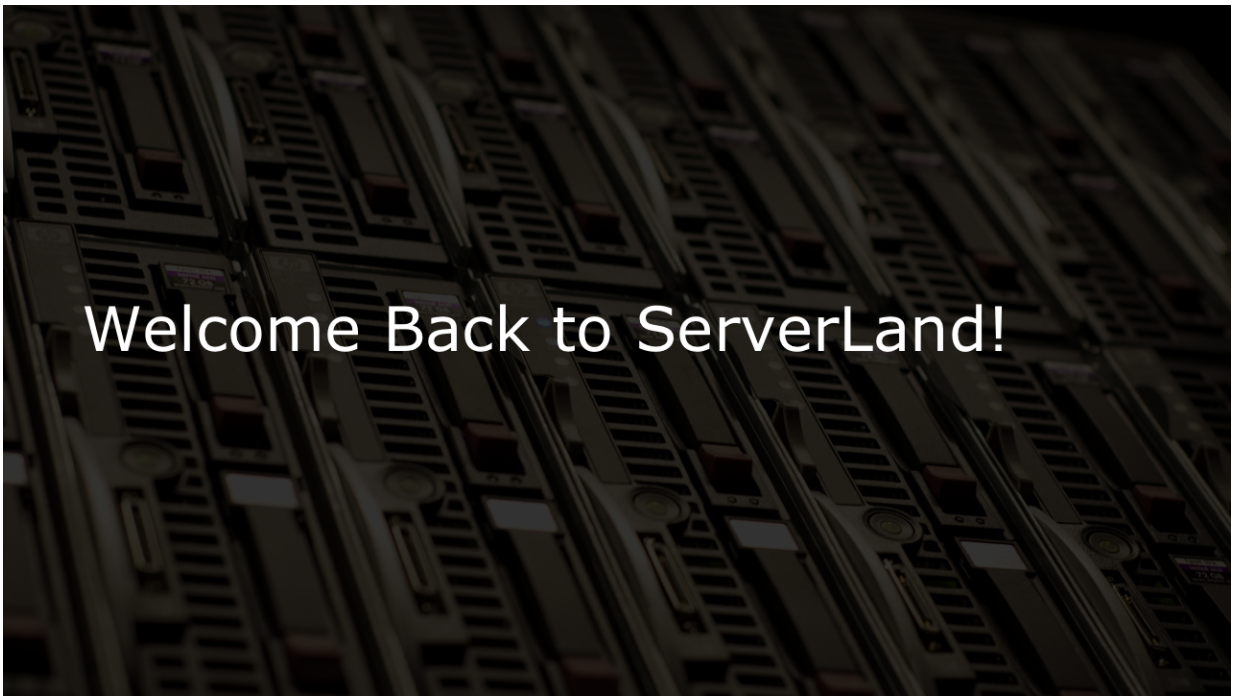

GDC
Start Scaling Your Servers!
Sela Davis
Senior Software Engineer, Vreal
Jennie Lees
Senior Software Engineer, Riot Games
GAME DEVELOPERS CONFERENCE® | MARCH 19-23, 2018 | EXPO: MARCH 21-23, 2018 #GDC18
UBM



- why this talk is different from last year
  - for *server engineers* and those who work with them
- what you will learn and who this is for
- pls dont DDOS yourself on launch day

Photo credit: https://www.flickr.com/photos/zagrobot/2731084578

- Architecture
  - 101: Client-Server Patterns
  - 201: Preparing For Scale
  - 301: 100M DAU? NP
- Deployment
  - 101: Getting Bits on Atoms
  - 201: Global Simultaneous Rolling Zero-Downtime Deploys, Oh My
- Operations
  - 101: Instrumentation and Triage
  - 201: Operations At Scale

This provides a framework for the speakers to own various sections.

https://www.flickr.com/photos/dancedancedancephotography/3356807821/

Introducing our Case Study

The image is programmer art - obviously

Our case study game has some important qualities which we'll call upon throughout this talk. As you might guess, it's a multiplayer battle royale game that can handle hundreds of concurrent players and matches. It's launched globally and is incredibly

popular, handling millions of users with spikes whenever we release new content.

What are you building and how are you building it?

Side of the picture

Photo Credit: Jennie Lees

(change: jennie -> sela)

AUTHORITATIVE SERVERS

If you saw our talk last year, we touched briefly upon security from the client's perspective. There's always a server component as well, which most server engineers in the room probably already know about. We mention this mostly for completion, and for the non-server engineers in the room.

There is some text about authoritative servers here and shared model. NEVER TRUST THE CLIENT.

Running a simulation on the server adds latency to all actions in the game, so you have to consider the tradeoffs involved. Will clients see visible discrepancies? Many games will perform predictions on the client and attempt to reconcile with the server; in any disagreement, the client should reset. At DropForge games, we ran our game logic on both the client and the server and sent inputs to the server. The client moved forward at the player's pace, showing no latency, and sent up a checksum. Checksum mismatches would result in the client resetting as a form of anti-cheat and an attempt to reduce desync issues.

If you want to learn more, check out Timothy Ford's excellent GDC 2017 talk on Overwatch's architecture, where he goes into detail about how they handle

reconciliation between the client's predictions and the server's authoritative results.

Charlie Bro

https://www.flickr.com/photos/lajuna/11119176759/in/

When you're thinking about security, you might want to think about what you can do on your requests. You also might want to think about how they affect your ability to scale upward. Take SSL, for instance. You probably want to use SSL to encrypt your requests, but what are the consequences? How much time will you spend establishing connections for each microservice? (Remember, you need to back-and-forth a few times

during the SSL handshake. In China, that can be veeeery slooooow.) And how much time will you spend doing crypto on the server? Ultimately, SSL > insecure connections, but it's worth understanding the cost.

The big advantage is reducing the information the client can see and modify. Any dedicated client can be malicious, but this is equivalent to locking your front door. We can pick the lock with a MITM certificate, but it becomes much harder. It also reduces the ability for malicious entities to learn what data is stored in your client vs. your server. You're also not limited to TCP - DTLS will allow you to send UDP packets over a secure connection.

http://info.ssl.com/article.aspx?id=10241 https://www.ssl.com/article/ssl-tls-handshake-overview/

How do the different security options

affect your ability to scale?

- Packet hashing
- SSL
- Authenticated requests

How do you authenticate with your own servers? How do you identify a user on signin? You can either federate with another system (for example, on Steam, any player of your game has an account on their platform) or implement it yourself. Security is scary - you don't want to mess this up - so I strongly recommend using your 3rd party platform holders as much as possible. But there can be good reasons to have

your own account systems, such as API access to 3rd party websites.

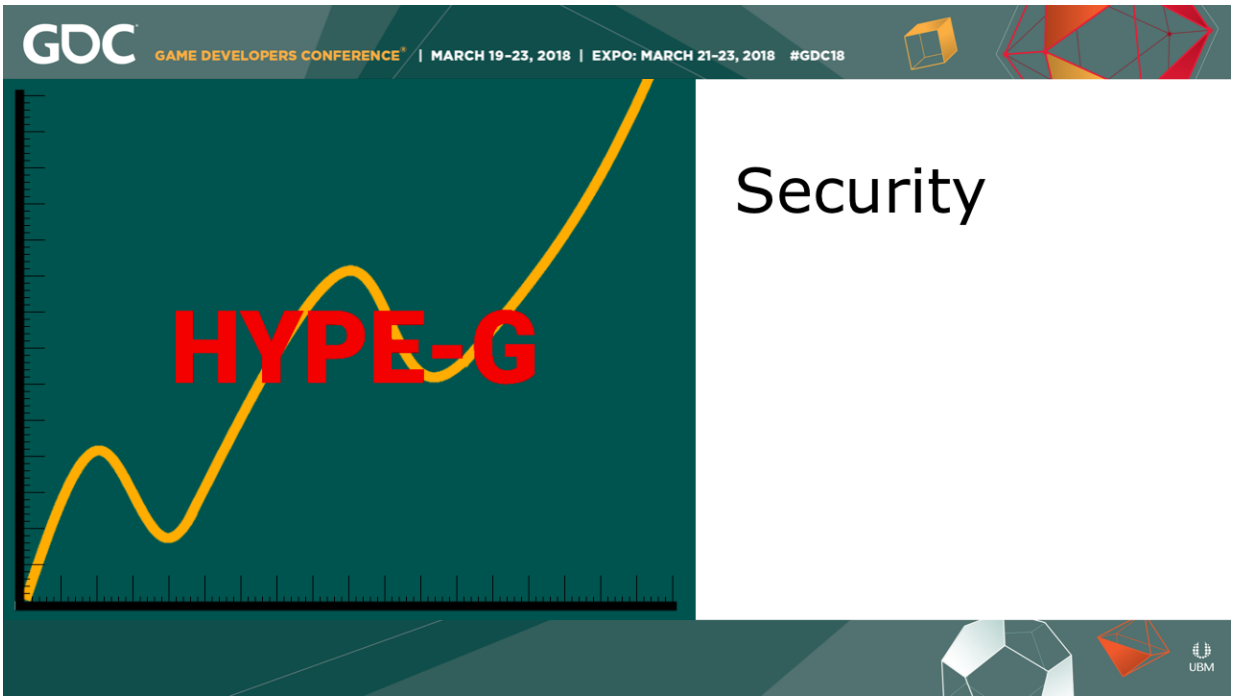
Either way, getting a user's authentication token can be useful in a number of ways. If you want to store any data for a user, such as purchases or bans, your auth token will let you assert that they are who they say they are. This will make them happy if they want to see the entitlements they have, or make them very sad if they are banned. You probably don't want to check all of these pieces of data on each request, but you'll find the right times to perform these checks. For instance, you probably only need to check a user's ban status in a multiplayer game if they are attempting to matchmake.

However, authentication is a bottleneck. If users can't sign in, what do you do? Do you reject all requests? Do you let them continue with limited permissions (say, read-only access to public data)?

This is something you need to determine on a case by case basis, but typically, the answer is “somebody is getting paged”.

OAuth is a common protocol used by a number of platforms. Twitch is a great example: https://dev.twitch.tv/docs/authentication (and https://dev.twitch.tv/get-started for samples). For Twitch, OAuth allows users to use their Twitch credentials to interact with the platform.

https://oauth.net/



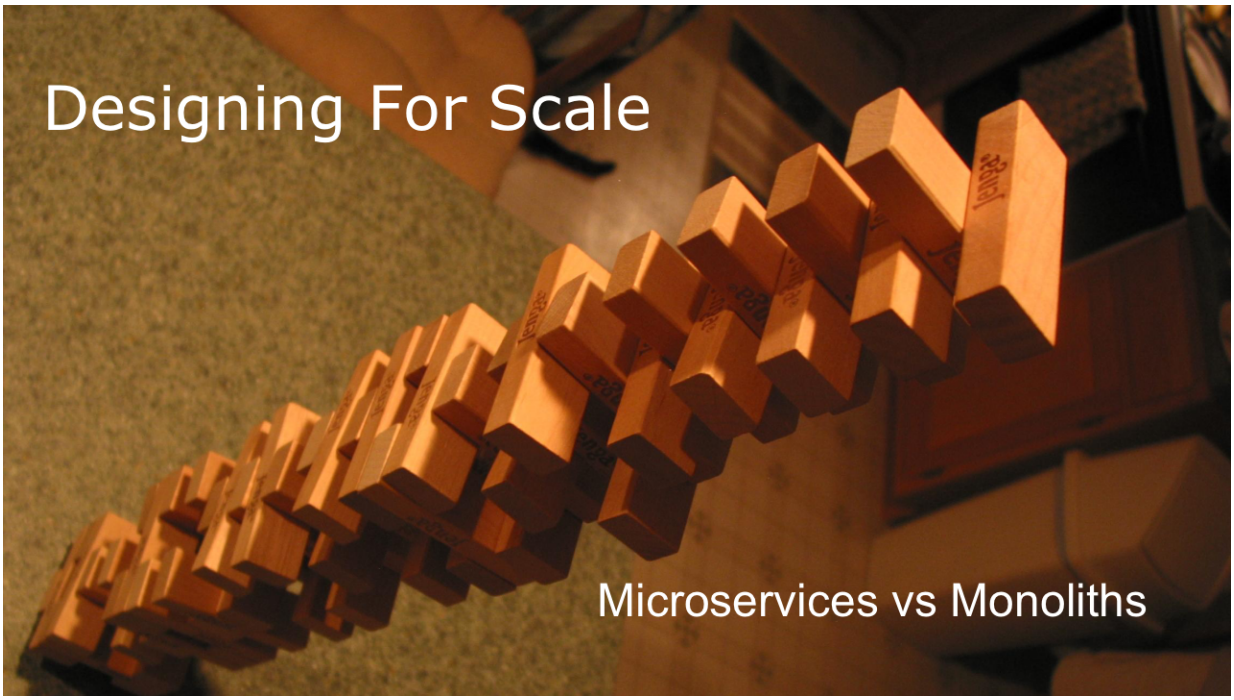
For Hype-G, we decided to build an authoritative server to fend off malicious clients from the beginning. Naturally, we simulated as much on the client-side as we could, but we decided to verify every action on the server by either performing all of the work on the server or by allowing a client to attempt an action, see the result, and roll it back if it failed to checksum on the server.

We authenticate users in our own system using in-house accounts, but we federate with our two released platforms - Xbox and Steam. Because these are tied to our own accounts, we can ban users ourselves even if a platform holder doesn't want to take any action. Likewise, if a user is malicious and is marked as a cheater by the platform (say, a VAC ban), we know they will be out of our system as well.

Users' security tokens are refreshed every few matches, which distributes the load on our token minting service while still ensuring that cheaters are ejected from our systems on a regular cadence. We also check their ban status at the beginning of a match. And because Hype-G is a multiplayer-only game, we don't have to worry about offline users doing anything they shouldn't.

Our two platforms enforce that users are on the latest version, so we don't have

to worry about old, stale, out of date versions. We keep both our client and server OSS up to date and patch when necessary.



- Microservices:
  - Small responsibility per service
  - Independent updates and deployments
  - Independent codebases
  - Lots of time spent communicating
  - Potential state differences between them
- Monoliths:
  - One server to rule them all
  - Requires a deployment for every change

- Operations never leave the machine, simplifying things

Clearer on when microservices and monoliths make sense

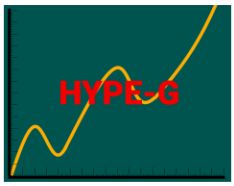
https://www.gdcvault.com/play/1024442/-Guild-Wars-Microservices-and
http://www.dwmkerr.com/the-death-of-microservice-madness-in-2018/

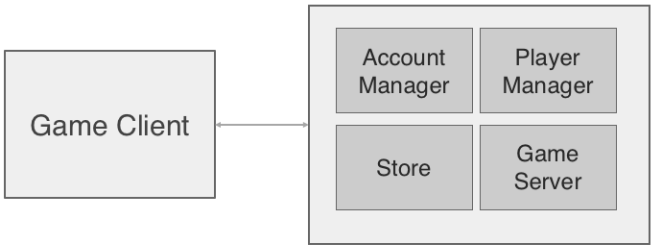
Why talk about this now? Because it's still an interesting technical decision!

Also - When to make things microservices
eg. Does the avatar become its own service or is it in player manager

PHOTO CREDIT
https://www.flickr.com/photos/chesswithdeath/307849468/
Jenga photo
https://www.flickr.com/photos/roblee/2697052/

(change: sela -> jennie)

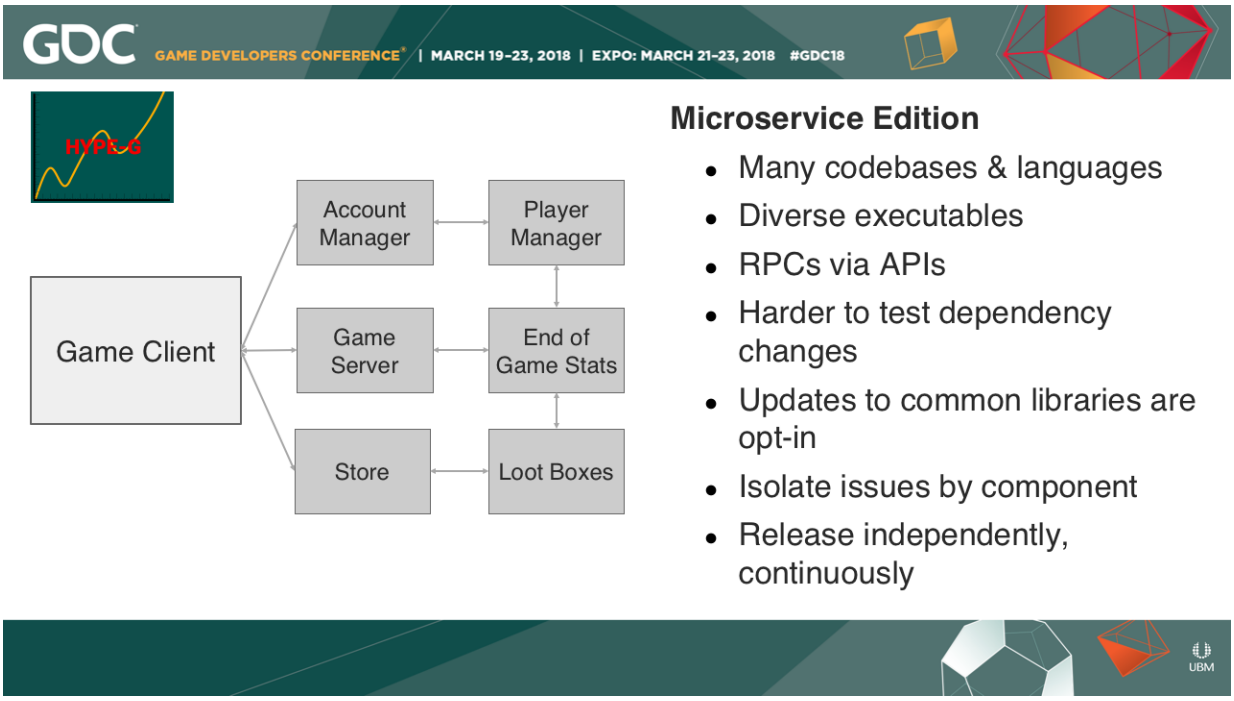




Monolith Edition

- One platform codebase
- One platform executable
- Exposes all API calls
- Uses internal RPCs
- Build breaks if dependencies are not met
- Updates to common libraries update globally
- Difficult to isolate issues
- Release slowly, all at once

- HYPEG "REAL EXAMPLE" - Matchmaking/Game server breaking down as we add more features eg game modes, spectator

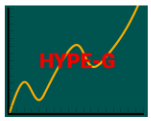


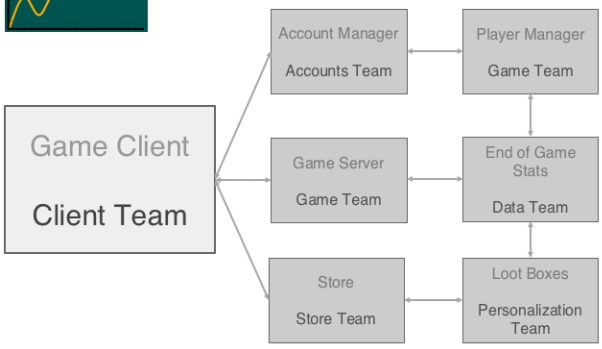
NOTE: Mention languages more explicitly. E.G. Riot's stack

And it's this last point that's really important here, and a huge advantage of microservices. While codebase diversity and being able to use the right tool for the job are attractive, they don't drive ROI. Whereas having the teams that own each component empowered to release their component independently and

frequently, within a wider release process of course, gets stuff to players - the new battle map, the bugfix - much faster. But, some of you may have spotted that the fact it’s harder to test dependency changes makes this more risky than in the monolith model, so you better have some good testing in place.

Hot spots become hard because there are hundreds of fan out requests





Microservices in Practice

A player doesn't receive their expected loot box after levelling up when a game ends.

Who gets paged first?

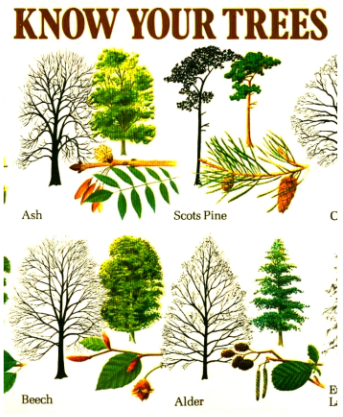
Whose problem is it really?

Wow, there are lots of teams involved here

paging - Hint - it's not the team whose thing broke. Example walk through when a player doesn't get their loot box at end of game. We realise a few things went wrong but it was actually a write to the PLAYER db that failed.

Preparing For Scale

- Know your service!
  - Hotspots
  - Expensive Calls
  - Calling Patterns
  - Autoscaling Strategy
  - Load Testing

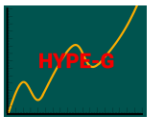


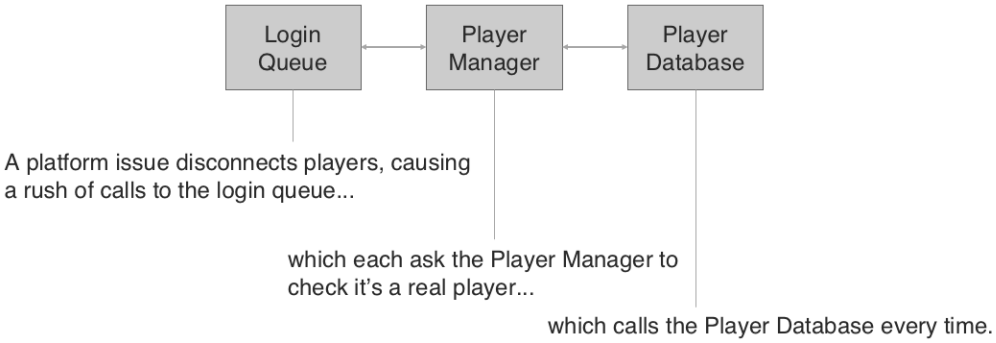
- Where are your hotspots? Which APIs are expensive? What calling patterns do your callers utilize now? (What do you expect new callers might utilize?)
- Autoscaling: add machines when traffic is heavy
- Load testing: know how your server performs at 50%, 60%, 70% utilization.
  Don't assume it scales linearly, even on a single machine.

Trying to write a load test is a good way to communicate with your clients about what typical and extreme call patterns look like

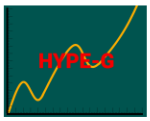
Gatling & Jmeter
Artillery

https://www.flickr.com/photos/30591976@N05/5617460238





Spot the Hotspot!



HOTSPOT 1! How many calls a minute can this service handle?

Login Queue

Player Manager

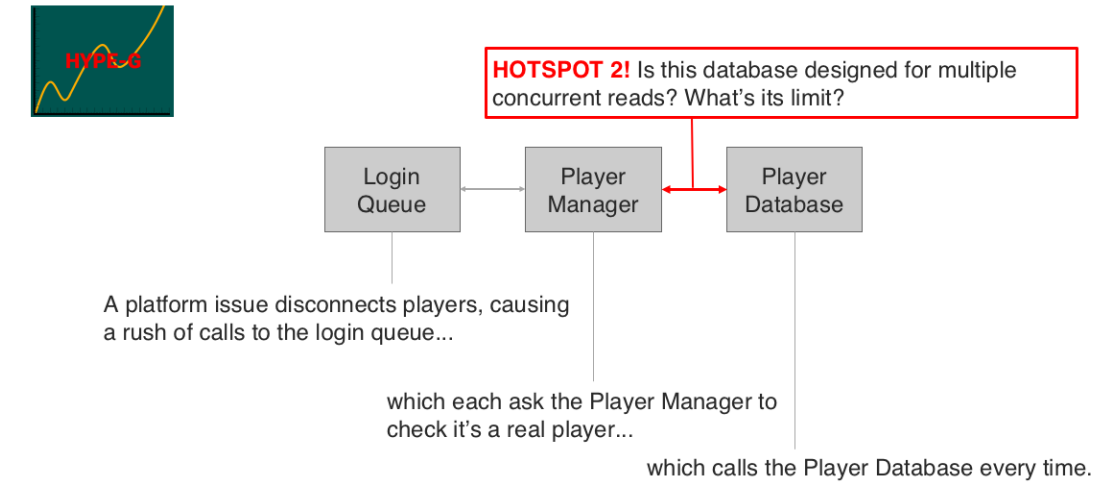
Player Database

A platform issue disconnects players, causing a rush of calls to the login queue...

which each ask the Player Manager to check it's a real player...

which calls the Player Database every time.

Spot the Hotspot!



Spot the Hotspot!

Spot how EVERYTHING is a potential hotspot!

What questions do you ask the other team to identify this

- Hotspot ProTip:
  - Know wtf the service DOES when it's in "hotspot"/"i can't even" mode
  - Have a strategy for dealing with this
    - Do you fast fail
    - Do you fail over
    - Do you expose the failure to players
    - Who needs to know when this happens and are they prepared?

- example strategies from reality

https://www.flickr.com/photos/question_everything/2516118991/

Consider utilizing well-known open-source technology.

- It's battle-tested at scale already.
- It's probably familiar to future hires. And your ops team. And your partner teams. Etc, etc.
- Examples: Kafka, Zookeeper, Redis, Elasticsearch, Hadoop (and many more).

Only build your own if you absolutely have to.

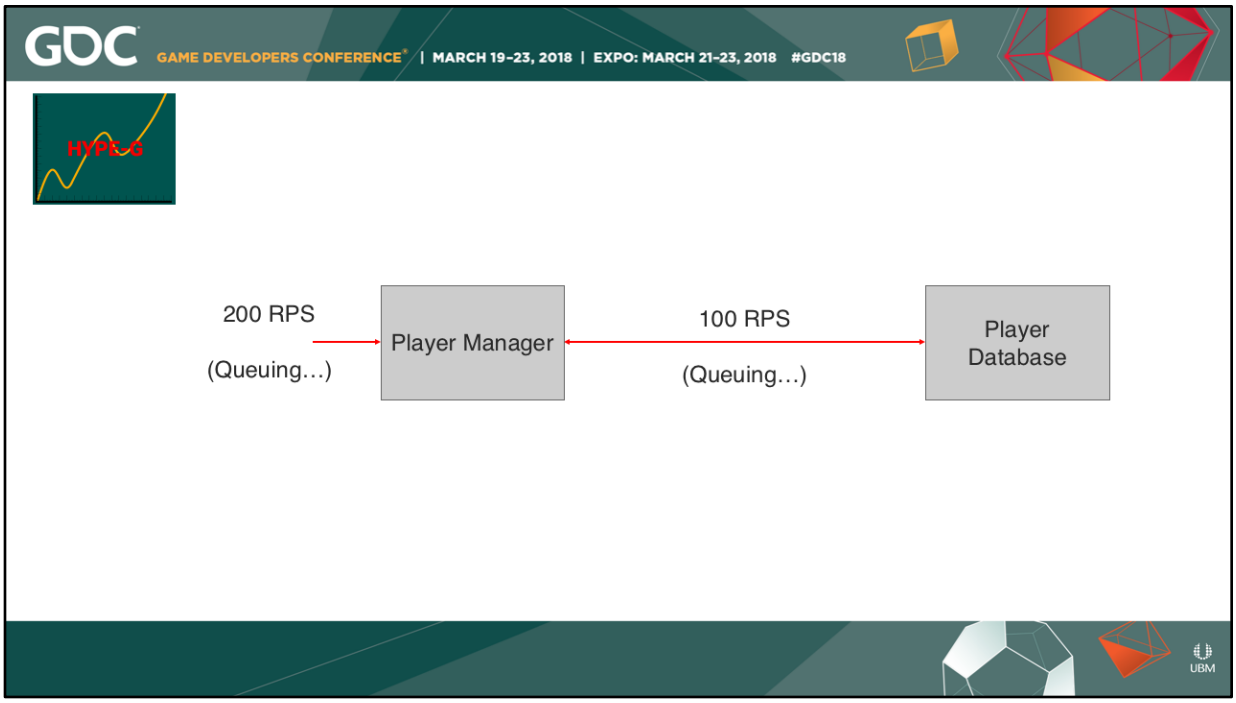
- And please open-source it so others can use it!
- Make sure you have dedicated the appropriate resources to build it AND to maintain it.

http://kafka.apache.org/images/apache-kafka.png
https://github.com/antirez/redis-io/tree/master/public/images

https://www.elastic.co/brand
https://svn.apache.org/repos/asf/zookeeper/logo/
https://svn.apache.org/repos/asf/hadoop/logos/out_rgb/

…these images need some cleanup, but you get the point

(change: jennie -> sela)



Storage and persistence become hard!

- You can't just slam the database on every request.
- Requests will queue.
- Major point of failure.
- Hardware limitations are a reality.

Figure out how to avoid it when you can. Become stateless where possible.

- One of my favorite tricks: some users can deal with stale data. If I update my profile, I want to see it immediately…but you might not need to see it update for a few minutes.

Make sure you don't build yourself into a corner

- If you partition your data into 10 buckets, each with a capacity of 5tb – what do you do with the 51st tb? Sounds like a nightmare of a migration…

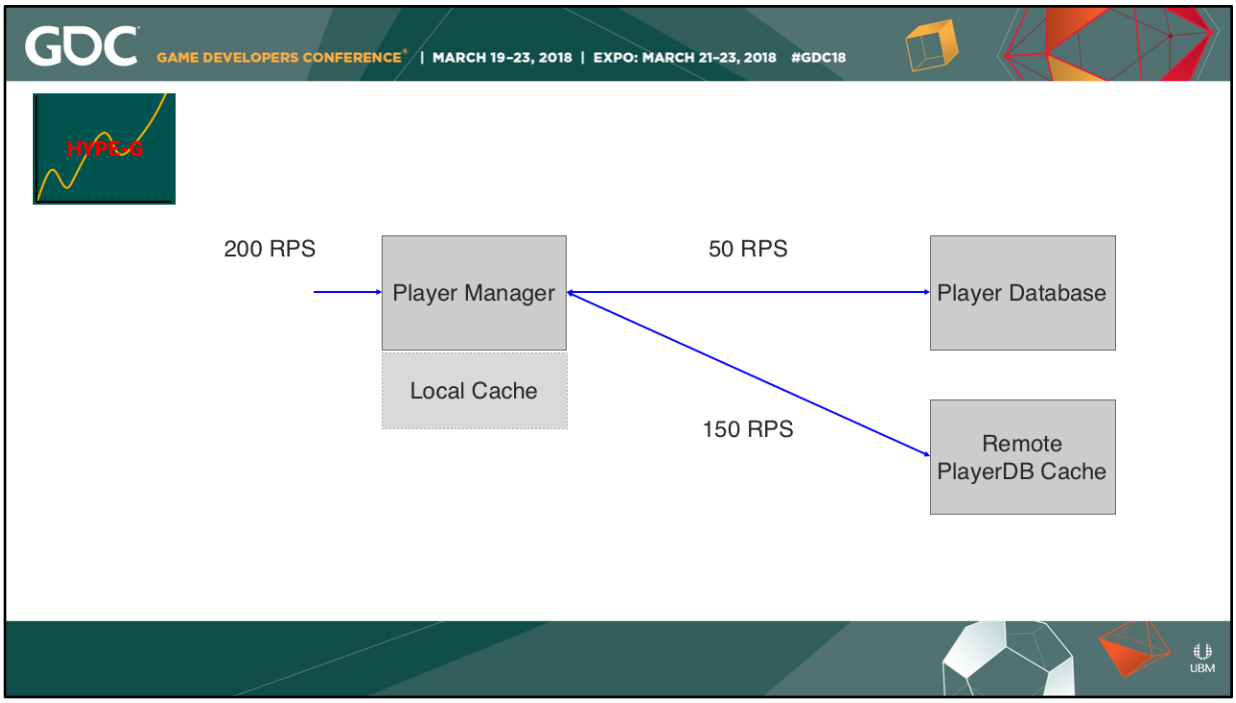
Sometimes you don't have a choice - you need to keep data in memory. Active game sessions are one example; databases and caches are another. When you need to do this, look at your technology options in the open source world. There's no reason to try and do this yourself. They're well tested and it's less you need to support later. Likewise, don't use the new hotness - they may be great, but you never know if they'll fail you at scale and you never know if you'll be supported in a few years. We used FoundationDB at DropForge games, and by the time I arrived at the company Apple had already purchased them and scoured all

information off the internet. Great technology, but it was a pain to maintain at that point.

If we want a story…

At DropForge games (Wargaming), we used FoundationDB as our data store – and continued to use it once it was purchased and scoured from the internet. We discovered scaling issues with our usage right before Smash Squad came out. We mitigated with hardware, but we determined a need for a new data storage for our next project. Because we wanted the opportunity to reuse user data, we were looking at a painful migration (potentially in-place if we hit a high scale on this title), and a need to slot in a new data store based on assumptions made for the previous data store. We considered many options, but opted to move toward a more general solution (DynamoDB) for general data to keep the data layer as simple as possible if we wanted to switch again in the future.

Once data is migrated, the tricky part is maintaining the previous behaviour (specific transactional behaviour, in our case) through a shim. This is where a great test suite comes in handy…



Storage and persistence become hard!

- You can't just slam the database on every request.
- Requests will queue.
- Major point of failure.
- Hardware limitations are a reality.

Figure out how to avoid it when you can. Become stateless where possible.

- One of my favorite tricks: some users can deal with stale data. If I update my profile, I want to see it immediately…but you might not need to see it update for a few minutes.

Make sure you don't build yourself into a corner

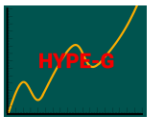
- If you partition your data into 10 buckets, each with a capacity of 5tb – what do you do with the 51st tb? Sounds like a nightmare of a migration…

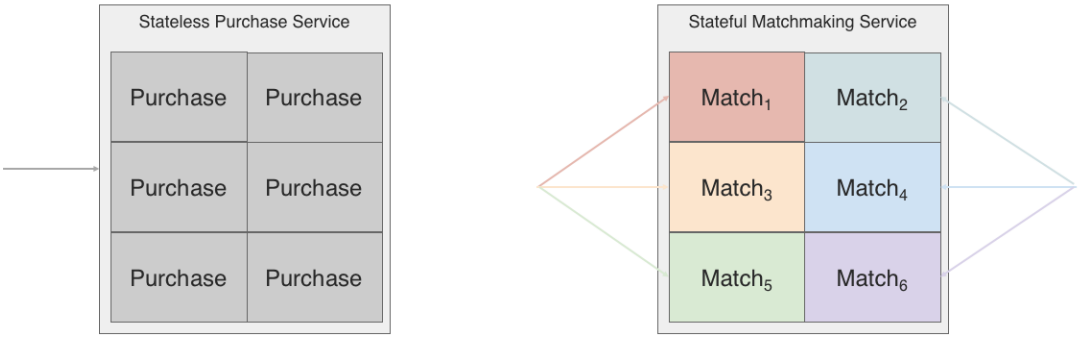
If we want a story…

At DropForge games (Wargaming), we used FoundationDB as our data store – and continued to use it once it was purchased and scoured from the internet. We discovered scaling issues with our usage right before Smash Squad came out. We mitigated with hardware, but we determined a need for a new data storage for our next project. Because we wanted the opportunity to reuse user data, we were looking at a painful migration (potentially in-place if we hit a high scale on this title), and a need

to slot in a new data store based on assumptions made for the previous data store. We considered many options, but opted to move toward a more general solution (DynamoDB) for general data to keep the data layer as simple as possible if we wanted to switch again in the future.

Once data is migrated, the tricky part is maintaining the previous behaviour (specific transactional behaviour, in our case) through a shim. This is where a great test suite comes in handy…





When do you actually begin to become stateful and shard your data? We recommend you wait as long as possible, and only do it when you don't have better options. Stateless servers are often easier to deal with than stateful ones.

When you start to shard your servers, you also need to think about what happens when you add or remove a machine from service. For a stateless service, it can just start taking traffic. For a stateful server, what else does it need to worry about? Does it need to dynamically change the ownership of data across a cluster? Does it need to wait for all open connections to drain - on a service that handles 30 minute game sessions? These are all things you need to plan around.

Stateful vs Stateless At Scale

Specific technology choice examples
= sharded cache

Sharding vs. stateless vs. stateful

Statefulness in the middle of a chain - repercussions

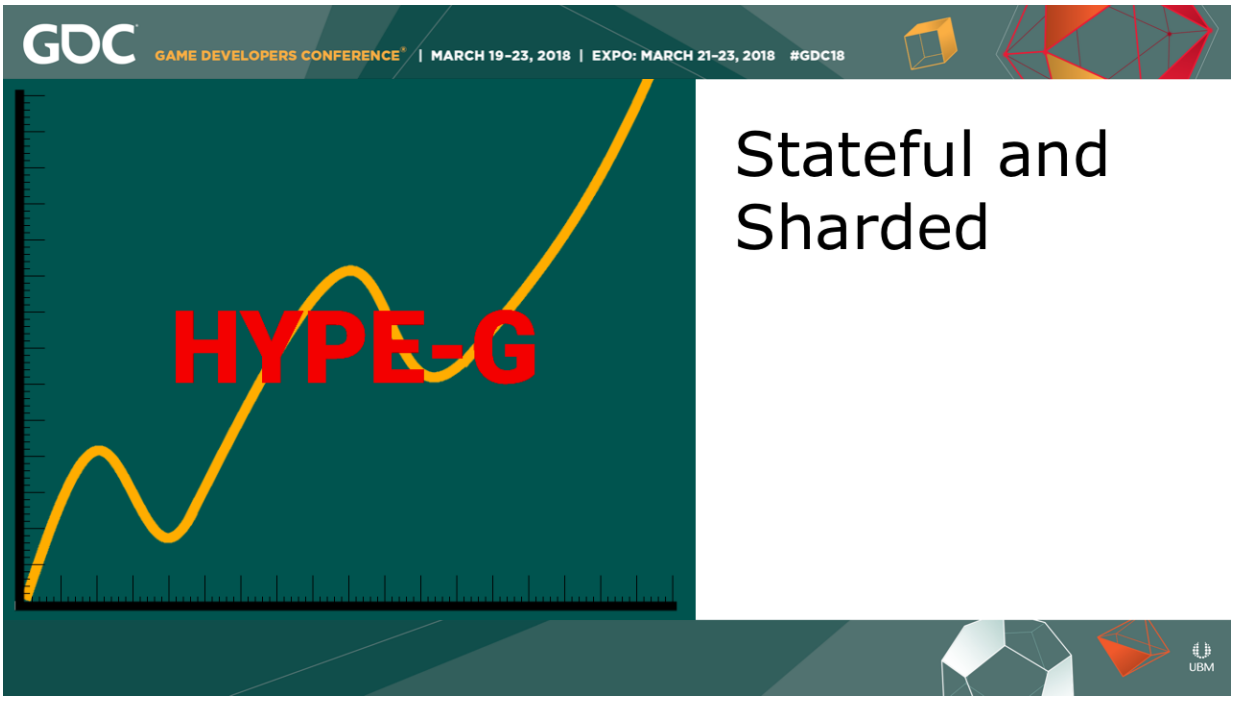
Cold starts - knock on effects

pros and cons of eg. hawt new cockroachdb vs mysql
know your costs of choosing this - training operators, hiring

memcached
hazelcast

- Sharding: When do you shard?
- Know when you need to have stateful data and when you don't – stateless will make your life easier.
- When you have to be stateful, look into various technologies and don't try to do it yourself.
    - What's the current hotness these days? DON'T USE IT
- Understand the costs of losing a machine, adding a new machine, etc.

HYPEG EXAMPLE Our curve is predictable downward curve from 128 to 1 players over time. So we can allocate game servers based on free mem etc. UNTIL WE ADD SPECTATOR MODE!! Interesting games mean people jump in to watch!



Let’s take a look at what we decided to keep stateful and/or sharded in Hype-G.

First, our game servers are stateful for the length of a match. All users in a match connect to the same machine. Matchmaking is run from the same machine, so it happens to be stateful as well. This lets us turn a match into a game as soon as the required number of users is present.

We have other stateful services, too. Our PubSub service is stateful, allowing players to remain connected to get information. So are our replay services, like our spectator distribution service. This allows us to fan data out to multiple consumers, so we want to maintain the data processing on specific servers.

Our databases are all sharded. We can shard the player accounts database by region, or by bucket hashing, or by alpha key. We need to keep in mind the relationships between our databases, too - if we shard player accounts in a specific way, we need to make sure anything (such as player entitlements) keyed on data in player accounts is sharded in the same way. However, data such as our catalog can be global.

- Case Study Example
- match is stateful

- player accounts DB is sharded by region
  - we could have sharded by alpha key prefix
  - or constant bucket hashing
- in game avatar (2D) in a cache

Stateful vs Stateless At Scale

Specific technology choice examples
= sharded cache

Sharding vs. stateless vs. stateful

Statefulness in the middle of a chain - repercussions

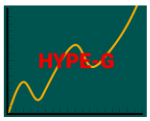
Cold starts - knock on effects

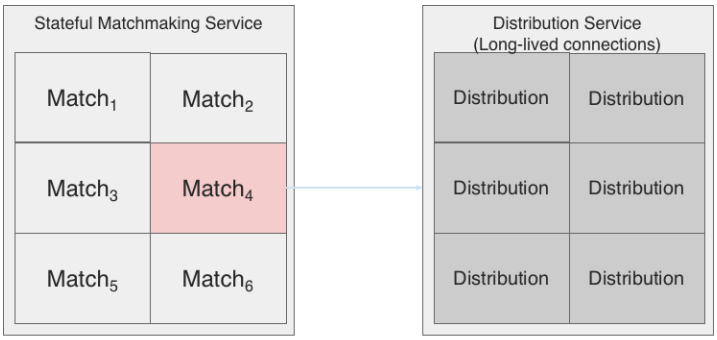
pros and cons of eg. hawt new cockroachdb vs mysql
know your costs of choosing this -

training operators, hiring

memcached
hazelcast

How would we design the architecture for the "Which hats can I wear" service?





Let's take a closer look at one of these.

When we built Hype-G, we knew that each game would only have 128 players at max at any given time. We built our models around how many game instances could run on a given piece of hardware based on these numbers, allowing us to budget per-player for things like memory usage, CPU, connections, etc on the server.

But then we added spectator mode.

Interesting games mean that people want to jump in to watch. We didn't want to limit the number of players who could spectate a match at any given time, so we needed to find a new way to do this without blowing out our models. A single machine can't scale to a very popular match, so we built a new service instead.

We decided to publish spectator data to a distribution service, allowing us to fan the data out to as many clients as desired and as many servers as needed, without requiring everybody to connect to a single stateful machine. 100,000 viewers on a single match? No problem. This also minimized additional network stress on our game instances, allowing active players to have the best experience possible.

As a side benefit, it’s allowed us to ingest additional business intelligence on our matches that we didn’t want to do in real-time on our game instances. Win-win!

HYPEG EXAMPLE Our curve is predictable downward curve from 128 to 1 players over time. So we can allocate game servers based on free mem etc. UNTIL WE ADD SPECTATOR MODE!! Interesting games mean people jump in to watch!

Become customer focused

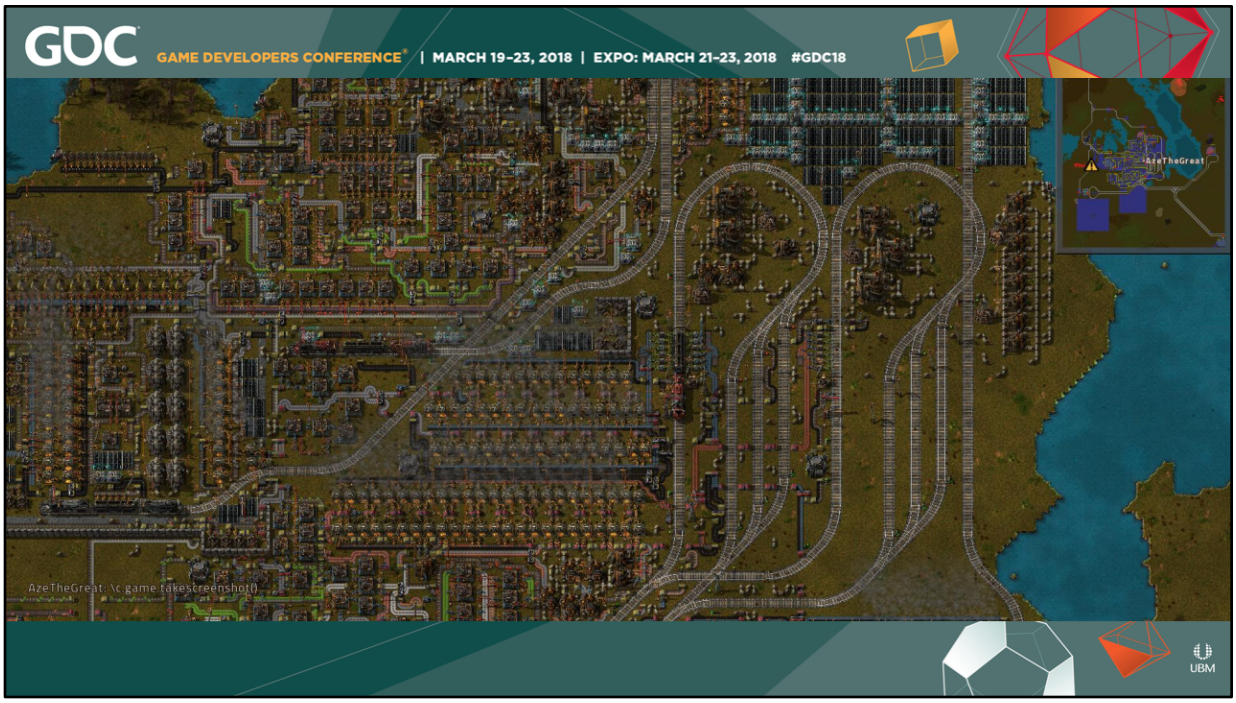
- Every edge case and every mistake affects players.
  - If they can't sign in, they can't play or pay.
  - Every piece of friction causes players to permanently drop off.
- At 10000 requests per second, a one in a million issue will happen on average every two minutes.

Hardware and software failure becomes a problem

- QA, QA, QA. Test early and often. Run through staging environments.
- DDOS protection and rate limiting become key.
  - You *will* be attacked and DDOSed. The question is not "how do you stop it?". The

question is “how do you mitigate it?”
Track requests per user, per client, etc and reject them early and often. The ability to allow only x requests per api key, or client, in a time window is key. (And you can grant extra requests to well-vetted customers like partners)

Image: https://cdn.cultofmac.com/wp-content/uploads/2016/09/160927142551-samsung-note-7-fire-1-780x439-780x439.jpg



Slide theme: “The human element” aka “it’s too complicated”

Infrastructure size: more Mastiff than Chihuahua

- One engineer cannot hold it all in her head anymore.
- Live service debugging becomes more and more challenging.

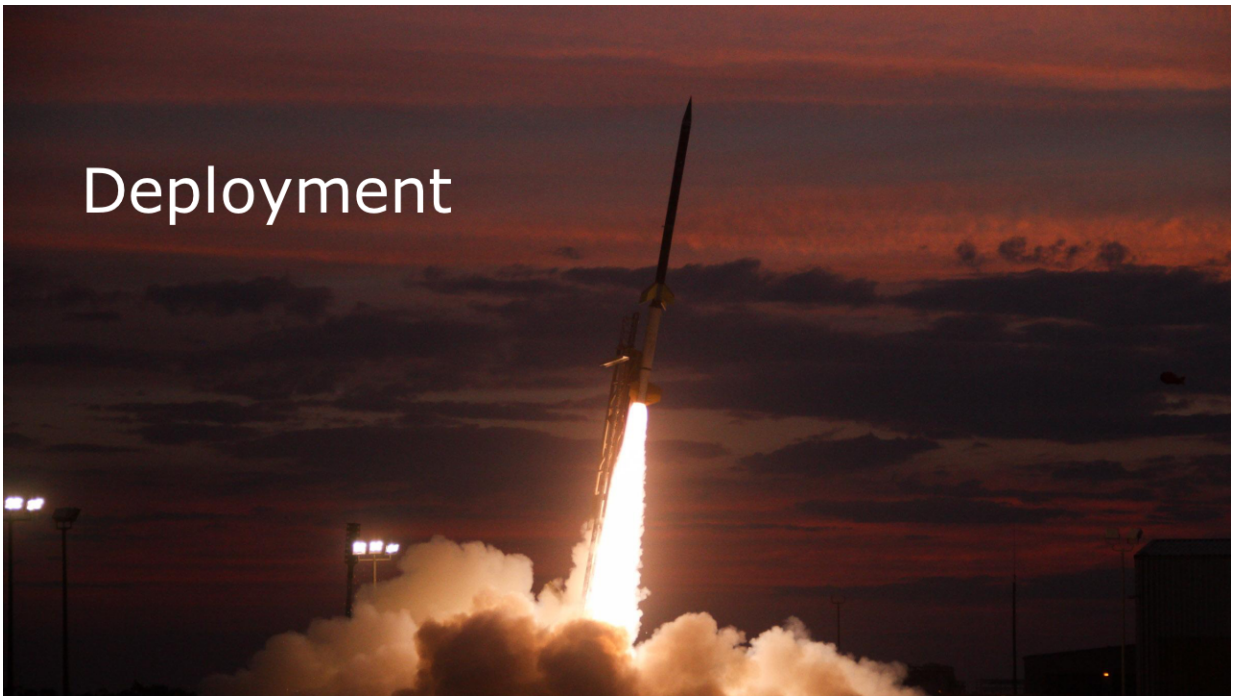
Communication becomes key.

- Standards, processes, and communication are necessary.
- You might even have to deal with cross-organizational politics.

Live site debugging becomes difficult.

- Common tooling and dashboards can help.

Image: http://i.imgur.com/JZuLqhy.jpg



(change: sela -> jennie)

Deployment

- Take the bits from over here
- And put them over here

https://memegenerator.net/img/instances/65215366/take-the-bits-from-here-and-put-them-over-there.jpg

- Run servers side by side!
  - Make sure your architecture and assumptions can handle this. e.g. service discovery version filtering
  - You can beta test!
  - You can test in production!
  - You can do zero downtime deployments!
  - Costs more but way way less risky than bad code everywhere
- Version strategy - Semver, Patch Commit
- Example - HYPEG Game server draining
- Blue Green Deploy Strategy
- Canary Deploy Strategy
- Downtime Windows

https://www.flickr.com/photos/fwc439h/

16053132734/

- How do you safely deploy brand new services for the first time?
- Dark Launch
- PBE
- What is your backout strategy?

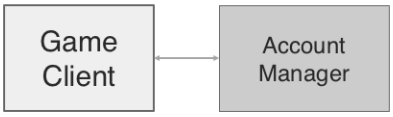
https://www.flickr.com/photos/nasahqphoto/6012071553/

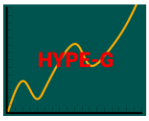
- Avoid “No Turning Back” scenarios whenever possible
  - Breaking database migrations are tricky
  - Turning a service off can happen in stages
- If it’s retiring a service, you can deploy the replacement and bleed over traffic
- DB migrations can be staged in many cases
- Think about schema changes and what you’re going to do with the cache, tokens, etc. Do you invalidate everything at once? If so, what happens?

- Deploy out of band from your client
  - …because someday you'll probably have more than one client
  - ...because it simplifies rollback
  - ...because it's not a great idea to tie them together (it requires player updates and downtime)
- Compatibility is key
  - Be able to run vCurrent and vNext at the same time, especially if there is an API change
- New Unity? Update server, run both, update client, turn old servers off OR Ship client that can handle both then roll server

- Know which parts of your client update less frequently and use gates to stop versions at the door (e.g LoL Login)

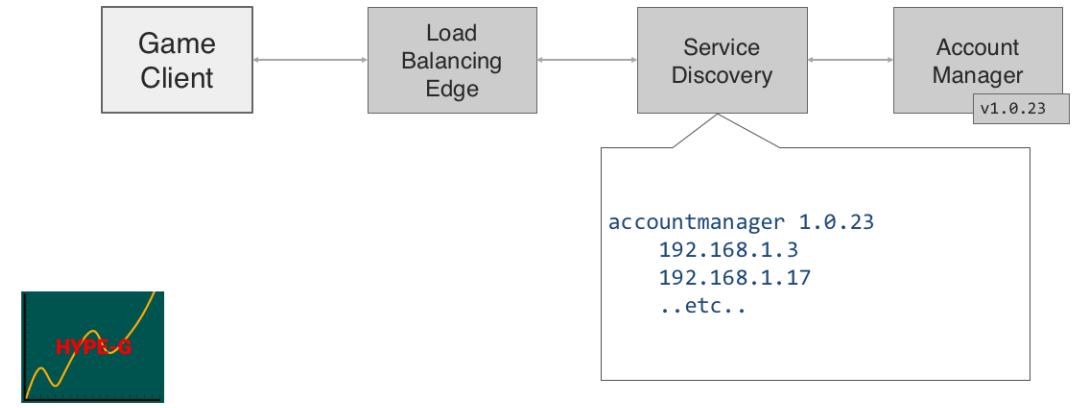
Global Simultaneous Rolling Zero Downtime Deploys, Oh My





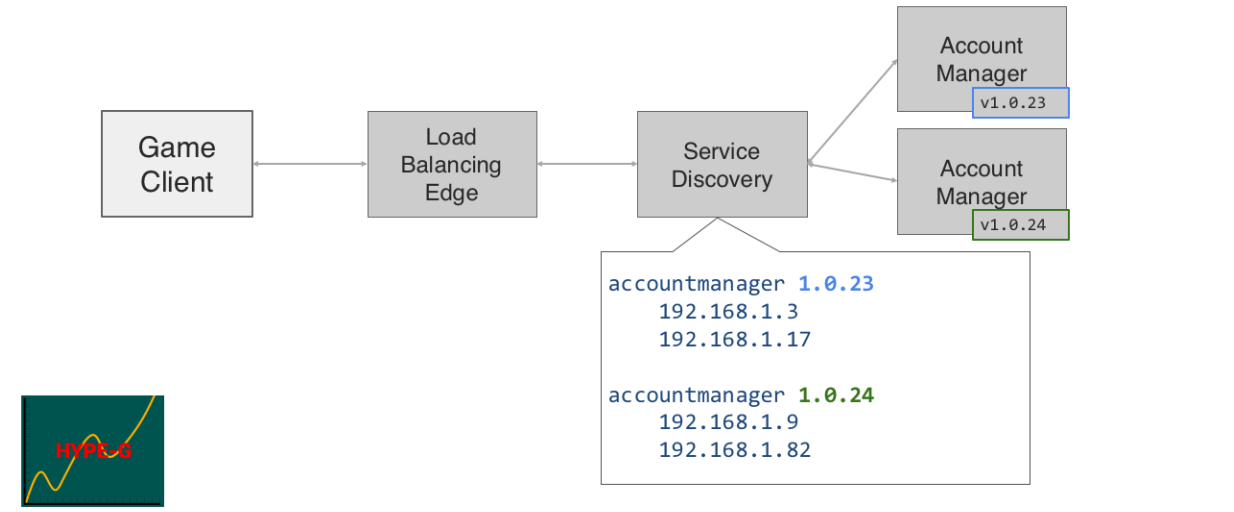
Tech deep dive into how a deploy actually works

Explain how HYPEG does these



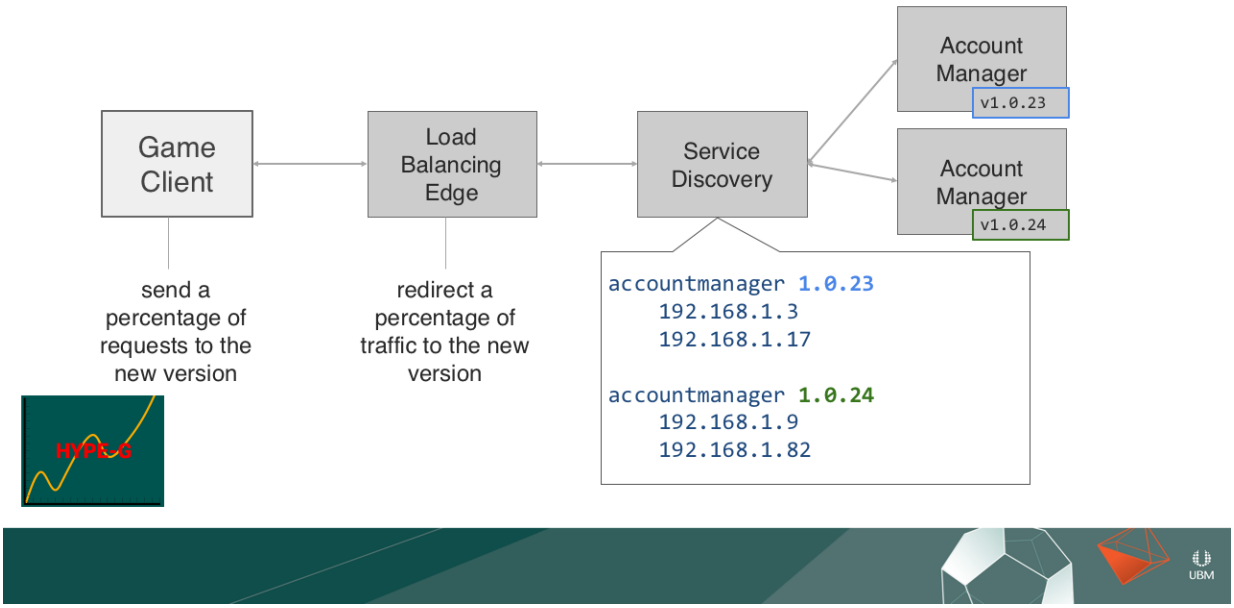
Tech deep dive into how a deploy actually works

Explain how HYPEG does these



Tech deep dive into how a deploy actually works

Explain how HYPEG does these



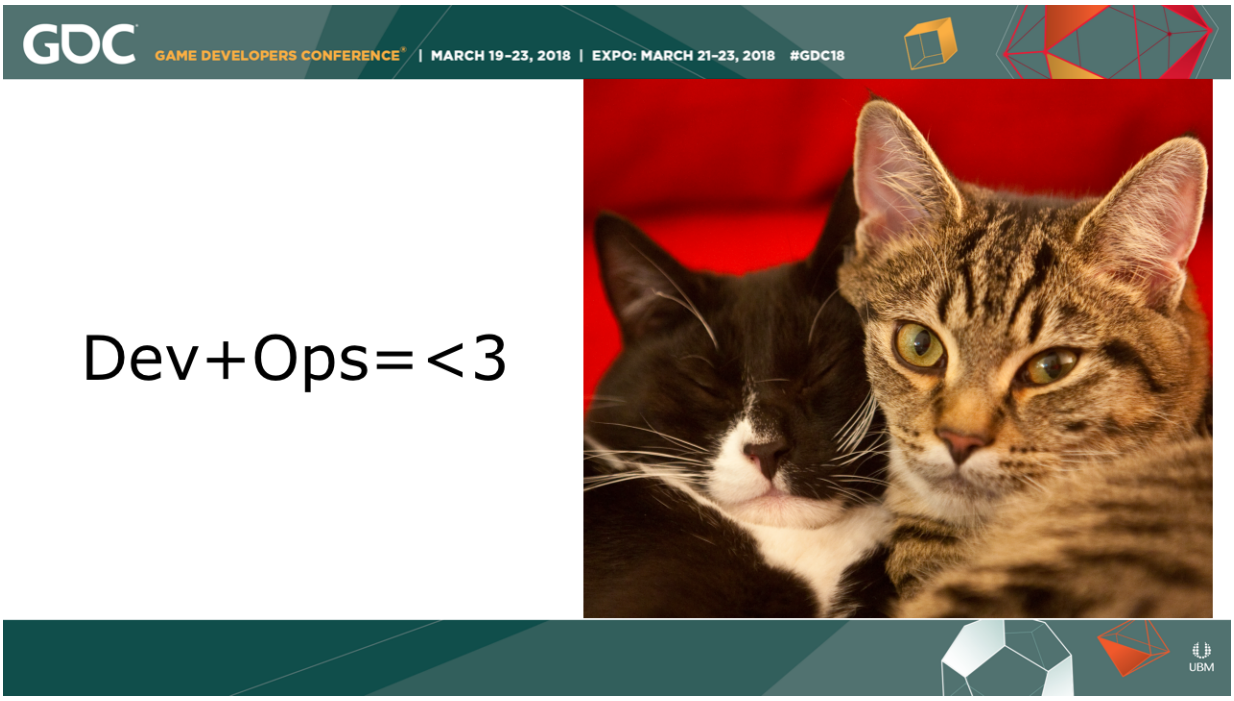
Tech deep dive into how a deploy actually works

Explain how HYPEG does these

https://www.flickr.com/photos/shawnzlea/2248316835

https://www.flickr.com/photos/25253506@N05/2820956884/

(change: jennie -> sela)



Operations

Operations is something that every engineer should worry about. We live in a world with a shared responsibility for production code, and working together will resolve every incident. Reliability also improves - instead of tossing code over a wall, we can all pitch in and solve problems more quickly. Nobody knows a service better than the developer, and an Ops team at this scale can be overwhelmed and

stretched thin by the number of services.

Not just something for “that other team”

Image credit: https://www.flickr.com/photos/jpovey/6649810681/in/photolist-b8C23i-68g7Yw-J6tjw-7qBgkT-659TMp-2hnVCN-tSYn6x-amkpoG-V4pHCH-8VAkvm-UAxoB4-CQLYA9-5bL7Ef-95juoQ-7g9CDD-qyRehs-csgu51-4YTptG-f6KvU-UgLoJ7-bjCAYA-bvxkYu-ntYEKM-8G9gXr-oEYaX3-52QXpY-5EWspw-apnTwU-RCMHrv-bVxXqw-8M91Ab-gxoNuE-ff7j82-5cSEQ4-Kazav5-cshExo-e8VxWd-5vZkDQ-ncusiY-95gpPe-5DFnFZ-qucnDF-aknLEV-dJsx9R-9g55q8-oZ2qzK-ko1sxk-aJKUt6-9tJgUk-m1vRsz

License: https://creativecommons.org/licenses/by/2.0/ (crop, no additional changes)

One thing we can do is build tooling to ensure that we know the status of a deployment. The more visibility we have into it, the better. If you identify that something is going wrong, you can roll it back - and if you can detect it programmatically, you can programmatically roll it back!

It's worth remembering that in most scenarios, it is safer to roll things back at the first sign of trouble and investigate out of band. If you build

your systems as Jennie suggested, you'll maintain compatibility between versions, and this should be safe to do. Trust your gut - I've left a service in production despite having a bad feeling, and it turns out I was right. Rebuilding the data a week later was much more difficult than deeper investigation.

How do we know the deployment went well? How do we know things are still going well?

https://giphy.com/gifs/rocket-landing-launch-26DNbCqVfLJbYrXIA/download
sourced from https://www.youtube.com/watch?v=wbSwFU6tY1c

If you pay attention to your metrics, you can catch issues as early as possible. Your service should attempt to make health checks before it goes into service, but health checks don't catch everything - especially when you are adding new features. If it passes the health checks, it should go into service and start to take traffic before the next batch is taken out of service for deployment. This is where the customer impact can occur.

I tend to watch my graphs related to outgoing and incoming traffic, as well as any service-specific error logs. In a bad case, I usually see one of the following rise up (and I prefer to keep a dashboard with all of this information visible on a single page):

4xx responses from a dependency
5xx responses from a dependency
4xx responses from my service
5xx responses from my service
Increased internal errors (handled or not) from my service
Increased latency on one or more APIs

These may be the source of problems or may be a symptom. Either way, increased failure means “cancel deployment, roll back, pull logs off both healthy and unhealthy machines so they don’t get lost”.

Pro tip: rollbacks can be automated

based on these metrics, but no system is perfect and you should always pay attention.

It’s even better when you can use shared infrastructure. There’s absolutely a place for custom technology that is designed for your needs, but there is also a place for consistency at scale. If you can simply correlate your outgoing requests’ failures with the downstream service’s failures because all of your metrics look the same, that’s fabulous.

It’s even better if you can coordinate

logs through a correlation id that gets logged in all places. A simple UUID that can be sent as part of your request protocol between machines is incredibly valuable. And if every service builds and logs them the same way, you can search your logs in a consistent way to filter by correlation id to see everything that happened in a single request from the client, merge client and server logs together so that you can see what happened when your request left your service, etc.

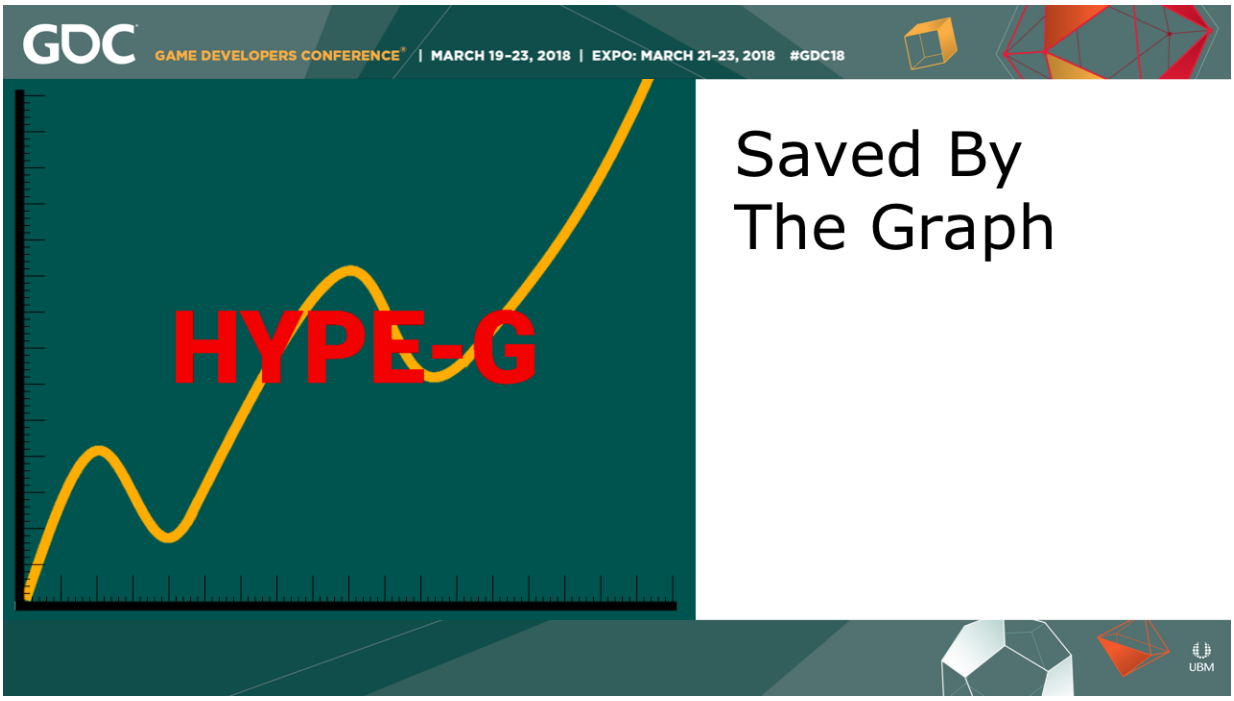
Remember, at scale, you are receiving so much data every second that it can be overwhelming. Anything you can do to reduce what you need to look at _without losing important data_ will save you time during incidents.

https://www.flickr.com/photos/154957955@N07/34642432020/in/photolist-UMew7G-Zq4uzR-8gGUpt-YeP2ki-U255Sf-UgDScz-4T8sYW-dFXEU8-

8f6Xi1-q1RjWn-9YFuoz-fJdvR6-RG6r-YeP5bP-RKhKJS-C9V5Fj-3vR3qm-9e6Vh8-qQAUEC-qZ8Wyh-cdP8nJ-ETK54M-c6hHDd-qguCeX-TjEnkN-em4c2M-cdP84y-fxqLsD-qJSBuU-9vjZqD-q58pp5-4Urum6-4DUZCM-b5JM4x-222bPNt-AUPj1-iLAV1A-dLEVfX-ebF9eH-f3FvTP-EBbBjm-SZPgN1-V3E5i8-dTZxGY-cmwary-9vnZx5-f2QFar-nT5wQW-p13gAL-oZHS8W

If you build dashboards around the metrics you use to verify service health, you can make sure that people who aren't you can tell if something bad is happening to the service. The simpler you can make a view, the better - remember that somebody is probably looking at a service they don't know at 2:00 in the morning, and they're probably very unhappy about it.

Remember, the more consistent your graphs are across your entire organization, the better off everybody will be in the end. Graphs incomprehensible to other teams take longer to react to, until you get overwhelmed. Just like Cat Sorter VR.



When we tried to roll out our latest changes to Hype-G, we saw unusual behavior. After the first batch of matchmaking services went back into service, we started seeing the occasional failed match across the cluster. Our 5xx rate jumped up from 0.1% to 0.4%, just underneath our threshold of 0.5% for alerting and automated rollback. Assuming that the 300% increase in failure was due to the single batch, we

manually triggered a rollback to reduce customer impact ASAP (even though the cluster would likely alert and cancel rollout as the next batch came into service). As we investigated the failures, we discovered a new edge case triggered by certain customers in production traffic, and we added appropriate testing to cover these cases.

- Talk about chaos of different boxes
  - Languages, datacenters/locations, frameworks, libraries...
  - It becomes a technical alignment problem
- Callback to microservice advantages/disadvantages - CHAOS CHAOS

(change: sela -> jennie)

Operations At Scale

Design everything as though you are going to be paged for it at 4am.

Then go back and design it so **somebody else** can handle that page.

- Empower other teams to help you in a pinch
  - How do you work with and empower a NOC?
  - How about SREs?
- High-level service dashboards help with “at-a-glance” info.
  - When 5xxs spike and request count shoots up, what is happening?
    - Retries on failure.

- Helps non-service-owners triage quickly

SME Escalation Path - Who gets paged??

WHAT IF YOU ARE NOT THE OPERATOR eg China

Design everything as if you will be using it at 3am on a saturday night

Runbooks - Pros and Cons

- difference between runbooks and identifying the actual cause of the issue
- alternatives to runbooks - automation, guidebooks
- how to diagnose and escalate
- up to date (HA)

- Investing in operations during development will pay off down the road
  - Log as much as you can
  - Track as many metrics as possible
  - Provide health checks for at-a-glance status checking
  - Be a good "customer" of the ops/NOC!
  - mention the types of 1 in a million events - HW failure, etc

At scale, "one in a million" events happen **all the time.**

At 10,000 requests per second,
a one in a million issue will happen,
on average, **every two minutes**.

(120 seconds * 10,000 rps = 1,200,000 requests)

- For non-server engineers: we hope you learned something about the inside of your servies
- For server engineers, we hope this makes sense and you learned something new about scale. If you think we missed something, we'd love to hear from you after this because we'd love to learn. We're all in this together.

THIS IS NOT GOSPEL TRUTH.

- Architecture
  - 101: Client-Server Patterns
  - 201: Preparing For Scale
  - 301: 100M DAU? NP
- Deployment
  - 101: Getting Bits on Atoms
  - 201: Global Simultaneous Rolling Zero-Downtime Deploys, Oh My
- Operations
  - 101: Instrumentation and Triage
  - 201: Operations At Scale

This provides a framework for the speakers to own various sections.

https://www.flickr.com/photos/dancedancedancephotography/3356807821/

(change: jennie -> sela)

Sela Davis

Senior Software Engineer
Vreal

Jennie Lees

Senior Software Engineer
Riot Games

https://www.flickr.com/photos/henryblomfield/12680815144/