A Series Of Unfortunate Container Events

Netflix's container platform lessons learned

NETFLIX

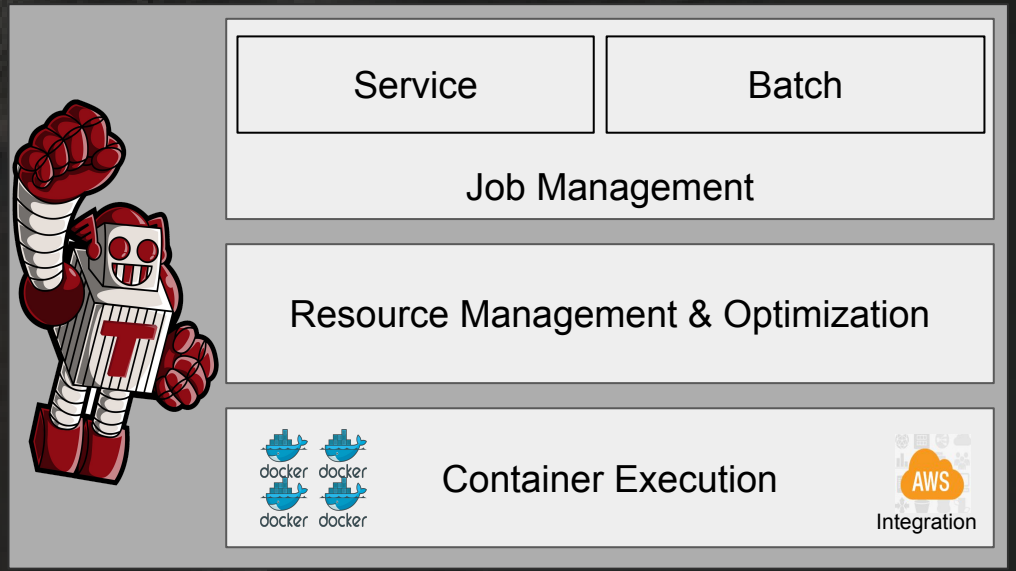
About the speakers and team

Follow along - @sargun @tomaszbak_ca @fabiokung

@aspyker @amit_joshee @anwleung

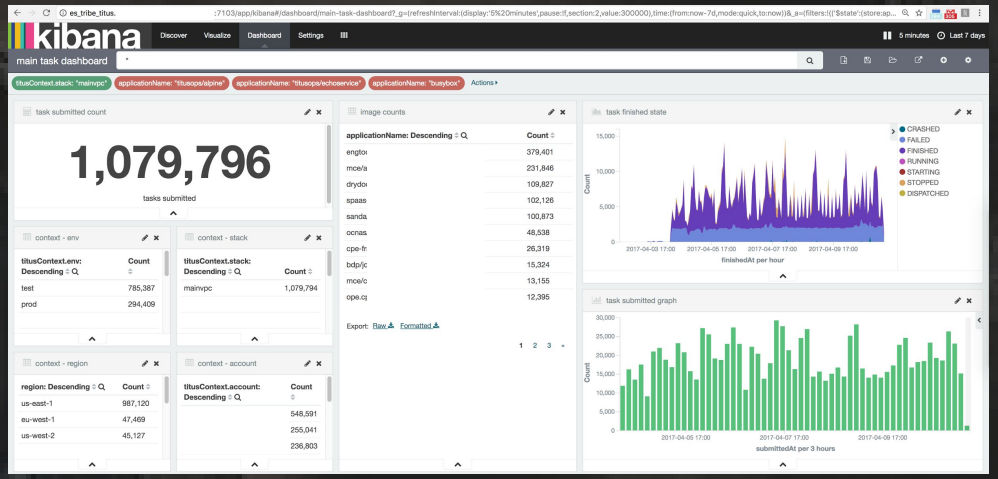
Netflix’s container management platform

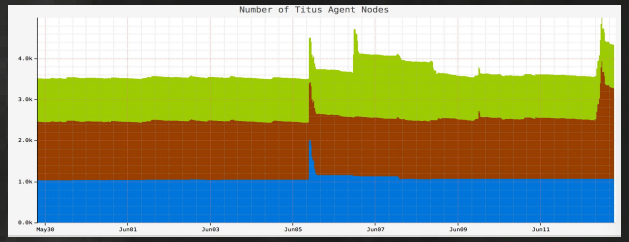
- Titus
- Scheduling
  - Service & batch jobs
  - Resource management
- Container Execution
  - Docker/AWS Integration
  - Netflix Infra Support

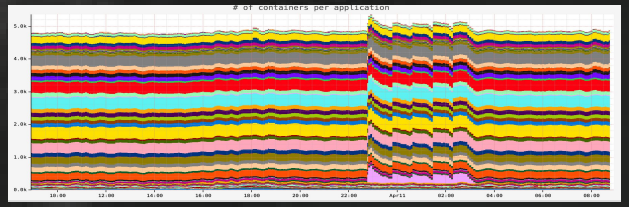


Containers In Production

Current Titus scale







- Deployed across multiple AWS accounts & three regions
- Over 5,000 instances (Mostly M4.4xls & R3.8xls)
- Over a week period launched over 1,000,000 containers
- Over 10,000 containers running concurrently

Single cloud platform for VMs and containers

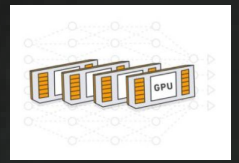
- CI/CD (Spinnaker)
- Telemetry systems
- Discovery and RPC load balancing
- Healthcheck, Edda and system metrics
- Chaos monkey
- Traffic control (Flow & Kong)
- Netflix secure secret management
- Interactive access (ala ssh)

Integrate containers with AWS EC2

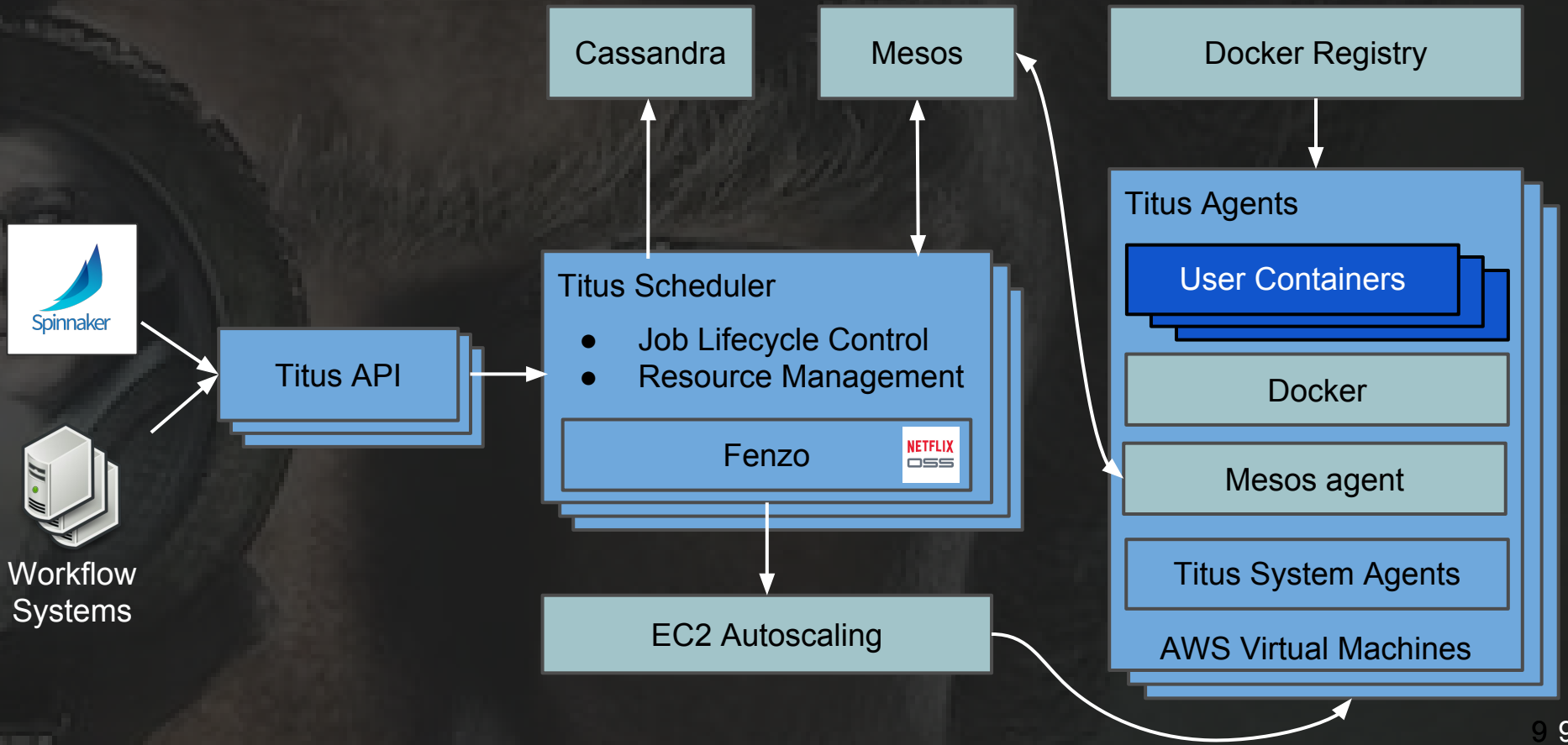
- VPC Connectivity (IP per container)
- Security Groups
- EC2 Metadata service
- IAM Roles
- Multi-tenant isolation (cpu, memory, disk quota, network)
- Live and S3 persisted logs rotation & mgmt
- Environmental context to similar to user data
- Autoscaling service jobs (coming)

Container users on Titus

- Service
  - Stream Processing (Flink)
  - UI Services (NodeJS)
  - Internal dashboards
- Batch
  - Personalization ML model training (GPUs)
  - Content value analysis
  - Digital watermarking
  - Ad hoc reporting
  - Continuous integration builds
  - Media encoding experimentation



Titus high level overview



Lessons learned from a year in production?

Look away, look away, Look away, look away

This session will wreck your evening, your whole life, and your day

Every single episode is nothing but dismay
So, look away, Look away, look away

Expect Bad Actors

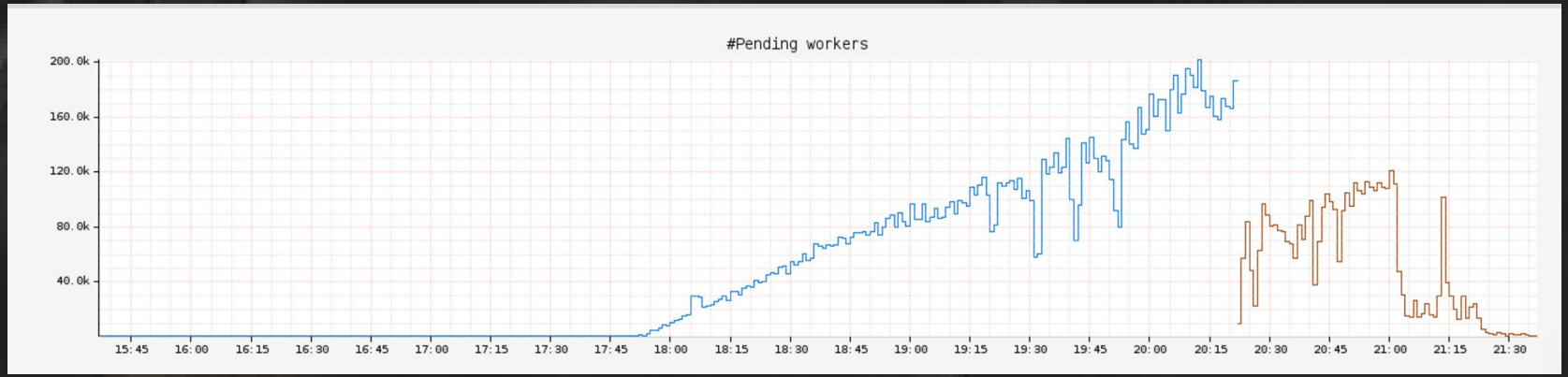
Run-away submissions

Submit a job, check status

If API doesn't answer

assume 404 and re-submit

Problem:



System perceived as infinite queue

Worked for our content processing job of 100 containers
Let's run our "back-fill" -- 100s of thousands of containers

Problems

- Scheduler runs out of memory
- All other jobs get queued behind

Solutions

- Scheduler capacity groups
- Absolute caps on number of concurrent live jobs
- Upstream systems doing ingest control

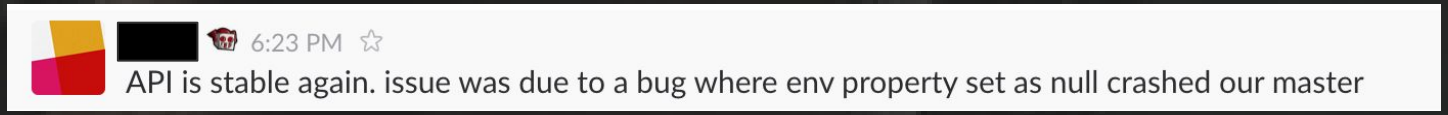
Invalid Jobs

Problems

- Scheduler crashes, fails over, crashes, repeat

Solutions

- Input validation, input fuzz testing, exception handling



Failing jobs that repeat

Image: “org/imagename:lateest”
Command: /bin/besh -c ...

Problems

- Containers can launch FAST! Can be restarted FAST!
- Scheduler works really hard
- Cloud resources allocated/deallocated FAST

Solutions

- Rate limiting of failing jobs

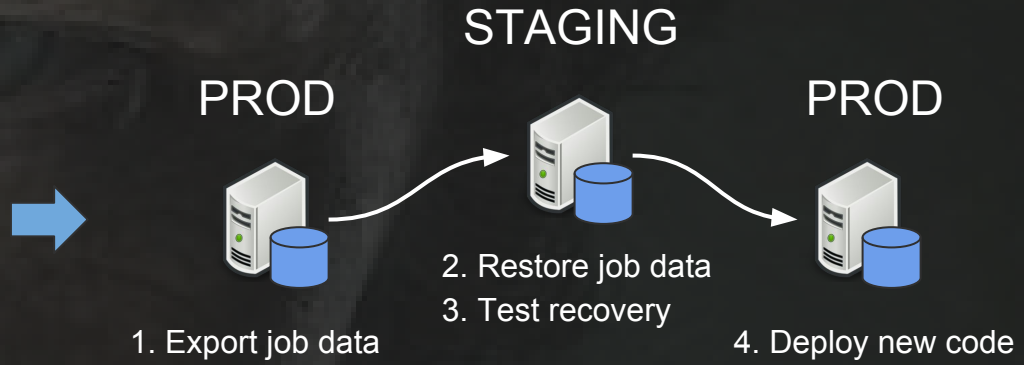
Testing for “bad” job data

Problems

- Scheduler fails, can’t recover due to “bad” jobs

Solutions

```
(cass_titus) ~ $ cqlsh4
Connected to cass_titus at 100.67.88.206:7104.
[cqlsh 5.0.1 | Cassandra 2.1.17.1428 | DSE 4.8.1
Use HELP for help.
cqlsh> delete from "JobStageData" where ...
```



Manual removal of bad job state? ✖

Test production data sets in staging

✔

Identifying bad actors

V2 API
- user (optional)

V2 Auditing
- Added collection of user performing action

V3 API
- Owner -> teamEmail (required)

Really bad actors - container escapes protections

- User Namespaces
  - Docker 1.10 - User Namespaces (Feb 2016)
  - Docker 1.11 - Fixed shared networking NSs
    - User id mapping is per daemon (not per container)
  - Deployed user namespaces recently
    - Problems - shared NFS, OSX LDAP uid/gid’s
- Locked Down hosts
  - Users only have access to containers, not hosts
  - Required “power user” ssh access for perf/debugging

The Cloud Isn’t Perfect

Cloud rate limiting and overall limits

Let's do a red/black deploy of 2000 containers instantly

Problems

- Scheduler and distributed host fleet ... no problem!
- Cloud provider … problem!

Solutions

- Exponential backoff with jitter on hosts
- Setting expectations of maximum concurrent launches
- Rate limiting of container scheduling and overall number of containers

Hosts start or go bad

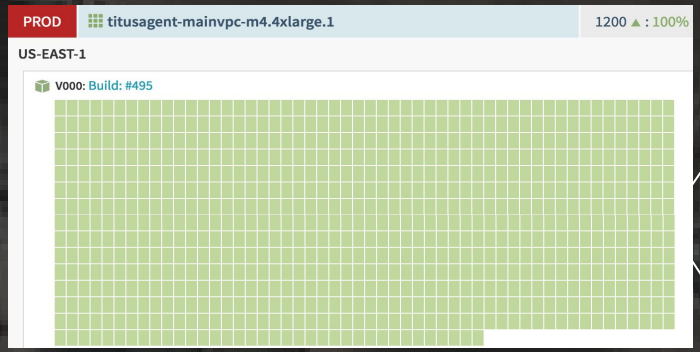
Problems

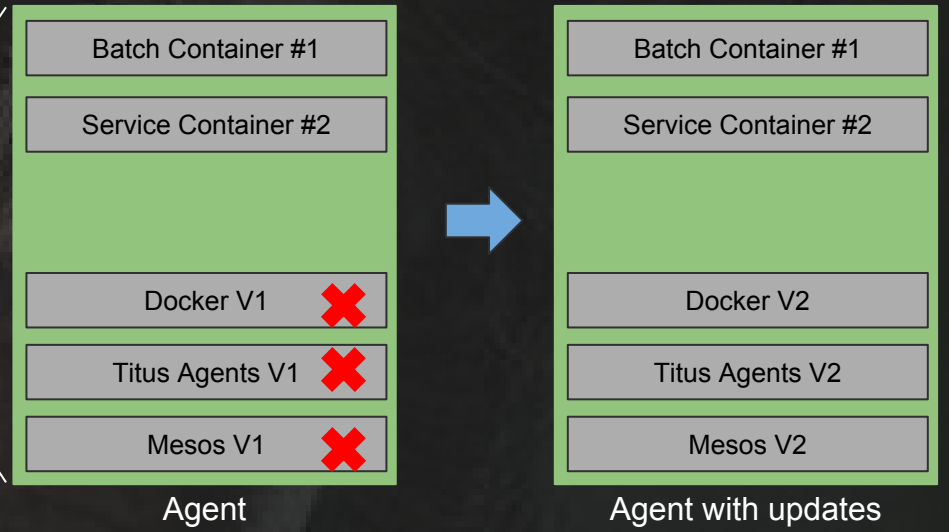
- Hosts come up with flakey networks
- Host disks come up and are slow
- Hosts go bad over time

Solutions

- Scheduler must be aware of host health checks
- Linux, storage, etc warming
- Auto-termination if hosts take too long to become healthy

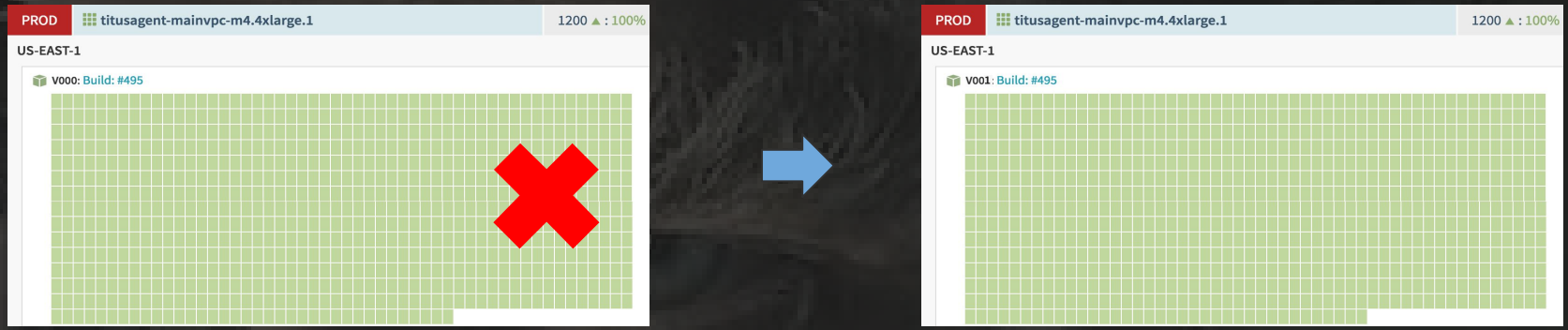
Upgrades - In place upgrades





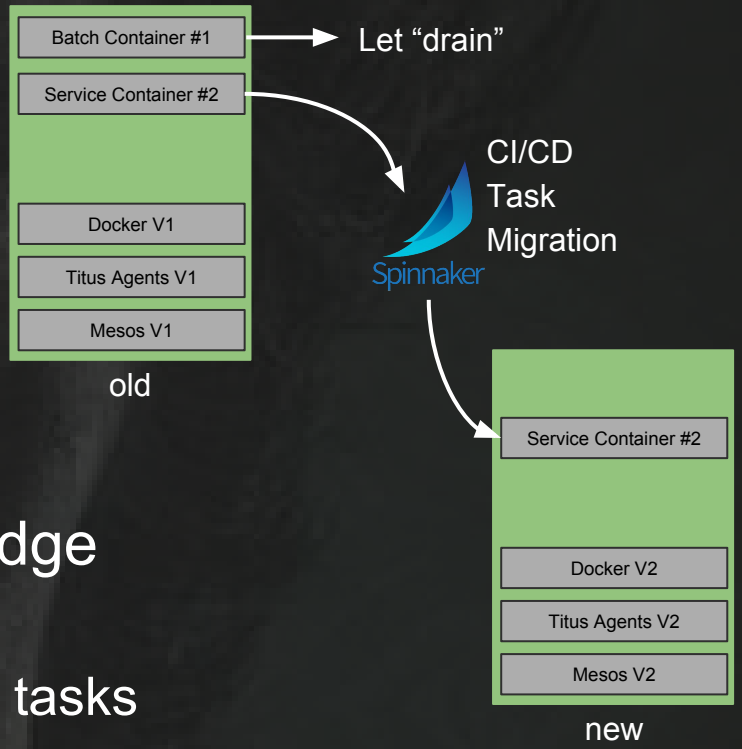
- Simpler for container users
- Infrastructure becomes mutable
- Doesn’t leverage elastic cloud
- How to handle rollback?

Upgrades - Whole cluster red/black



- Full red/black takes hours
- Costly (duplicate clusters)
- Insufficient Capacity Exception (ICE)
- Rollback requires **ALL** containers to move twice

Upgrades - Partitioned cluster updates



- Requires complex scheduler knowledge
  - Batch jobs to have runtime limits
  - Service jobs with Spinnaker migration tasks
- Starting point for fleet cluster management

Our Code Isn’t Perfect

Disconnected containers

Problem

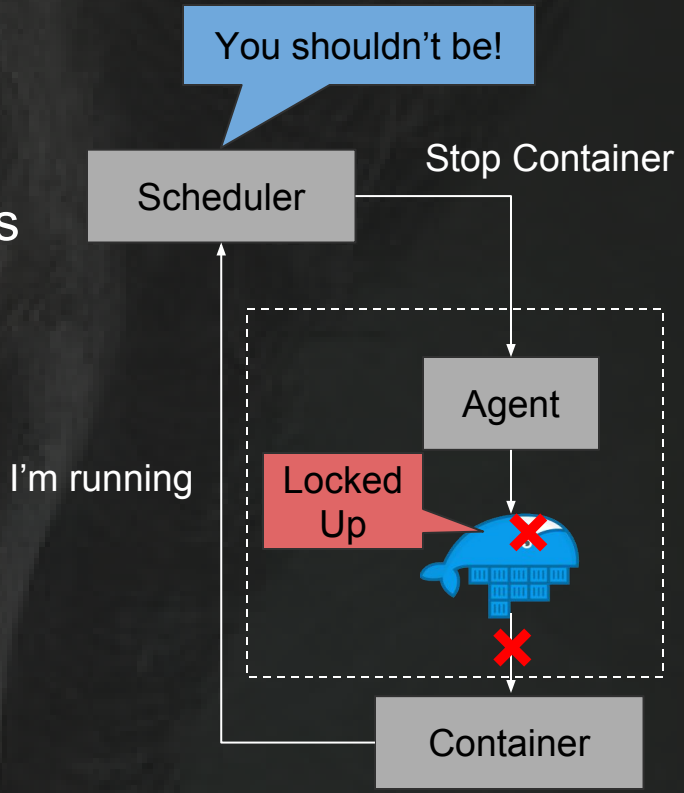
- Host agent controllers lock up
- Control plane can’t kill replaced containers

User

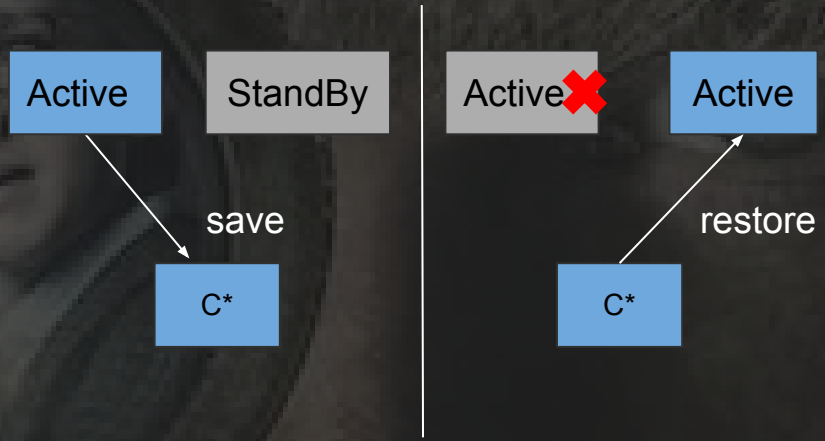
- Why is my old code still running?

Solutions

- Monitor and alert on differences
- Reconcile to this system as aggressively as possible



Scheduler failover speed is important



Scheduler failover time increased with scale

- Loss of API availability
- Reconciliation bugs caused task crashes

Solutions:

- Data sharding (current vs old tasks)
- Do as little as possible during startup

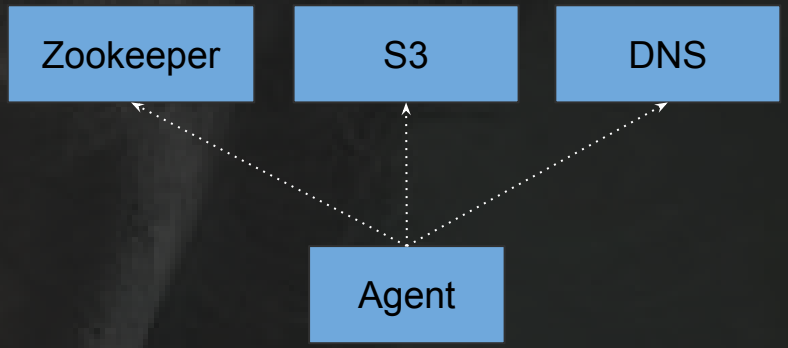
Know your dependencies

Problems

- Container creation errors
- Logs upload failure
- Task crashes

Solutions

- Retries
- Rate limiting
- Isolation



Containers require kernel knowledge

- Containers start with Docker .. end with the kernel

- Best container runtimes deeply leverage kernel primitives
  - Resource Isolation
  - Security
  - Networking etc

- Debugging tools (tracepoints, perf events) not container aware
  - Need for BPF, Kprobe

Strategy: Embrace chaos

Problems

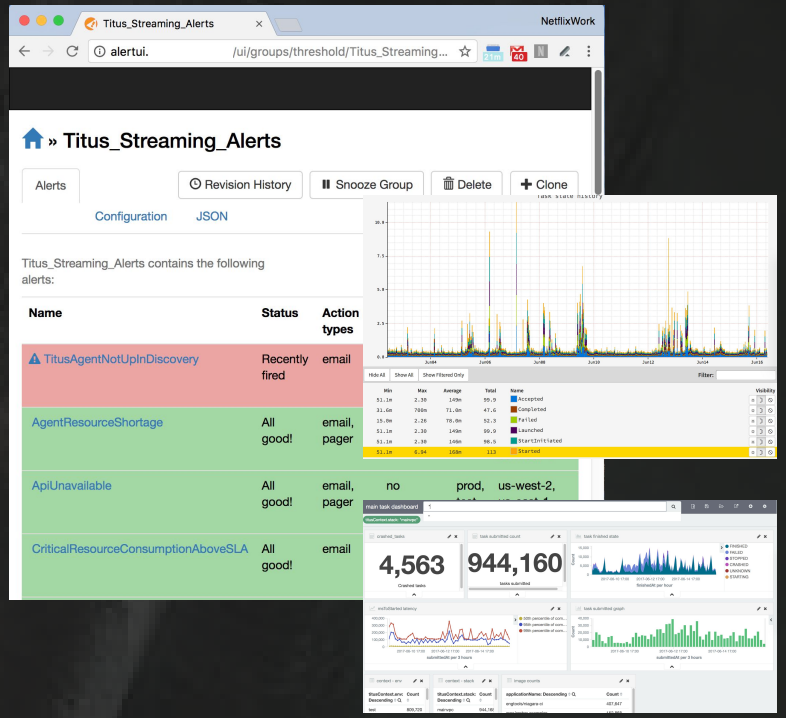
- Our instances fail
- Our code fails
- Our dependencies fail

Solutions

- Learn to love the Chaos Monkey
- Enabled for prod and all services (even our scheduler)

Alerting and dashboards key

Telemetry system	Number
Metrics	100’s
Dashboard Graphs	70+
Alerts	50+
Elastic Search Indexes	4



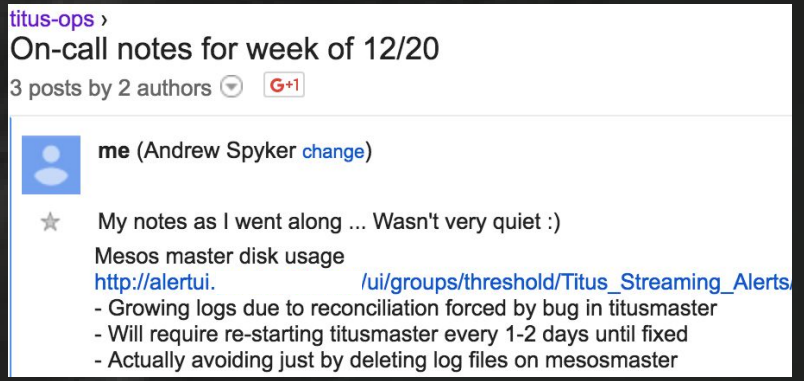
Very complex system == very complex telemetry and alerting

Continuously evolving

- Based on real incidents and resulting deep analysis

Temporary ad hoc remediation

- Manual babysitting of scheduler state
- Pin high when auto-scaling and capacity management isn't work
- Automated for each ssh across all nodes
  - Detecting and remediating problems


titus-ops ›
On-call notes for week of 12/20
3 posts by 2 authors
G+1
me (Andrew Spyker change)
My notes as I went along ... Wasn't very quiet :)
Mesos master disk usage
http://alertui. /ui/groups/threshold/Titus_Streaming_Alerts
- Growing logs due to reconciliation forced by bug in titusmaster
- Will require re-starting titusmaster every 1-2 days until fixed
- Actually avoiding just by deleting log files on mesosmaster

What has worked well?

Solid software

Docker Distribution

- Our Docker registry
- Simple redirect on top of S3

Zookeeper

- Leader Election
- Isolated

Apache Mesos

- Extensible resource manager
- Highly reliable replicated log

Cassandra

- CDE Internal Service
- Careful of access model

Managing the Titus product

Focus on core business value

- Container cluster management

Features are important

- Deciding what not to do …
  Just as important!

Deliberately chose NOT to do

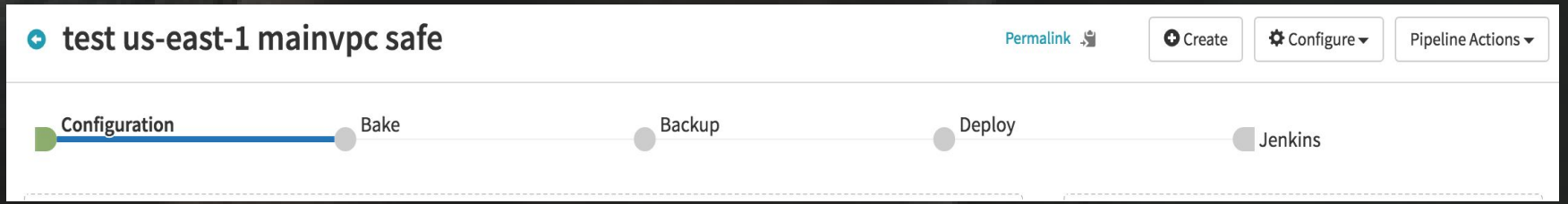
- Service Discovery / RPC
- Continuous Delivery

Ops enablement

Phase 1: Manual red/black deploys

Phase 2: Runbook for on-call

Phase 3: Automated pipelines



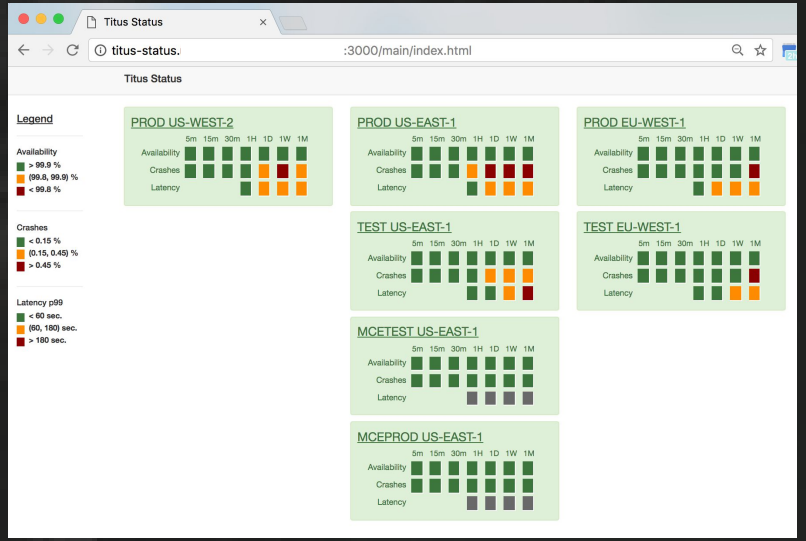
Service Level Objectives (SLOs)

Problem

- If you aren't measuring, you don't know
- If you don't know, you can't improve

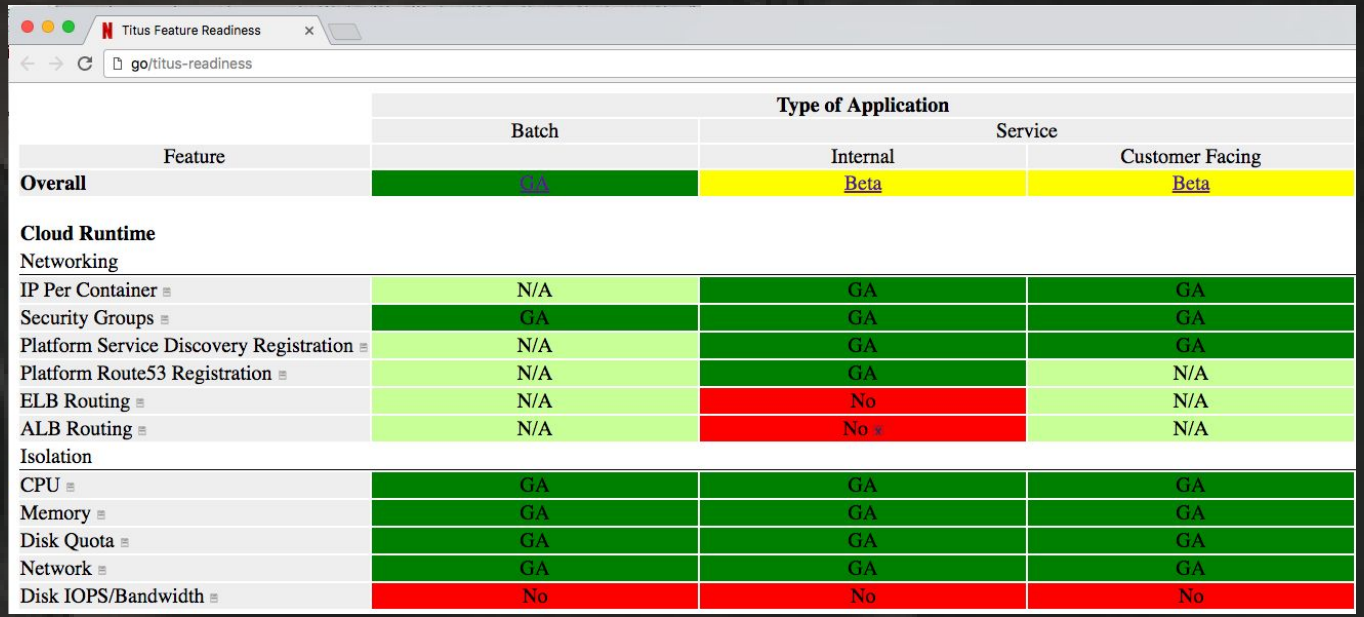
Solutions

- Our SLOs
  - Start Latency
  - % Crashed
  - API Availability
- Once we started watching, we started improving



Onboarding slowly

Documenting container readiness



	Type of Application		
	Batch	Service	
Feature		Internal	Customer Facing
Overall	GA	Beta	Beta
Cloud Runtime			
Networking			
IP Per Container	N/A	GA	GA
Security Groups	GA	GA	GA
Platform Service Discovery Registration	N/A	GA	GA
Platform Route53 Registration	N/A	GA	N/A
ELB Routing	N/A	No	N/A
ALB Routing	N/A	No *	N/A
Isolation			
CPU	GA	GA	GA
Memory	GA	GA	GA
Disk Quota	GA	GA	GA
Network	GA	GA	GA
Disk IOPS/Bandwidth	No	No	No

- Broken down by type of application and feature set
- Readiness expressed in
  - Alpha (early users), beta (subject to change), GA (mass adoption)

Growing usage slowly, carefully

First Scale Production Service

4Q 2016

Netflix Customer Facing Service

2Q 2017

NETFLIX

shadow

Titus Created Batch GA

4Q 2015

Service Support Added

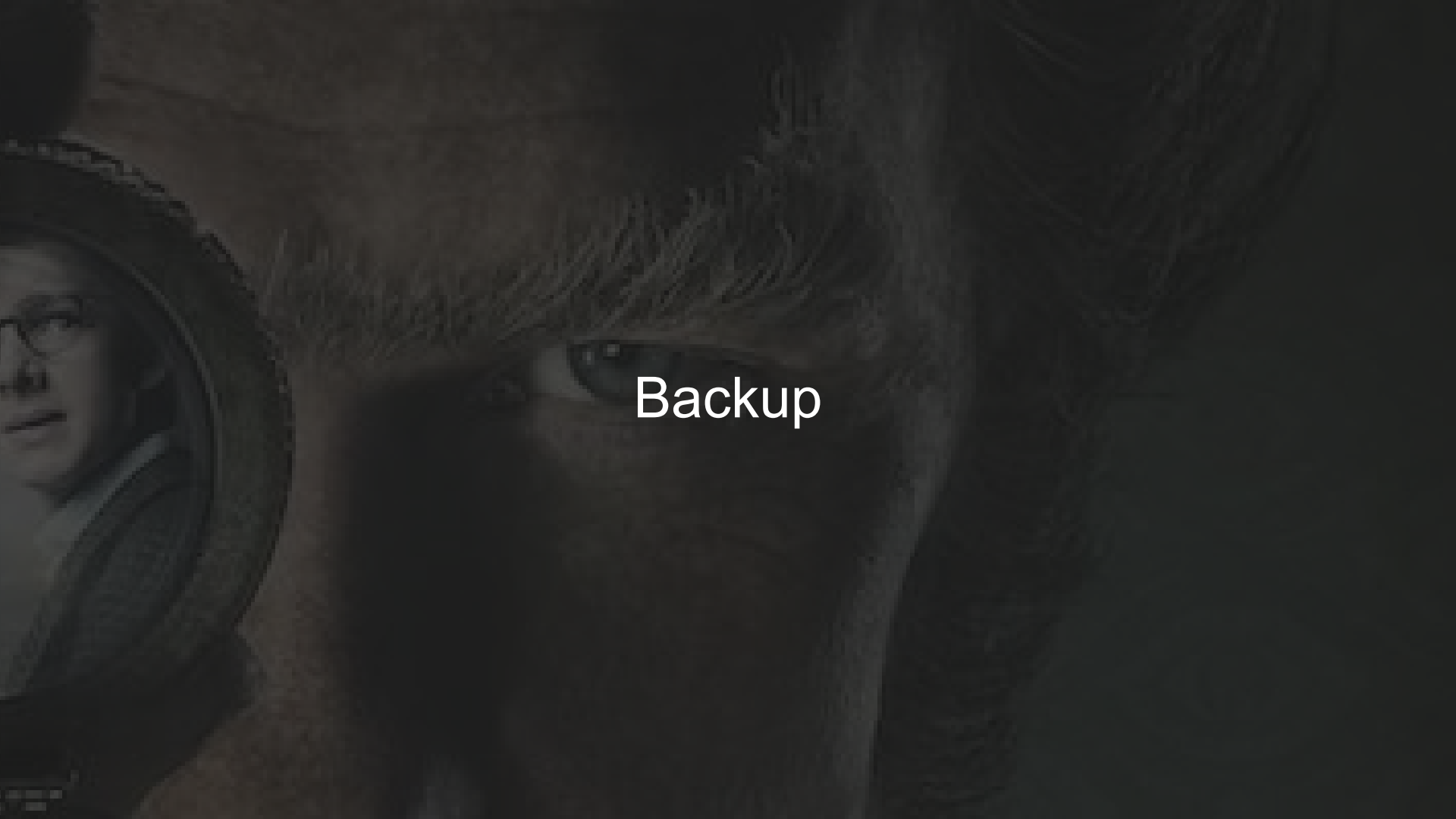
1Q 2016

Key takeaways

- Expect problematic containers & workloads
- Continued need for cloud to evolve for containers
- Container schedulers, runtime are complex
- Ops enablement key for production systems
- Users need help adopting containers responsibly

- Worth the effort due to value containers unlock

Questions?


Backup

Titus High Level Overview