SLO WORKSHOP

Outage Maths **4**

How SLOs help… **5**

The SLI Equation **6**

Translating a user journey to SLI specifications 7

Specifying an Availability SLI **8**

Request / Response 8

Other Availability SLIs 8

Specifying a Latency SLI **9**

Request / Response 9

Other Latency SLIs 9

Specifying a Quality SLI **10**

Request / Response 10

Specifying a Freshness SLI **11**

Data Processing 11

Measuring Data Freshness as Response Quality 11

Specifying a Coverage SLI **12**

Data Processing 12

Specifying a Correctness SLI **13**

Data Processing 13

Measuring SLIs **14**

Server-side Logging 14

Application Server Metrics 14

Front-end Infrastructure Metrics 15

Synthetic Clients (Probers) or Data 15

Client Instrumentation 15

Developing SLOs and SLIs **16**

Example SLO Worksheet 16

User Journey: Home Page Load 16

Stoker Labs Inc. **18**

Mission Statement 18

Our Game: Tribal Thunder 18

Service Architecture 19

User Journeys 19

View Profile Page 19

Buy In-Game Currency 20

App Launch 21

Manage Settlement 22

Play PvP Battle 23

Generate Leaderboards 24

Postmortem: Blank Profile Pages! 25

Impact 25

Root Causes and Trigger 25

Detection 25

Lessons Learned 25

Action Items 25

Profile Page Errors and Latency 26

Outage Maths

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

Boxes shaded red allow less than one hour of complete outage.

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

How SLOs help...

...your business engineer for reliability

The **product** perspective:

*If reliability is a feature, **when do you prioritise it** versus other features?*

The **development** perspective:

*How do you **balance the risk to reliability** from changing a system with the requirement to build new, cool features for that system?*

The **operations** perspective:

*What is the **right level of reliability** for the system you support?*

The SLI Equation

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

*The proportion of **valid events** that were **good**.*

Expressing all SLIs in this form has a couple of useful properties.

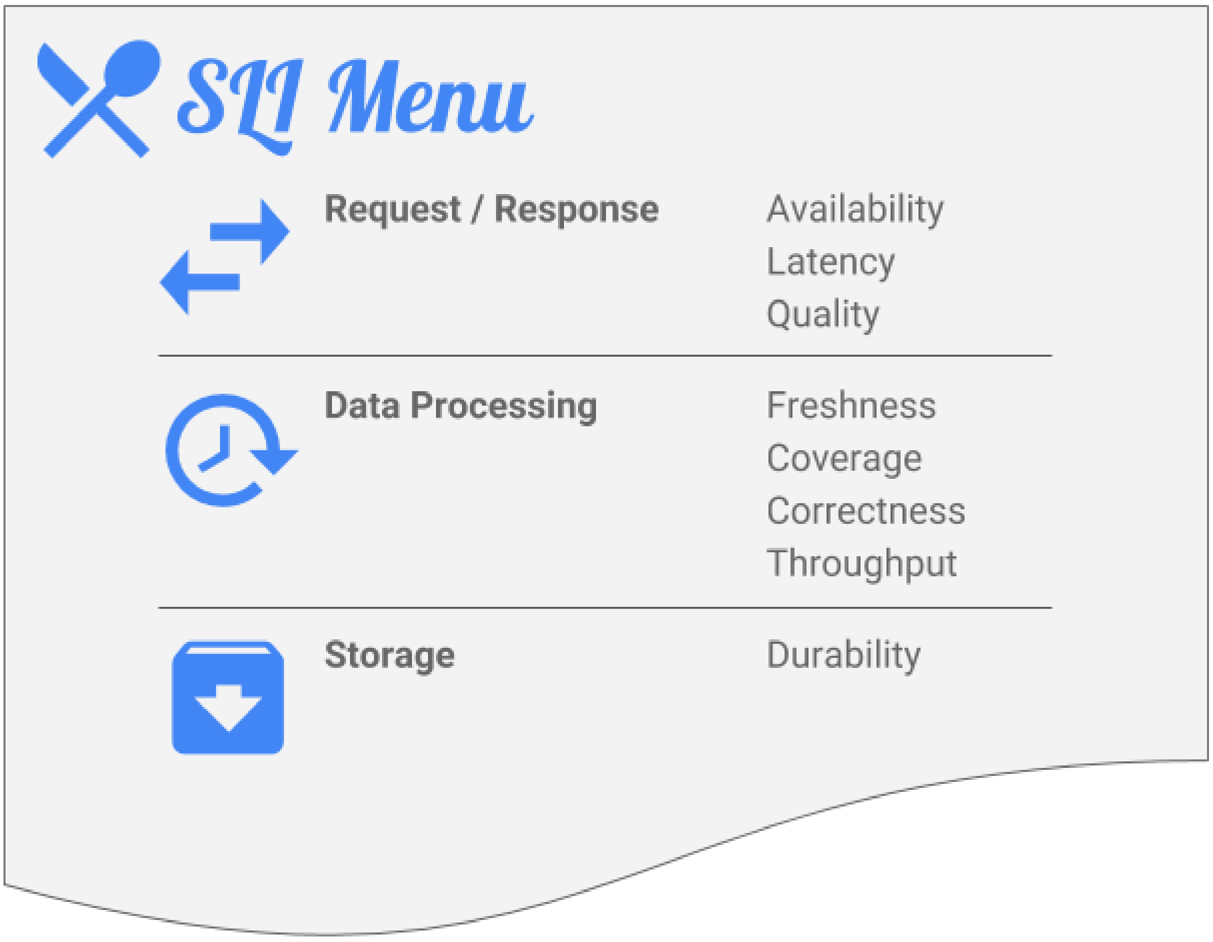
1. **SLIs fall between 0% and 100%.**
   0% means nothing works, 100% means nothing is broken. This scale is intuitive to reason about and directly translates to percentage-reliability SLOs and error budgets.

2. **SLIs have a consistent format.**
   Consistency allows common tooling to be built around 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 SLO threshold.

Events can be prevented from counting against an error budget either by including them in the numerator or by excluding them from the denominator. The former is achieved by classifying some events good, the latter by classifying some events invalid.

Typically, for systems serving requests over HTTP(S), validity is determined by request parameters like hostname or request path, to scope the SLI to a particular set of serving tasks or response handlers.

Typically, for data processing systems, validity is determined by input parameters, to scope the SLI to subsets of the data.



SLI Menu

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

Translating a user journey to SLI specifications

A 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 are likely to want to measure for a given user journey.

Once 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**.

Specifying an Availability SLI

The availability of a system serving interactive requests from users is a critical reliability measure. If your system is not responding to requests successfully, it's safe to assume it is not meeting your users' expectations of its reliability.

Request / Response

The suggested specification for a request/response Availability SLI is:

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

Turning this specification into an implementation requires making two choices: which of the requests this system serves are **valid** for the SLI, and what makes a response **successful**?

The definition of success tends to vary widely depending on the role of the system and the choice of how to measure availability. One commonly used signifier of success or failure is the status code of an HTTP or RPC response. This requires careful, accurate use of status codes within your system so that each code maps distinctly to either success or failure.

When considering the availability of an entire user journey, care must be taken to enumerate and measure the ways that users can voluntarily exit the journey before completion.

Other Availability SLIs

Availability is a useful measurement concept for a wide range of scenarios beyond serving requests. The availability of a virtual machine could be defined as the proportion of minutes that it was booted and accessible via SSH, for example.

Sometimes, complex logic is required to determine whether a system is functioning as a user would expect it. A reasonable strategy here is to write that complex logic as code and export a boolean availability measure to your SLO monitoring systems for use in a bad-minute style SLI like the example above.

Specifying a Latency SLI

The latency of a system serving interactive requests from users is an important reliability measure. A system is not perceived as "interactive" by its users if their requests are not responded to in a timely fashion.

Request / Response

The suggested specification for a request/response Latency SLI is:

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

Turning this specification into an implementation requires making two choices: which of the requests this system serves are **valid** for the SLI, and what **threshold** marks the difference between requests that are fast enough and those that are not?

Setting a threshold for fast enough is dependent on how accurately measured latency translates to the user experience. Systems can be engineered to prioritize the *perception* of speed, allowing relatively loose thresholds to be set. Requests may be made in the background by applications, and thus have no user waiting for the response.

It can be useful to have multiple thresholds with different SLOs. When a single threshold is used it often targets long-tail latency. But it can also be useful to target median latency because the translation of perceived latency to displeasure usually follows an S-curve rather than being binary.

Other Latency SLIs

Latency can be equally important to track for data processing or asynchronous work-queue tasks. If you have a batch processing pipeline that runs daily, that pipeline probably shouldn't take more than a day to complete. Users care more about the time it takes to complete a task they queued than the latency of the queue acknowledgement.

One thing to be careful of here is only reporting the latency of long-running operations on their eventual success or failure. If the threshold for operation latency is 30 minutes but the latency is only reported when it fails after 2 hours, there is a 90 minute window where that operation was missing expectations but not *measurably* so.

Specifying a Quality SLI

If your system has mechanisms to trade off the quality of the response returned to the user for e.g. lower CPU or memory utilization, you should track this graceful degradation of service with a quality SLI. Users may not be consciously aware of the degradation in quality until it becomes severe, but their subconscious perceptions may still have an impact on your business if e.g. degrading quality means serving less relevant ads to users, reducing click-through rates.

Request / Response

The suggested specification for a request/response Quality SLI is:

*The proportion of **valid** requests served without **degrading quality**.*

Turning this specification into an implementation requires making two choices: which of the requests this system serves are **valid** for the SLI, and how to determine whether the response was served with **degraded quality**.

In most cases, the mechanism used by the system to degrade response quality should also be able to mark responses as degraded or increment metrics to count them. It is therefore much easier to express this SLI in terms of "bad events" rather than "good events".

Similar to measuring latency, if the quality degradation falls along a spectrum it can be useful to set SLO targets at more than one point from that spectrum. For a somewhat contrived example of this, consider a service that fans out incoming requests to 10 optional backends, each with a 99.9% availability target and the ability to reject requests when they are overloaded. You might choose to specify that 99% of service responses must be served with no missing backend responses and 99.9% must be served with no more than one missing response.

Specifying a Freshness SLI

When batch-processing data, it is common for the utility or relevance of the outputs to degrade over time as new input data is generated by the system or its users. The users, in turn, have expectations that the outputs of the system are up-to-date with respect to those inputs. Data processing pipelines must be run regularly or perhaps even rebuilt to process small increments of input data continuously to meet those expectations. A freshness SLI measures the system's performance against those expectations and can inform those engineering decisions.

Data Processing

The suggested specification for a data processing Freshness SLI is:

> *The proportion of **valid** data updated **more recently** than a **threshold**.*

Turning this specification into an implementation requires making two choices: which of the data this system processes are **valid** for the SLI, and the **threshold** after which generated data should be considered stale.

For a batch-processing system, freshness can be approximated as the time since the completion of the last successful processing run. More accurate freshness measurements for batch systems usually require augmenting processing systems to track generation and/or source age timestamps. Freshness for incremental streaming processing systems can also be measured with a watermark that tracks the age of the most recent record that has been fully processed.

Measuring Data Freshness as Response Quality

Stale serving data is a common way for response quality to be degraded *without* a system making an active choice to do so. Measuring stale data as degraded response quality is a useful strategy: if no user accesses the stale data, no expectations around the freshness of that data can have been missed. For this to be feasible, the parts of the system responsible for generating the serving data must also produce a generation timestamp that the serving infrastructure can check against a freshness threshold when it reads data.

Specifying a Coverage SLI

A coverage SLI functions similarly to an availability SLI when processing data in a system. When users have expectations that data will be processed and the outputs made available to them, you should consider using a coverage SLI.

Data Processing

The suggested specification for a data processing Coverage SLI is:

*The proportion of **valid** data processed **successfully**.*

Turning this specification into an implementation requires making two choices: which of the data this system processes are **valid** for the SLI, and how to determine whether the processing of a particular piece of data was **successful**.

For the most part, the system doing the processing of the data ought to be able to determine whether a record that it began processing was processed successfully and output counts of success and failure. The challenge comes from identifying those records that should have been processed but were skipped for some reason. This usually requires some way of determining the number of valid records that resides outside of the data processing system itself, perhaps by running the equivalent of `COUNT(*)` on the data source.

Specifying a Correctness SLI

In some cases it can be important to measure not just that a processing system processes all the data it should have, but that it produces the correct outputs while doing so. Correctness is something best ensured proactively via good software engineering and testing practice, rather than detected reactively *in absentia*. However, when users have strong expectations that the data they are accessing has been generated correctly—and have ways of independently validating that correctness—having an SLI to measure correctness on an ongoing basis can be valuable.

Data Processing

The suggested specification for a data processing Correctness SLI is:

*The proportion of **valid** data producing **correct output**.*

Turning this specification into an implementation requires making two choices: which of the data this system processes are **valid** for the SLI, and how to determine the **correctness** of output records.

For a correctness SLI to be useful, the method of determining correctness needs to be independent of the methods used to generate the output data. Otherwise, it is probable that any correctness bugs that exist during generation will also exist during validation, preventing the detection of the resulting incorrectness by the SLI. A common strategy is to have "golden" input data that produces known-good outputs when processed. If this input data is sufficiently representative of real user data, and is designed to exercise most of the processing system's code paths, then this can be sufficient to estimate overall correctness.

Measuring SLIs

Broadly speaking, there are five ways to measure an SLI, each with their own set of advantages and disadvantages. Like many engineering decisions there is no one right choice for all situations, but with a good understanding of the trade-offs involved it is possible to choose SLI implementations that meet the requirements of the system.

These classes of measurement methods are presented in decreasing order of their distance from the user. In general, an SLI should measure the user experience as closely as possible, so proximity to the user and their interactions with the system is a valuable property.

Server-side Logging

Processing server-side logs of requests or processed data to generate SLI metrics.

Pros	Cons
+ Existing request logs can be processed retroactively to backfill SLI metrics. + Complex user journeys can be reconstructed using session identifiers. + Complex logic to derive an SLI implementation can be turned into code and exported as two much simpler "good events" and "total events" counters.	– Application logs will not contain requests that did not reach servers. – Processing latency makes logs-based SLIs unsuitable for triggering an operational response. – Engineering effort is needed to generate SLIs from logs; session reconstruction can be time-consuming.

Application Server Metrics

Exporting SLI metrics from the code that is serving requests from users or processing their data.

Pros	Cons
+ Often fast and cheap (in terms of engineering time) to add new metrics. + Complex logic to derive an SLI implementation can be turned into code and exported as two much simpler "good events" and "total events" counters.	– Application servers are unable to see requests that do not reach them. – Measuring overall performance of multi-request user journeys can be difficult if application servers are stateless.

Front-end Infrastructure Metrics

Utilizing metrics from load-balancing infrastructure (e.g. GCP's layer 7 load balancer) to measure SLIs.

Pros	Cons
+ Metrics and recent historical data most likely already exist, so this option probably requires the least engineering effort to get started. + Measures SLIs at the point closest to the user still within serving infrastructure.	– Not viable for data processing SLIs, or in fact any SLIs with complex requirements. – Can only measure approximate performance of multi-request user journeys.

Synthetic Clients (Probers) or Data

Building a client that sends fabricated requests at regular intervals and validates the responses. For data processing pipelines, creating synthetic known-good input data and validating outputs.

Pros	Cons
+ Synthetic clients can measure all steps of a multi-request user journey. + Sending requests from outside your infrastructure captures more of the overall request path in the SLI.	– Approximates user experience with synthetic requests. – Covering all corner cases is hard and can devolve into integration testing. – High reliability targets require frequent probing for accurate measurement. – Probe traffic can drown out real traffic.

Client Instrumentation

Adding observability features to the client the user is interacting with and logging events back to your serving infrastructure that track SLIs.

Pros	Cons
+ Provides the most accurate measure of user experience. + Can quantify reliability of third parties, e.g. CDN or payments providers.	– Client logs ingestion and processing latency make these SLIs unsuitable for triggering an operational response. – SLI measurements will contain a number of factors outside of direct control. – Building instrumentation into the client can involve lots of engineering work.

Developing SLOs and SLIs

For each **critical user journey**, stack-ranked by **business impact**

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**

Example SLO Worksheet

*Make sure that your SLIs have an **event**, a **success criterion**, and specify **where and how** you record success or failure. Describe your specification as the proportion of **events** that were **good**. Make sure that your SLO specifies both a **target** and a **measurement window**.*

User Journey: Home Page Load

SLI Type: Latency

SLI Specification:

Proportion of **home page requests** that were served in **< 100ms**

(Above, "[home page requests] served in <100ms" is the numerator in the SLI Equation, and "home page requests" is the denominator.)

SLI Implementations:

- Proportion of **home page requests** served in **< 100ms**, as measured from the 'latency' column of the **server log**.
  (**Pros/Cons:** This measurement will miss requests that fail to reach the backend.)
- Proportion of **home page requests** served in **< 100ms**, as measured by **probers** that execute javascript in a browser running in a virtual machine.
  (**Pros/Cons:** This will catch errors when requests cannot reach our network, but may miss issues affecting only a subset of users.)

SLO:

99% of home page requests in **the past 28 days** served in < 100ms.

EXAMPLE SERVICE

Stoker Labs Inc.

Mission Statement

Our company mission is to *"replace conflict with games"*. The tribalism and division we see throughout society becomes harmful when people take life too seriously. We aim to provide an outlet for the competitive urges and us-versus-them mentality so central to human nature via the medium of mobile video gaming. We firmly believe that providing ways for people to sublimate these urges in a manner that is fun rather than psychologically and physically harmful will bring about a more cooperative and successful world.

Our Game: Tribal Thunder

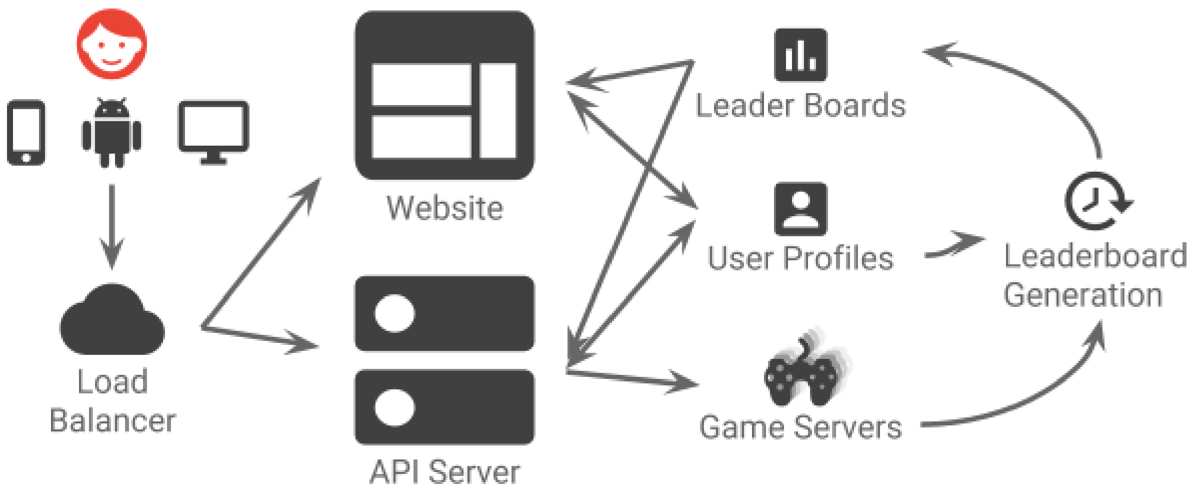
The rise of the vampires has taken a devastating toll on humanity, forcing those who survived to cluster together in the few remaining habitable regions far from previous centres of civilization. As the leader of a tribe of survivors, you must recruit people to your cause, secure and upgrade your settlement, raid vampire-occupied cities, and battle other tribes for control of resources.

The game world is split up into a number of areas with varying rewards and challenges. Access to areas with better rewards is gated by overall play-time, settlement size and in-game currency expenditure. Each area has its own leaderboard ranking the top tribes.

We have around 50M 30-day active users playing, with between 1M and 10M players online at any given time. We add new world areas once per month, which drives a spike in both traffic and revenues.

The primary revenue stream stems from the exchange of real-world money for in-game currency via in-app purchases. Players can earn currency without paying for it by winning PvP battles, playing mini-games, or over time via control of in-game resource production. Players can spend in-game currency on settlement upgrades, defensive emplacements for PvP battles, and by playing a recruitment mini-game that gives them a chance of recruiting highly-skilled people to their tribe.

Service Architecture

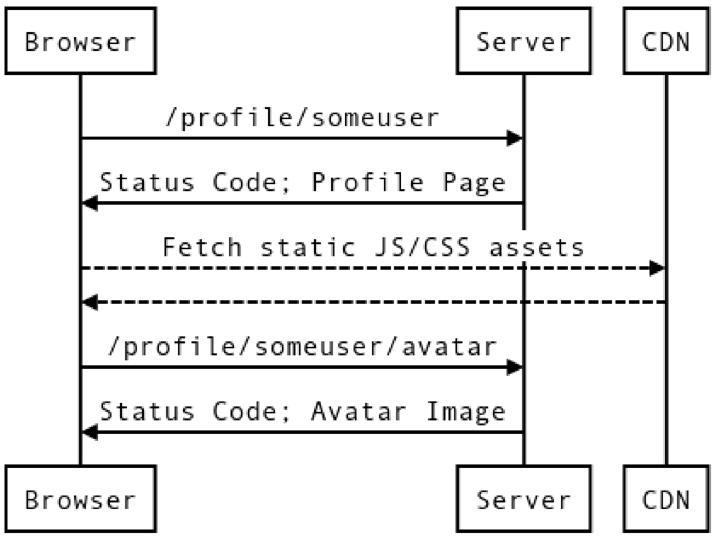


The game has both a mobile client and a web UI. The mobile client makes requests to our serving infrastructure via JSON RPC messages transmitted over RESTful HTTP. It also maintains a web socket connection to receive game-state updates. Browsers talk to the web servers via HTTPS. Leaderboards are updated every 5 minutes.

User Journeys

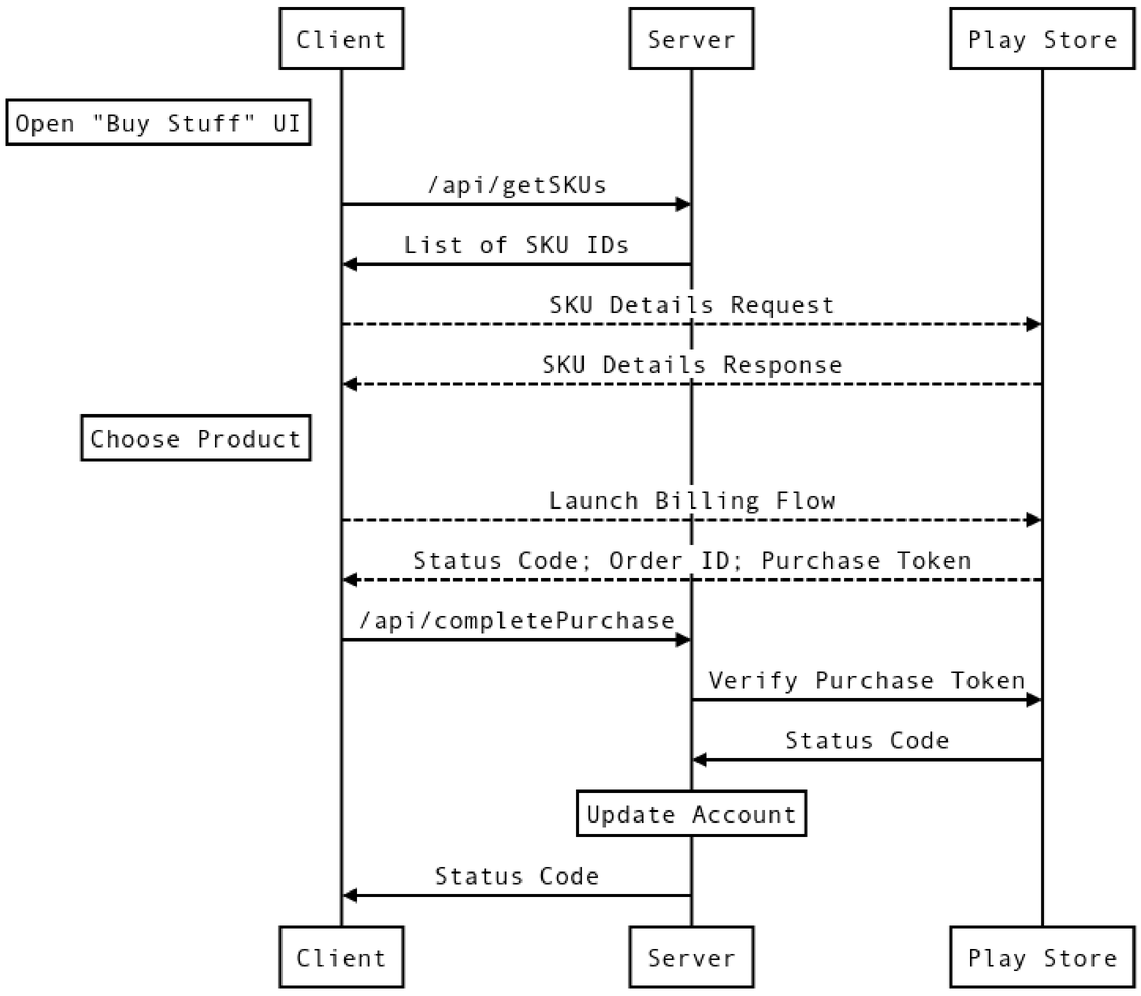
View Profile Page

Players can log in to their game account, view their settlement and make profile changes from a web browser. A player loading their profile page is a simple journey that we will go through together in the workshop.


Browser
Server
CDN
/profile/someuser
Status Code; Profile Page
Fetch static JS/CSS assets
/profile/someuser/avatar
Status Code; Avatar Image
Browser
Server
CDN

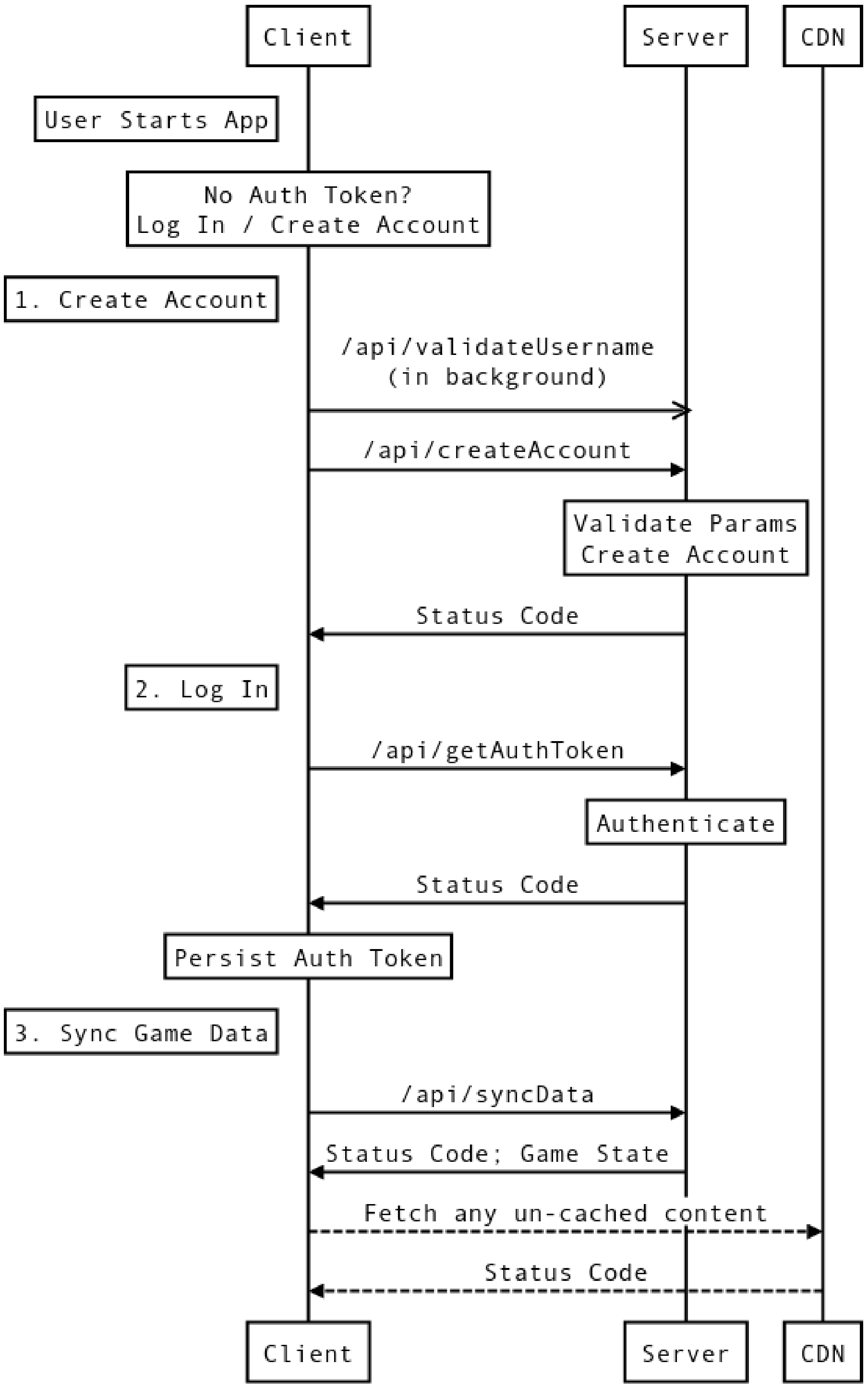
Buy In-Game Currency

Our most important user journey is the one that generates all our revenue: users buying in-game currency via in-app purchases. Requests to the Play Store are only visible from the client. We see between 0.1 and 1 completed purchase every second; this spikes to 10 purchases per second after the release of a new area as players try to meet its requirements.



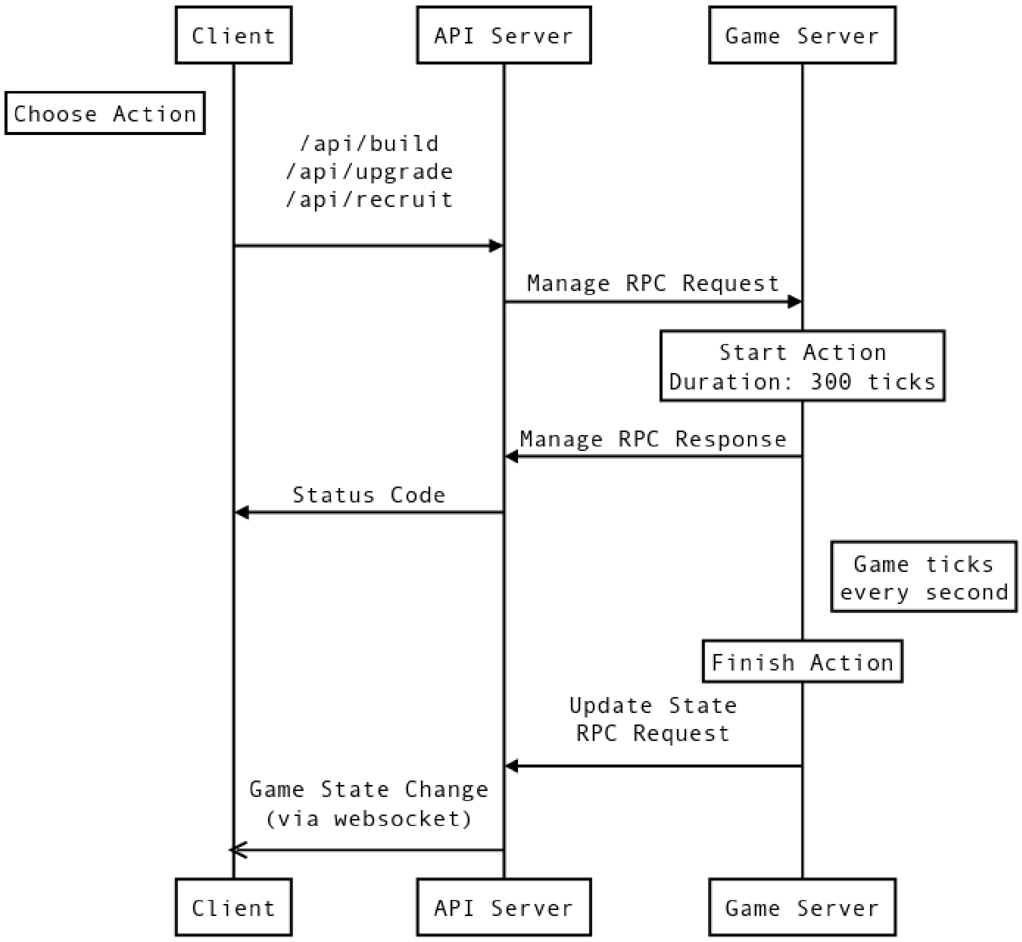
App Launch

There are three parts to the app launch process, depending on whether the user already has an account and whether that account has been previously accessed on the current device. Account creation and auth token request rates are low, but we see between 20 and 100 QPS of syncData requests, spiking to 1000 after the release of a new area.



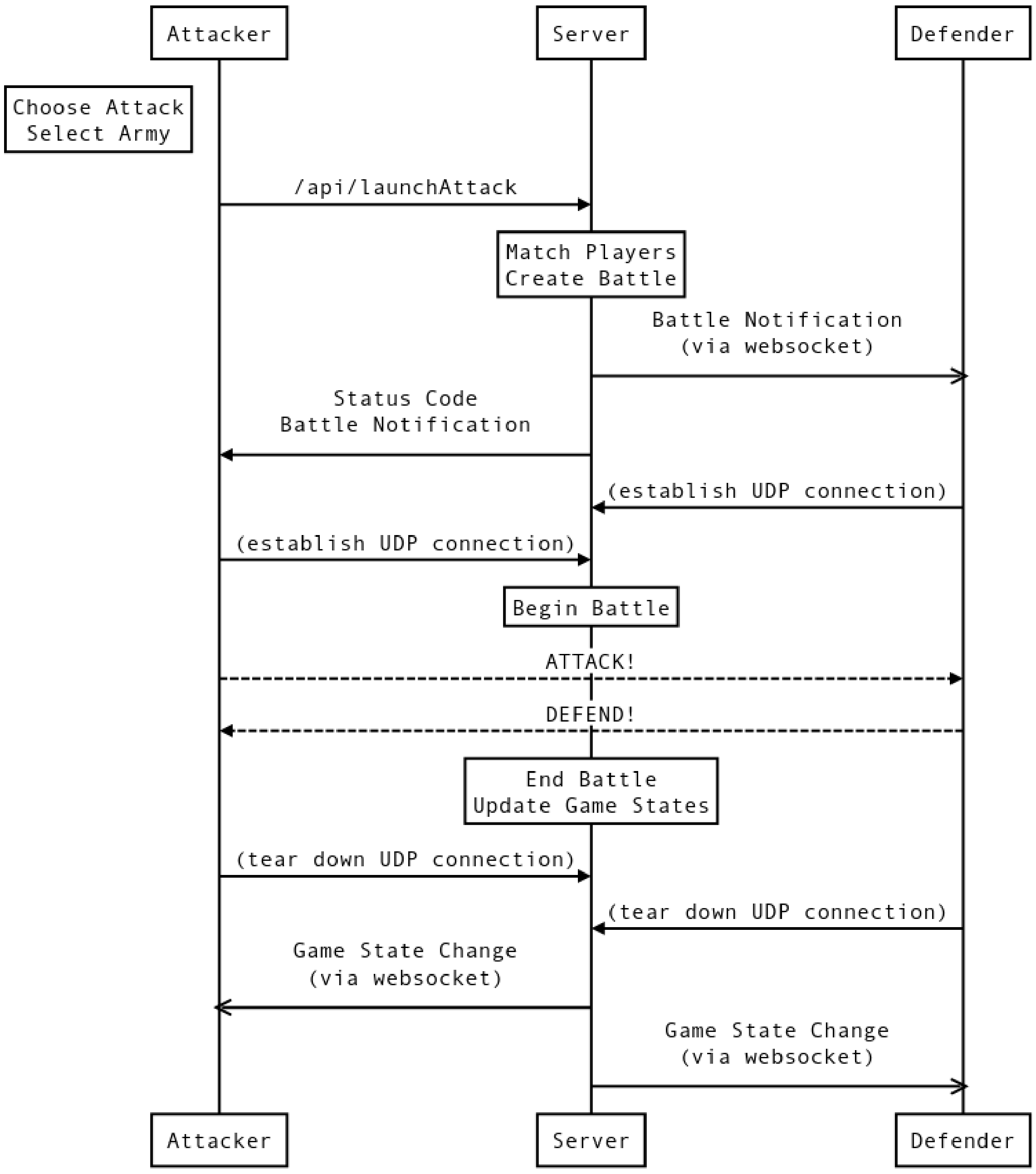
Manage Settlement

Settlement management is a trio of relatively simple API requests. Upgrading settlements, building defenses and recruiting tribespeople consumes wall-clock time as well as in-game currency. Players spend a lot of time managing their settlements: we see 2000-3000 requests per second across all these API endpoints, spiking up to 10,000/s. If the game servers consistently take more than a second to compute a tick, users will notice their buildings not completing on time.



Play PvP Battle

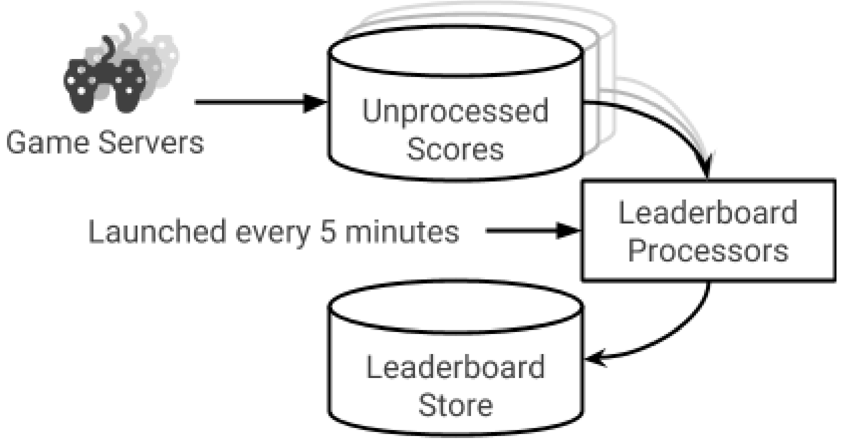
Launching an attack on another player spins up an RTS-like tower defence battle where the attackers troops try to overrun the defenders' emplacements. The defender can deploy some of their troops to aid their defence. Both sides get points in proportion to the amount of damage they dealt to the opposition. We see around 100 attacks launched every second.



Generate Leaderboards

Competition for the top spots is fierce because players in a given area primarily battle each other and are of similar skill levels. PvP battles are scheduled across the pool of game servers on a least-loaded basis: there is no guarantee that a battle between two players from a given game area will be hosted on the game server which also ticks their settlements.

Battle scores are written to an append-only log of unprocessed scores on the local game server when the battle ends. A set of background tasks run every five minutes to rotate the append-only logs, archive old logs to a cloud storage service, aggregate the unprocessed scores and merge them into the leaderboards.



Postmortem: Blank Profile Pages!

Impact

From 08:43 to 13:17, users accessing their profile pages received incomplete responses. This rendered them unable to view or edit their profile.

Root Causes and Trigger

The proximate root cause was a bug in the web server's handling of unicode HTML templates. The trigger was commit a6d78d13, which changed the profile page template to support localization, but at the same time accidentally introduced unicode quotation marks (U+201C “, U+201D ”) into the template HTML. When the web server encountered these instead of the standard ascii quotation mark (U+0022 "), the template engine aborted rendering of the output.

Detection

Because the aborted rendering process did not throw an exception, the HTTP status code for the incomplete responses was still `200 OK`. The problem thus went undetected by our SLO-based alerts. The support and social media teams manually escalated concerns about a substantially increased level of complaints relating to the profile page at 12:14.

Lessons Learned

Things that went well:

- Support/SM teams were able to find the correct escalation path and successfully contact ops team.

Things that went poorly:

- HTTP status code SLIs cannot detect incomplete responses.
- Web server uses a severely outdated vendored version of the templating engine with substantially broken unicode support.

Where we got lucky:

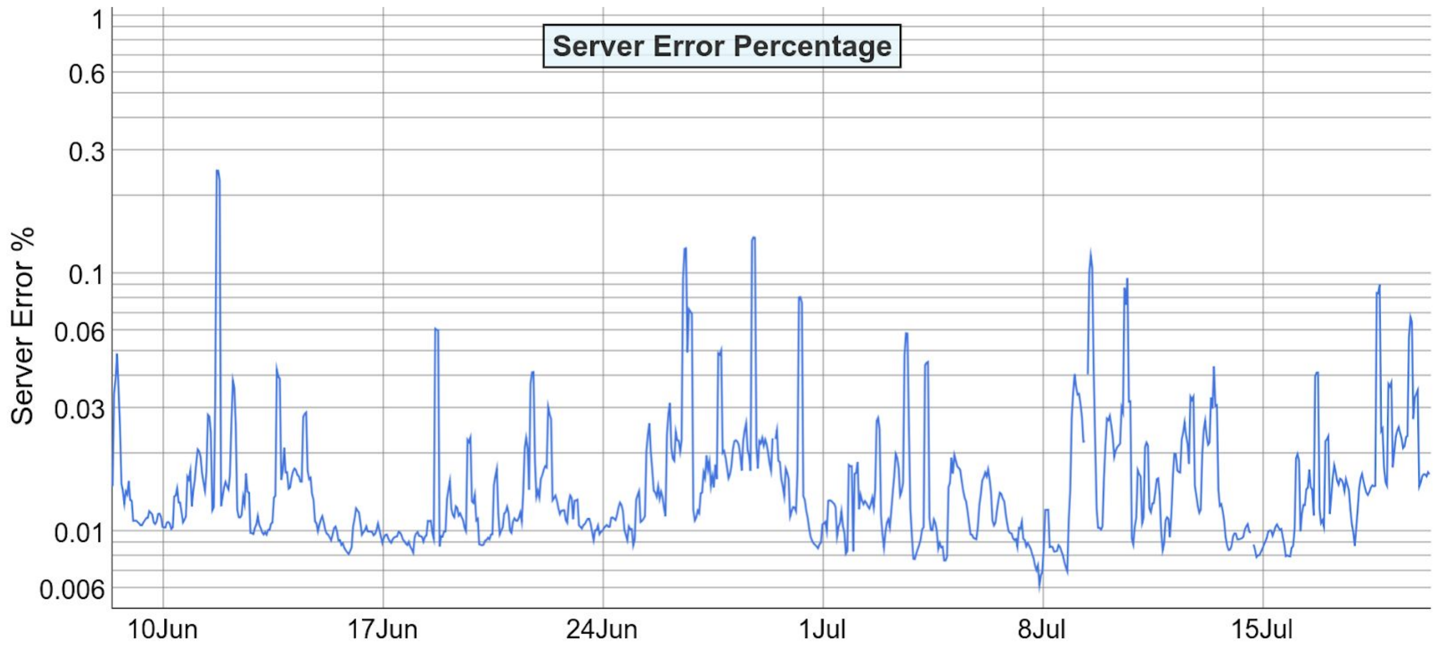
- User profile page is relatively unimportant to our revenue stream.

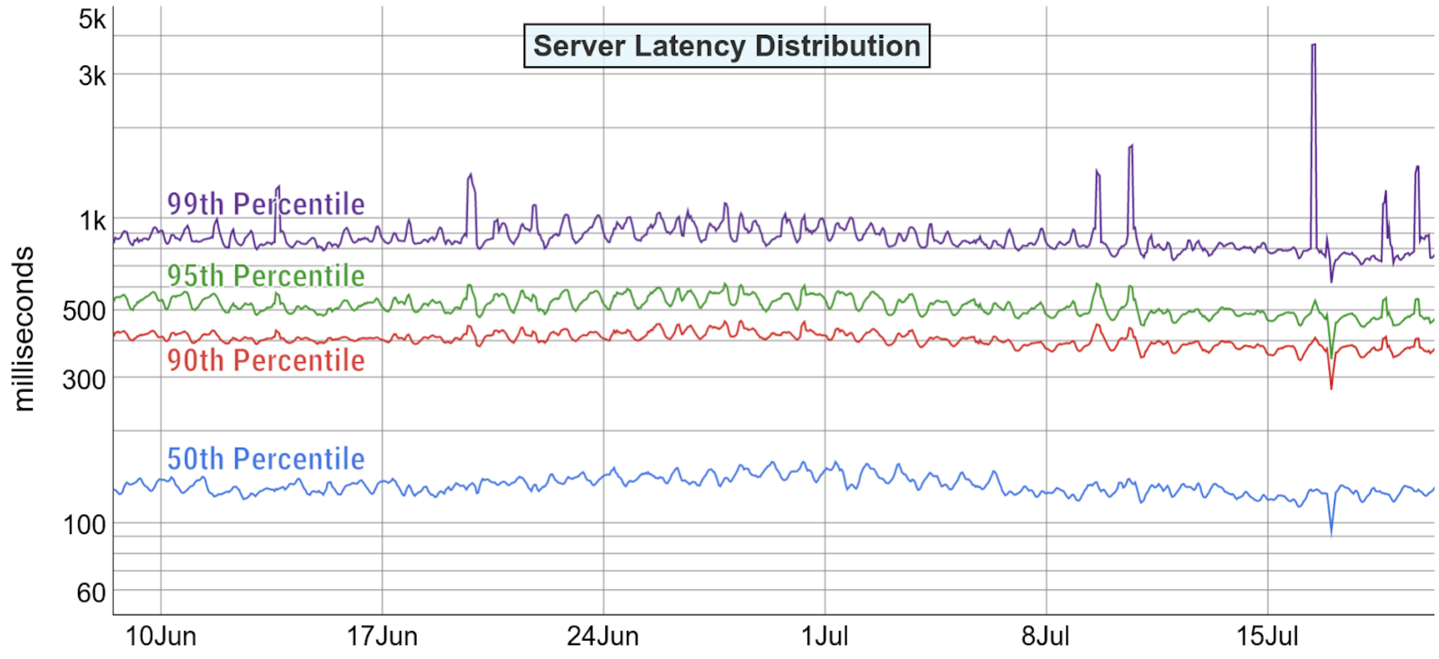
Action Items

… to be determined!

Profile Page Errors and Latency

NOTE: both these graphs have logarithmic Y axes!


Server Error Percentage
Server Error %
10Jun
17Jun
24Jun
1Jul
8Jul
15Jul


Server Latency Distribution
milliseconds
5k
3k
1k
99th Percentile
95th Percentile
90th Percentile
50th Percentile
10Jun
17Jun
24Jun
1Jul
8Jul
15Jul