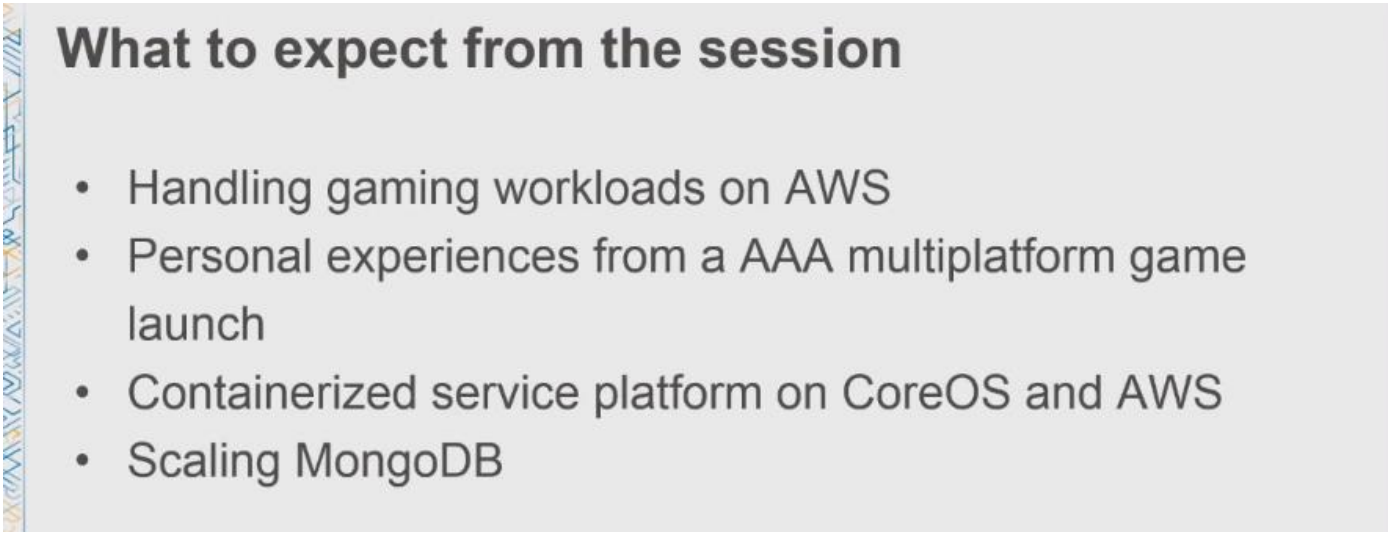
Turbine shares lessons learned from their new microservice game platform, which used Docker, Amazon EC2, Elastic Load Balancing, and Amazon ElastiCache to scale up as the game exceeded expectations. Learn about their Docker-based microservices architecture and how they integrated it with a legacy multiplatform game-traffic stack. Turbine shares how they gracefully degraded their services rather than going down and how they dealt with unpredictable client behavior. Hear how they resharded their live MongoDB clusters while the game was running. Finally, learn how they broke their game-event traffic into a separate Kafka-based analytics system, which handled the ingestion of over two billion events a day.



Challenges

1. Managing multiple games on AWS
2. Scale
3. Pick your battles
4. MongoDB at scale
5. Staying up through launch

A bit about us!

- Founded in 1994, acquired by WB in 2010
- Pioneer in online games
- Work with huge properties (LoTR, D&D, DC Comics)
- Launching our first mobile game

We support lots of WB titles

Turbine's platforms

WBPlay game (WBG): Customized cross-platform, cross-studio game back end and analytics

Mobile game platform (MGP): Next gen game back end for Turbine games

Analytics platform: Analytics data crunching for WBIE

WBPlay identity: Identity and e-commerce

We will be discussing the first two

WBG and MGP

Currently powering

- *Batman: Arkham Underworld*
- Unnamed AAA multiplatform game

Provides:

- Authentication
- Profiles
- Matchmaking
- Match history
- Guilds
- Cross-platform unlocks
- Feeds
- Social identity mapping
- Analytics events

Why AWS?

- Data center capacity
- Unknown infrastructure footprint
- Platform automation
- Bursty gaming workloads
- Multiple Availability Zones

Challenge 1: How to set up AWS?

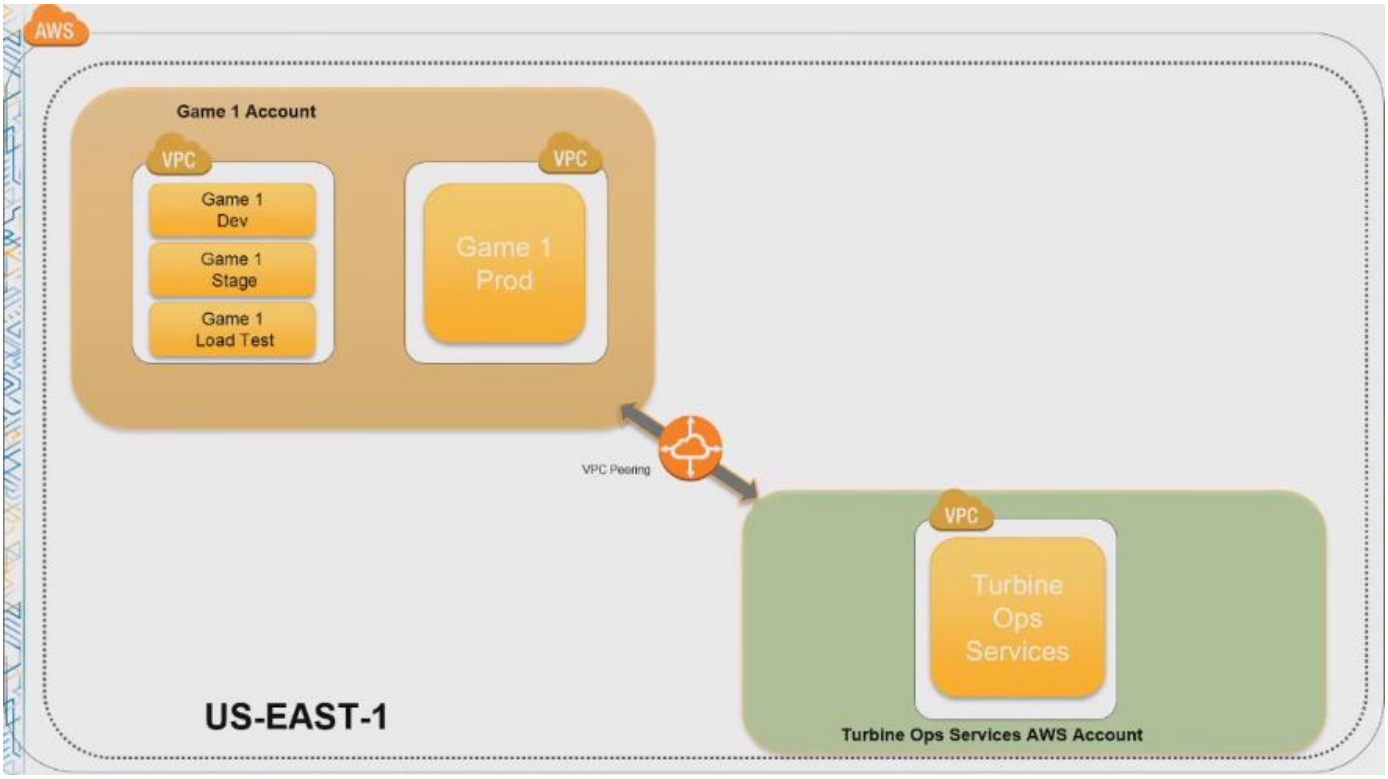
- Support multiple games, studios, environments, and teams
- One game / environment cannot affect another
- Access control
- Centralized services access

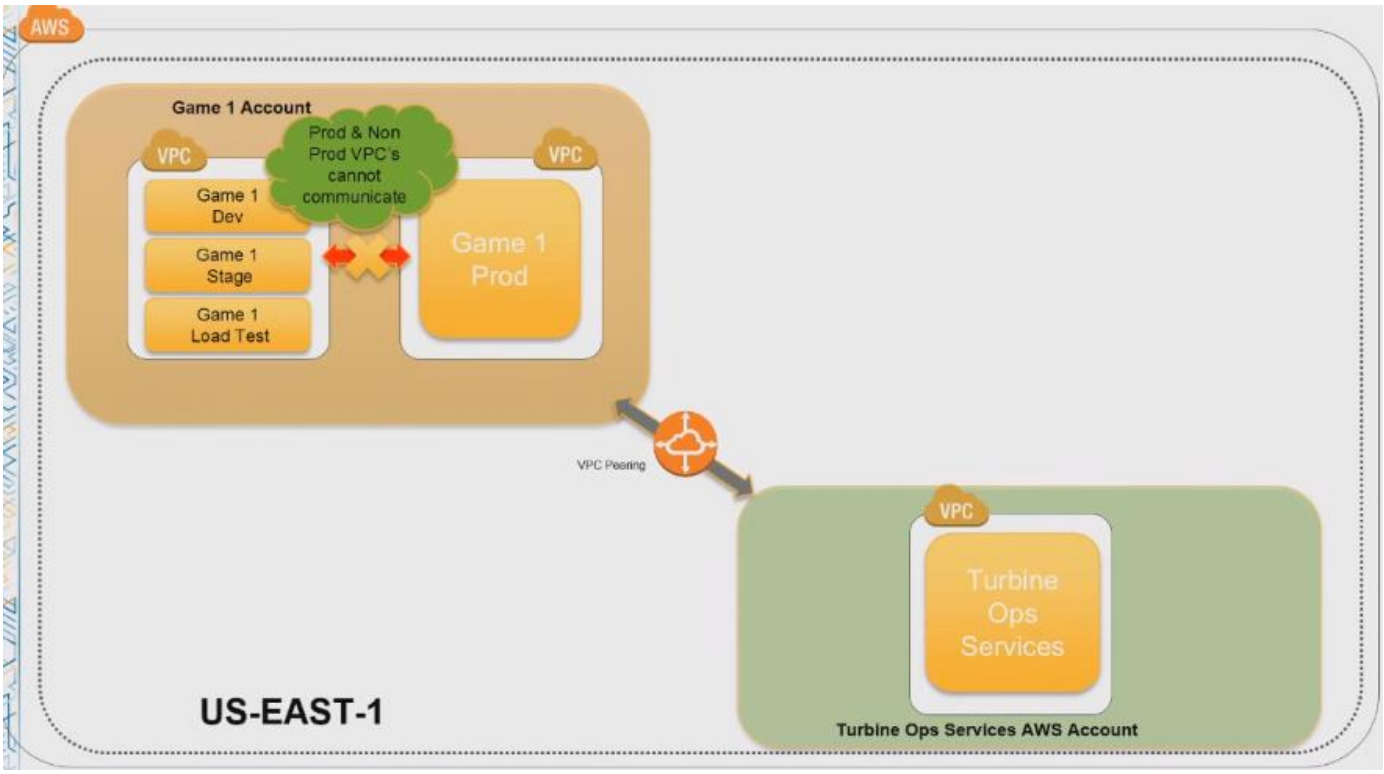
Multiple AWS accounts

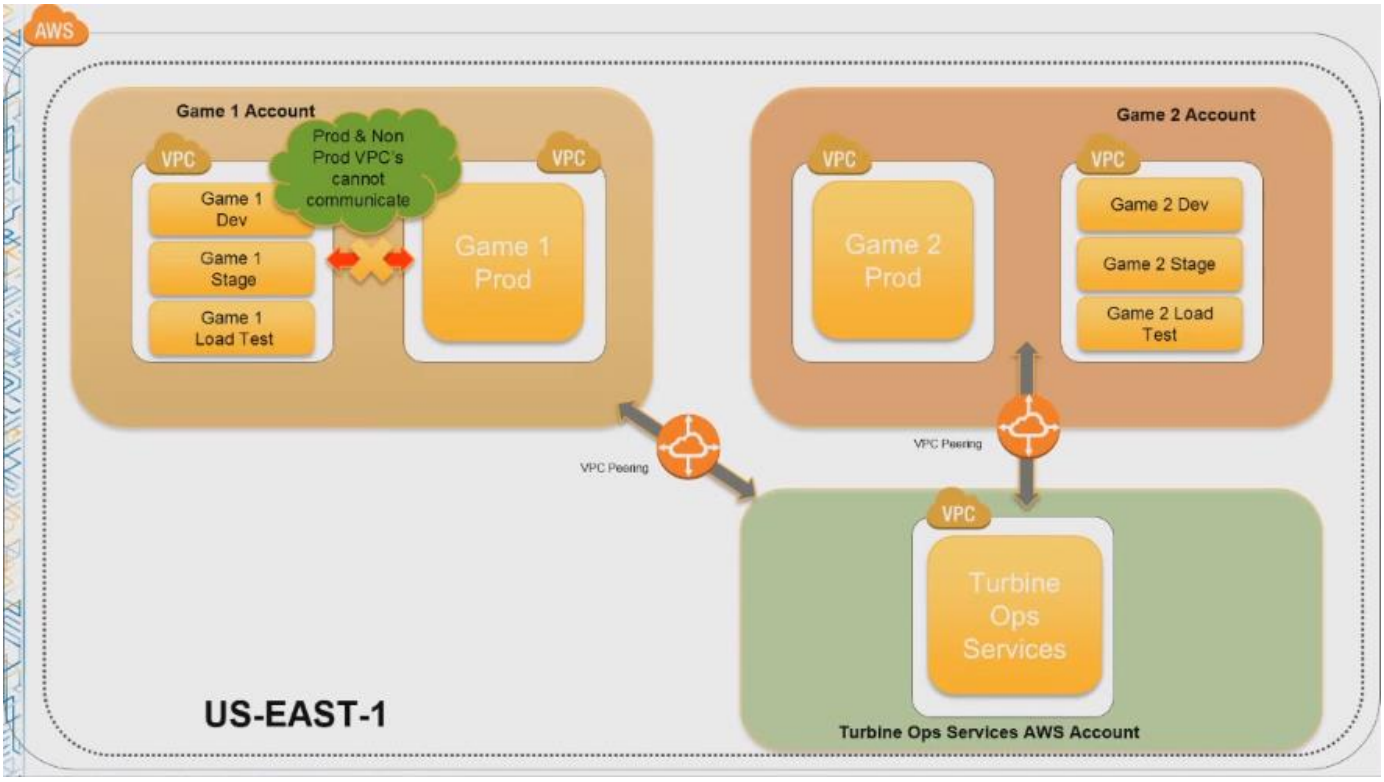
- One account per game
- One VPC per environment
- One account for internal core services

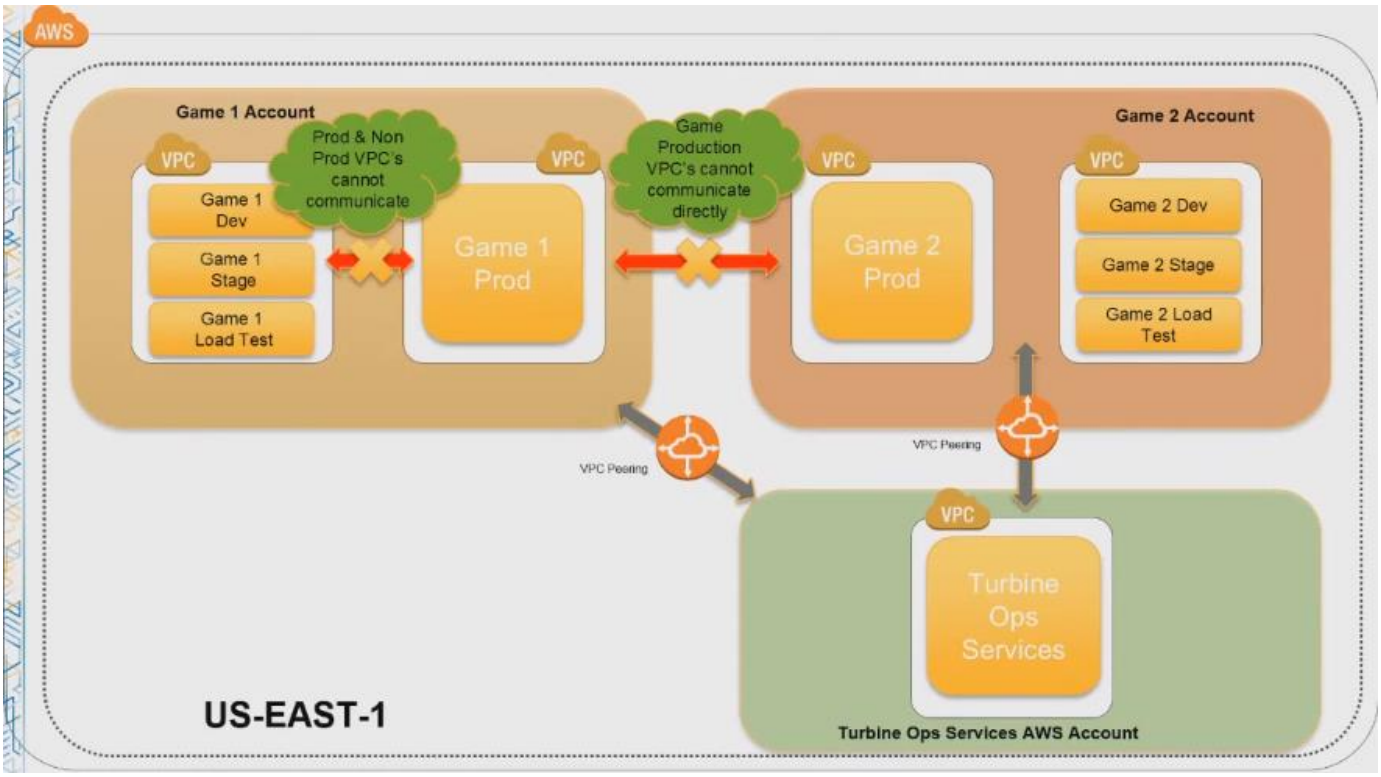
Pros
Service limits can't clash
Simplified billing allocation
Digestible chunks of infrastructure

Cons
Management overhead
Must be automated



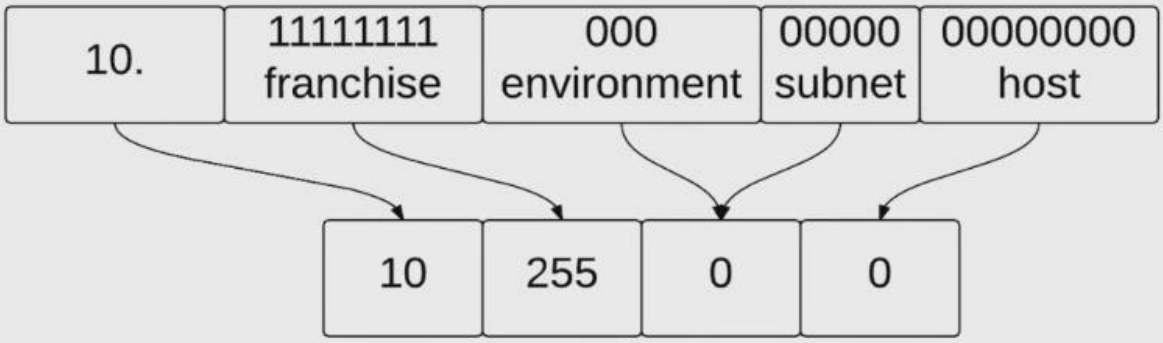

AWS
Game 1 Account
VPC
Prod & Non Prod VPC's cannot communicate
VPC
Game 1 Dev
Game 1 Stage
Game 1 Load Test
Game 1 Prod
VPC Peering
VPC
Turbine Ops Services
Turbine Ops Services AWS Account
US-EAST-1


AWS
Game 1 Account
VPC
Prod & Non Prod VPC's cannot communicate
VPC
Game 1 Dev
Game 1 Stage
Game 1 Load Test
Game 1 Prod
Game 2 Account
VPC
Game 2 Prod
VPC
Game 2 Dev
Game 2 Stage
Game 2 Load Test
VPC Peering
VPC Peering
VPC
Turbine Ops Services
Turbine Ops Services AWS Account
US-EAST-1

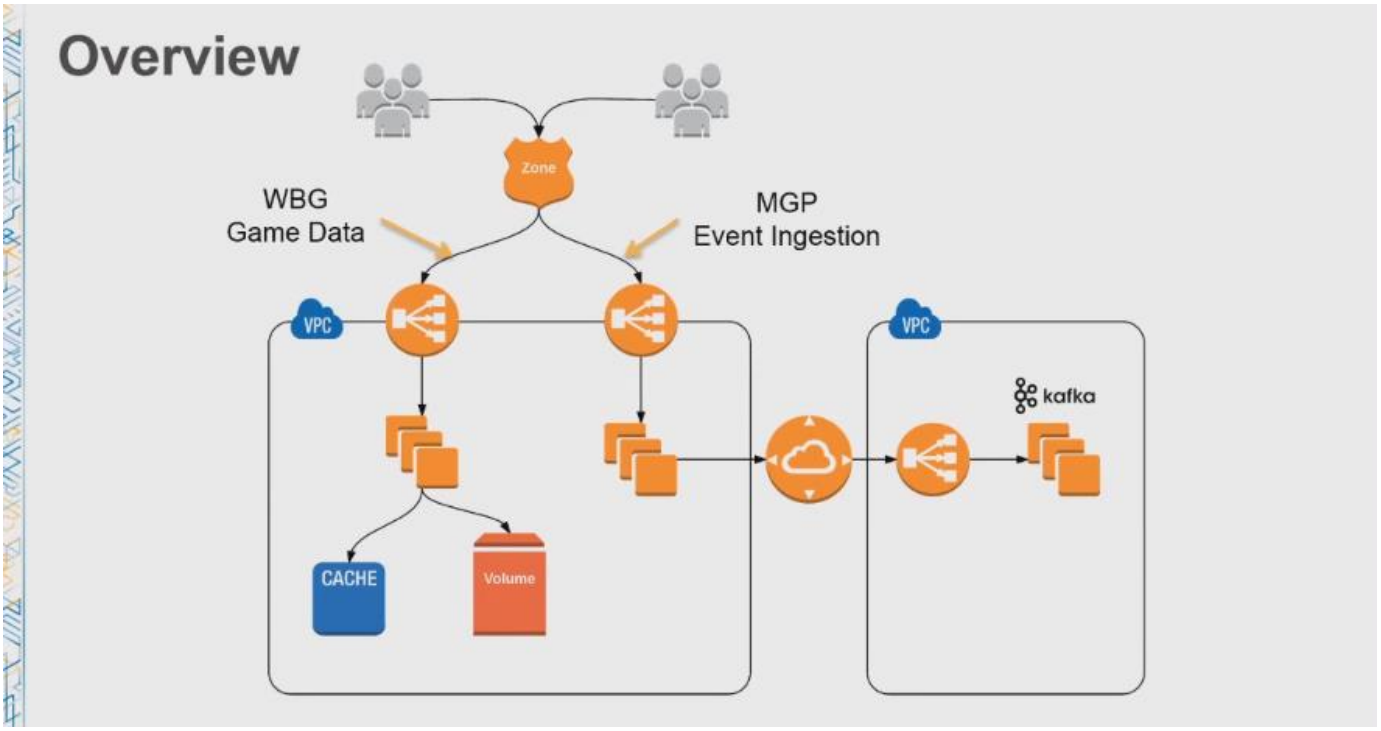


Globally unique addressing

- Central route table entries must be unique
- Provisioned 5 / 20s for 5 environments up front
- VPC IP space divided into 3 public and 3 private subnets in 3 AZs
- Bitmask to precompute consistent VPC blocks

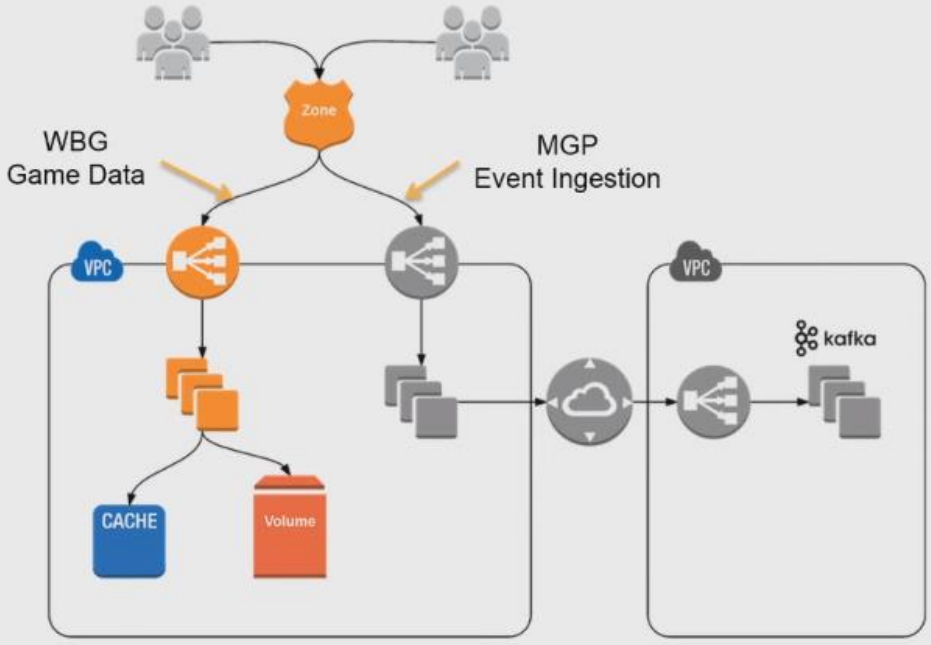


Architecture Overview



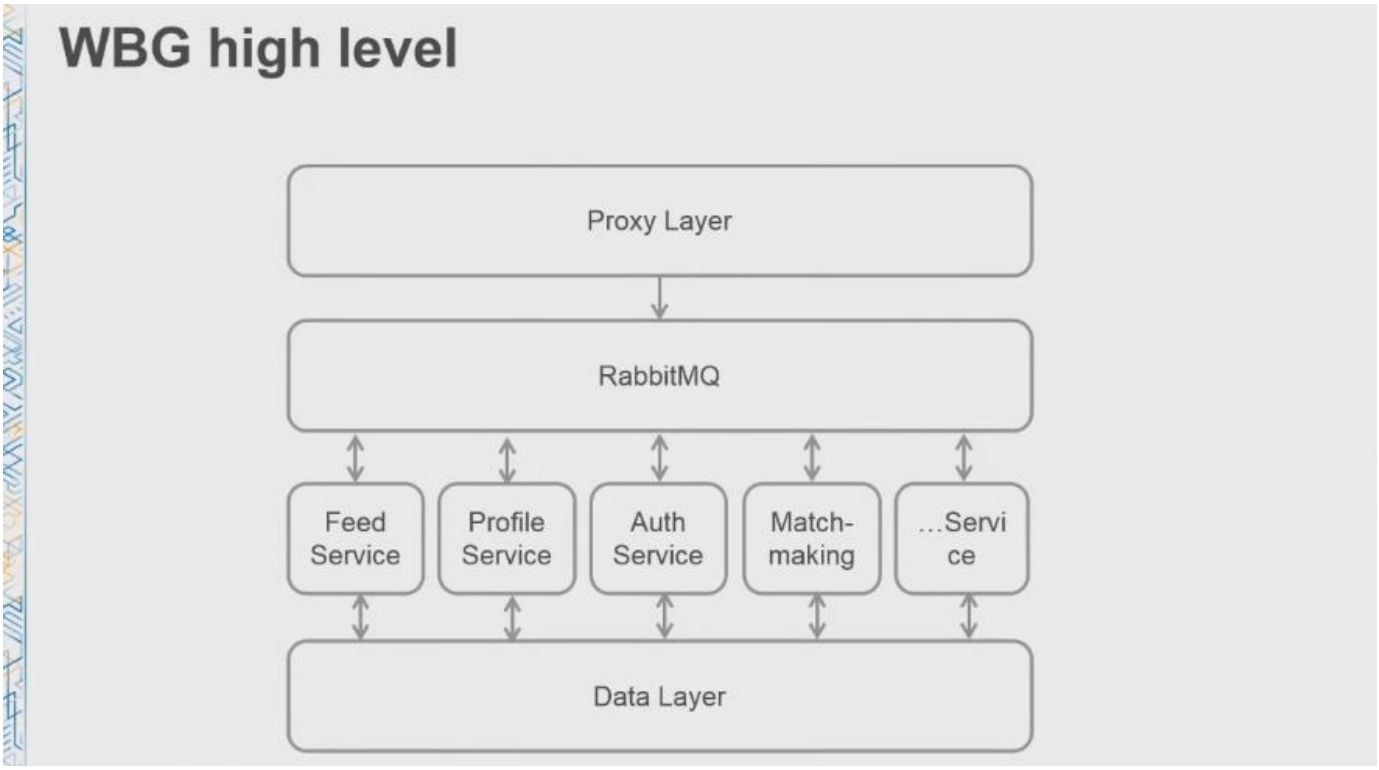
This is the 10,000 feet view of our architecture

WBG

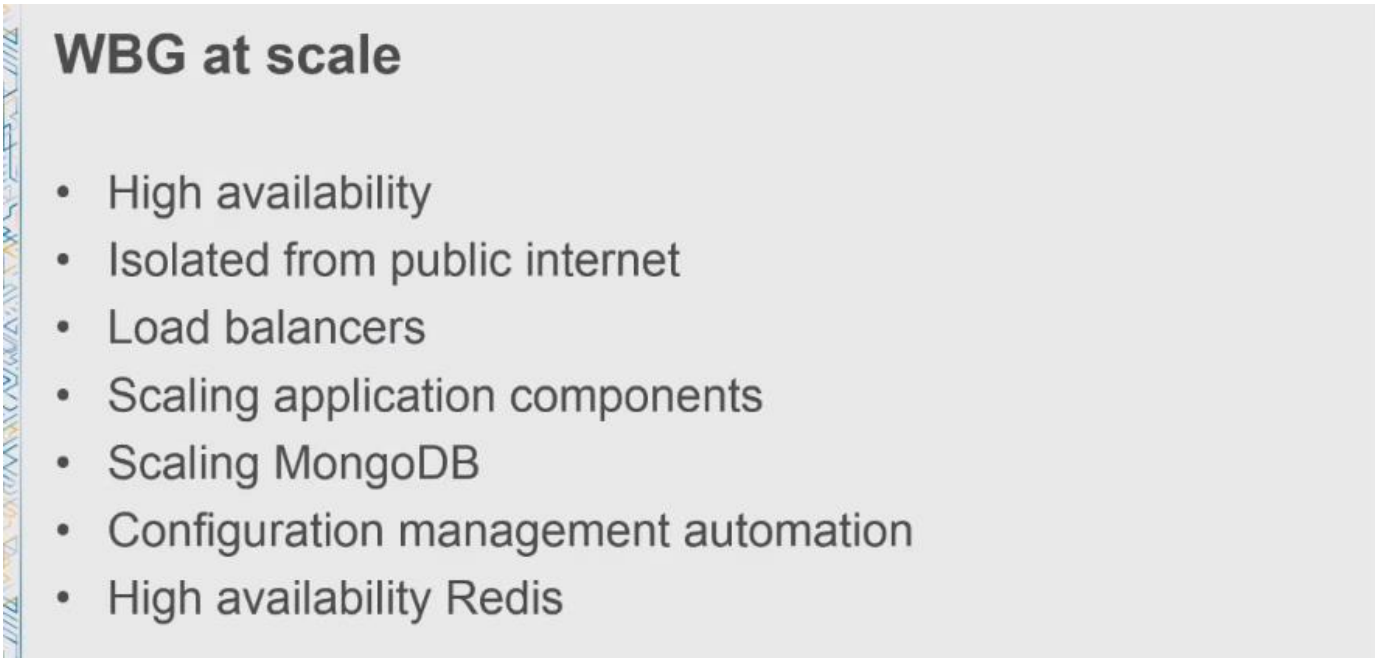


WBG overview

- Game back end and analytics
- Monolithic code base
- Python on Linux
- MongoDB and Redis
- Custom binary message formats
- Console and Mobile SDK
- Chef (solo)



Conceptually we have a proxy layer that handles traffic and hands them over to RabbitMQ which is the message bus that supports the MQTT protocol, this routes traffic from the proxy layer to any of the services registered with the message bus. All the registered services are doing their work using a shared data layer which is MongoDB in this case,

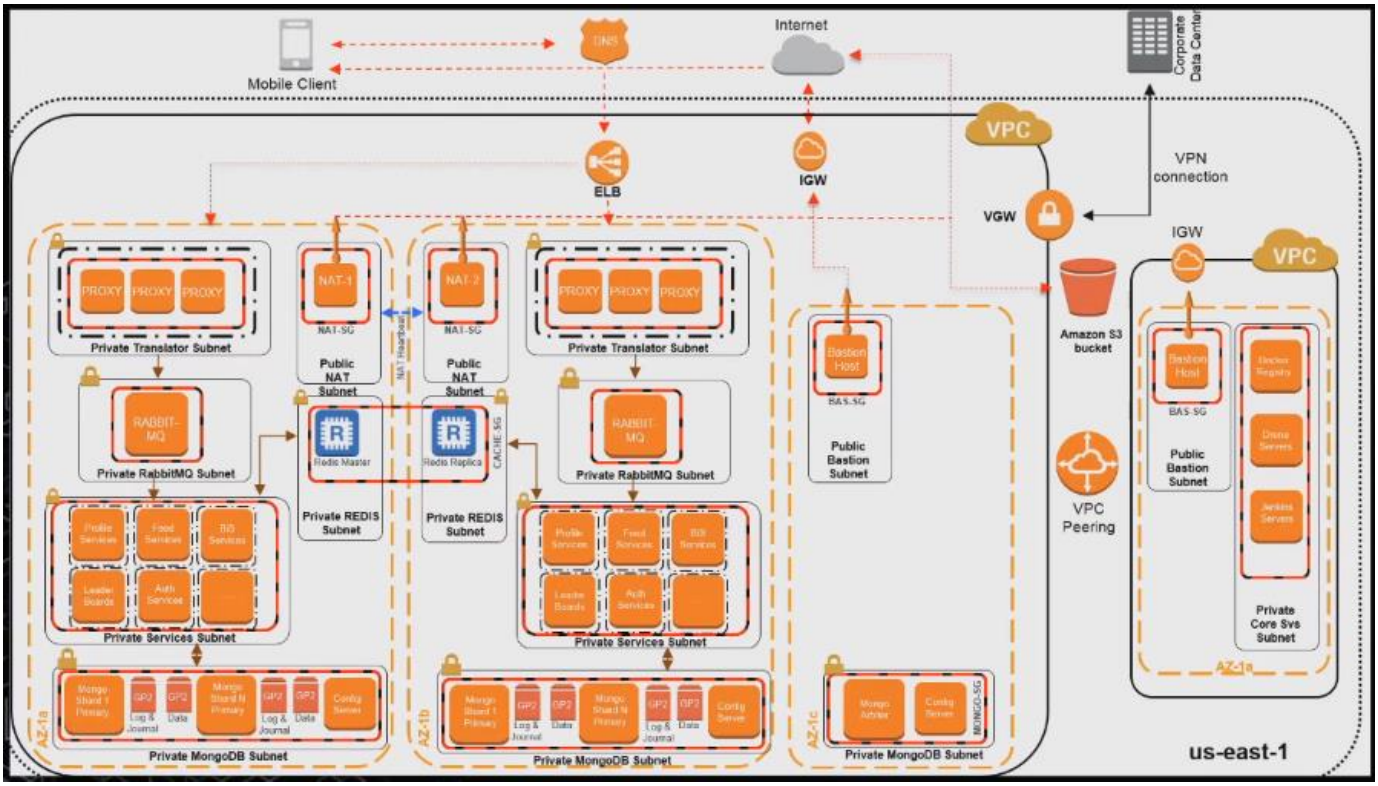


We need all the above in order to use the platform for gaming.

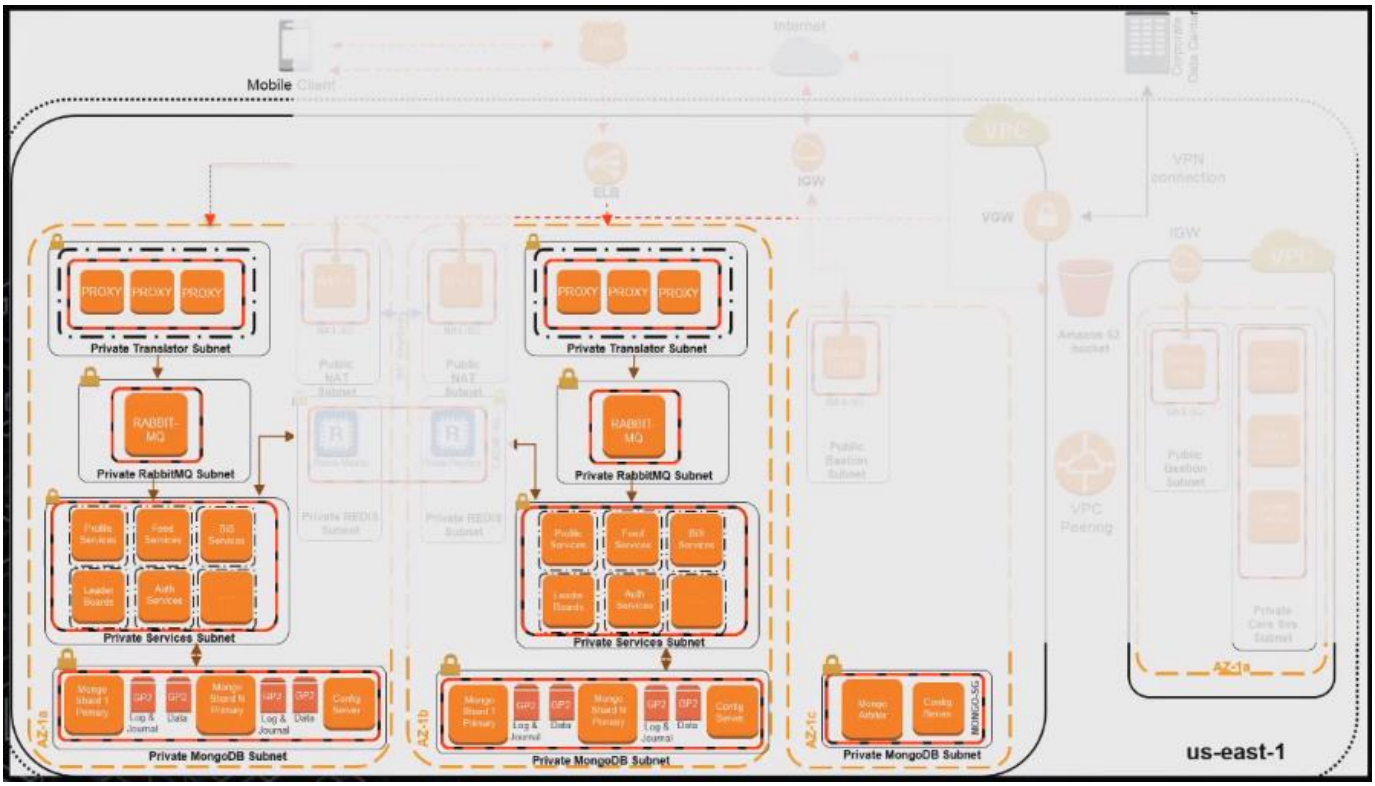
WBG at scale

- ~~High availability~~
- ~~Isolated from public internet~~
- ~~Load balancers~~
- Scaling application components
- Scaling MongoDB
- ~~Configuration management automation~~
- ~~High availability Redis~~

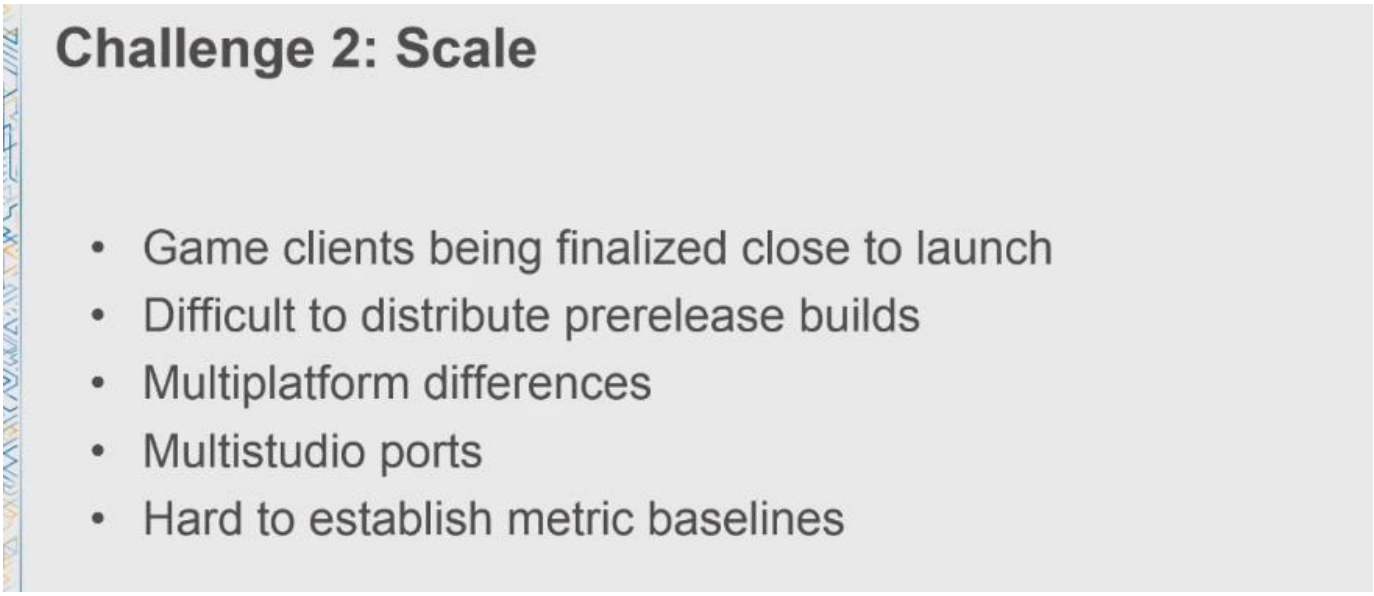
With AWS, we use ASGs to solve the HA problem, we use multiple VPCs to solve the isolation problem, we use ELBs for load balancing and also use baked AMIs so that we only have to configure things once when creating the AMIs. We use Elasticache for highly available Redis.



This is what it looks like when implemented



Using the AWS services, we are left with the core application stack issues we really needed to focus on above



When doing a multi-platform gaming application launch, scale becomes an issue.

Pre-emptive AWS capacity planning

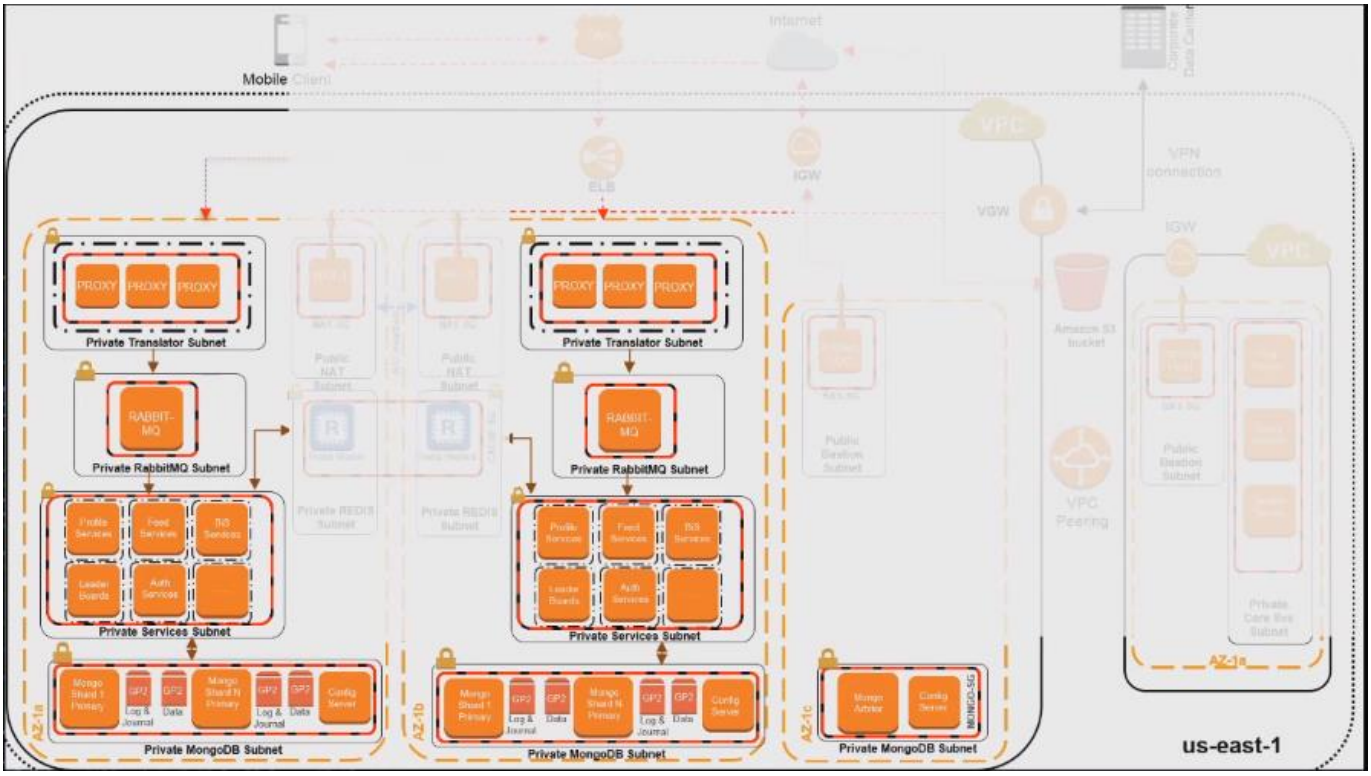
- Elastic Load Balancing prewarming
- Amazon EC2 limit increases
  - 500 C4 and R3, per size
- Amazon Elastic Block Store (EBS) limit
  - 192TB GP2
- Amazon S3 bucket partitioning

Instrumentation

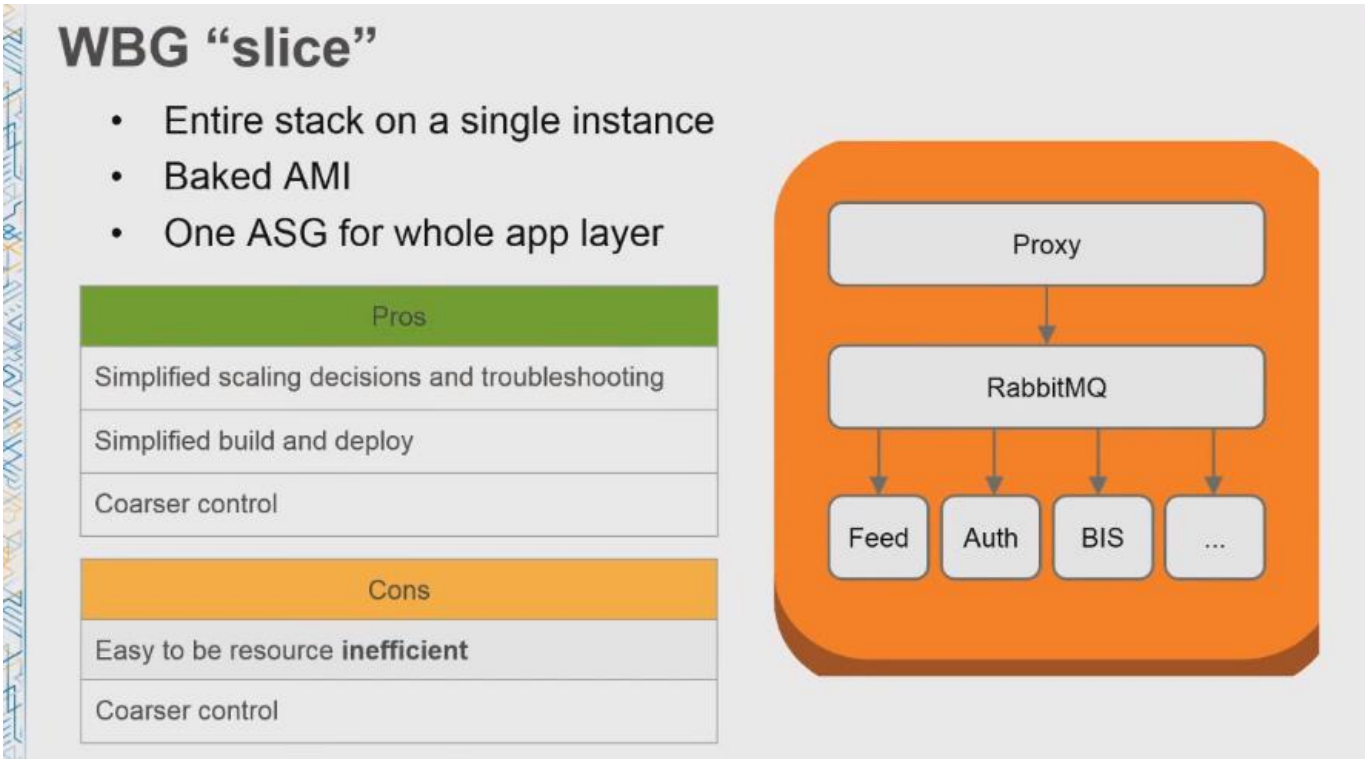
- Preproduction metrics measured externally
  - Load test clients
  - Amazon CloudWatch
- Late addition of internal metrics
- Statsd -> Librato
- No baselines for key metrics, so we instrumented all the things

Operational complexity

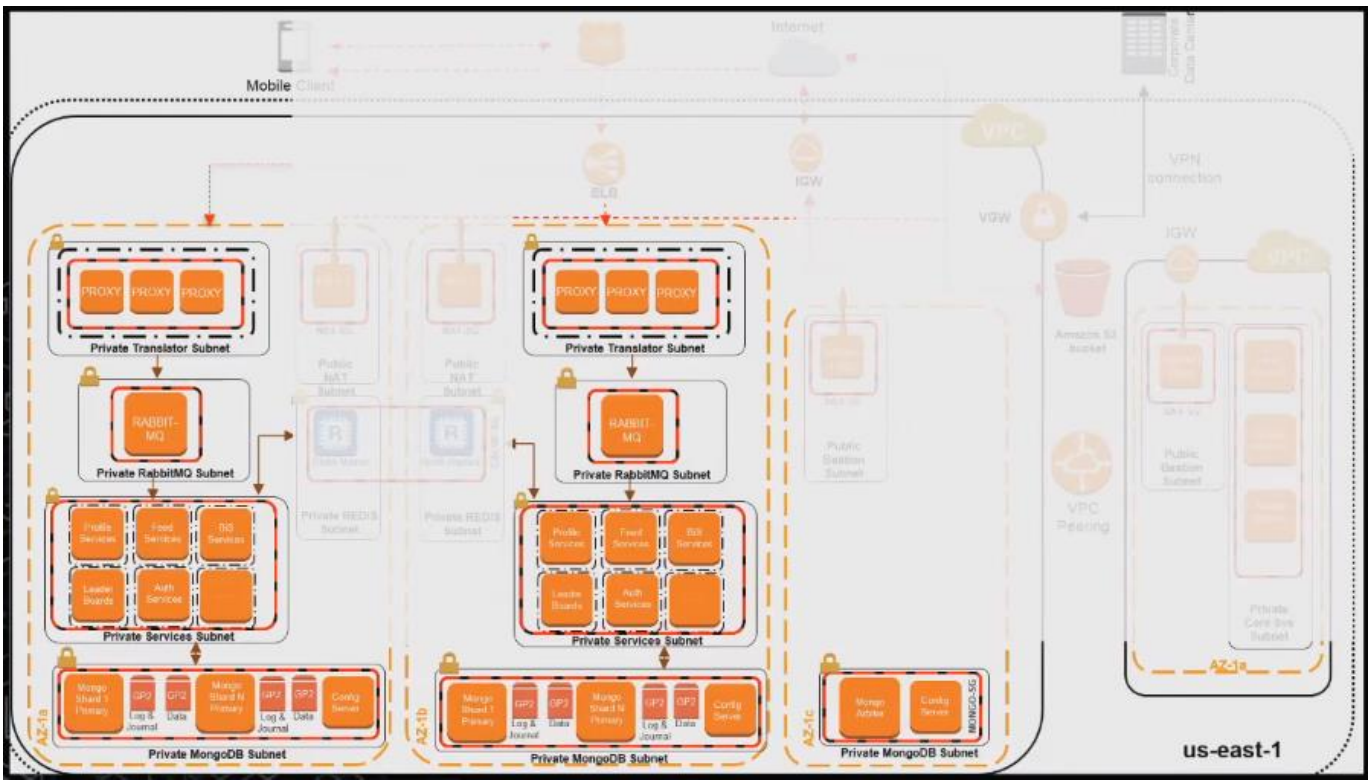
- Nine scalable components for just the app layer
- Scaling HA RabbitMQ
- MongoDB sharding
- Redis sharding



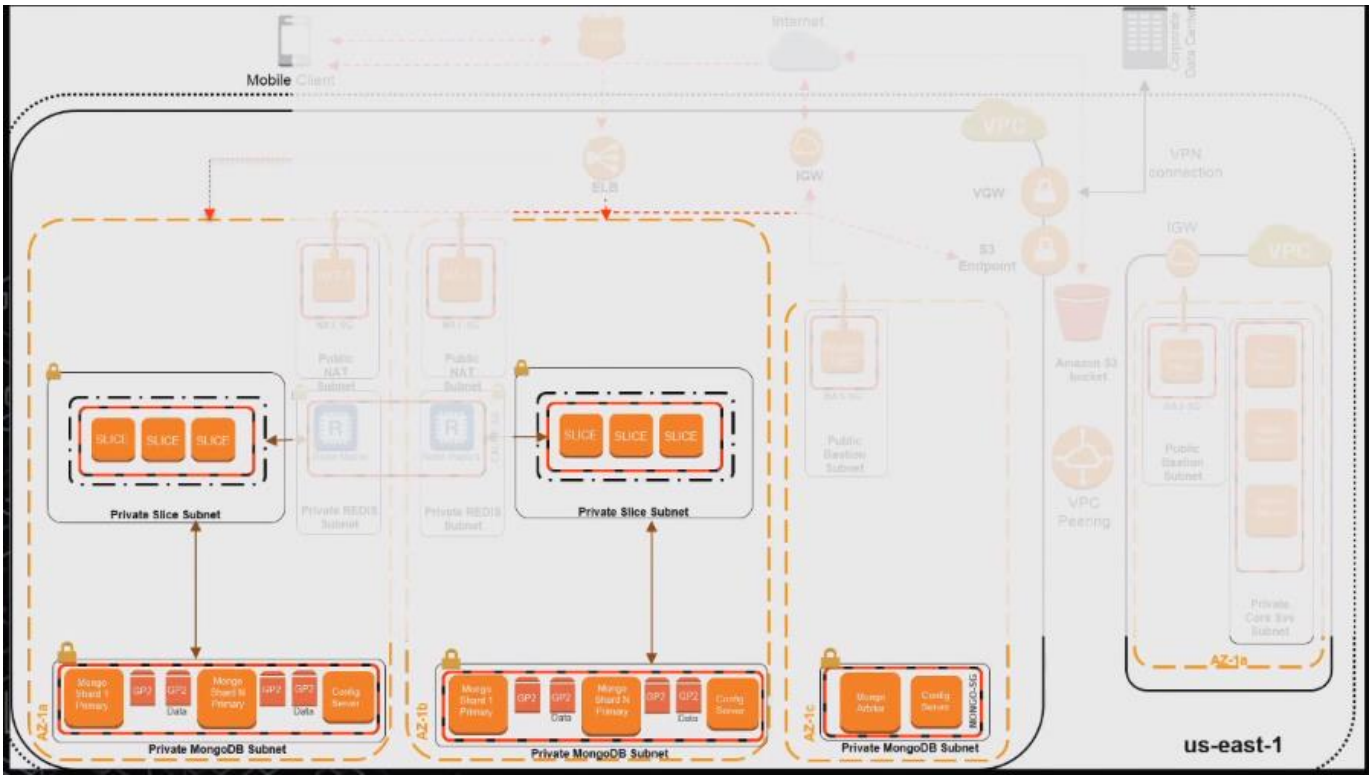
The key issues here are about how to scale RabbitMQ and scale MongoDB, all other components are in ASGs that automatically scale as needed.



We took all the application stack except MongoDB and baked them into an AMI, we can then scale this whole unit with an ASG.



We now reduced this to below



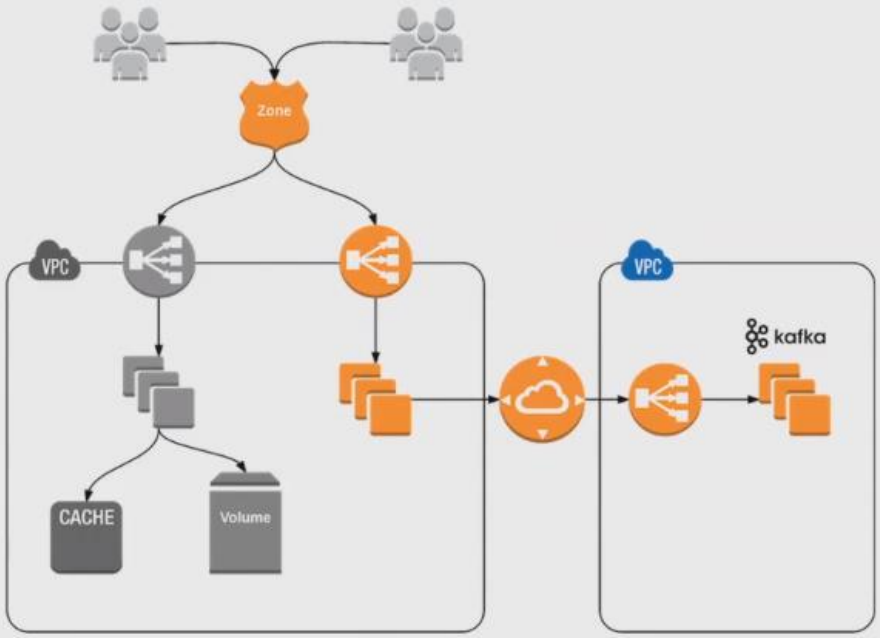
Challenge 3: Event log

- WBG services fire events, stored in Mongo
- Batch process scrapes and dumps daily

Pros
We already have the data “here”
Events from server-side callbacks
Clients already integrated

Cons
Huge Mongo performance hit
Huge Mongo storage footprint
Rabbit message rate amplification
Duration of scraper process
Large halo

MGP



Mobile game platform team

- Microservices architecture
- Go, Docker, CoreOS, AWS
- Next generation platform being written and deployed in parallel to WBG
- HTTP + REST
- Green-field

MGP Microservices

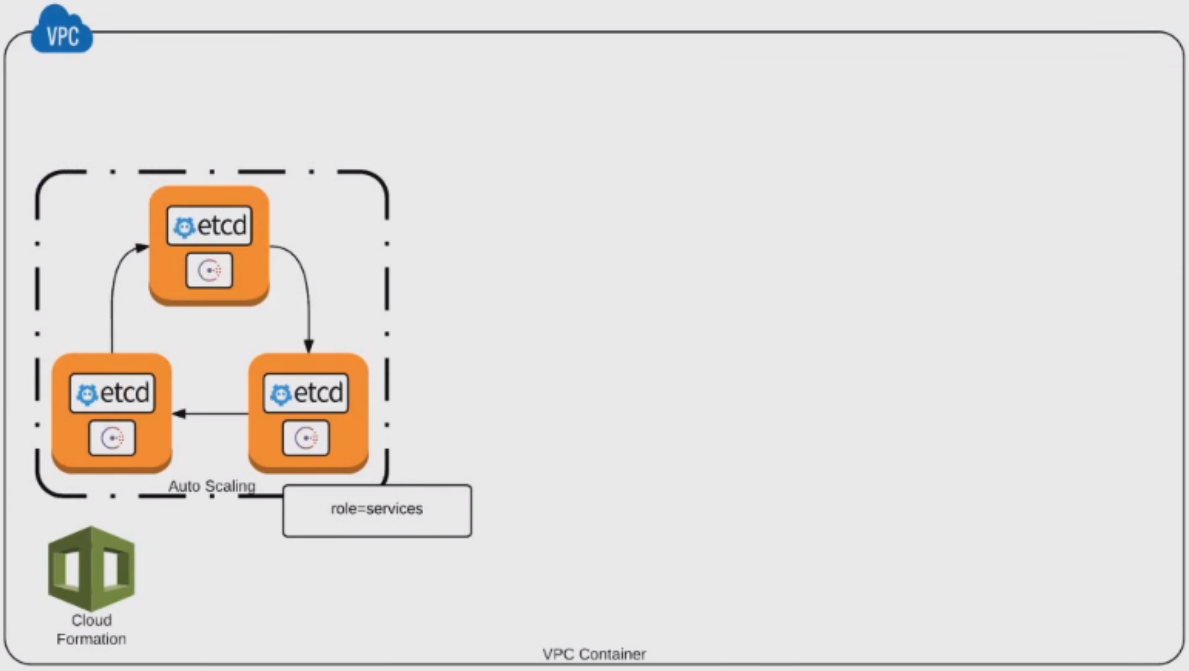
Feature	Benefit
HTTP + REST	API contracts minimize surprises
Do “One Thing”	Focused code base sanity
Datastore per-service	Per-service tuning and scaling
Config in Consul	HA central management with a good API
Config cached at startup	Quick lookups, protects from outages
Encryption key rotated every deploy	Automatic and easy secure secrets

At startup, each service will load its config from Consul (a K/V store) from Hashicorp.

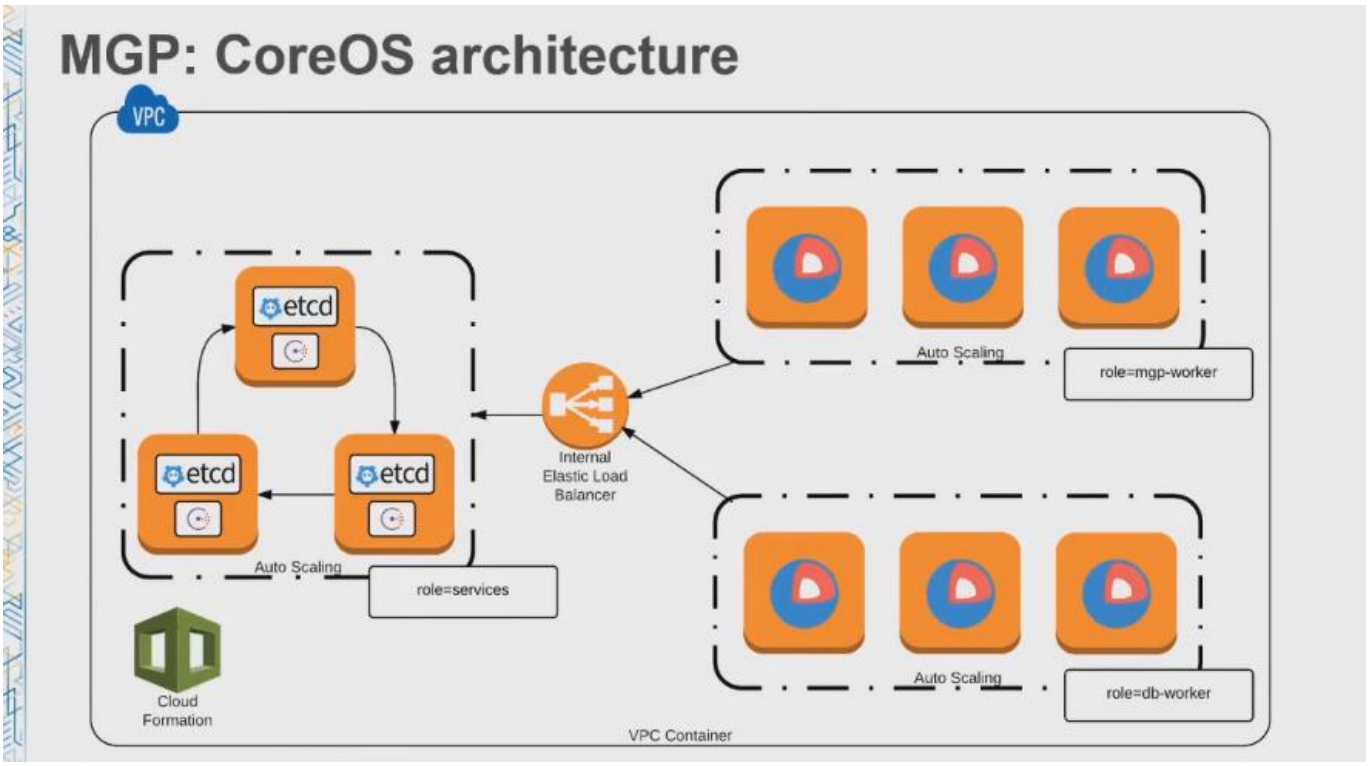
How we use containers

Feature	Benefit
From scratch containers	No OS in the container means tiny containers
<1MB on the wire	Tiny containers means ultra-fast deploys
Deployment conformity	Proprietary and third party apps deploy the same way
Deployment certainty	Git sha in tag ensures app version
Docker "--cap-drop-all"	Improved security stance

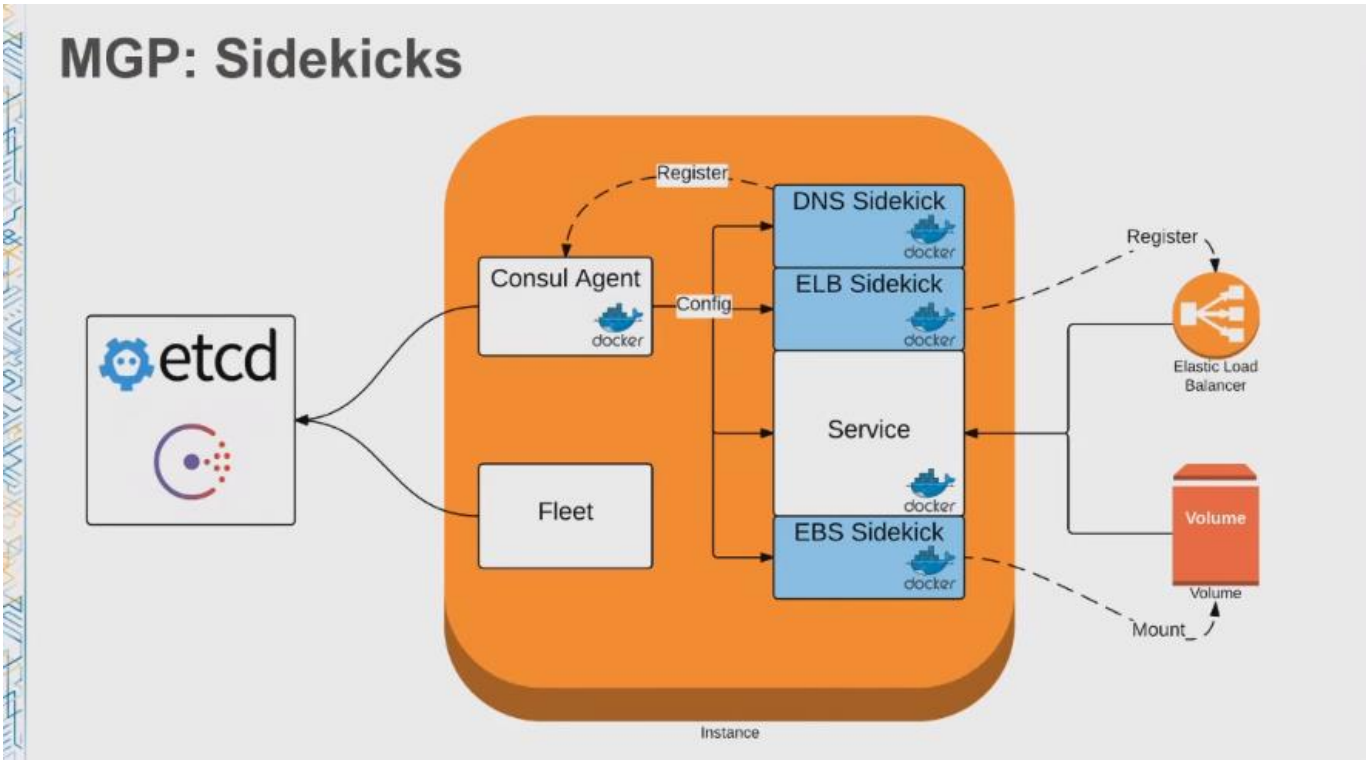
MGP: CoreOS architecture



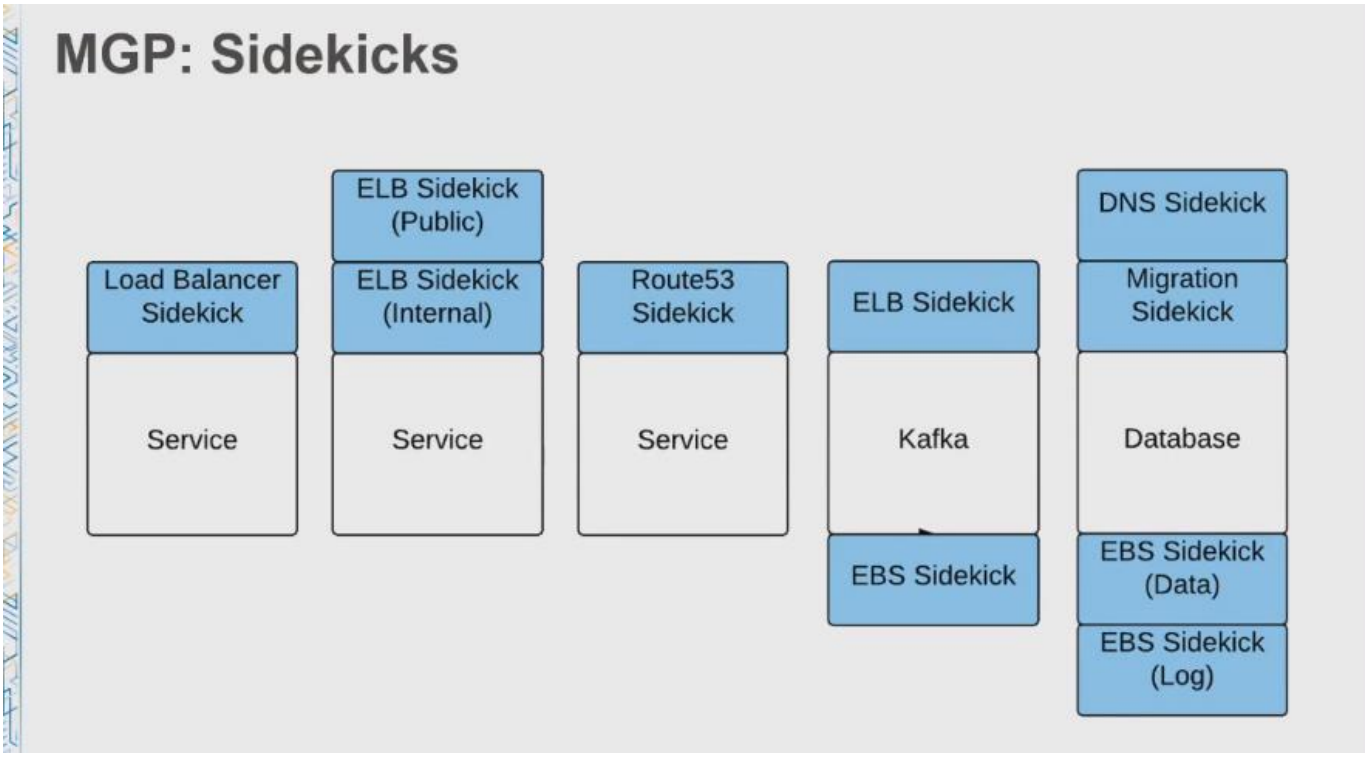
We run a central service cluster that helps maintains quorum and maintains cluster state, it runs Consul that we use for our configurations and for DNS lookups, it also runs etcd cluster for DNS access and healthchecks

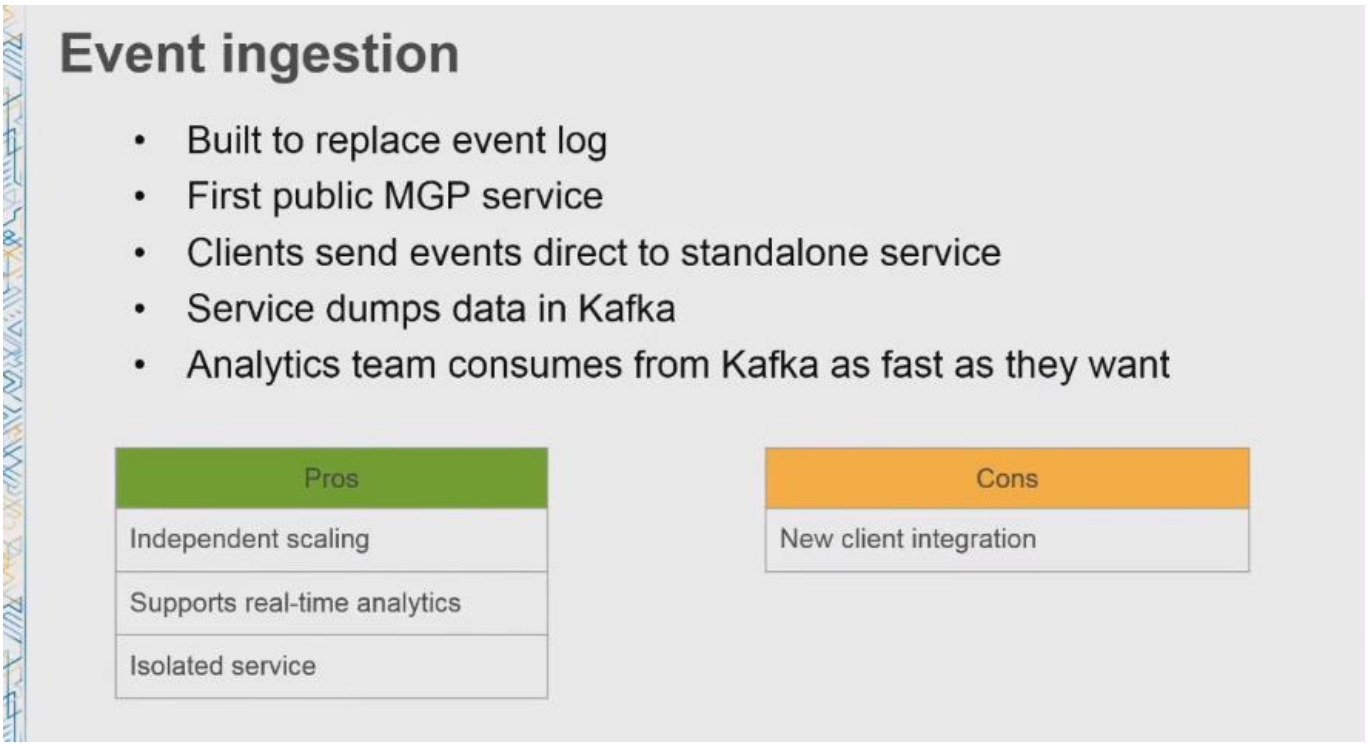


We then have a set of isolated ASGs for things like databases running within containers

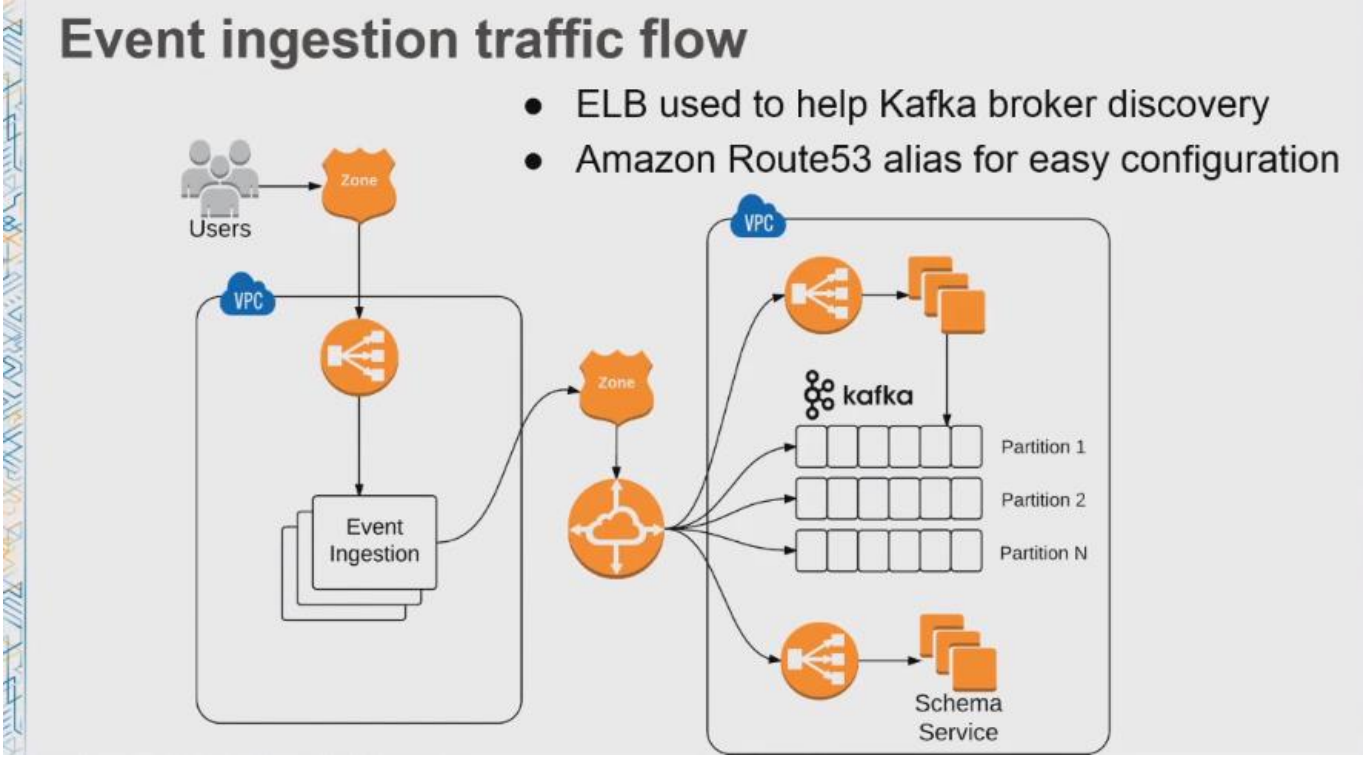


Each service uses sidekicks (separate containers) but its only job is to serve requests, the sidekicks help do things like getting EBS storage for the service, also a config sidekick that fetches the config for the service, the DNS sidekick that helps get DNS details for the service.

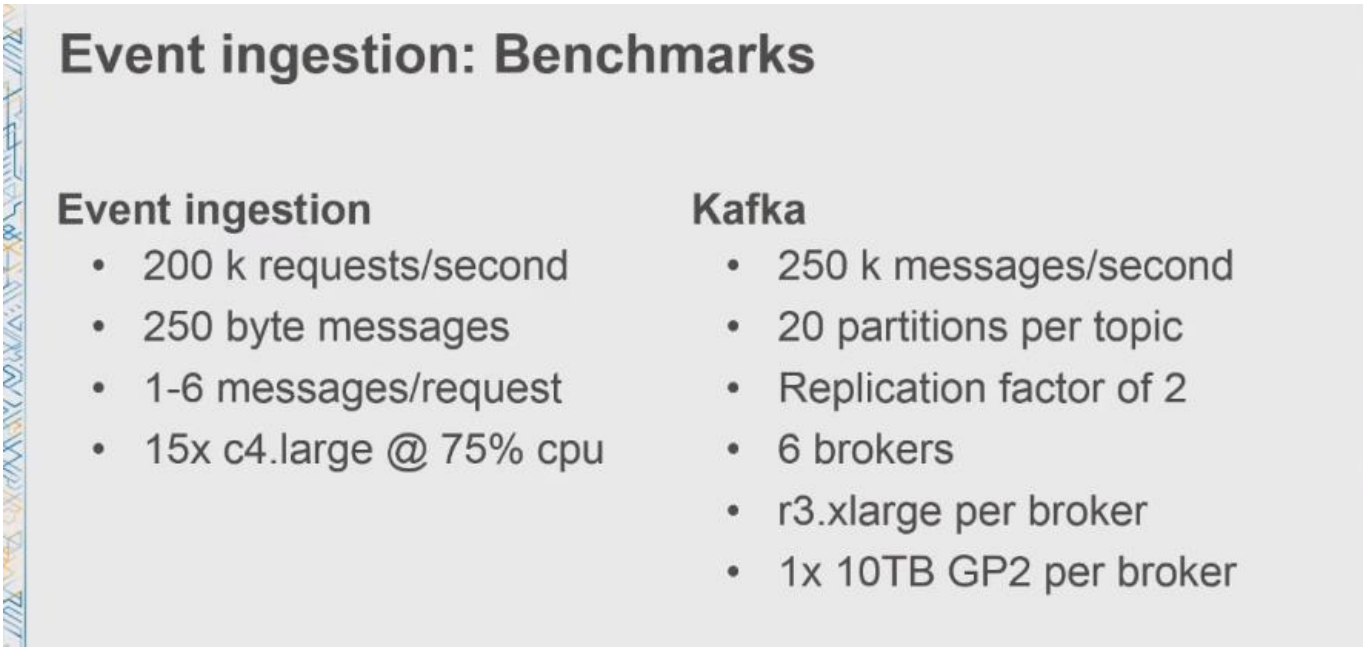




We ended up building an Event Ingestion service as above

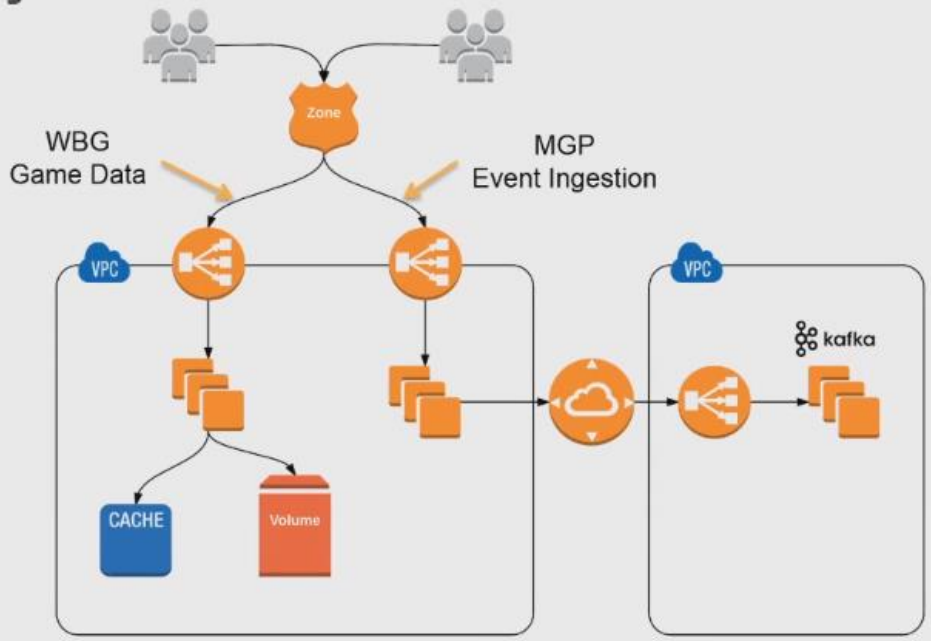


Traffic comes in through the ELB, we talk to the Event Ingestion service, then we go over the VPC Peering link to talk to Kafka. We also have a Schema service that holds Avro schemas which is a way we can have the game team publish what their events look like ahead of time so that once the events data gets to Analytics team, they don't have to infer what the events and data types mean. We can also reject events coming in with bad schemas.



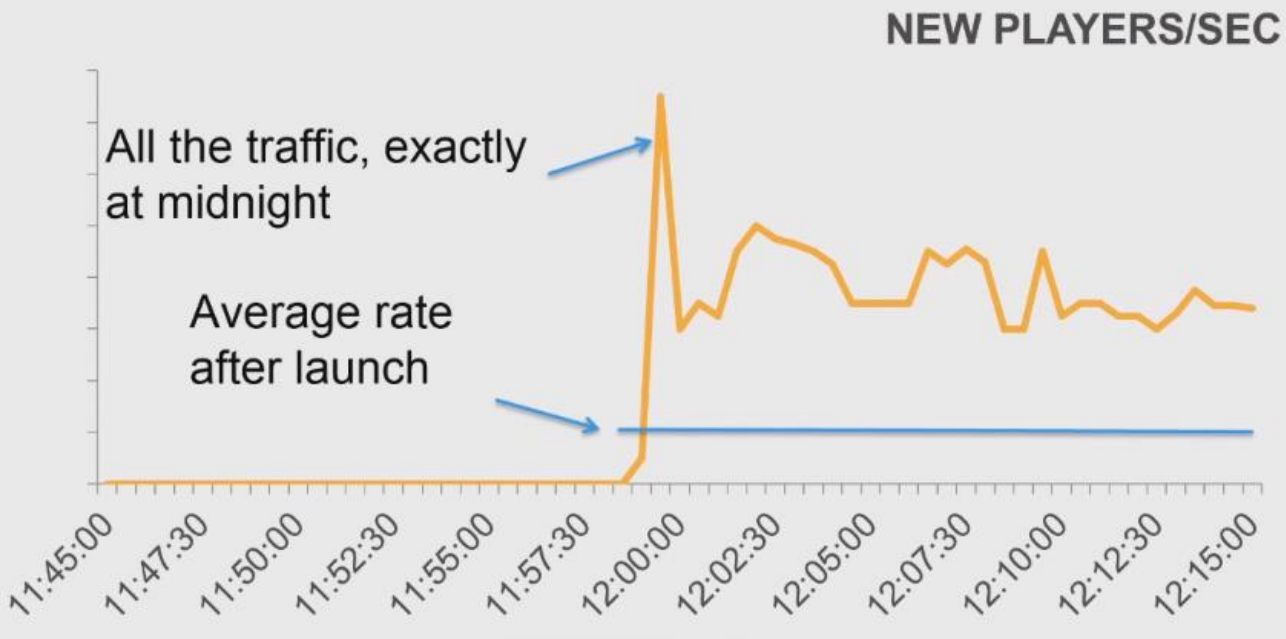
Kafka is used by all the games and the analytics stack, we can add more brokers to scale up for more messages

Side by side



More Challenges…Launch Time

Midnight release!



Challenge 4: MongoDB

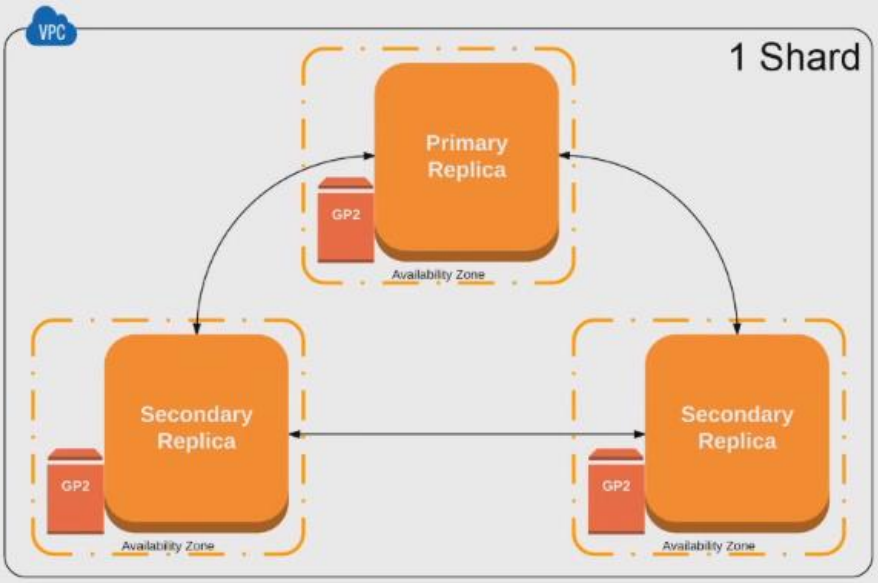
Three problems at scale

1. Mongodb Oplog data
2. Read queues under load causing service backlogs
3. Frequent Mongodb failovers

MongoDB architecture

- 3 node sharded replica set
- Multiple shards
- Mongo 2.6.9

- r3.2xl
- 2TB GP2
- 3 AZ's



MongoDB: Oplog data

Symptoms

- up to 100 of GB/h in oplog data per shard!
- TB of data per day to back up
- Unable to use off-the-shelf backup solutions

MongoDB: Oplog data

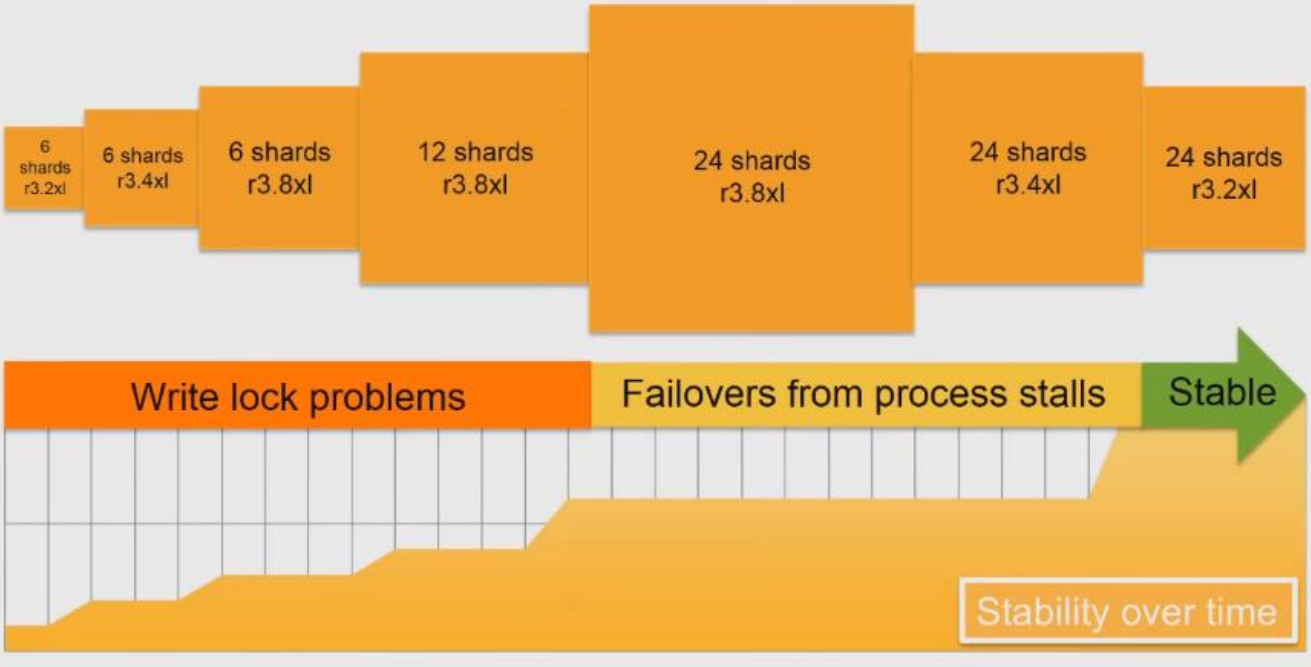
Why?

- Application bug read / wrote too much data
- Unexpectedly chatty clients
- Clients that did not send diffs

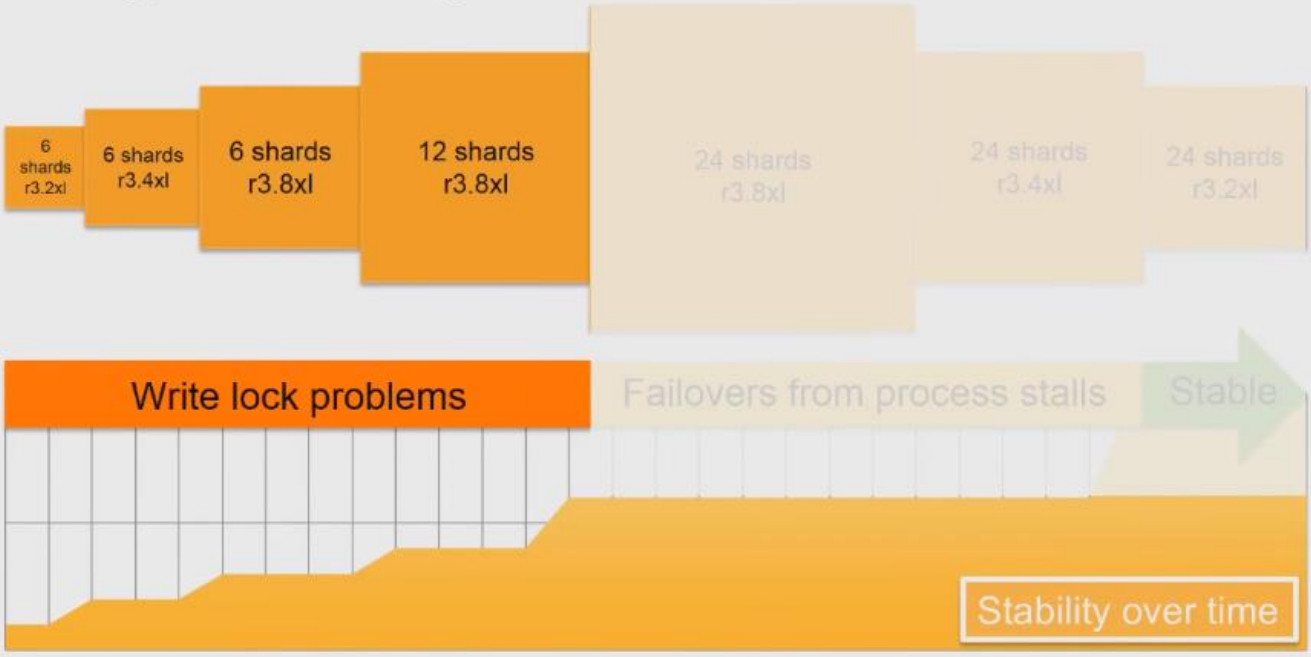
Solution

- Fixed bug (update only document keys that were changed)
- Patched client to send significantly less updates

MongoDB scaling timeline



MongoDB scaling timeline



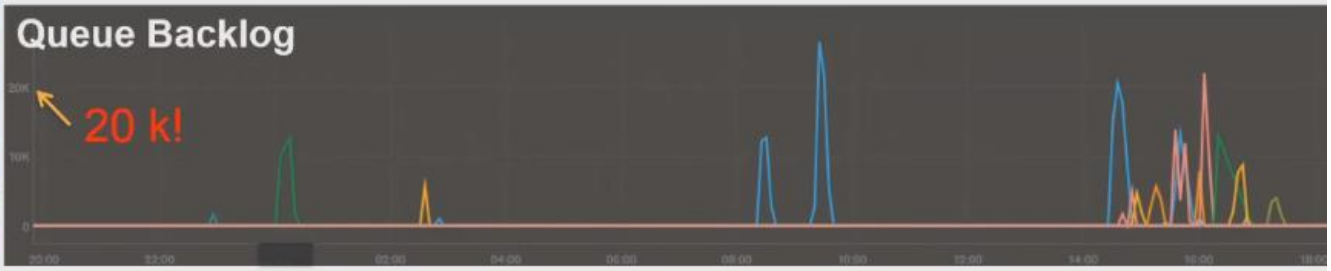
MongoDB: Read queue backlog

Symptoms

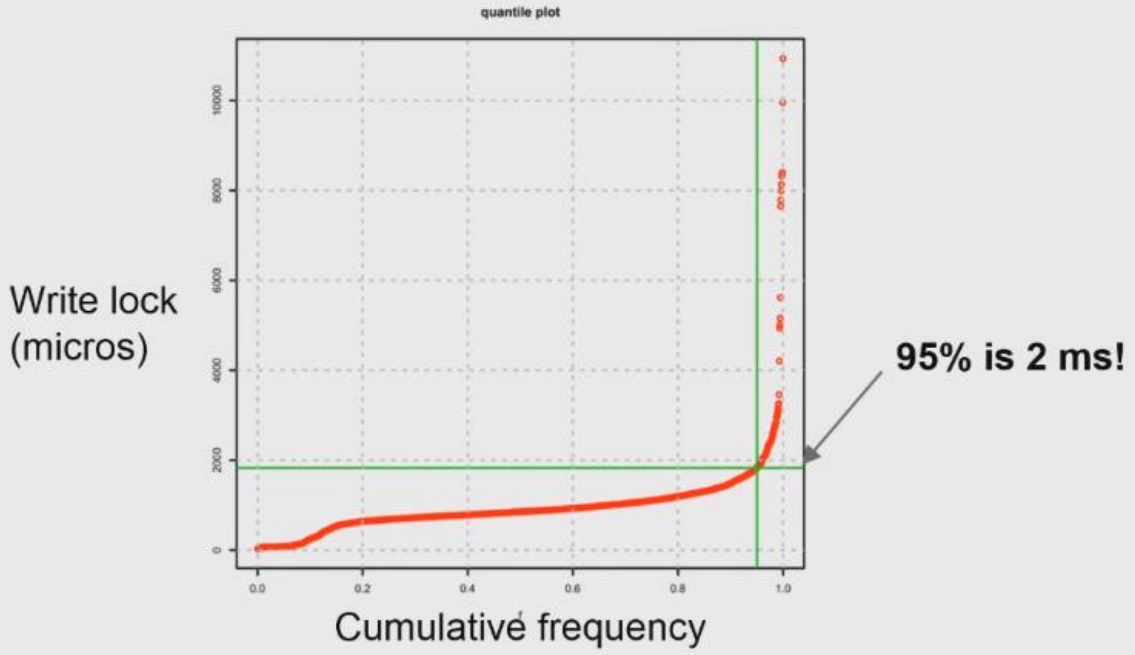
- Huge spikes in queued operations
- Predominantly read operations
- Relatively low lock percentage (~20-30)%

MongoDB: Read queue backlog

- 80 k reads/sec
- 3 k writes/sec
- 50 k–100 k write size
- 20% lock (average)
- No I/O bottleneck



MongoDB: Write lock



MongoDB: Read queue backlog

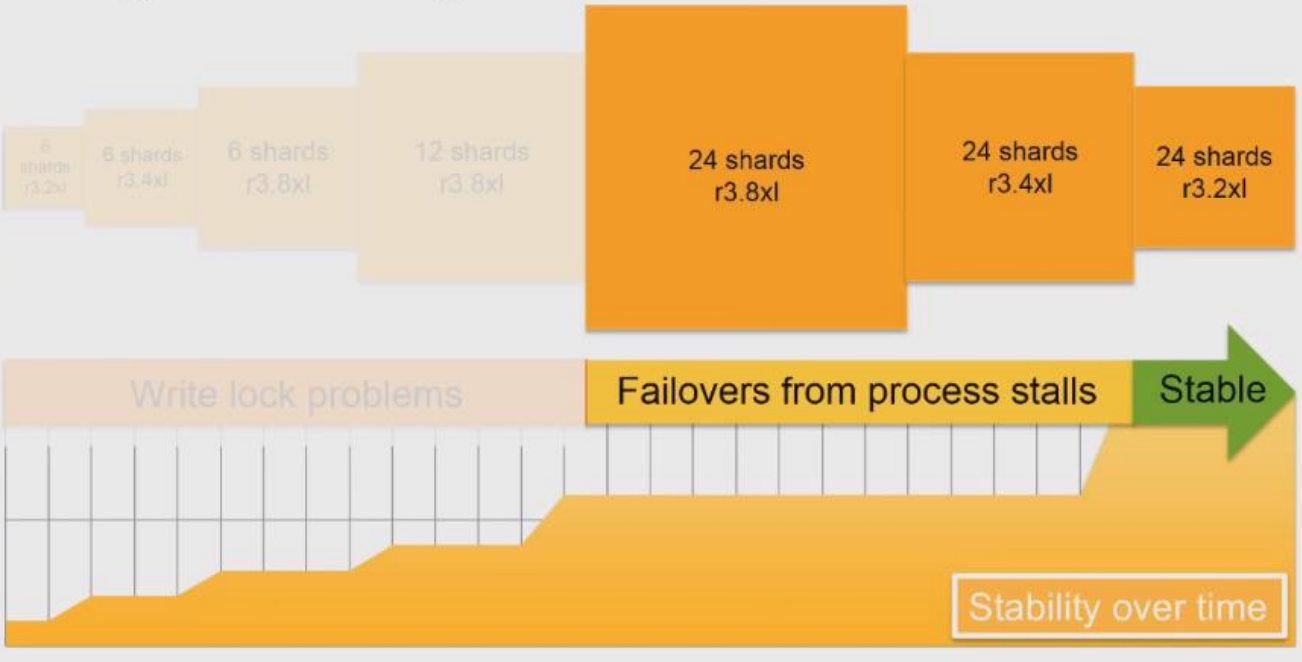
Why?

- Very long write locks increased impact of low lock percentage
- Linux kernel known issue, frequent and slow context switches
- Document moves

Solutions

- Newer Linux kernel (halved the number of context switches)
- **Doubled number of shards twice**

MongoDB scaling timeline



MongoDB: Mystery failovers

Symptoms

- Spikes in system CPU
- Mongod stall
- Mongo elections
- “Network” partition to replica set members
- Application queue backups

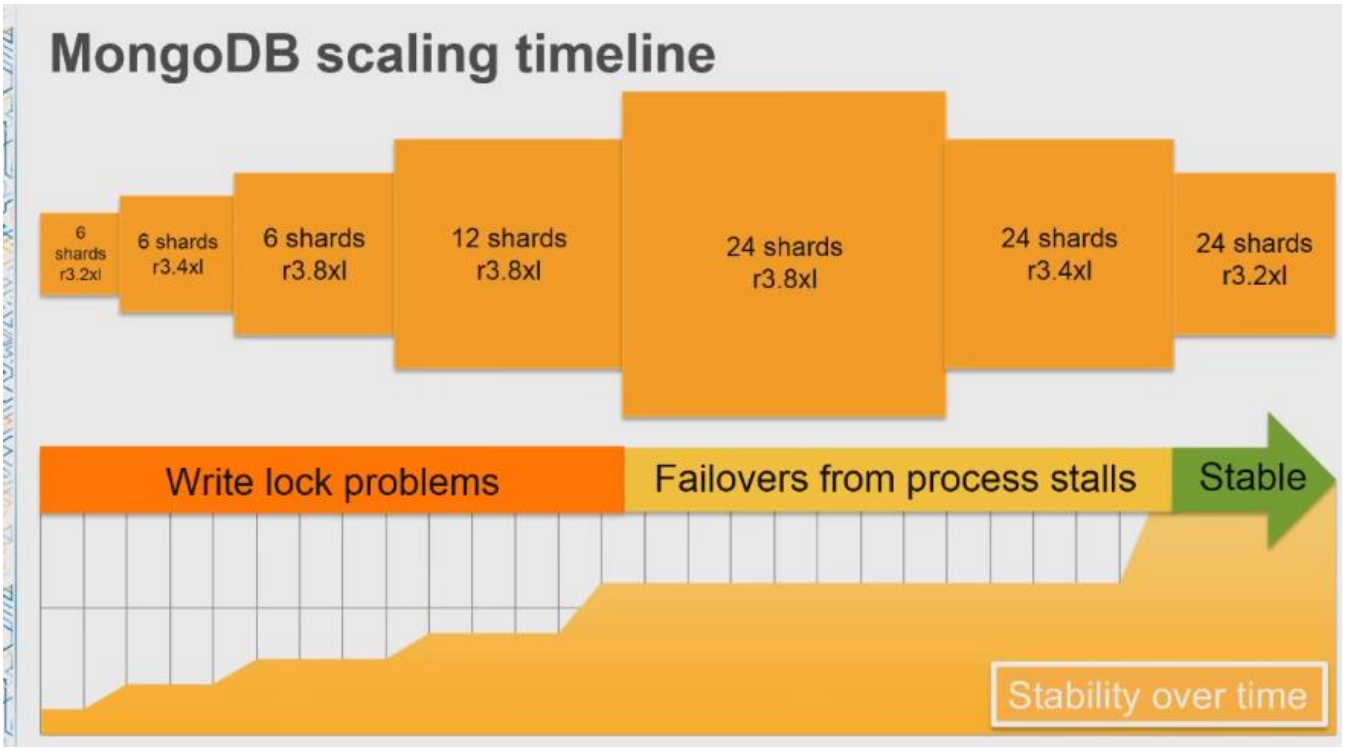
MongoDB: Mystery failovers

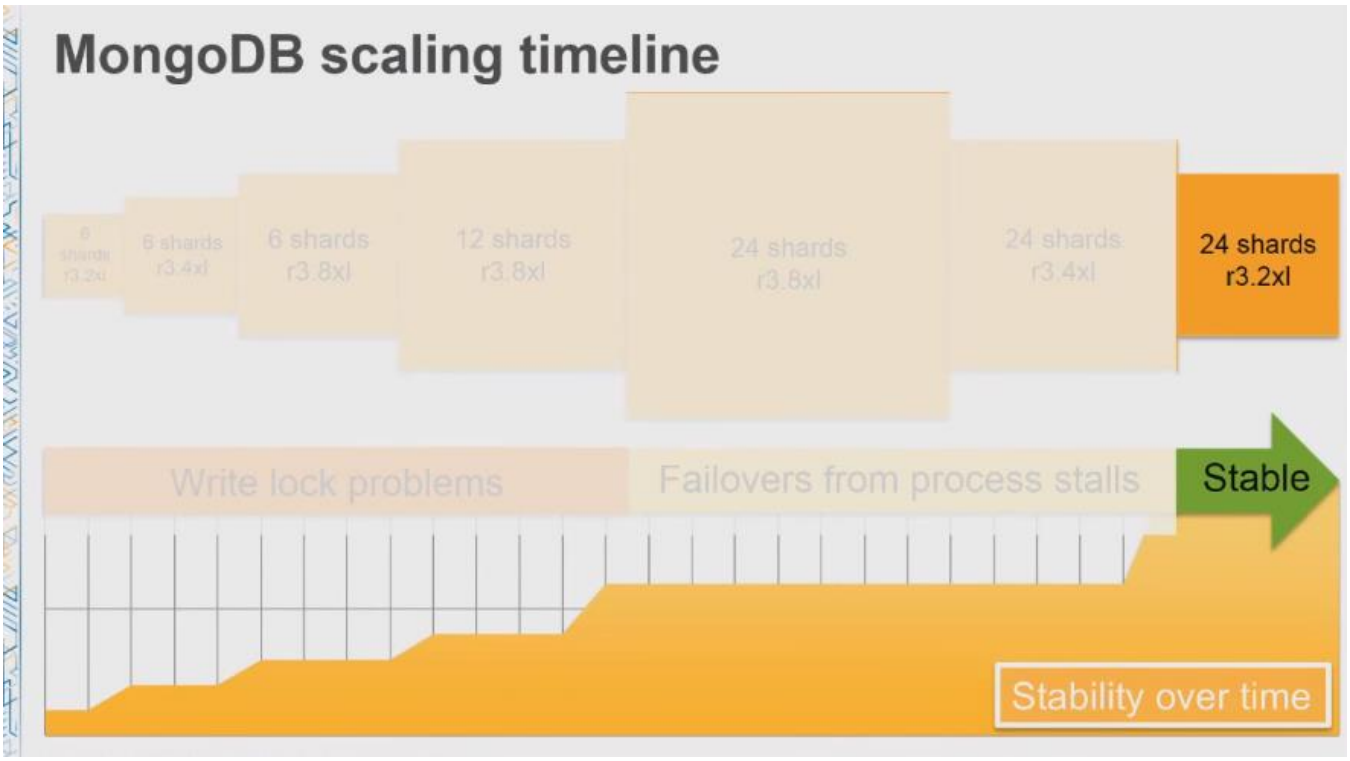
Why?

- Too much RAM
- Linux memory compaction taking too long
- Busy processes (mongod) had to wait

Solution

- Reduced instance size to r3.2xl
- Less memory = faster and more frequent memory compaction
- Scale out widely with smaller instances


MongoDB scaling timeline
6 shards r3.2xl
6 shards r3.4xl
6 shards r3.8xl
12 shards r3.8xl
24 shards r3.8xl
24 shards r3.4xl
24 shards r3.2xl
Write lock problems
Failovers from process stalls
Stable
Stability over time


MongoDB scaling timeline
24 shards r3.2xl
Write lock problems
Failovers from process stalls
Stable
Stability over time

CDF's are awesome for bottleneck analysis

```
import numpy as np
from pylab import *

writelocks = np.loadtxt("writelocks.csv")

num_bins = 100
counts, bin_edges = np.histogram(writelocks, bins=num_bins, density=True)
cdf = np.cumsum(counts*np.diff(bin_edges))

idx = next(i for i, v in enumerate(cdf) if v >= 0.95)

plot(bin_edges[1:], cdf)
plot([0, 10000],[0.95, 0.95])
plot([bin_edges[idx], bin_edges[idx]], [0, 1])
ylim([0,1])
show()
```

Challenge 5: Selective degradation

- Needed to buy time to solve back-end issues
- Game is live; we can't pull it
- Specific calls were causing stack problems
- Some calls were low value to the client, but high impact on the back end

Nginx is your friend

- Caching of "uncacheable" calls
- Rate limiting by single endpoint or service

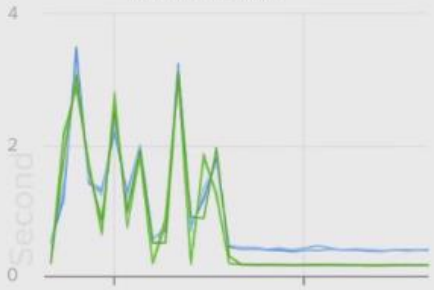
- Confd+Consul allowed for easy updates

Caching

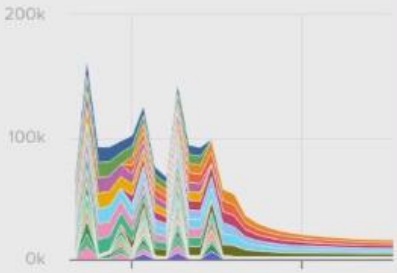
- Different definitions of “real time”
- Caching 2 endpoints saved 50% app layer network throughput
- Smoothed out everything

Mongo Lock %

ELB Latency



Established TCP Connections



Caching with Nginx

Some Problems...

- Nginx caches on disk
- Large requests are not good for Nginx proxy defaults
- Caching too much; needed to tune proxy_min_uses

- Small EBS root volume became an unexpected bottleneck
- Unexpected disk usage pattern also consumed all inodes with default ext4 settings

Throttling

- Focused on highest impact calls first
  - Poorly indexed queries
  - Chatty writes from clients
- Reduced the number of writes to allow reads
- Blocked unexpectedly bad but low-customer-impact calls
- Used detailed metrics

Problems with throttling

- Clients behave in different ways
- Multistep auth caused more failures than individual rates would imply
- Induced client retry storms
- Single failure forced some clients into offline mode

Post Mortem

Metric overload

- Don’t display graphs unless there’s a problem or someone will find one
- Measure everything = mysteries everywhere
  - Outliers
  - Because the Internet
  - Measure everything, show important things

Soft launch if you can

- Real players create real data
- Flexible APIs let clients do “flexible” things
- Look at your soft launch data and usage patterns

Your blast radius scales too

- Vendor contacts before launch
- Other internal team contacts too
- Let them know before something big happens
- Blast radius can be more than just you

Pick your battles

- Solve your biggest problem first
- Use AWS scalability to buy time to find root causes
- Make sure you have someone dedicated to triage
- Don’t need to rewrite everything
- Share APIs, not data

The future

- More microservices
- Embrace immutability
- More cross-team collaboration
- More Amazon support (Thanks, Dhruv)
- Come work for us!

Special thanks

Turbine: WBPlay
Turbine: MGP
Turbine: Cloud solutions and integration

Amazon – TAMs and Dhruv are the best
Librato support
MongoDB support
Splunk Cloud support

Contact us

Evan Pipho:
epipho@turbine.com

Romesh McCullough
rmccullough@turbine.com

William Day:
wday@turbine.com

www.turbine.com/careers

AWS re:Invent

Thank you!