses for audience questions after each 30-minute section. Try to keep the questions limited to about 5 minutes per section, or you'll run out of time. The speaker notes contain both bullets and prose. You can read the prose to the audience as-is if ad-libbing makes you uncomfortable :-)

There are a number of interactive activities in the deck that are intended as prompts to get the audience thinking and talking. If you've had a lot of questions and are running short on time, these activities can be sprinted through or skipped entirely; if your audience is quiet or you're running fast you can dwell on them and try to tease out differences of opinion.

In the afternoon, we recommend that all tables work on the "Buy In-Game Currency" problem first, since this is the user journey that has an example answer provided in the deck. Participants should be able to cover this journey and one other in the 90 minute session from the example timings on page 4, so we suggest you allow each table to decide amongst themselves which of the other problems to attempt.

If you're planning to run many of these workshops for your organization, delegate someone to record notes, questions and slide timings for a post-workshop retrospective so next time goes more smoothly.

Homework

The goal of this workshop is for everyone to leave convinced that they should define some SLOs for their services, and with knowledge of how to achieve this in practice. So, if you're running this workshop for people at your company—it's hard to set homework for conference attendees—it can be useful to ask them to take a shot at setting some initial SLOs for the services they are responsible for.

Frequently-Asked Questions

Similar questions tend to crop up regularly when we run this workshop. Answering them all in the slides could easily double its length, and answering them all here could produce a (short) book. We're including a list of them so you can think about them and be prepared!

Questions About SLOs

- How can we measure user happiness?
- What steps should we follow to get started with SLOs?
- Is there any software or tooling that can do this for us?
  - Should we also have SLOs for our monitoring?
- How should we document our SLOs?
  - Should we keep our old SLOs around after iterating?
- What is the difference between Coverage and Correctness?
  - (and other variations on this theme)
- What happens if the "fix" means more errors in the short term?
- What's the difference between "planned" / "accidental" downtime?
- Should we set SLO targets based on competitors performance?
  - Is five nines enough?
- How can I create SLOs for systems with third-party dependencies?
  - Should we enact consequences when third party is at fault?
  - What about things like CDNs between users and our service?
- I've got 100+ microservices, do they all need SLOs?
  - Is it better to define SLOs bottom-up or top-down?
  - How can we aggregate fine grained low-level SLOs upwards?
- Can we have SLO targets that vary based on time of day/year?
  - What about weighting different types of requests unequally?
  - How do we represent degradation of service with an SLO?

Questions About SRE

- What's the biggest mistake people make setting up SRE teams?
- What do you look for when recruiting SREs?
- How do you make sure that there is a consistent level of under-standing of these concepts across your organization?
- How do you push back against project managers with deadlines?

Volunteering at a Workshop

Around a week before the workshop is due to take place, you should gather all the volunteers together and run a "train-the-trainers" session. The goal of this session is to familiarize the volunteers with the game's serving infrastructure and the five user journeys participants can create SLOs for. Beforehand, give each volunteer a copy of both handbooks to refer to, and ask them to read through the detailed write-up of the "Buy In-Game Currency" journey on page 15 of this handbook. The training session will take approximately an hour, with the following agenda:

XX:00–XX:10	Logistics and ground rules
XX:10–XX:15	Game serving infrastructure
XX:15–XX:30	Overview of user journeys
XX:30–XX:35	Introduce four step process
XX:35–XX:55	"Buy In-game Currency" deep dive

Logistics and Ground Rules

You'll want your volunteers to turn up towards the end of lunchtime—before the workshop session begins—and stick around for most of the afternoon. Now is a good time to sort out which of them will be participating in the Q&A session, if you're running one.

Volunteers should be prepared to keep the discussion on topic and drive a convergence of opinion on each table. A common failure mode is for participants to focus on designing the serving infrastructure in ever more detail. They'll make far better forward progress if they make a minimal set of assumptions that allow them to define reasonable SLOs. Volunteers should keep an eye on the clock and draw participants back to the SLO worksheets and the four step process if they are obviously sidetracked or they've spent more than half an hour on SLIs for a single journey.

Conversely, if a table blazes through all of the journeys, they've most likely not gone into enough depth on any of them. They will almost certainly find that the SLIs they've come up with are unable to capture at least some of the failure modes of the serving infrastructure, so volunteers should challenge them to think up undetectable failures.

We've compiled a list of useful things for volunteers to keep in mind while they're helping out:

- Remember everyone's here to learn and have fun, so *be nice* ;-)
- Sit in the middle of the table rather than at one end.
- Make sure everyone can hear you and is participating equally.
- Don't give people answers, coax them forwards with questions.
- Encourage people to make *justifiable* assumptions about parts of the architecture that are ill-defined to keep things moving forward.
- Encourage discussion of trade-offs, particularly around choosing measurement strategies. There's no single perfect approach.

Serving Infrastructure and User Journeys

The workshop handbook contains a basic infrastructure block diagram and descriptions of the five user journeys that participants will have to grapple with. It's important that all volunteers know how the serving infrastructure is expected to work and have some ideas around SLIs for each of the journeys before the workshop begins. The next section of this booklet has some considerations for each journey, as well as detailed write-ups for both the "Buy In-Game Currency" and "App Launch" journeys.

It's important to remember that these write-ups cover the journeys in more depth than groups will be able to manage in 45 minutes. Instead, they're intended to give some insights into the thought process that underlies the engineering decisions around choices of measurement strategy or prioritization. Producing a similarly detailed write-up for one of the remaining journeys is an excellent piece of homework to challenge your volunteers with before the workshop!

The Four Step Process

Workshop participants have the four step process from page 14 of the handbook demonstrated to them with a very simple journey. After the practical session, they are shown a condensed version of the detailed write-up. In the second half of the training session, you should go through slides 76-92 of the workshop presentation and ensure your volunteers are happy they understand the journey and the example answer. Are they confident they can help your workshop participants come up with SLIs?

The User Journeys in Brief

Buy In-Game Currency

The critical things to take into account are:

- Two critical request/response interactions are with the Play Store rather than the game's server-side infrastructure.
- Users will generally not buy things after they've opened the SKU list. You can assume purchase rates of 1-2%.
- Many of the non-OK status codes the Play Store responds with may not be errors from the perspective of the business or the user.

If you want to make this harder for your group, ask them about purchase correctness, but be prepared to give them a lot of help. The Play Store provides transaction lists in CSV form daily so it should be possible to determine whether the system gave the correct in game items to users.

Possible SLIs:

- **Availability** and **Latency** for "Get SKUs", measured with a combination of load balancer metrics and a synthetic client.
- **Availability** and **Latency** for "Purchase SKU" measured with client-side instrumentation.
- **Correctness** for "Purchase SKU" measured with a reconciliation pipeline running daily over Play Store purchase logs.

App Launch

The critical things to take into account are:

- The large QPS disparity between (register + auth) and sync means these can't be aggregated together simply.
- Username validation is done interactively in the client, so will need strict latency guarantees that are not true of the rest of the flow. But server-side validation is necessary: this may not need an SLI!
- The business impact of failed registrations is higher than that of failed authentication: those users may never come back!
- Downloading game state from our servers should be relatively quick, but CDN assets can be substantial for a fresh install, so measuring overall sync latency client side will run into problems.

If you want to make this harder for your group, consider adding OAuth calls to 3rd party authentication providers like Facebook/Google in the first and second steps.

Possible SLIs:

- **Availability** of the full journey measured with a synthetic client.
    - Create new account, authenticate as the new user, synchronise game state, then validate empty account data.
    - Authenticate as a known "golden" user, synchronise game state, then validate the response matches the known data.
- **Availability** and **Latency** of (create, auth) and (sync) phases separately, measured with load balancer metrics.
    - Load balancers do standard "5xx codes are failures" availability. Remember to ask about 4xx codes for e.g. "bad password" or "unknown user" which shouldn't count against SLI.

Manage Settlement

The critical things to take into account are:

- Asynchronous success, from the user's perspective. They care about the building being finished, but this happens 300s later.
- There's 3 request/response pairs here: aggregation is important.

If you want to make this harder for your group, consider challenging them to create a Correctness SLI for these actions. How many building actions initiated by a user did not result in their building being completed? This could potentially happen if e.g the game server hosting the player's settlement suddenly dies in the middle of a transaction…

Possible SLIs:

- Composite **Availability** and **Latency** for all endpoints, measured at the load balancers and aggregated together.
- Composite **Availability** SLI measured by a synthetic client.
    - Client detects builds are completed via side-channel to DB.
- **Throughput** SLI for game servers, measured server-side.
    - Expressed as proportion of game ticks executed in <Xms.
- **Correctness** SLI for game actions, measured by publishing a Pub/Sub feed of game actions from the API servers and having a listener validate that those actions were reflected in game state after the expected build times elapsed.

Play PvP Battle

The game design around this battle mechanic is extremely incomplete, so you will need to encourage people to make assumptions when they tackle this journey! Here are some things to think about.

Should defenders be given a choice about participating in the battle? The attacker has one, so the defender should probably get one, but how does this change the matching algorithm and the latency expectations for the set-up phase?

Should game state be updated in real-time alongside the battle? This would ensure that losses incurred during a battle would persist through (potentially deliberate) disconnections. But what does this mean for the system design and SLIs?

How long should each game last? We assumed 3-5 minutes, with attackers being allowed to launch successive waves of troops every 30 seconds or so and defenders being allowed to place towers whenever.

The critical things to take into account are:

- Whether the PvP battle traffic goes via the game servers. We think that it probably would in a real-world implementation, because:
    - Device-to-device networking will be an *Absolute Nightmare™* given the prevalence of NAT within carrier networks.
    - Proxying battle traffic allows for in-game analytics, which are likely desirable from a business perspective.
    - Making the server the source-of-truth for game state prevents malicious clients "hacking" the battles.
- Player behaviour. Players that are losing may well deliberately drop their connectivity to attempt to avoid negative consequences. Naïve SLIs may not take this or other connectivity problems into account. More generally, the definition of "success" is not as clear-cut as people may initially assume.
- How to treat the HTTP request/response and UDP phases of the user journey. Should each UDP packet be considered a separate "event" for SLIs?

It's unlikely you'll need to make this harder for your group, but if they blaze through this ask them to consider throughput SLIs for the game servers. The reason these might be important is that the game servers will be handling many of these battles simultaneously. Battles are likely to have some form of time quantization, even if users perceive them as real-time. When this quantization slips, e.g. due to overload, users will notice their games running slowly.

Possible SLIs:

- **Availability** of launchAttack endpoint, measured at load balancers.
- **Latency** of player matching, measured at the API servers.
- **Latency** of battle setup, measured at the API servers.
    - This assumes the defender gets a choice, so we can't measure the latency of the overall attack launch.
- **Latency** of in-battle actions, measured at the game client.
- **Throughput** of battle actions measured at the game servers.

Generate Leaderboards

This journey is loosely based on Apache Beam's mobile game example[1]. Participants will need to make some assumptions around how scoring data is stored and processed. The easy option is to assume the Leader-boards store is something like BigQuery and the serving snapshots are created by running queries and writing the results to BigTable.

The critical things to take into account are:

- There are lots of different moving parts, but which ones do the players really care about?
- Some additional data needs to be propagated through the process-ing architecture to support measuring SLIs.

Possible SLIs:

- **Freshness** of serving snapshots, the time delta between now and the most recent game completion reflected in the snapshot.
- **Coverage** of game completions in leaderboard store and archives.
- **Correctness** of the per-area top score tables, validated by recon-structing them from archived score data.

[1] https://beam.apache.org/get-started/mobile-gaming-example/

Buy In-Game Currency

The "buy flow" involves 5 http request/response pairs. Only 3 of these are visible from the server-side, but for the flow to be successful, all must be successful, the most important of which is the request to the Play Store[2]. SLOs based solely on server-side or front-end metrics will not provide enough coverage to give a good idea of overall journey reliability.

It's possible to build a synthetic client for purchases on the Play Store using the Play Billing testing APIs[3]. This won't be particularly realistic, but it could be used as a basic "is play billing working" signal. Testing getSKUs, the Play Store SKU details request, and completePurchase with a synthetic client is more tractable. Presuming the success of the Play Store billing flow, the synthetic client can send a known, fake purchase token which the API servers can special-case where necessary.

Since this is an important revenue-generating journey, building some client-side instrumentation is a reasonable investment and will yield better overall coverage of the user experience than synthetic clients will. It's important to consider that our users may object to their device sending this sort of telemetry, so we'll need a setting in the game that allows them to consent to it.

Purchase Availability

Splitting the journey into two parts ("Display Available SKUs" and "Purchase SKU") for availability purposes is important, because many people who load the list of SKUs will not buy anything. The real value for the company is in "Purchase SKU", since each successful transaction generates measurable revenue, so it is most important to have **Availability** measured for this part of the journey. Because some of the errors the Play Store can return are for valid reasons[4] it's important to be specific about what's "successful"—it's not just OK.

Availability SLI: The proportion of launched buy flows from consenting users where the request to the play store results in one of the following successful codes:

- `OK`
  `Hopefully self-explanatory`

[2] https://developer.android.com/google/play/billing/billing_library_overview
[3] https://developer.android.com/google/play/billing/billing_testing
[4] https://developer.android.com/reference/com/android/billingclient/api/BillingClient.BillingResponseCode

- `FEATURE_NOT_SUPPORTED`
  `User is using an unsupported android / play store version, we can't fix this.`
- `ITEM_UNAVAILABLE`
  `Either a race condition (we disabled the item since they listed the available SKUs) or someone faking SKUs.`
- `USER_CANCELED`
  `User gave up trying to buy the SKU. This conveniently includes all forms of payment declined errors, because the buy flow prompts users to try other forms of payment and forces them to cancel if none work.`

AND the request to /api/completePurchase results in one of the following HTTP status codes:

- `200 OK`
  `Hopefully self-explanatory, again`
- `Any 4xx code`
  `Client errors. Specifically we use 402 Payment Required to signify to the requestor that the validation of their purchase token with the Play Store failed.`

AND the JSON response is parsable and valid, as measured by the game client and reported back asynchronously.

Purchase Latency

Since the device-side billing flow includes a nondeterministic amount of non-request "user poking their device with a finger" time, we are going to measure the latency of the completePurchase API call for our purchase latency SLO. It involves a relatively cheap call to the Play Store and a database write. We will measure this at the load balancers to avoid capturing highly variable and uncontrollable device → load balancer latency.

Latency SLI: The proportion of completePurchase API requests where the time from the load balancers receiving the request to the complete response being sent back from the server is less than 1000ms.

getSKUs and SKUDetails

Neither of these SLIs covers the getSKUs / SKU details side of things. To meet our guidelines of 3-5 SLIs per journey, we're going to cheat a little! We'll assume we have already-existing general "Read-Only API Availability" and "Read-Only API Latency" SLIs which aggregate many API endpoints together. The getSKUs API call is a good candidate for inclusion into an aggregate SLI like this. All these API requests will be covered with a combination of synthetic client requests and load balancer metrics.

We should have our synthetic client also make requests to the Play Store to track "SKU Details" availability, but we may not want to include this into our API availability SLI. Instead, if our game displays an error message like "could not contact the Play Store" when this request fails, the user's unhappiness will not be directed towards our services. This is an accurate reflection of the underlying problem, so we're not lying to our users, and we can't do anything meaningful when the Play Store is failing anyway.

Purchase Correctness

Correctness is a good fit for this journey: (1) it is business critical and revenue-generating, (2) the interactions have important side-effects (incrementing in-game currency) that our users care deeply about and the flow of those interactions is complex enough that users can exit at a number of points, and (3) we have an external source of data (the Play Store) to validate against.

There's a lot of value in a cross-check so we can be sure that everything has occurred as it should have. Users may successfully complete the Play Store billing flow but fail to send /api/completePurchase for a while—the Play Store guidelines suggest retrying purchases in this state on application start-up.

The Play Store generates financial reports[5] which contain records of every transaction made in the game in a given month. These reports are updated daily. This makes it possible to build a reconciliation pipeline that compares real account balances with ones calculated from this data. It will take a snapshot of the amount of in-game currency each user has at the beginning of a day and keep track of the currency they spend or generate organically during that day. At the end of the day, it can combine this with the purchases of new currency they made that day from the financial report to arrive at the amount of currency they should have at the end of that day.

Correctness SLI: The proportion of users whose real account balance matches a synthetic balance calculated from their real balance from the previous day, Play Store order data and currency spend history over the day.

[5] https://support.google.com/googleplay/android-developer/answer/6135870?hl=en-GB

SLO Targets

All SLOs use a 28 day rolling window, providing a good balance between long-term and short-term prioritization needs. If we decide to drive incident response with these SLOs, we will also measure them over 1h, 12h and 1 week rolling windows.

Purchase Availability: 99.95% of purchases are successful in the past 28 day rolling window.

We assume 99.99% availability of the Play Store billing flow, so a 99.99% target here would allow for no other unavailability. A 99.95% target gives scope for other failure, including a larger buffer for in-flight client telemetry, while still being appropriately strict for an important, revenue critical user journey.

Purchase Latency: 99% of completePurchase API requests are served in <1000ms in the past 28 day rolling window.

We are targeting the long tail with this SLO. In-app purchases have to feel fast so users will spend more money; this validation step must not hold up the overall purchase flow.

Purchase Correctness: 99.999% of users have real balances that tally up with their synthetic counterparts in the past 28 day rolling window.

The holy grail, five nines! We have 50M 30-day active users. Assuming that 1% of them make a purchase on any given day, this target gives us an error budget of around 140 inconsistent balances per month. This is a low enough number that our support teams can deal with individual cases.

App Launch

Breaking the Journey Down

The first thing to do when contemplating SLIs for this journey is to make some assumptions about the relative importance of each phase of the application launch process. All three phases must complete successfully before the user is able to play the game, but only the final one, syncData, occurs every time they start the app. An outage affecting this API call will affect more users than either of the other two, so we'll consider this part of the journey to be the most important.

Users who already have an account and are just migrating to a new device will presumably return to try again if they encounter an error. In the meantime, they can continue playing on their old device as long as syncData is still working. So this is probably the least important phase of the journey. Users who receive an error in the account creation phase will most likely not come back to try again unless they have some social pressure to do so, e.g. wanting to play with friends.

Practically speaking, we will need to measure the availability of the sync phase separately from the other two, due to the difference in request rates. We'll ignore username validation entirely since it happens in the background as the user types their desired name. The client should perform validation asynchronously, set a short request timeout to curb tail latency, and simply not display anything if a validation request fails. We'll have to do validation server-side on account creation anyway to avoid race conditions.

Choosing a Measurement Strategy

It's plausible to assume the game client already has some telemetry features given how important data on user interaction is for game development. We might even be generous and assume that the server-side infrastructure receiving this telemetry data is built with reliability in mind, and provides the kind of low-loss ingestion suitable for building high-availability SLOs on top of it. These assumptions would appear to remove one of the major downsides of client-side instrumentation (cost of implementation) as a source of SLI metrics. Why don't we just default to that as our measurement strategy for all the SLIs we develop to cover this journey?

In the Art of SLOs, we show that an SLI is only useful when the range of values it has during normal operation is perceivably different from the range of values it has when the system is failing. Client-side telemetry data, especially from mobile clients, is inherently noisy—mobile network quality is highly variable, connectivity comes and goes unexpectedly, and there is a broad range of available devices with wildly different capabilities. Many of the factors that cause this variability are outside of our control and hard to isolate from each other. Including highly variable factors in our SLI increases the probability that these ranges will overlap, decreasing the signal-to-noise ratio.

In practice, creating a viable SLI from client-side instrumentation usually requires pre-processing the telemetry data to attempt to remove the variability we don't care about and reveal changes in aspects we do care about. This type of data processing is complex and takes time to create and fine-tune. For this user journey, the engineering cost to build and maintain the filtering and processing heuristics necessary to derive a high-quality SLI from the underlying data—likely to be multiple person-quarters of effort—outweighs the marginal additional benefits from measuring our SLIs closer to the user.

A more effective trade-off is to use an SLI based on HTTP status codes measured at the load balancer to provide reasonable coverage for less engineering effort. This measurement strategy has well-known drawbacks, primarily the inability to detect malformed responses. But, assuming that load balancer metrics are already in place, the effort required to build synthetic clients that cover that gap is probably of the order of a few person-weeks. A single synthetic client SLI can easily cover all of the journey at once, validating that the overall flow works correctly and closing the measurement gaps inherent in load balancer SLIs. Coupling per-endpoint load balancer SLIs with a single whole-journey synthetic client SLI is a common pattern, and one we would recommend for many scenarios.

SyncData Load Balancer Availability

We determined previously that the most important phase of this journey was syncing the game state to the device, so let's tackle that first. We'll treat 4xx responses as successes because we expect user errors and fake (or even actively malicious) clients to far outweigh the probability of our servers returning e.g. 403 or 404 for a legitimate request. This background noise would obscure the signal from a slightly elevated rate of 5xx responses if it were included in the SLI.

The risk here is that code or configuration bugs might result in large-scale "authentication is broken for everyone" or "the load balancers are null-routing /api/syncData" problems. Fortunately, these will be caught by our synthetic client.

SyncData Load Balancer Availability: The proportion of /api/syncData requests that result in a 2xx, 3xx or 4xx response code, measured at the load balancers.

SyncData Availability SLO: >99.95% of /api/syncData requests are successful during the past 28 days.

Authentication Logs-based Availability

Before we dig into the details of a synthetic client that validates the entire journey, let's consider the other phases of the journey separately. The real-world rate of account creation and token exchange is quite low, according to the handbook. Let's assume that means roughly one "create user" and one "new device" every 10 seconds, which would result in 3 requests every 10 seconds with a 2:1 bias in favour of token exchange because each account creation needs a subsequent auth token request.

The problem here is that any synthetic client we build is going to create approximately the same amount of traffic to these endpoints as our real users will, and our load balancer metrics won't discern between the two traffic sources.

One way we could deal with this is to base our authentication availability SLI on log analysis instead of load balancer metrics. We can have our prober use a well-known, unique UserAgent like "wooden-stake" (this is a vampire game, after all) and ignore those requests as invalid when processing logs to count "good events" and "all events". Our synthetic client SLI can be used in an SLO with a small time window to drive short-term operational response, and the logs-based SLI (which might suffer from ingestion and processing latency) can be used in an SLO with a longer time window to drive prioritization decisions.

Auth Logs-based Availability SLI: The proportion of *valid* request logs which were *successful*.

- *Valid* requests have a UserAgent of `/^wooden-stake v[\d.]+$/` and a request path of /api/createAccount or /api/getAuthToken.
- *Successful* requests also have a 2xx, 3xx, or 4xx HTTP status code.

Auth Logs-based Availability SLO: >99.95% of real user auth requests are successful during the past 28 days.

App Launch Synthetic Client Availability

A synthetic client that attempts to exercise the entire journey will provide a backstop for the status-code based SLIs that cover individual parts of it. We can validate that journey is working correctly by having our synthetic client:

1. create a new account;
2. request an authentication token for that new account;
    a. validate this token works by requesting /api/syncData;
    b. validate the response represents an empty account;
3. request another auth token for a known "prober_user" account;
    a. validate this token works by requesting /api/syncData;
    b. validate the response matches a known-good game state;
4. delete the newly-created account;

We'll generate the known-good state for "prober_user" pseudorandomly as part of our build process to ensure it doesn't get stale and push it to the user's account and the synthetic client when the build is released to production. The client will use the state to validate syncData responses it receives from the API servers. The account should not be modified between releases!

App Launch Synthetic Client Availability SLI: The proportion of synthetic client application launch flows where all HTTP status codes are 200 OK and all validation steps complete successfully.

App Launch Synthetic Client Availability SLO: >99.8% of synthetic app launches are successful during the past 12 hours.

Here, we choose a lower availability target because we have a much shorter measurement window and we're only completing one synthetic app launch journey every 10 seconds. In 12 hours that's 4320 app launches, so setting a 99.95% availability target results in missing that target once 3 or more launches fail. These could be isolated connectivity problems that might go mostly unnoticed by users.

A 99.8% target gives us an error budget of 8 launches across the twelve hours, or nearly 90 seconds of downtime in a 12 hour period before the SLO is impacted. We're making an explicit trade-off to lower sensitivity here, so that we only trigger an operational response when one is necessary. We can't have this happen regularly because we're choosing to only fire an alert when we burn ~7% of our error budget over 12 hours, but our other SLOs will take care of that side of things.

App Launch API Composite Latency

Latency-wise, users will care most about the overall app launch time, i.e. from "I pressed the icon on my home screen" to "I can see my settlement". This latency is an ideal measure of our user experience, but it will be highly variable, depending on things like the speed of the user's device, how much content needs to be synced and their connection speed. As with other client-side instrumentation, isolating a meaningful, actionable "too slow" signal from this noise is likely to need some complex data processing.

Instead, as a compromise, we'll track the latency of all app launch rel-ated API calls, measured at the load balancers. While in theory this is not as ideal as an SLI on app launch time, in practice the consistency and reliability of this measurement makes it a better choice for our initial SLI. If we find that this compromise SLI is not representative enough, i.e. users are unhappy with launch times but this is not reflected in the SLO, we can always revisit the implementation of our SLI and invest some additional effort to derive a reliable signal from the highly variable app launch time data.

Tracking latency at the load balancer assumes that these API calls have relatively similar latency profiles and request rates. Realistically, getAuthToken is likely to be faster than syncData, which is again likely to be faster than createAccount, but the differences are likely to be of the magnitude of tens-of-milliseconds, and we're mostly concerned about latency of the magnitude of seconds.

The problem here is again the difference in request rates between syncData and the other two handlers. We want them to have approximately equal weights in the composite SLI/SLO. The easiest way to achieve this goal is to create separate latency SLIs and SLOs for each API call and aggregate them as SLOs rather than SLIs using a "bad minute" approach. This also allows us to set separate latency thresholds for each SLI, so we no longer need to worry about the correctness of our assumptions.

Each successive minute of our rolling window where all three SLIs are above their respective SLO targets is considered "good", while minutes where any or all are below target are "bad". We then set a target for how many "bad" minutes we allow per month.

LB createAccount Latency SLI: The proportion of /api/createAccount requests where the complete response is sent in <2000ms, measured at the load balancers.

LB getAuthToken Latency SLI: The proportion of /api/getAuthToken requests where the complete response is sent in <400ms, measured at the load balancers.

LB syncData Latency SLI: The proportion of /api/syncData requests where the complete response is sent in <2000ms, measured at the load balancers.

App Launch API Composite Latency SLI: The proportion of minutes where:

- >99% of /api/createAccount requests are <2000ms, and
- >99% of /api/getAuthToken requests are <400ms, and
- >99% of /api/syncData requests are <2000ms.

App Launch API Composite Latency SLO: API requests will meet their individual SLOs >99.9% of the time during the past 28 days.

Recreating Handbook PDFs

You might think this would be easy, right? It is... *mostly* easy. The problems arise when you want to create an A5, Letter or Half-Letter version of the (A4) handbook. Changing the page size in Google Docs leaves you needing to re-layout the document basically from scratch, but it is possible to resize the PDF that Google Docs can generate for you using GhostScript[6].

The wrinkle to overcome is that when you download the PDF the fonts are not embedded, which makes it impossible to re-render the document correctly at a smaller size. We use Raleway and Roboto, which are freely available from the Google Fonts Library[7]. Once you have both the PDF and the fonts in the same directory, the following Magic Incantation will produce a resized version of your modified handbook document with the fonts embedded correctly:

```
gs \
  -dSAFER \
  -sDEVICE=pdfwrite \
  -dPDFSETTINGS=/prepress \
  -dColorConversionStrategy=/LeaveColorUnchanged \
  -dSubsetFonts=true \
  -dEmbedAllFonts=true \
  -sFONTPATH=. \
  -dFIXEDMEDIA \
  -dPDFFitPage \
  -sPAPERSIZE="a5" \
  -o "OUTPUT.pdf" \
  -f "INPUT.pdf"
```

Using a PAPERSIZE parameter of "letter" or "halfletter" instead of "a5" will generate these sizes.

Some printing shops will complain if the document you want to print as a booklet isn't a multiple of 4 pages long. If they do, you can add a single padding page to the output PDF by appending `-c showpage` to the above command. For multiple padding pages, repeat this multiple times.

[6] https://ghostscript.com/download/gsdnld.html or via your friendly local package manager.
[7] https://fonts.google.com/specimen/Raleway and https://fonts.google.com/specimen/Roboto