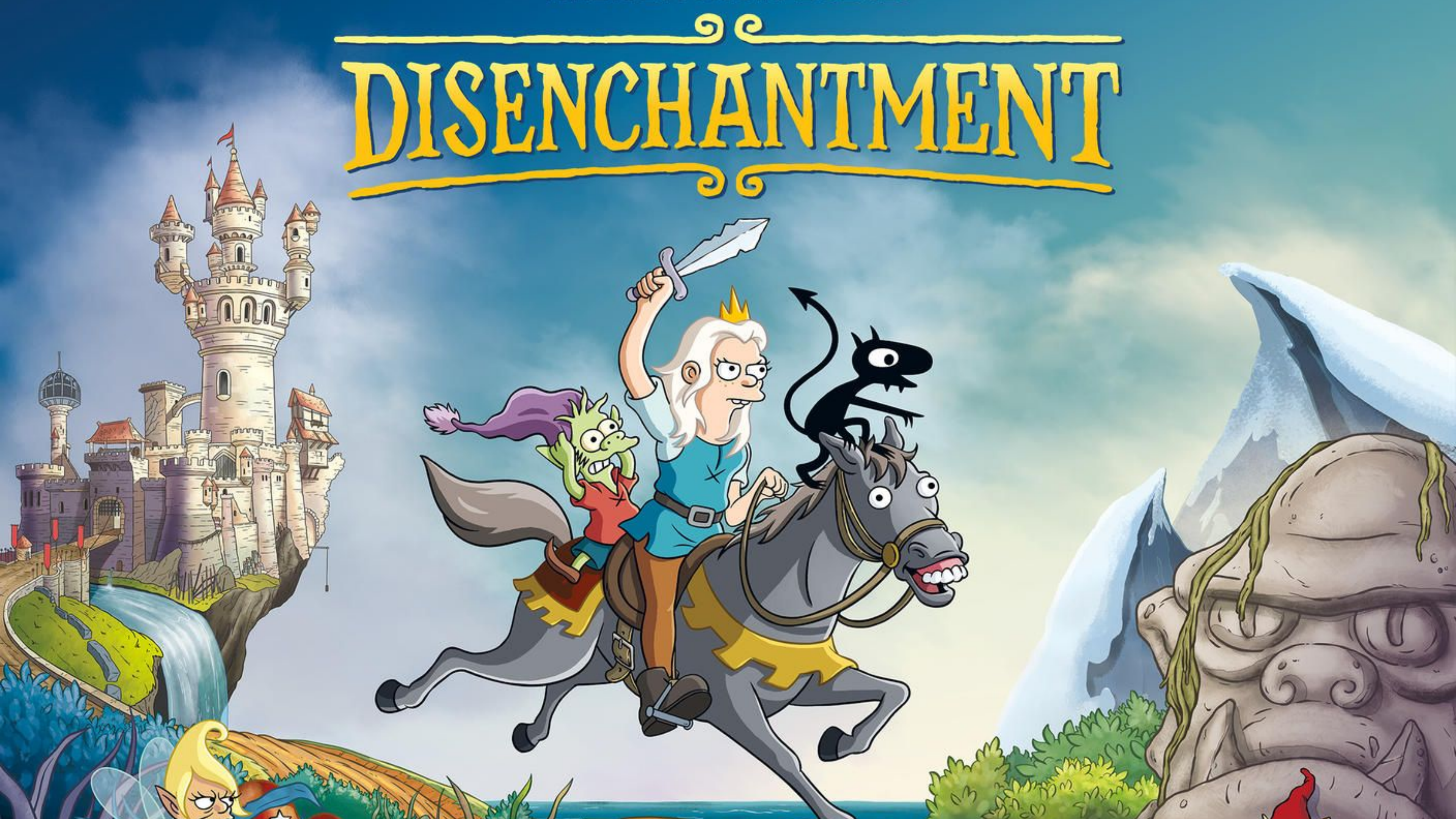
DISENCHANTMENT™

Netflix Titus, its Feisty Team, and Daemons


DISENCHANTMENT

Titus - Netflix's Container Management Platform

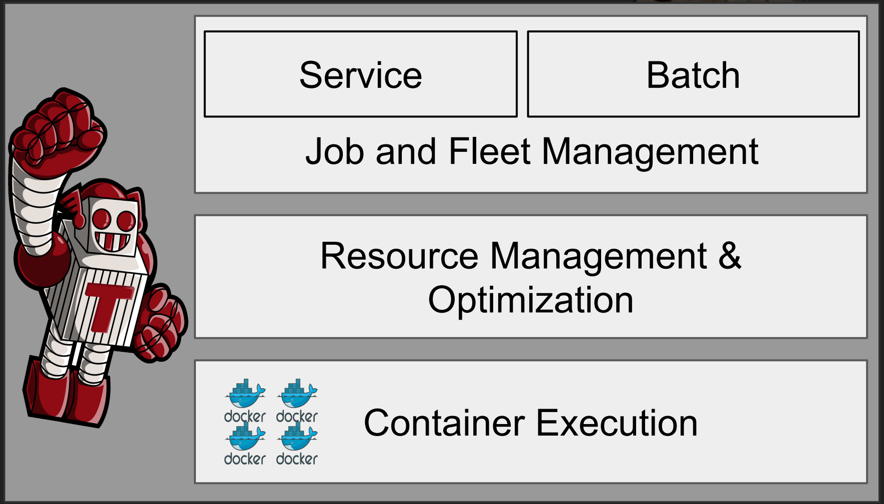
NETFLIX

Scheduling

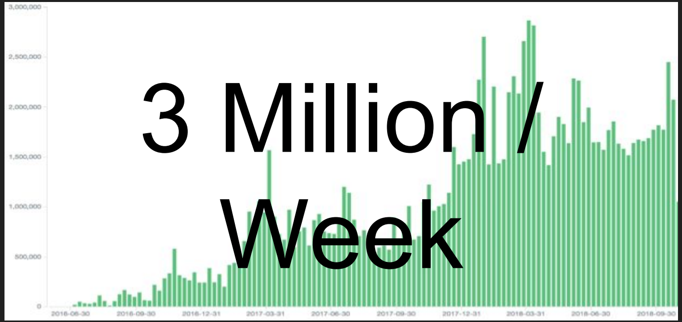
- Service & batch job lifecycle
- Resource management

Container Execution

- AWS Integration
- Netflix Ecosystem Support

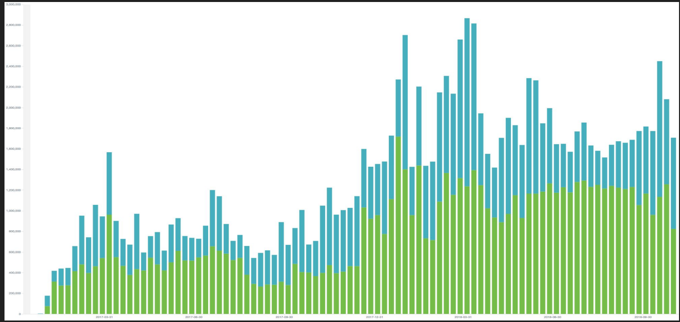


Stats



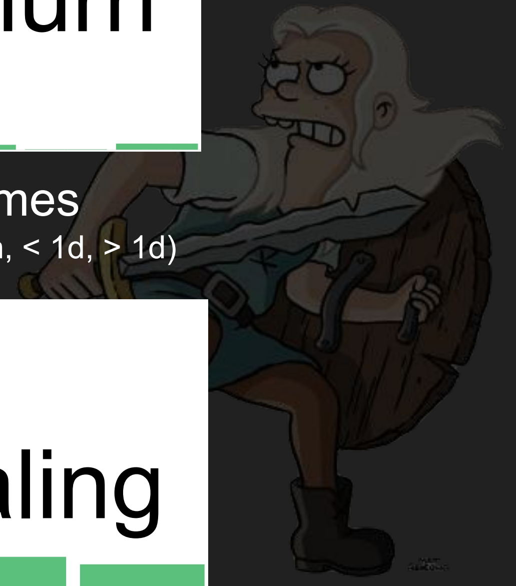
Containers Launched
Per Week

Batch runtimes
(< 1s, < 1m, < 1h, < 12h, < 1d, > 1d)



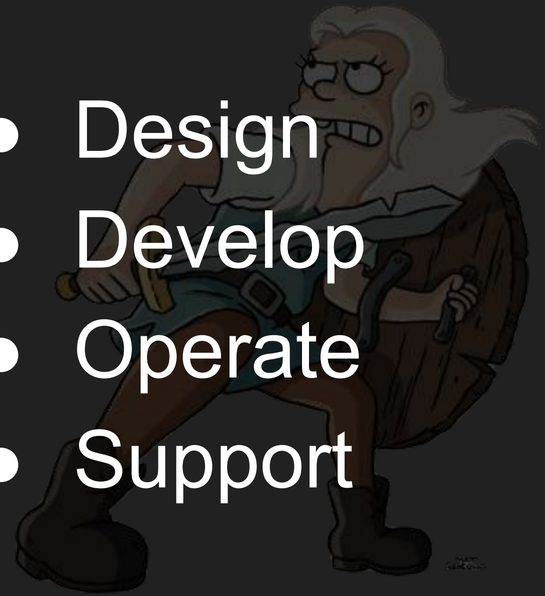
Batch vs. Service

Service runtimes
(< 1 day, < 1 week, < 1 month, > 1 month)



The Titus team

- Design
- Develop
- Operate
- Support

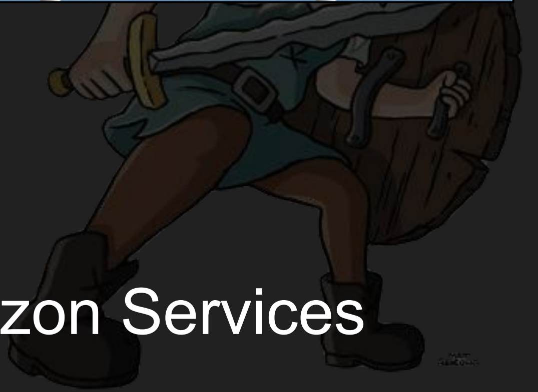


* And Netflix Platform Engineering and Amazon Web Services

Titus Product Strategy

Ordered priority focus on

- Developer Velocity
- Reliability
- Cost Efficiency



Easy migration from VMs to containers

Easy container integration with VMs and Amazon Services

Focus on just what Netflix needs

Deeply integrated AWS container platform

IP per container

- VPC, ENIs, and security groups

IAM Roles and Metadata Endpoint per container

- Container view of 169.254.169.254

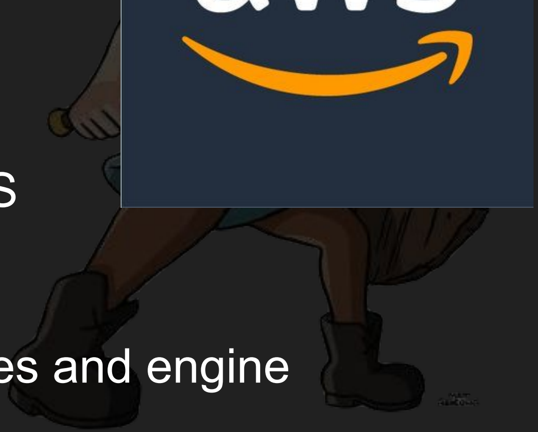
Cryptographic identity per container

- Using Amazon instance identity document, Amazon KMS

Service job container autoscaling

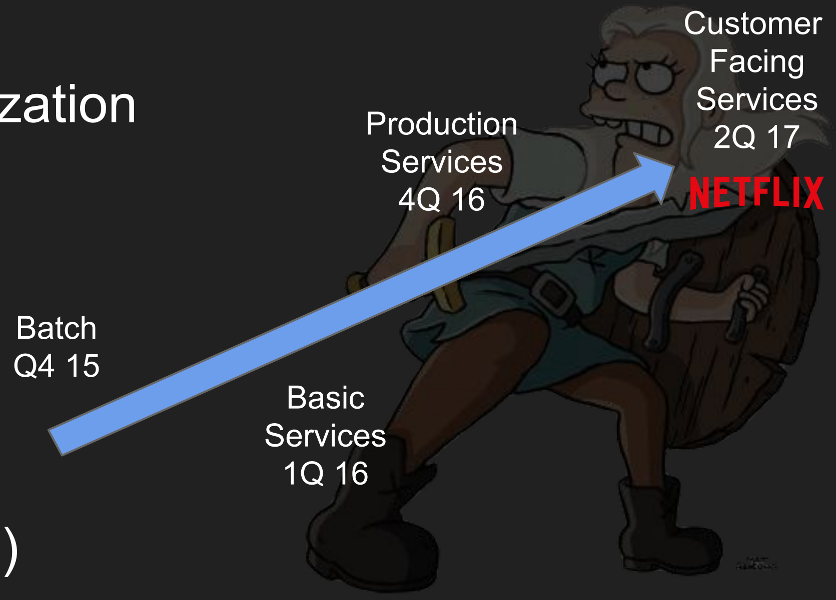
- Using Native AWS Cloudwatch, SQS, Autoscaling policies and engine

Application Load Balancing (ALB)



Applications using containers at Netflix

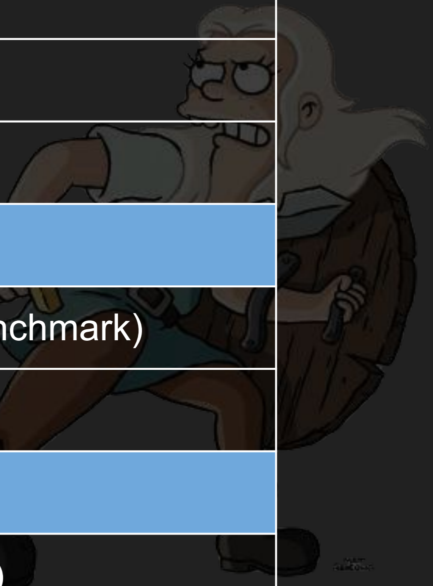
- Netflix API, Node.js Backend UI Scripts
- Machine Learning (GPUs) for personalization
- Encoding and Content use cases
- Netflix Studio use cases
- CDN tracking, planning, monitoring
- Massively parallel CI system
- Data Pipeline and Stream Processing
- Big Data use cases (Notebooks, Presto)



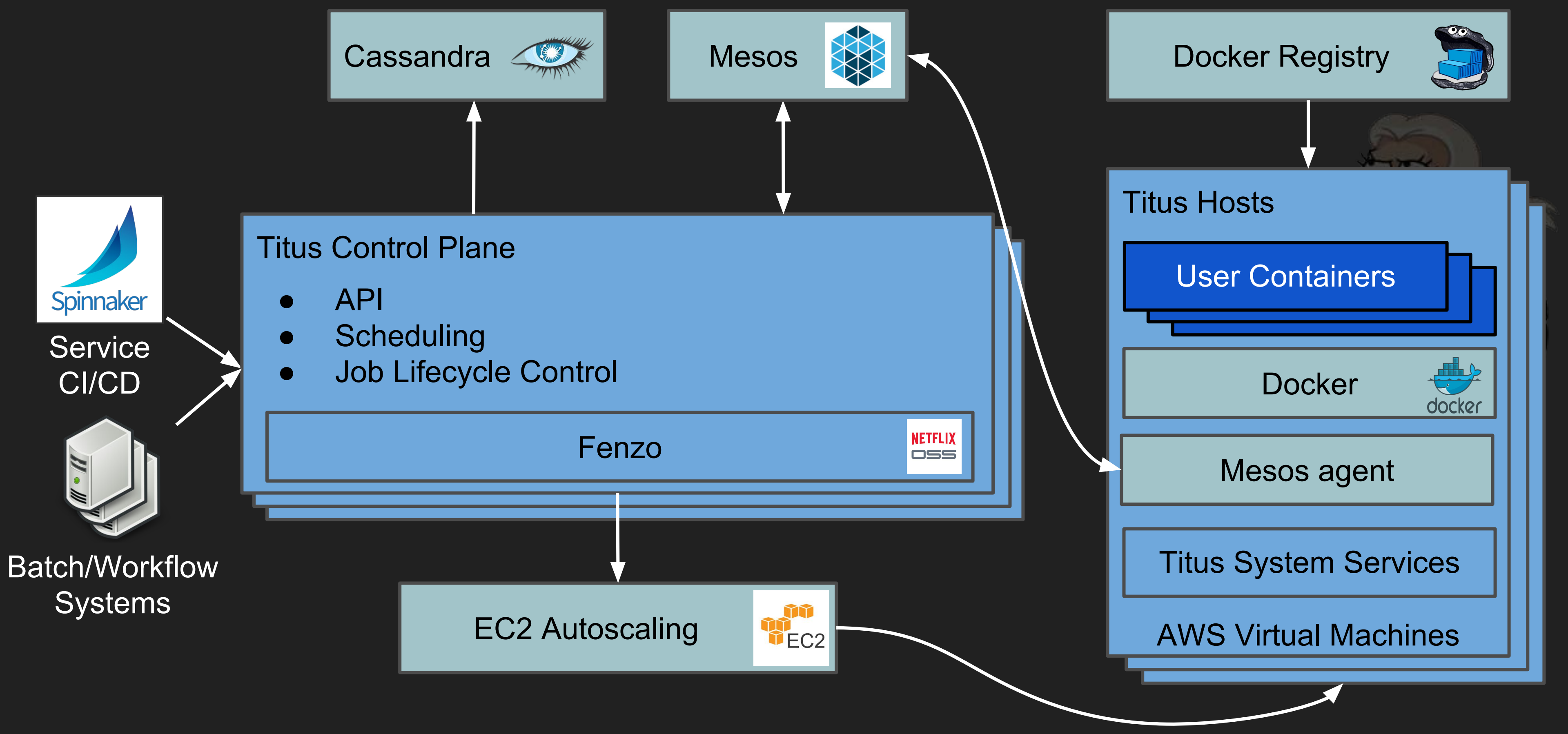
Q4 2018 Container Usage

NETFLIX

Common	
Jobs Launched	255K jobs / day
Different applications	**1K+ different images**
Isolated Titus deployments	7
Services	
Single App Cluster Size	5K (real), 12K containers (benchmark)
Hosts managed	7K VMs (**435,000 CPUs**)
Batch	
Containers launched	450K / day (**750K / day peak**)
Hosts managed (autoscaled)	55K / month



High Level Titus Architecture



Open Source

Open sourced April 2018

Help other communities by sharing our approach

Lessons Learned

End to End
User Experience

Our initial view of containers

What about?


Ad hoc Performance analysis
Run Container Workload
Change Campaigns
Monitor Containers
Security Scanning
Deploy new Container Workload
Image Registry Publish

What about?

End to end tooling

NETFLIX

Container orchestration only part of the problem

For Netflix …

- Local Development - Newt
- Continuous Integration - Jenkins + Newt
- Continuous Delivery - Spinnaker
- Change Campaigns - Astrid
- Performance Analysis - Vector and Flamegraphs

Tooling guidance

- Ensure coverage for entire application SDLC
  - Developing an application before deployment
  - Change management, security and compliance tooling for runtime
- What we added to Docker tooling
  - Curated known base images
  - Consistent image tagging
  - Assistance for multi-region/account registries
  - Consistency with existing tools

Operations and High Availability

NETFLIX

Learning how things fail

- Single container crashes
- Single host crashes
- Control plane fails
- Control plane gets into bad state

Learning how things fail

- Single container crashes
- Single host crashes
  - Taking down multiple containers
- Control plane fails
- Control plane gets into bad state

NETFLIX

Learning how things fail

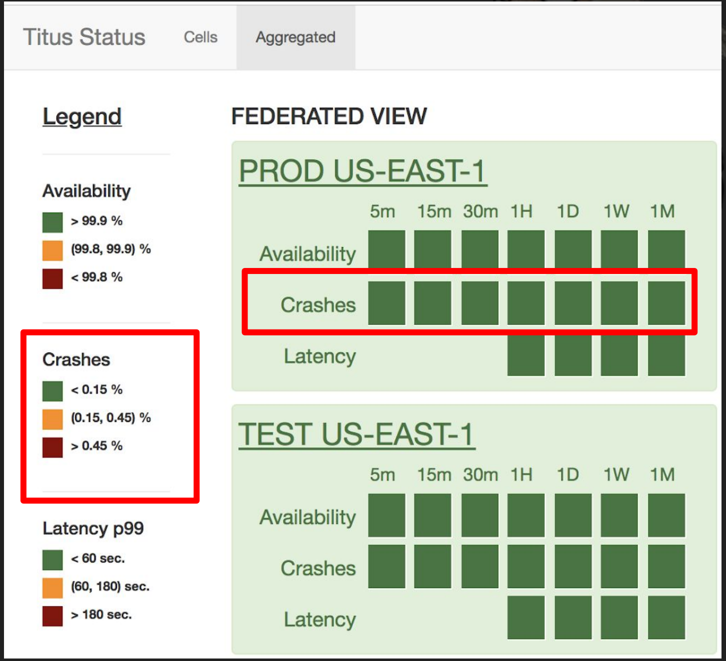
- Single container crashes
- Single host crashes
- Control plane fails
  - Existing containers continue to run
  - New jobs cannot be submitted
  - Replacements and scale ups do not occur
- Control plane gets into bad state

Learning how things fail

- Single container crashes
- Single host crashes
- Control plane fails
- Control plane gets into bad state
  - Can be catastrophic

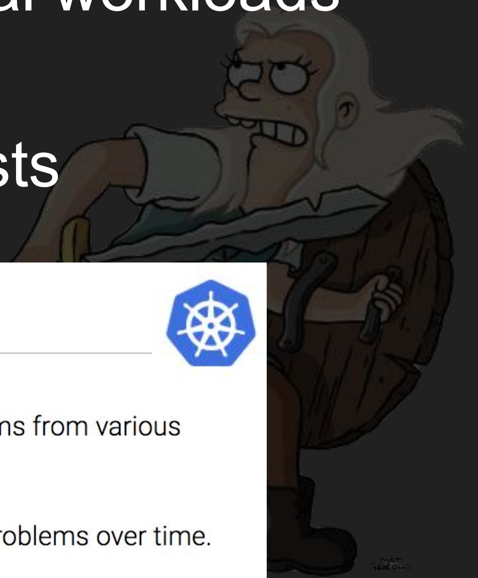
Case 1 - Single container crashes

- Most orchestrators will recover
- Most often during startup or shutdown
- Monitor for crash loops



Case 2 - Single host crashes

- Need a placement engine that spreads critical workloads
- Need a way to detect and remediate bad hosts



Monitor Node Health

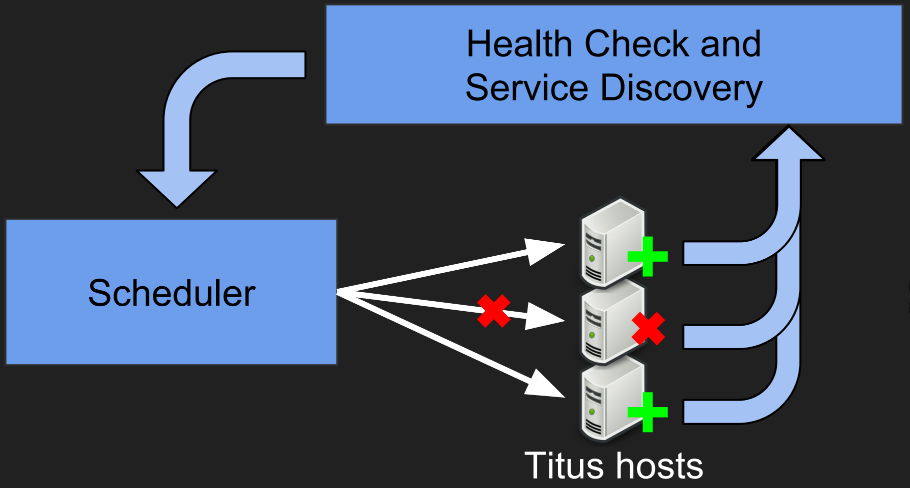
Node problem detector is a DaemonSet monitoring the node health. It collects node problems from various daemons and reports them to the apiserver as NodeCondition and Event.

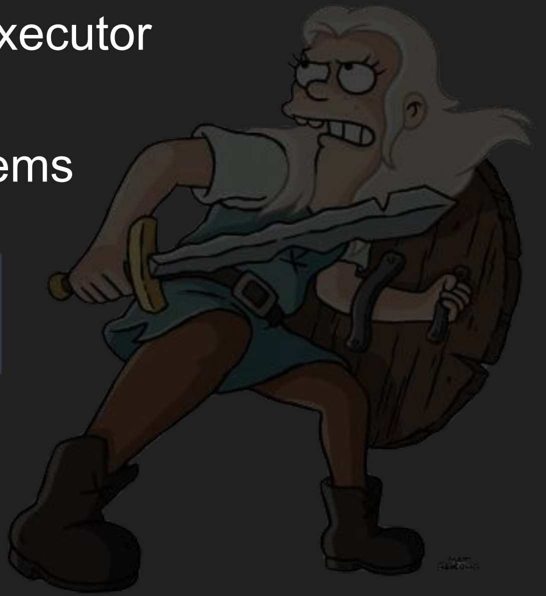
It supports some known kernel issue detection now, and will detect more and more node problems over time.

Currently Kubernetes won't take any action on the node conditions and events generated by node problem detector. In the future, a remedy system could be introduced to deal with node problems.

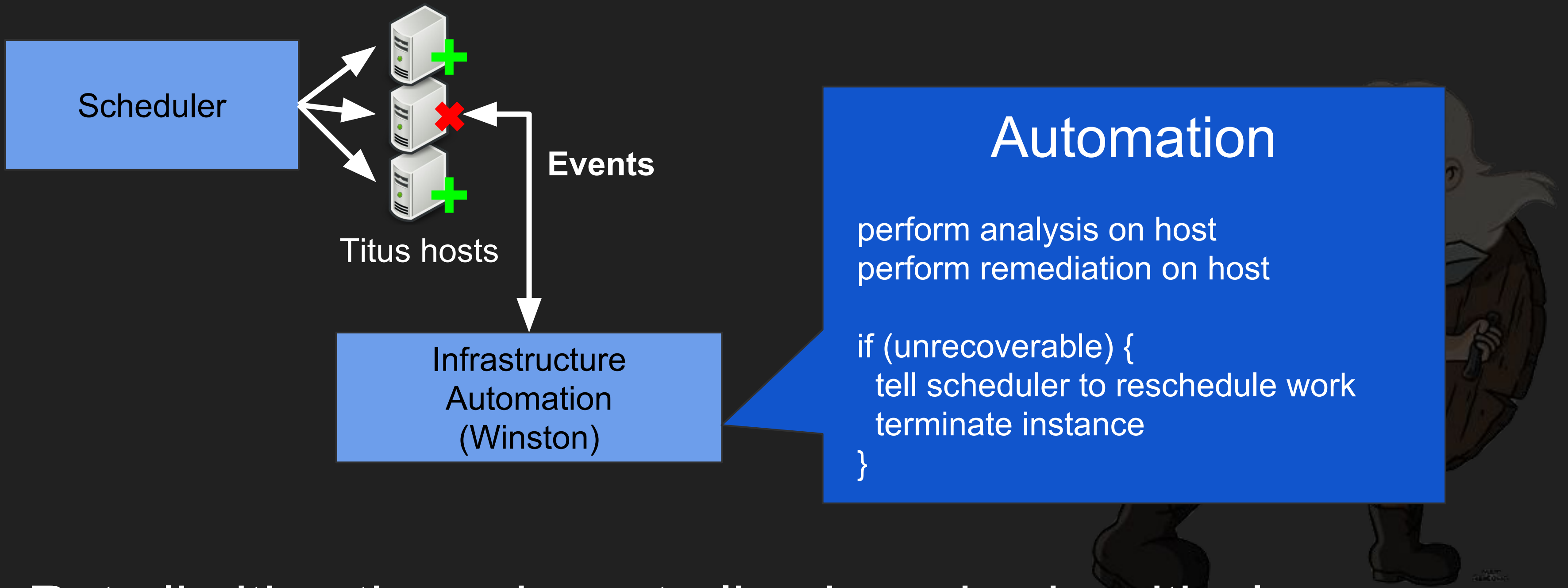
Titus node health monitoring, scheduling

- Extensive health checks
  - Control plane components - Docker, Mesos, Titus executor
  - AWS - ENI, VPC, GPU
  - Netflix Dependencies - systemd state, security systems





Titus node health remediation

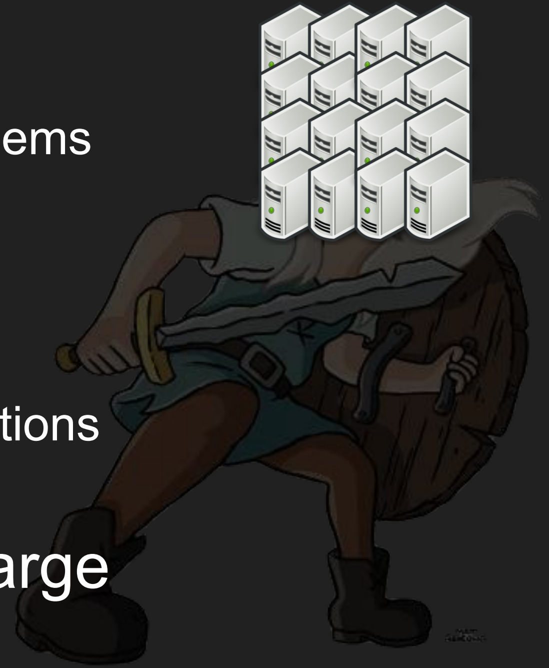


- Rate limiting through centralized service is critical

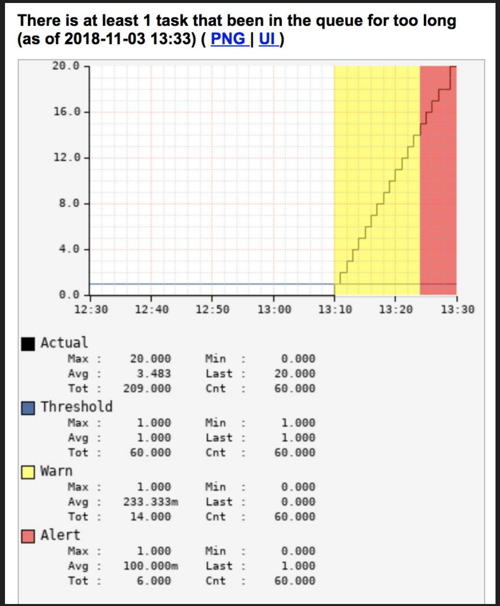
Spotting fleet wide issues using logging

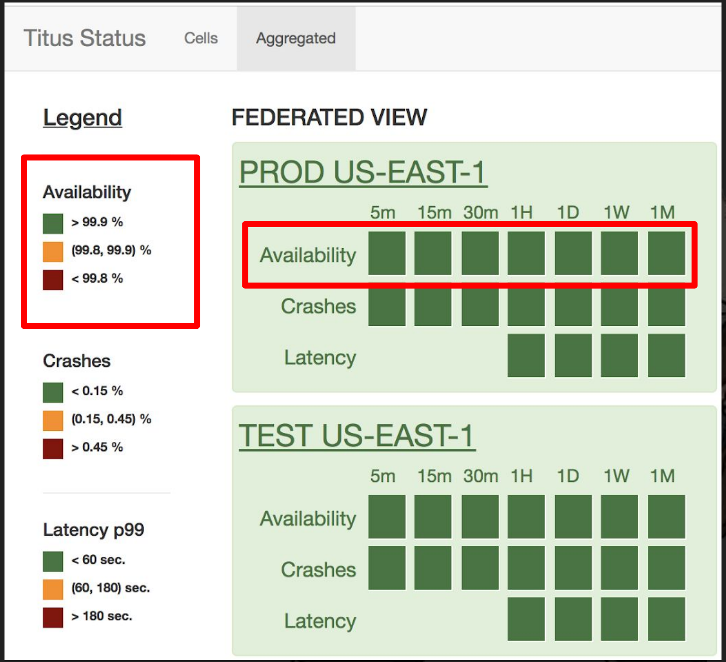
NETFLIX

- For the hosts, not the containers
  - Need fleet wide view of container runtime, OS problems
  - New workloads will trigger new host problems
- Titus hosts generate 2B log lines per day
  - Stream processing to look for patterns and remediations
- Aggregated logging - see patterns in the large



Case 3 - Control plane hard failures





White box - monitor time bucketed queue length

Black box - submit synthetic workloads

Case 4 - Control plane soft failures

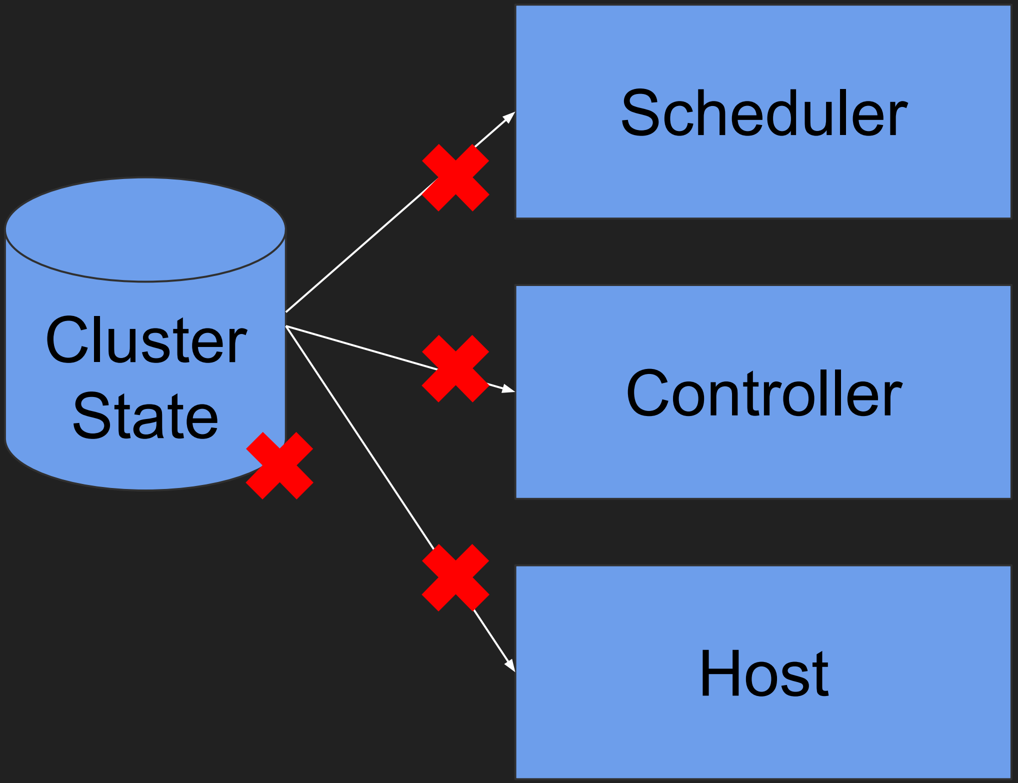
I don’t feel so good!

But first, let's talk about Zombies

Disconnected containers

- Early on, we had cases where
  - Some but not all of the control plane was working
  - User terminated their containers
  - Containers still running, but shouldn’t have been
- The “fix” - Mesos implicit reconciliation
  - Titus to Mesos - What containers are running?
  - Titus to Mesos - Kill these containers we know shouldn’t be running
  - System converges on consistent state 👍

But what if?



Backing store gets corrupted

Or

Control plane reads store incorrectly

Bad things occur

Running Containers

Not running Containers

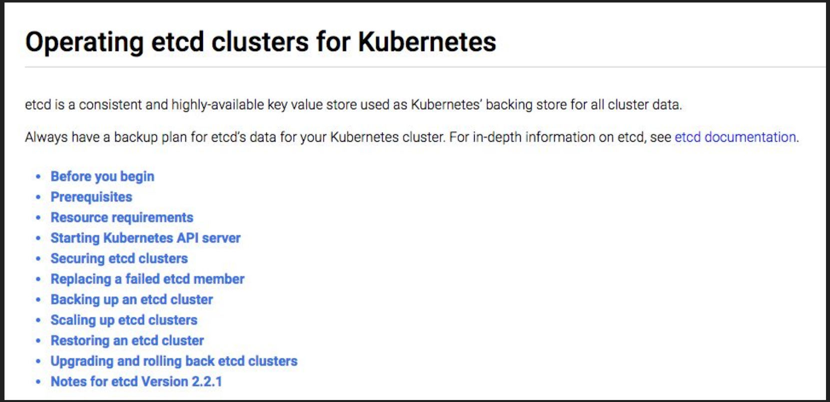
12,000 containers “reconciled” in < 1m

An hour to restore service

Guidance

NETFLIX

- Know how to operate your cluster storage
  - Perform backups and test restores
  - Test corruption
  - Know failure modes, and know how to recover

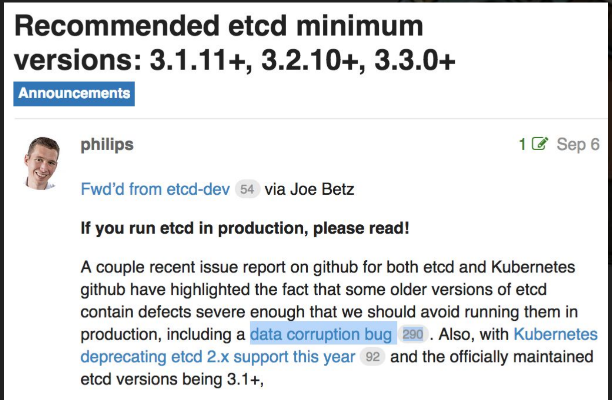


Operating etcd clusters for Kubernetes

etcd is a consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.

Always have a backup plan for etcd's data for your Kubernetes cluster. For in-depth information on etcd, see etcd documentation.

- Before you begin
- Prerequisites
- Resource requirements
- Starting Kubernetes API server
- Securing etcd clusters
- Replacing a failed etcd member
- Backing up an etcd cluster
- Scaling up etcd clusters
- Restoring an etcd cluster
- Upgrading and rolling back etcd clusters
- Notes for etcd Version 2.2.1



Recommended etcd minimum versions: 3.1.11+, 3.2.10+, 3.3.0+

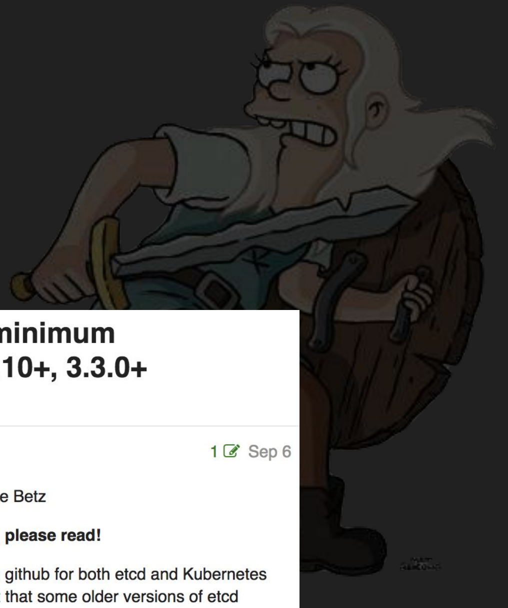
Announcements

philips 1 Sep 6

Fwd'd from etcd-dev 54 via Joe Betz

If you run etcd in production, please read!

A couple recent issue report on github for both etcd and Kubernetes github have highlighted the fact that some older versions of etcd contain defects severe enough that we should avoid running them in production, including a data corruption bug 290 . Also, with Kubernetes deprecating etcd 2.x support this year 92 and the officially maintained etcd versions being 3.1+,



At Netflix, we ...

- Moved to less aggressive reconciliation
- Page on inconsistent data
  - Let existing containers run
  - Human fixes state and decides how to proceed
- Automated snapshot testing for staging

Security

Reducing container escape vectors

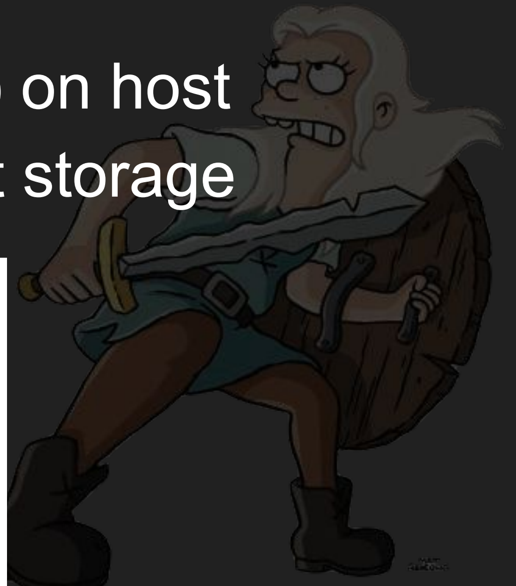
- Enforcement
  - Seccomp and AppArmor policies
- Cryptographic identity for each container
  - Leveraging host level Amazon and control plane provided identities
  - Validated by central Netflix service before secrets are provided

NETFLIX

Reducing impact of container escape vectors

User namespaces

- Root (or user) in container != Root (or user) on host
- Challenge: Getting it to work with persistent storage



NAME top

user_namespaces - overview of Linux user namespaces

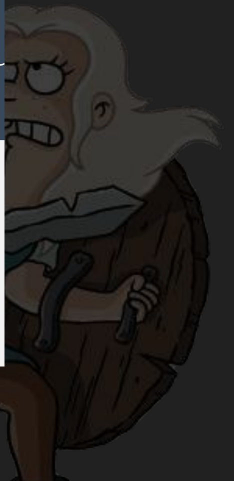
DESCRIPTION top

For an overview of namespaces, see namespaces(7).

User namespaces isolate security-related identifiers and attributes, in particular, user IDs and group IDs (see credentials(7)), the root directory, keys (see keyrings(7)), and capabilities (see capabilities(7)). A process's user and group IDs can be different inside and outside a user namespace. In particular, a process can have a normal unprivileged user ID outside a user namespace while at the same time having a user ID of 0 inside the namespace; in other words, the process has full privileges for operations inside the user namespace, but is unprivileged for operations outside the namespace.

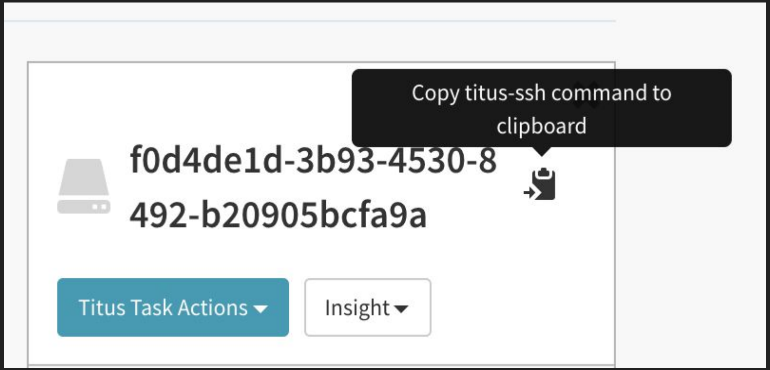
user_namespaces (7)

Lock down, isolate control plane

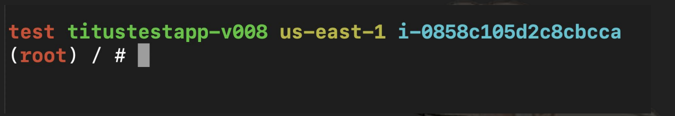


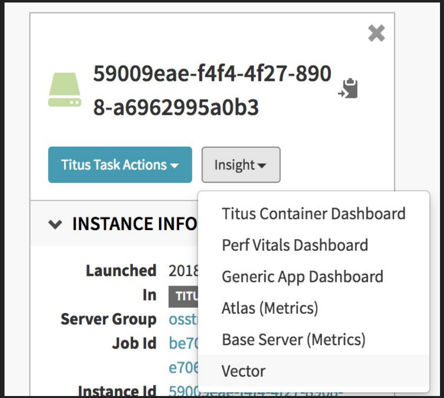
- Hackers are scanning for Docker and Kubernetes
- Reported lack of networking isolation in Google Borg
- We also thought our networking was isolated (wasn't)

Avoiding user host level access

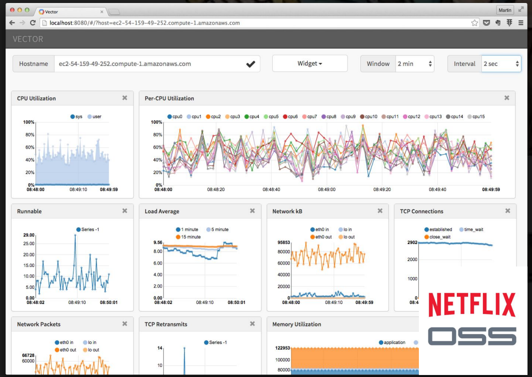


ssh





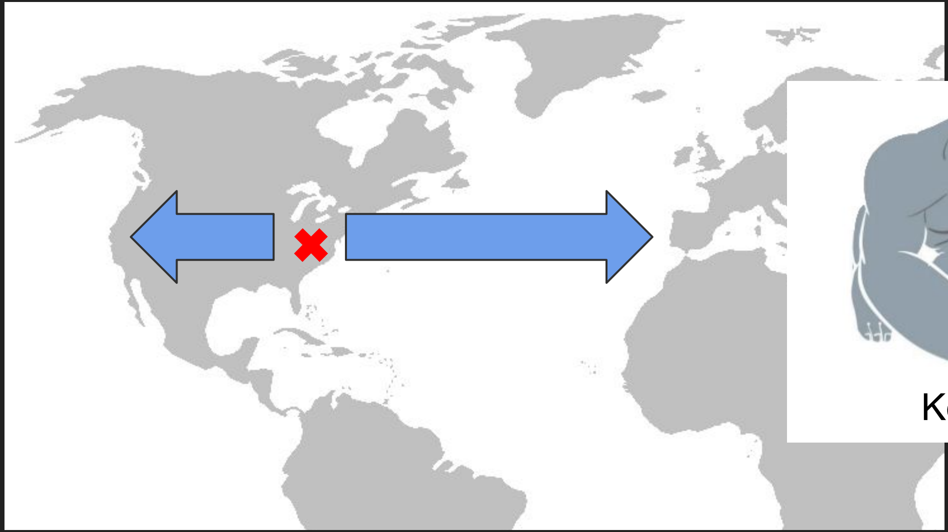
perf tools

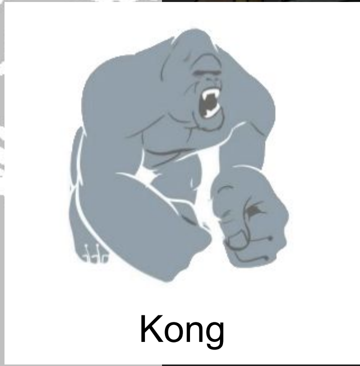


Vector

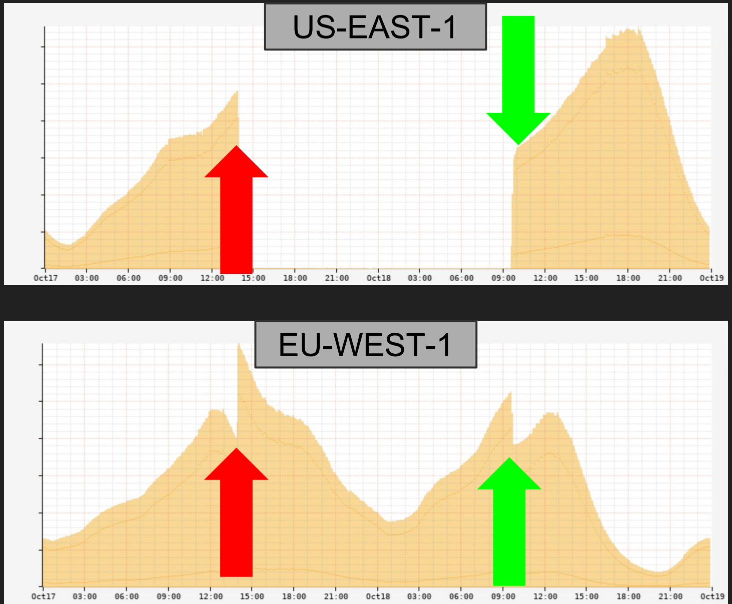
Scale - Scheduling Speed

How does Netflix failover?





Netflix regional failover



API Calls Per Region

- Kong evacuation of us-east-1
  Traffic diverted to other regions
- Fail back to us-east-1
  Traffic moved back to us-east-1

Infrastructure challenge

- Increase capacity during scale up of savior region
- Launch 1000s of containers in 7 minutes

Easy Right?

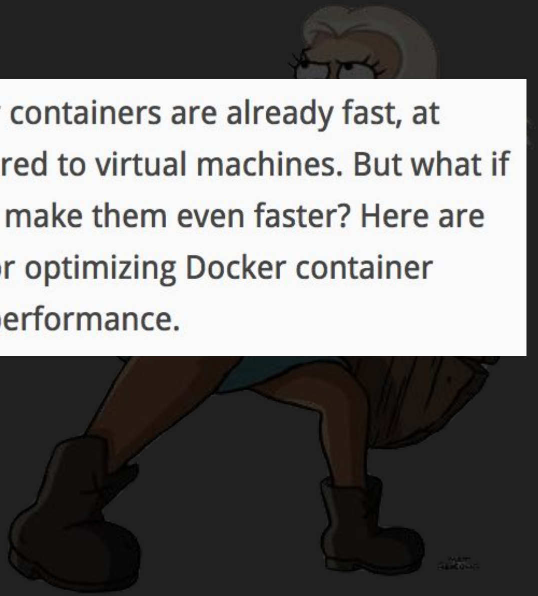
Improving Kubernetes Scheduler Performance

February 22, 2016 • By Hongchao Deng

"we reduced time to schedule 30,000 pods onto 1,000 nodes from 8,780 seconds to 587 seconds"

The Million Container Challenge

HashiCorp scheduled 1,000,000 Docker containers on 5,000 hosts in under 5 minutes with Nomad, our free and open source cluster scheduler.



Your Docker containers are already fast, at least compared to virtual machines. But what if you want to make them even faster? Here are strategies for optimizing Docker container speed and performance.

Easy Right?

Improving Kubernetes Scheduler Performance

February 22, 20

"we r
p
8,7

The M

HashiCorp sc
in under 5 m
cluster scheduler.

fast, at
But what if
? Here are
ntainer

Synthetic benchmarks missing

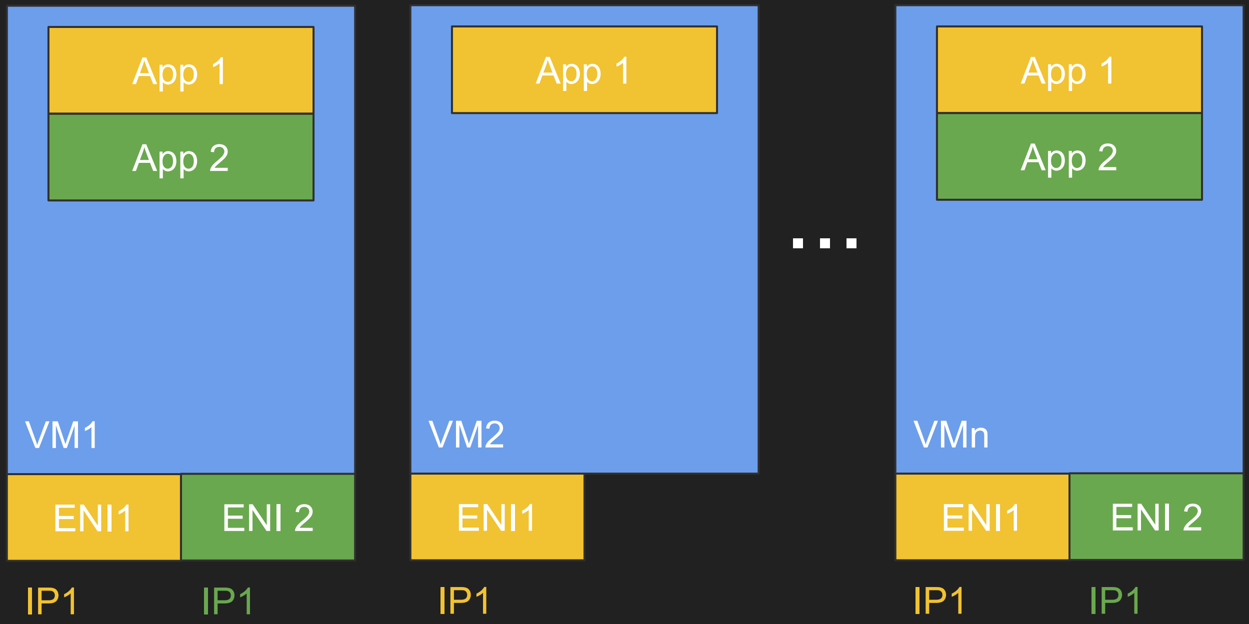
1. Heterogeneous workloads
2. Full end to end launches
3. Docker image pull times
4. Integration with public cloud networking

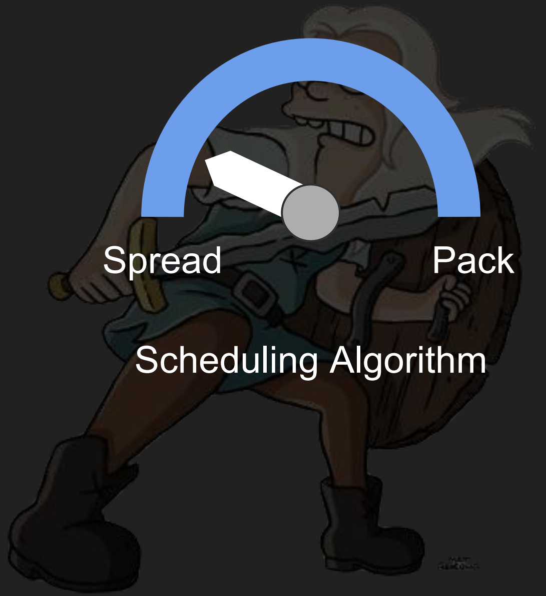
Titus can do this by ...

- Dynamically changeable scheduling behavior
- Fleet wide networking optimizations

Normal scheduling

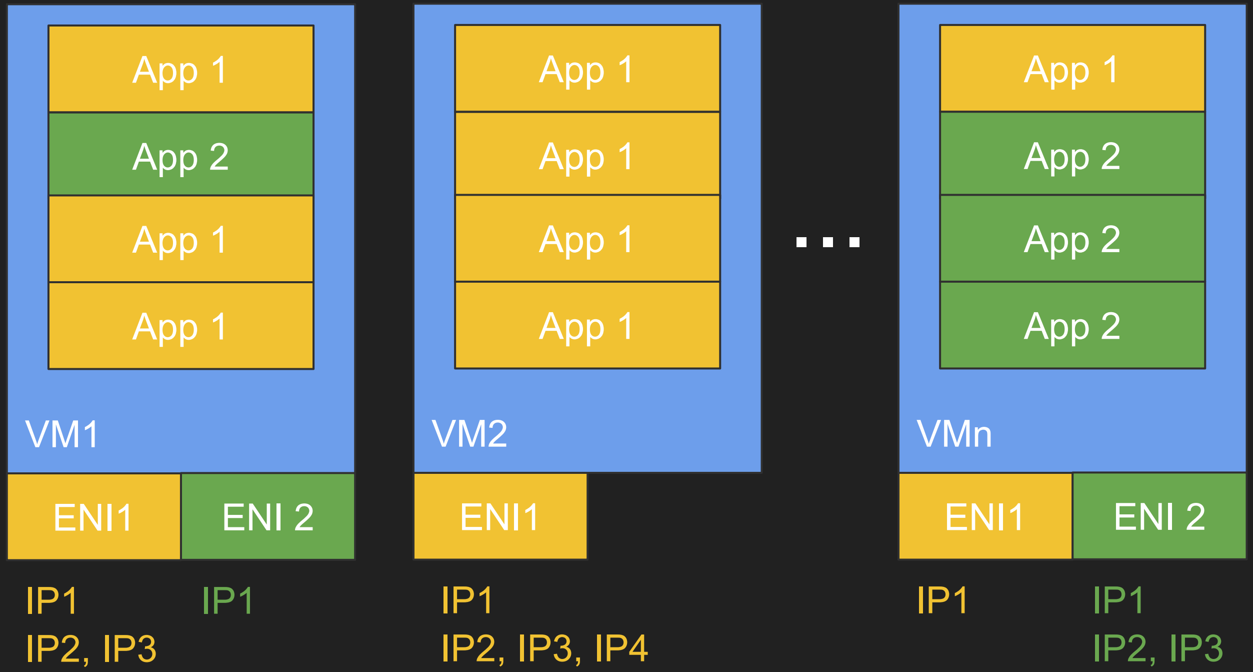
NETFLIX

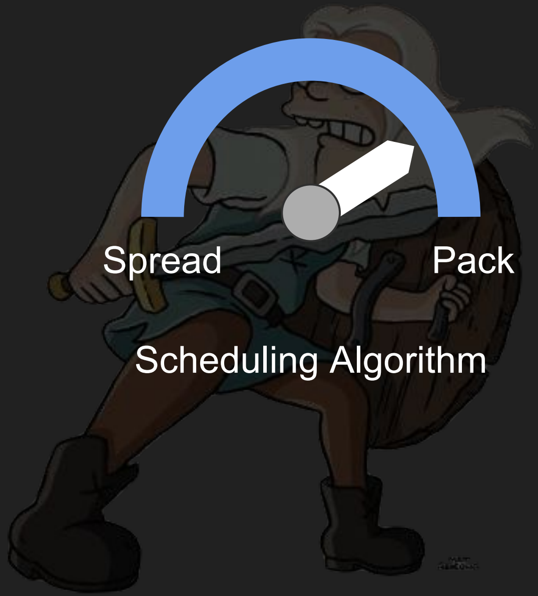




Trade-off for reliability

Failover scheduling





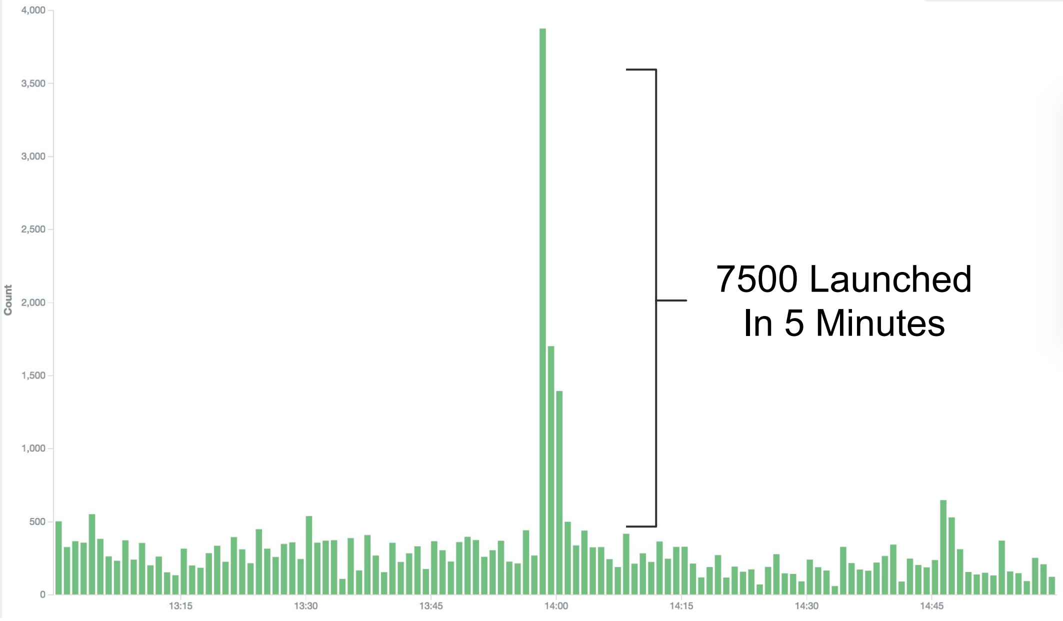
Trade-off for speed

NETFLIX

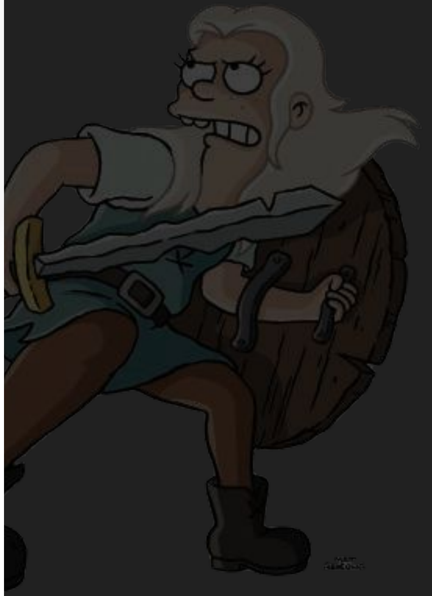
On each host

- Due to normal scheduling, host likely already has ...
  - Docker image downloaded
  - Networking interfaces and security groups configured
- Need to burst allocate IP addresses
  - Opportunistically batch allocate at container launch time
  - Likely if one container was launched more are coming
  - Garbage collect unused later

Results



us-east-1 / prod
containers started per minute



Scale - Limits

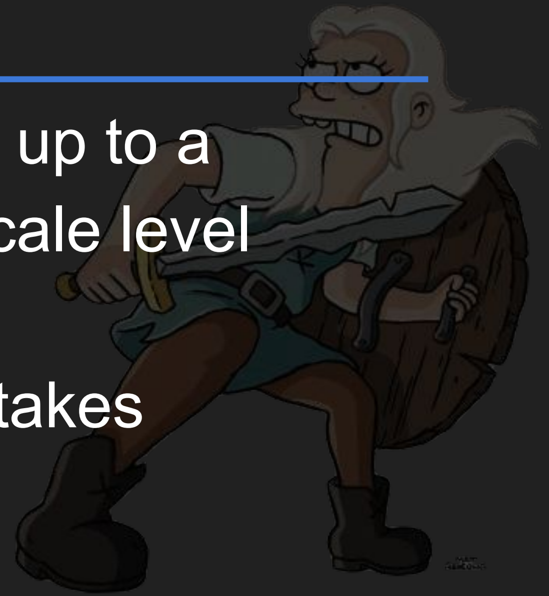
How far can a single Titus stack go?

NETFLIX

- Speed and stability of scheduling
- Blast radius of mistakes

Scaling options

Idealistic	Realistic
Continue to improve performance	Test a stack up to a known scale level
Avoid making mistakes	Contain mistakes



Titus “Federation”

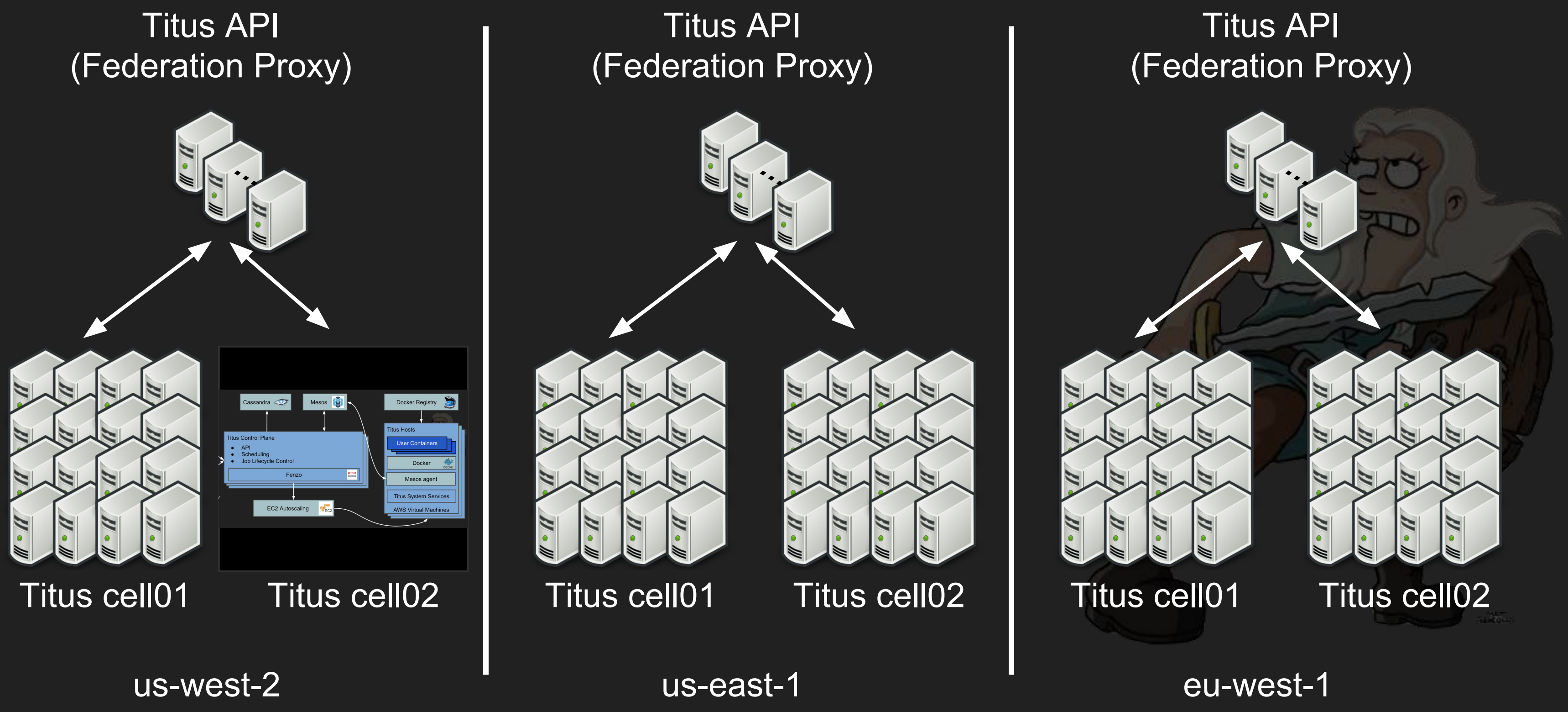
- Allows a Titus stack to be scaled out
  - For performance and reliability reasons
- Not to help with
  - Cloud bursting across different resource pools
  - Automatic balancing across resource pools
  - Joining various business unit resource pools

Federation Implementation

NETFLIX

- Users need only to know of the external single API
  - VIP - titus-api.us-east-1.prod.netflix.net
- Simple federation proxy spans stacks (cells)
  - Route these apps to cell 1, these others to cell 2
  - Fan out & union queries across cells
  - Operators can route directly to specific cells

Titus Federation



How many cells?

A few large cells

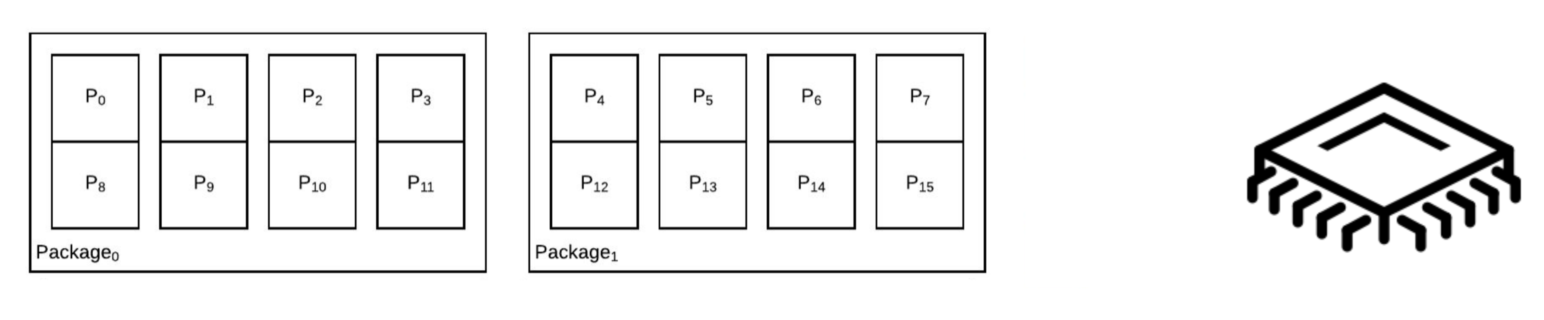
- Only as many as needed for scale / blast radius

Why? Larger resource pools help with

- Cross workload efficiency
- Operations
- Bad workload impacts

Performance and Efficiency

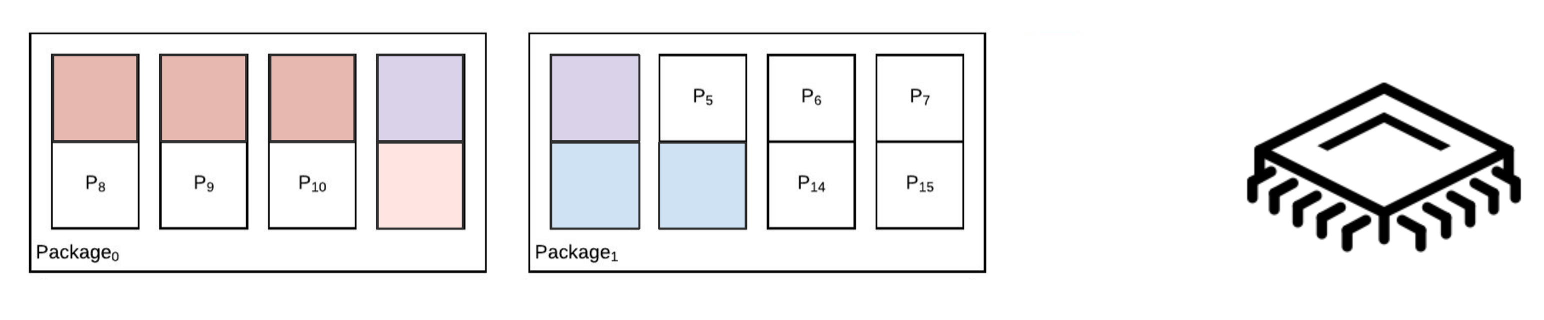
Simplified view of a server



- A fictional “16 vCPU” host

- Left and right are CPU packages
- Top to bottom are cores with hyperthreads

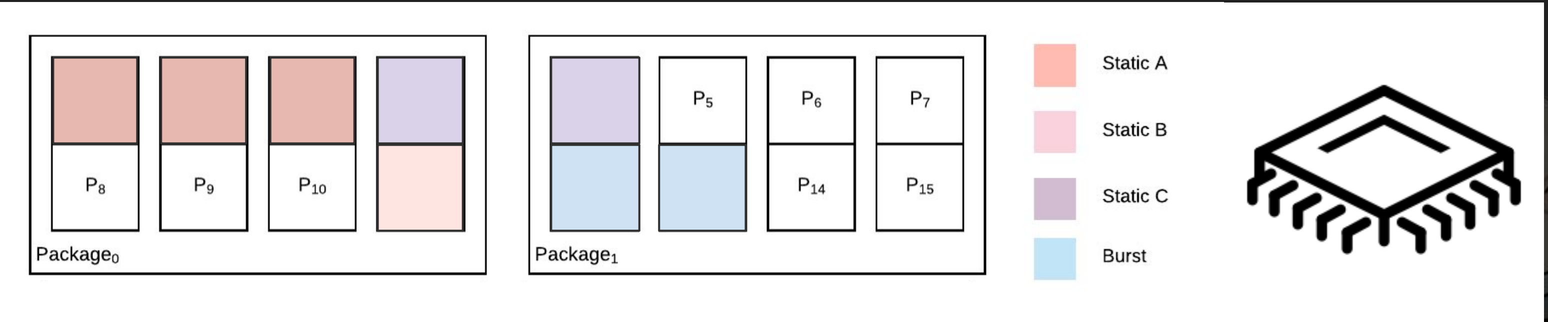
Consider workload placement



- Consider three workloads
  - Static A, B, and C all which are latency sensitive
  - Burst D which would like more compute when available
- Let’s start placing workloads

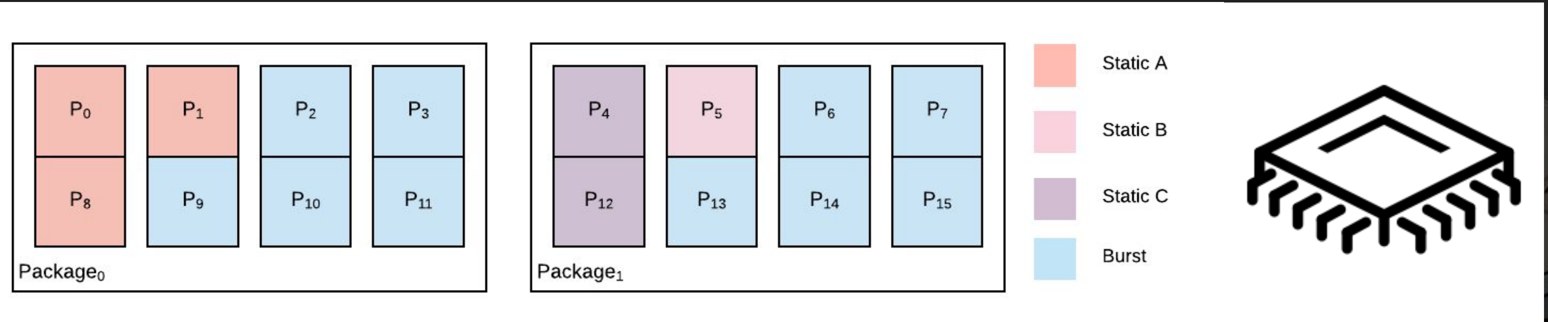
Problems

NETFLIX



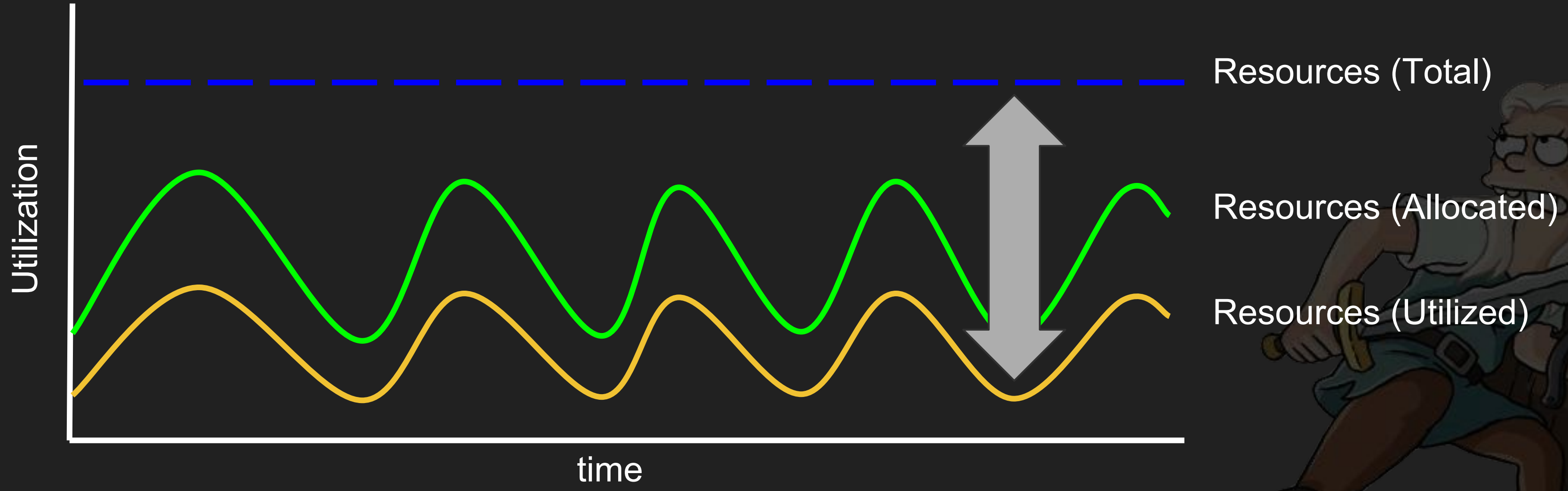
- Potential performance problems
  - Static C is crossing packages
  - Static B can steal from Static C
- Underutilized resources
  - Burst workload isn't using all available resources

Node level CPU rescheduling



- After containers land on hosts
  - Eventually, dynamic and cross host
- Leverages cpusets
  - Static - placed on single CPU package and exclusive full cores
  - Burst - can consume extra capacity, but variable performance
- Kubernetes - CPU Manager (beta)

Opportunistic workloads



- Enable workloads to burst into underutilized resources
- Differences between utilized and total

NETFLIX

Questions?

Follow-up: @aspyker