# TINDER'S MOVE TO KUBERNETES

Chris O'Brien - Senior Engineering Manager
Chris Thomas - Engineering Manager

Cloud Infrastructure

# Why

Over two years ago, Tinder decided to move its platform to Kubernetes. Kubernetes afforded us an opportunity to drive Tinder Engineering toward containerization and low-touch operation through immutable deployment. Application build, deployment, and infrastructure would be defined as code.

We were also looking to address challenges of scale and stability. When scaling became critical, we often suffered through several minutes of waiting for new EC2 instances to come online. The idea of containers scheduling and serving traffic within seconds as opposed to minutes was appealing to us.

# How

Starting January 2018, we worked our way through various stages of the migration effort. We started by containerizing all of our services and deploying them to a series of Kubernetes hosted staging environments. Beginning in October, we began methodically moving all of our legacy services to Kubernetes. By March the following year, we finalized our migration and the Tinder Platform now runs exclusively on Kubernetes.

# Overview

- Legacy Architecture
- Building Images
- CI/CD
- Original Architecture
- Migration
- Learnings
- Load Balancing / Envoy
- Current Architecture
- Monitoring Stack
- Future Architecture

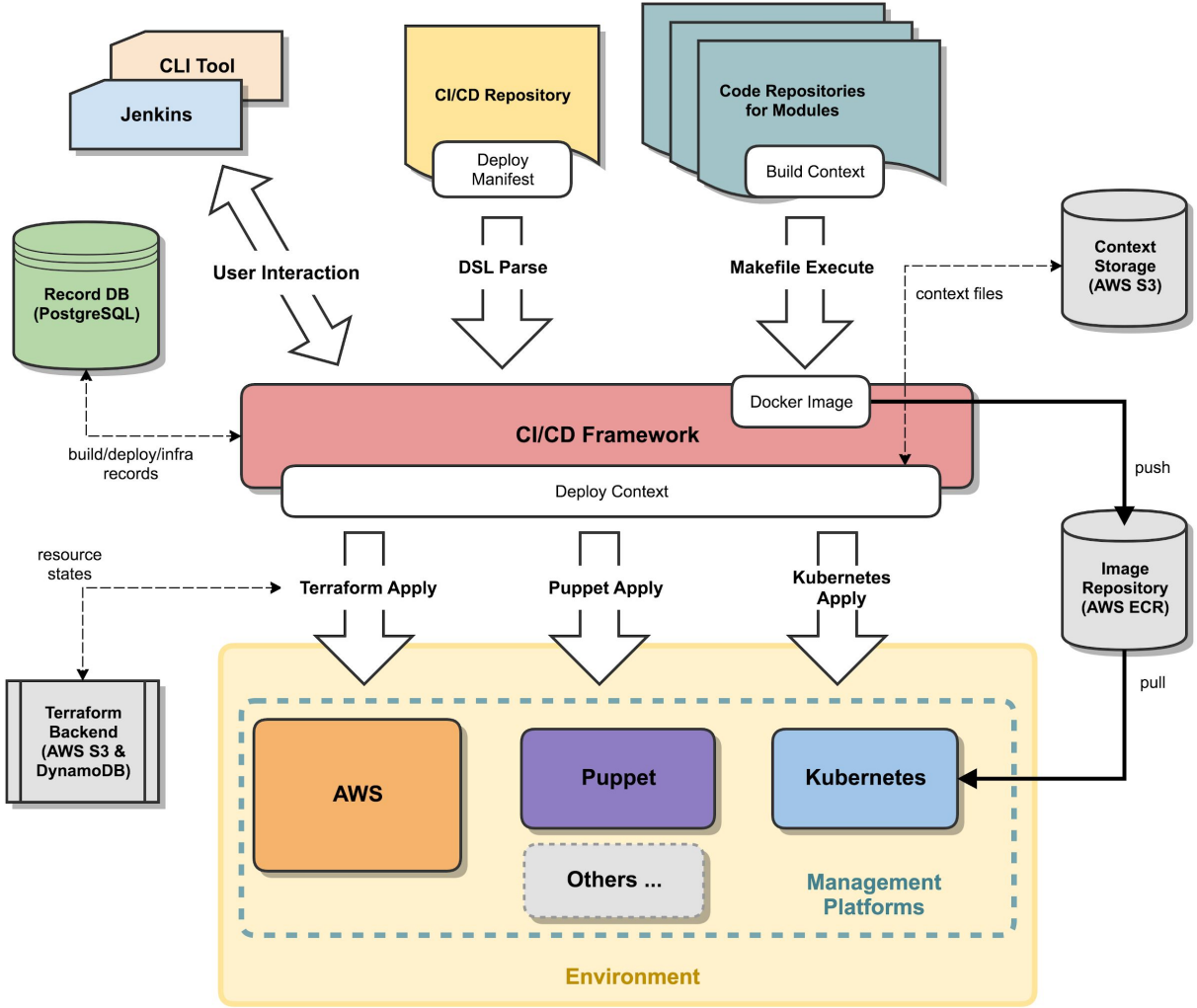
# Legacy Architecture

- EC2 Auto Scaling Groups
  - Fronted by ELB per-service
  - Scaling off CPU usage
  - Minimal tooling to automate provisioning of new ASGs for services
- Puppet for bootstrapping node configuration
- 2 Prometheus nodes in an ASG per-service
- Code deployments pushed a new version to nfs mount and triggered service restart

# Building Images

- Microservices
- More than 30 source repositories
    - Node.js, Java, Scala, Go
- Fully customizable build context with standardized format (yaml)
- Same build process for development and production
- Builder container

# CI/CD


CLI Tool
Jenkins
CI/CD Repository
Deploy Manifest
Code Repositories for Modules
Build Context
Record DB (PostgreSQL)
User Interaction
DSL Parse
Makefile Execute
Context Storage (AWS S3)
context files
Docker Image
CI/CD Framework
Deploy Context
build/deploy/infra records
push
resource states
Terraform Apply
Puppet Apply
Kubernetes Apply
Image Repository (AWS ECR)
pull
Terraform Backend (AWS S3 & DynamoDB)
AWS
Puppet
Kubernetes
Others ...
Management Platforms
Environment

# Original Architecture

- kube-aws for provisioning
- Initially one node pool
- Quickly separated into different sizes and types
- Running fewer heavily threaded pods (Java) together yielded better performance than letting them colocate with Node.js
- Settled on:
    - c5.4xlarge Controlplane Masters (3 nodes)
    - c5.4xlarge Etcd (3 nodes)
    - c5.4xlarge for Node.js
    - c5.2xlarge for Java and Go
    - m5.4xlarge for monitoring (Prometheus)

# Migration

- Change existing service-to-service calls to new ELBs
  - Peered to K8s VPC
  - Granularly migrate modules with no regard to order
  - Endpoints used weighted DNS records with CNAME to ELB
- For migration, TTL was lowered and weight was adjusted to slowly shift traffic to new K8s ELB
  - Java honored low TTL but Node.js did not
  - New connection pool code that would refresh the pools every 60s

# Learnings - ARP

- January 8, 2019 - unrelated scale up earlier in the day left the cluster at a larger size than before
- **ARP cache exhaustion** once pod and node counts reached a certain point
- Resulted in:
  - Dropped packets
  - Entire flannel /24s missing from ARP tables
- Raised values for **gc_thresh1/2/3** on all nodes
- Restarted **flannel** on all nodes

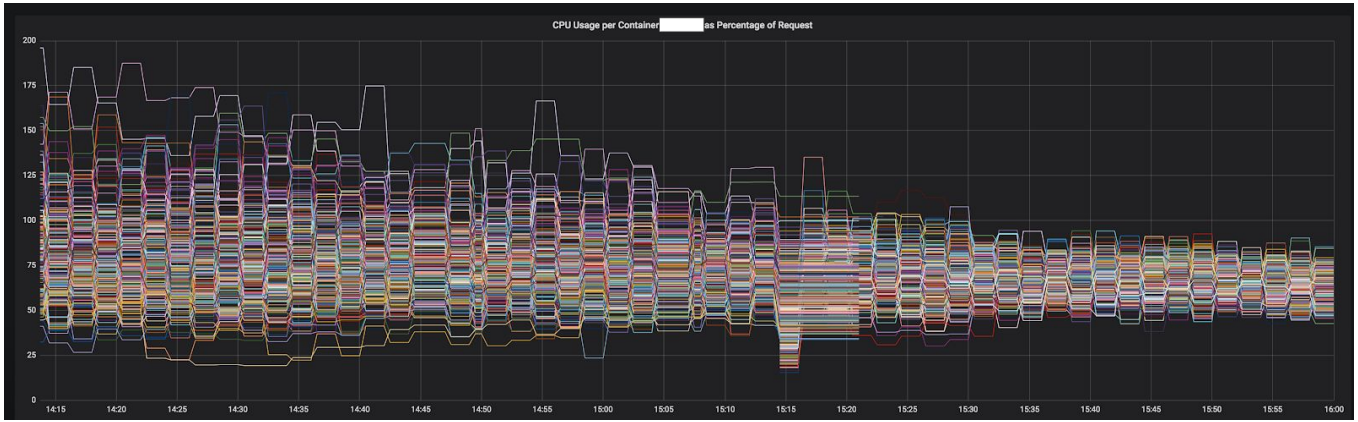
# Learnings - DNS

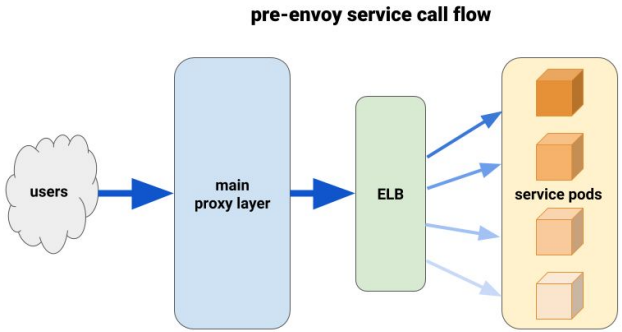
- **DNS timeouts** due to conntrack insertion failures - SNAT / DNAT
- Issues were amplified by ndots defaulting to 5 and causing many subsequent lookups
- Scaling attempts and ndots mitigations helped but only went so far - peaked at 250k/sec and 120 cores across 1000 CoreDNS pods
- CoreDNS redeployed as DaemonSet and injected node IP into resolv.conf - via kubelet config

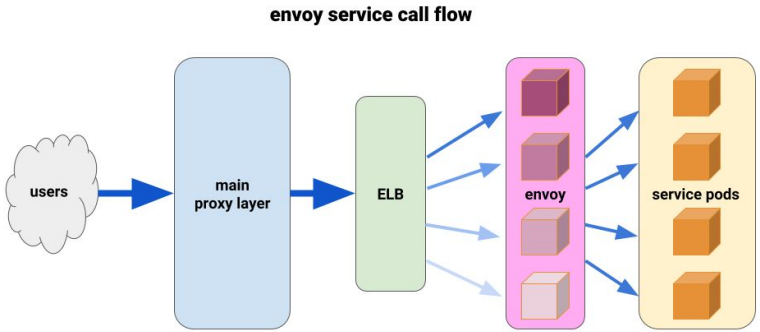
# Load Balancing

- Unbalanced load across pods due to ELB connections sticking to the first ready pods of each deployment
- Multiple temporary mitigations attempted
  - MaxSurge 100%
  - Inflated resource requests
- Internal POCs for Envoy proved successful so this gave us the chance to leverage it in a limited fashion
- Envoy sidecar alongside service
- Small fleet of proxies per-service, one deployment in each AZ
- Fronted by TCP ELB
- preStop hook on sidecar
  - calls health check fail admin endpoint
  - small sleep to allow inflight conns to complete and drain

# Envoy


CPU Usage per Container as Percentage of Request
200
175
150
125
100
75
50
25
0
14:15
14:20
14:25
14:30
14:35
14:40
14:45
14:50
14:55
15:00
15:05
15:10
15:15
15:20
15:25
15:30
15:35
15:40
15:45
15:50
15:55
16:00

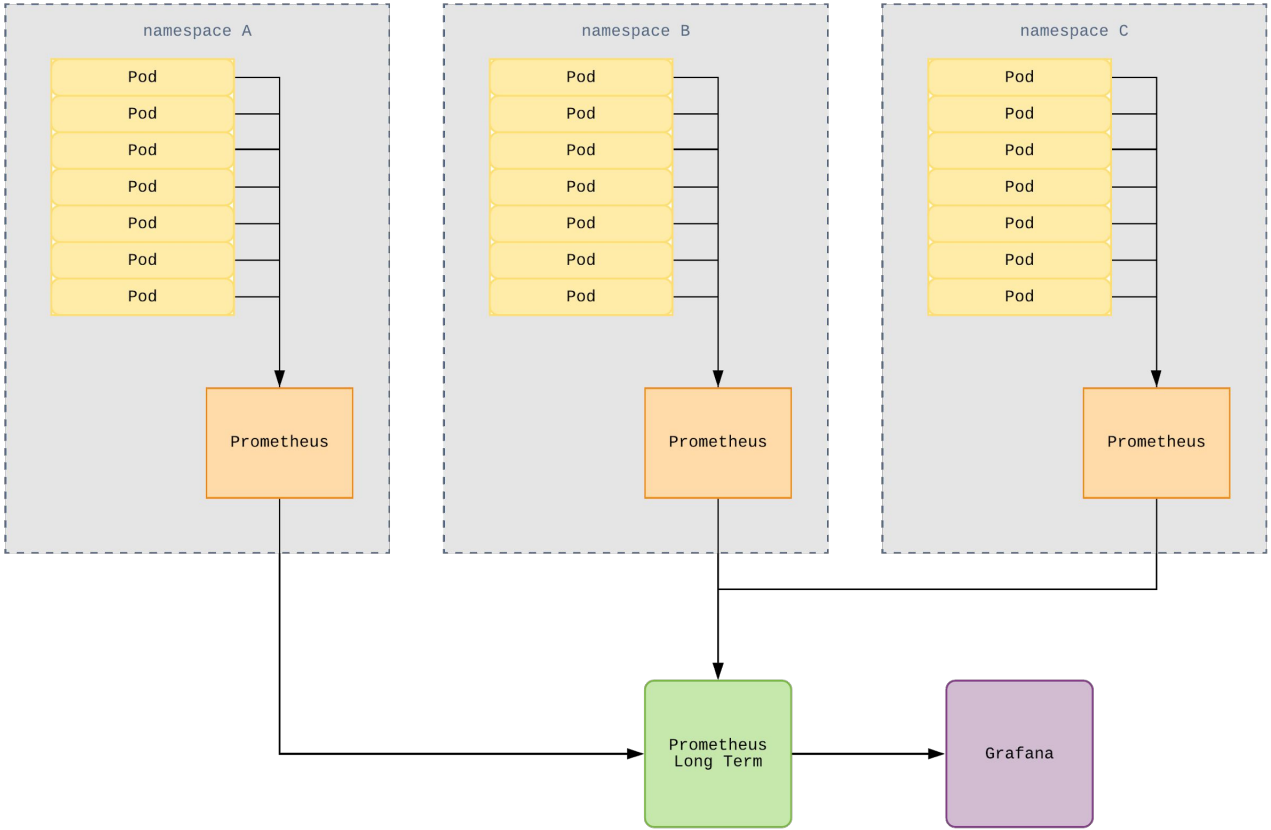

pre-envoy service call flow
users
main
proxy layer
ELB
service pods


envoy service call flow
users
main
proxy layer
ELB
envoy
service pods

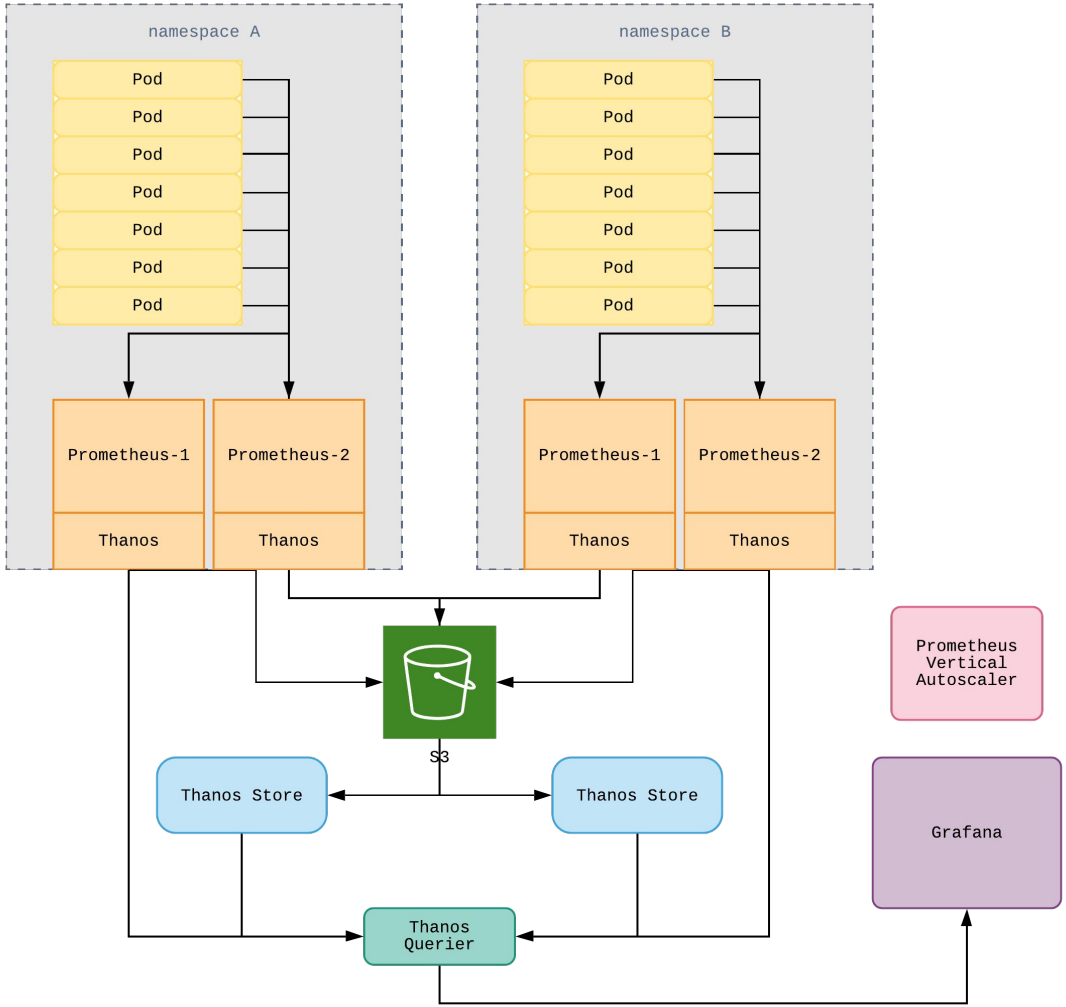
# Current Architecture

- ~2000 nodes / 18000 cores
- 6 Controlplane Masters
- 25K - 30K Pods
- 115K - 130K Containers
- 750K samples / sec - Prometheus stack
- ~ 5 TB / day log ingestion
- Migration to Envoy based Service Mesh

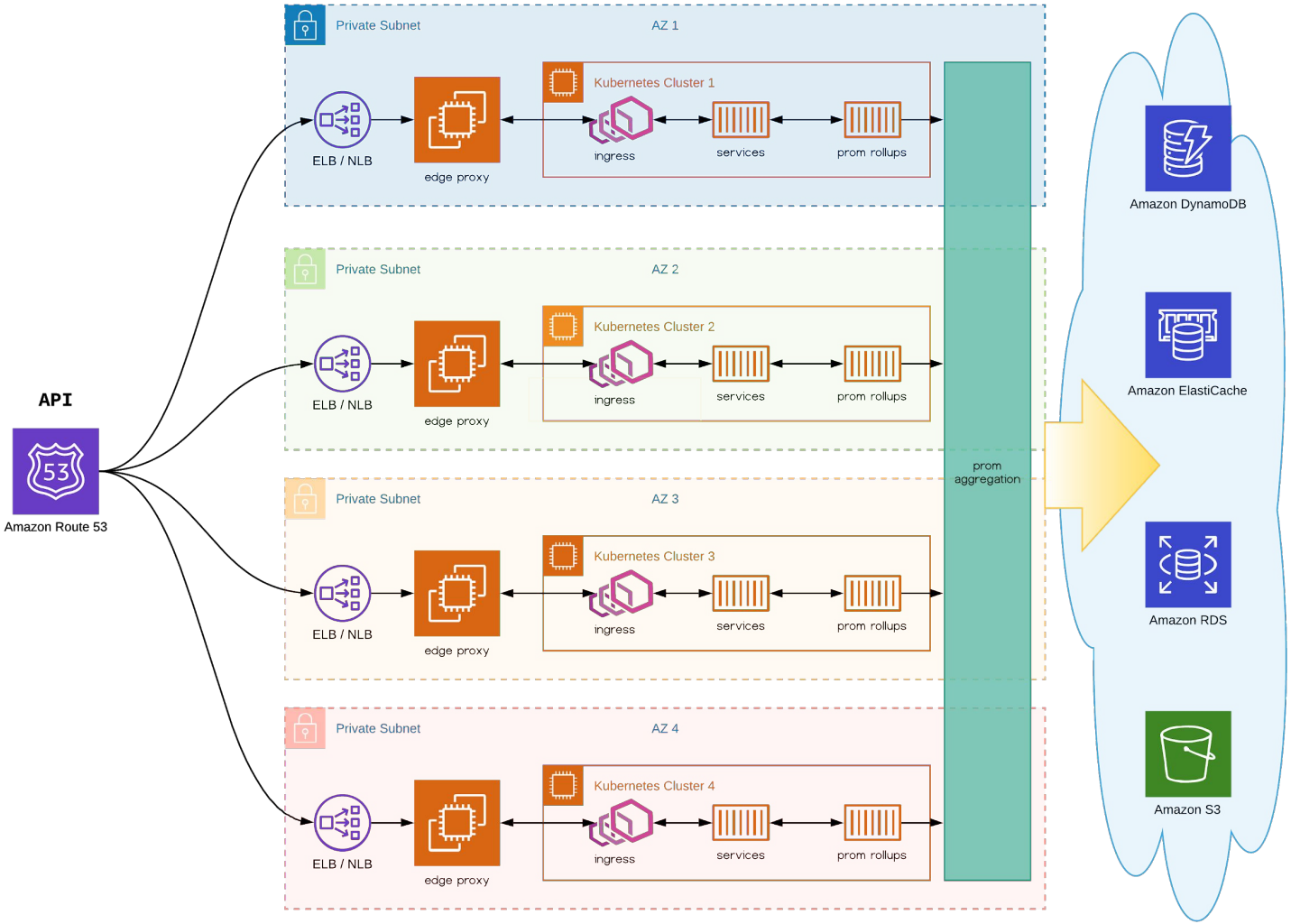
# Old Monitoring Stack



# New Monitoring Stack



# Future


API
Amazon Route 53
Private Subnet
AZ 1
ELB / NLB
edge proxy
Kubernetes Cluster 1
ingress
services
prom rollups
Private Subnet
AZ 2
ELB / NLB
edge proxy
Kubernetes Cluster 2
ingress
services
prom rollups
Private Subnet
AZ 3
ELB / NLB
edge proxy
Kubernetes Cluster 3
ingress
services
prom rollups
Private Subnet
AZ 4
ELB / NLB
edge proxy
Kubernetes Cluster 4
ingress
services
prom rollups
prom aggregation
Amazon DynamoDB
Amazon ElastiCache
Amazon RDS
Amazon S3