AWS re:Invent

NOV. 28 – DEC. 2, 2022 | LAS VEGAS, NV

SEC312

Deploying egress traffic controls in production environments

Kevin Park (he/him)
Software Engineer, Security
Robinhood

Houston Hopkins (he/him)
Sr. Staff Security Engineer
Robinhood

Graham Zulauf (he/him)
Principal Solutions Architect
AWS

Agenda

Why we need additional egress controls

AWS Network Firewall primer

Where we were

Our roadmap to the solution

Key steps we took and decisions we made

How we solved the problem

Where we are today

Why restrict egress?

• **Zero-days**	Log4j (1)
• **C2 frameworks**	Cobalt Strike (2) (3) (4)
• **Ransomware**	Data exfiltration/double extortion (5)

1. https://aws.amazon.com/blogs/security/using-aws-security-services-to-protect-against-detect-and-respond-to-the-log4j-vulnerability/
2. https://attack.mitre.org/software/S0154/
3. https://blog.talosintelligence.com/2020/09/CTIR-quarterly-trends-Q4-2020.html
4. https://malpedia.caad.fkie.fraunhofer.de/details/win.cobalt_strike
5. https://www.cybereason.com/blog/rise-of-double-extortion-shines-spotlight-on-ransomware-prevention

AWS Network Firewall

AWS managed deep packet inspection firewall

Managed infrastructure for high availability

Flexible protection through fine-grained controls

Consistent policy across VPCs and AWS accounts

Network Firewall is built for the cloud

Scales automatically, AWS managed infrastructure

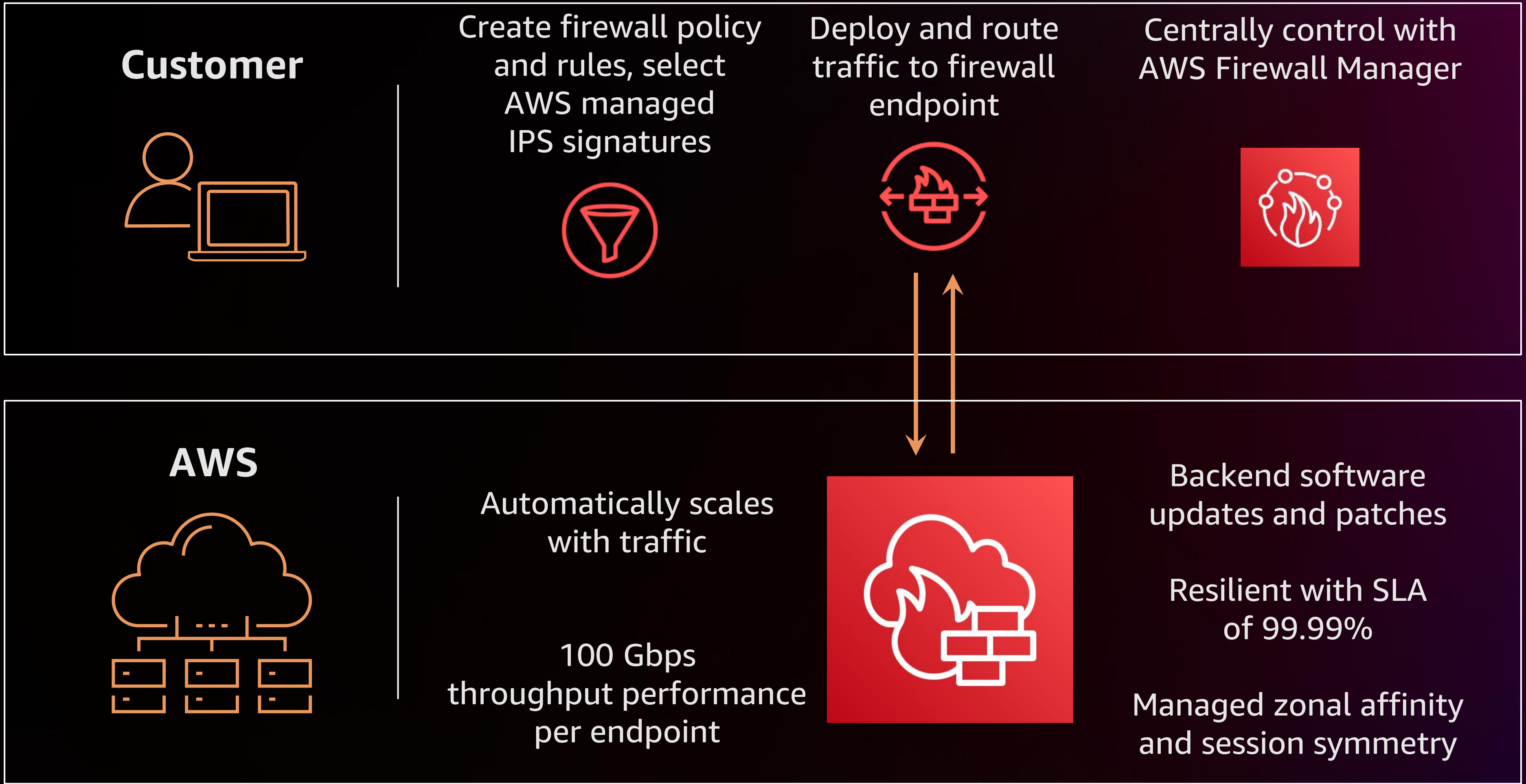
Deep packet inspection

AWS managed IPS signatures and threat intelligence

Centrally managed policies, real-time monitoring, increased visibility

No upfront commitments and pay only for what you use

Network Firewall at a glance



Customer

Create firewall policy and rules, select AWS managed IPS signatures

Deploy and route traffic to firewall endpoint

Centrally control with AWS Firewall Manager

AWS

Automatically scales with traffic

100 Gbps throughput performance per endpoint

Backend software updates and patches

Resilient with SLA of 99.99%

Managed zonal affinity and session symmetry

Network Firewall features

Advanced filtering

- Domain filtering
- Suricata IDS/IPS rules
- AWS managed threat signatures
- Protocol detection and enforcement
- Large-scale 5-tuple rules

Visibility and reporting

- Amazon CloudWatch rule metrics
- Full network flow logs
- Event, rule-based logs
- Log collection to Amazon S3, Amazon CloudWatch Logs, or Amazon Kinesis Data Firehose
- Amazon CloudWatch Contributor Insights

Central management

- Cross-account management and rule visibility using AWS Firewall Manager
- AWS CloudFormation and Terraform templates

Network Firewall top customer use cases

Egress security

- Software supply chain security
- Domain/FQDN filtering
- DenyListing Known-Bad and AllowListing of Known-Good
    - FQDNs (HTTP, HTTPS, DNS)
    - CIDRs
    - ccTLDs
    - TLS JA3/S hashes
    - TLS server certs fingerprint
    - Ports (e.g., 1389, 4444)
- Ensure ports are used only by their legitimate protocol
- Block vulnerable versions of TLS
- Block direct to IP communications
- Threat hunting/reverse stack ranking

Environment segmentation

- VPC to VPC
- Prod to dev, dev to prod
- VPC to on premises, on premises to VPC

Intrusion prevention

- Running IDS/IPS signatures from open-source repositories, partners, or both
- AWS managed IPS rules
- Auto block IPs seen brute forcing by Amazon GuardDuty

The journey

Problem: proliferation of unique point solutions (proxies) for egress filtering and desire for centralized security tooling for detection and response

Solution: complementary to existing cloud security capabilities

Flexible implementation: guardrail approach, prevent the known bad, reduce risk

Evolve rapidly; don't break prod

Egress control requirements

Capable

North-South inspection

Flexible rules engine

Deep packet inspection

Centralized orchestration options

Reliable

Multi-AZ, multi-Region

Cellular architecture

Managed infrastructure

Scalable

Pay as you go

Infrastructure-as-code support

Evaluation

	Network Firewall	Third-party appliance w/GWLB	NACLs and security groups
Centralized orchestration	✓	partial	✗
Managed capabilities	✓	✓	n/a
AWS integration	✓	partial	✓
Pay as you go (no contract)	✓	partial	✓
Quotas	documented	unknown	documented
Ruleset transparency	medium	low	high
Management	low	high	high
Automatically inherit future capabilities	✓	low	high
Domain level filtering	✓	✓	✗

Phased goals for egress controls

Enhance visibility

Deploy firewalls

Implement telemetry

Dashboards

Blocking and tackling

Establish runbooks, playbooks

Templates for blocking in incident response scenarios

Approval chains

Gracefully move towards positive security model

Deny Known-Bad

Mining data for allow list for egress traffic

Further isolation of nonconforming traffic

Implementation

Implementation goals

- Capture, monitor all egress traffic
  - Provide insight into all outgoing traffic
    - Packet inspection shows protocol details
  - Block known malicious traffic ASAP
- Scalable
  - Must not be a major bottleneck
- **Non-goal**
  - Monitor, capture ingress traffic

Option 1 – Centralized

Pros

One firewall deployment

Single point of management

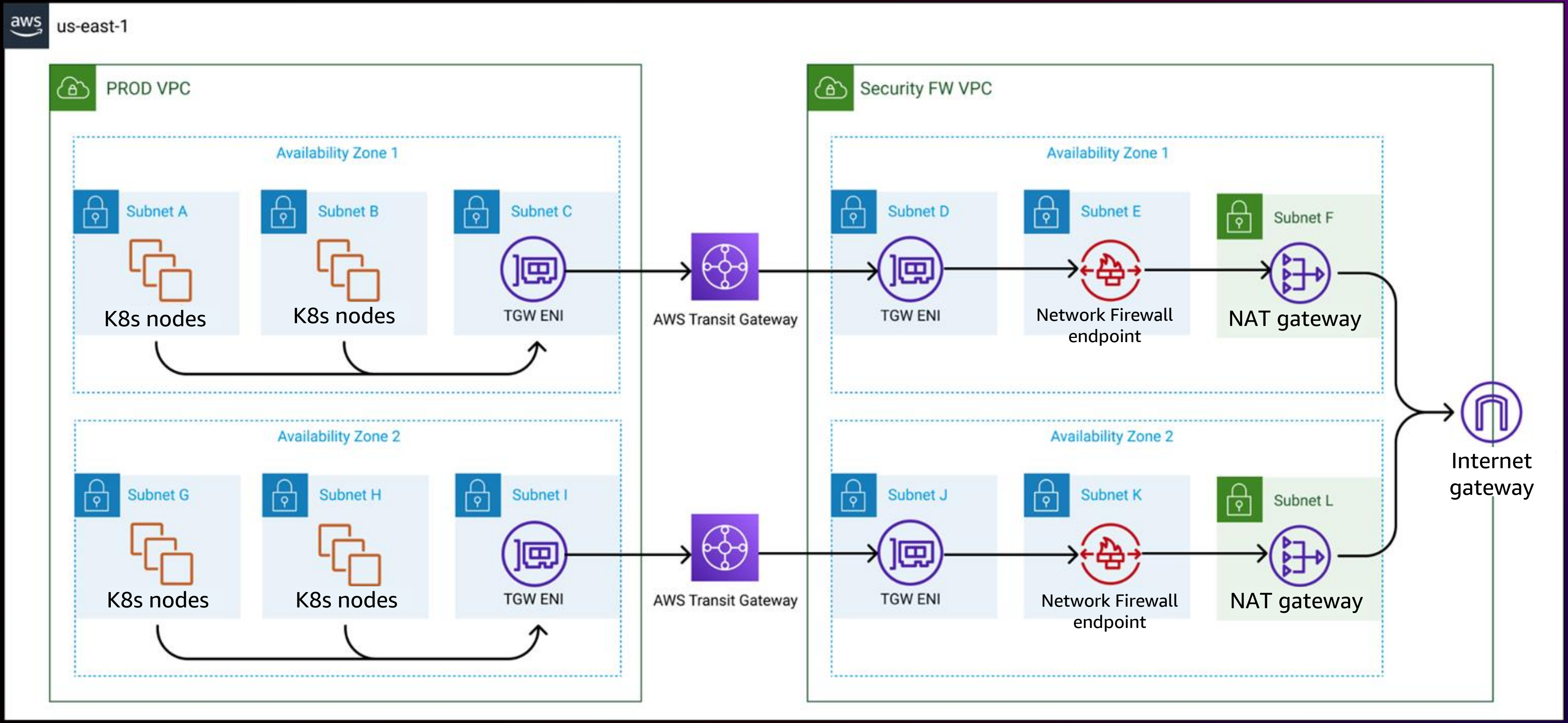
Requires fewer FW endpoints, saves costs

Reduce number of NAT GWs

Cons

All traffic goes through a single set of choke points

Greater dependency on centralized components



Option 2 – Distributed

Pros

Firewall is deployed in place

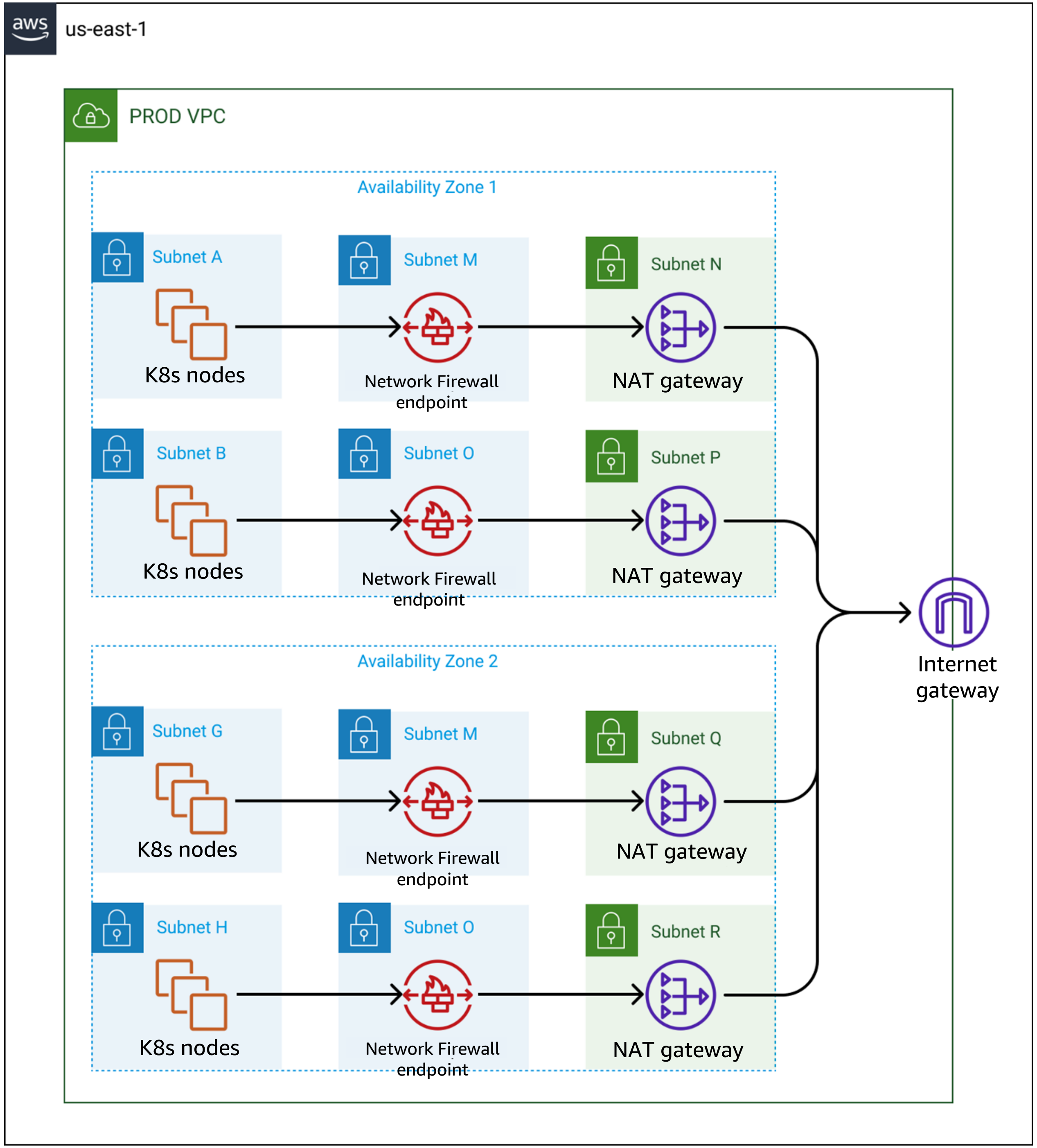
1–1 scaling

Phased rollout of the firewall and rules

Cons

Multiple deployments, managed components

More complex firewall rule management



Winner: Distributed

Max bandwidth

NAT – 45 Gbps

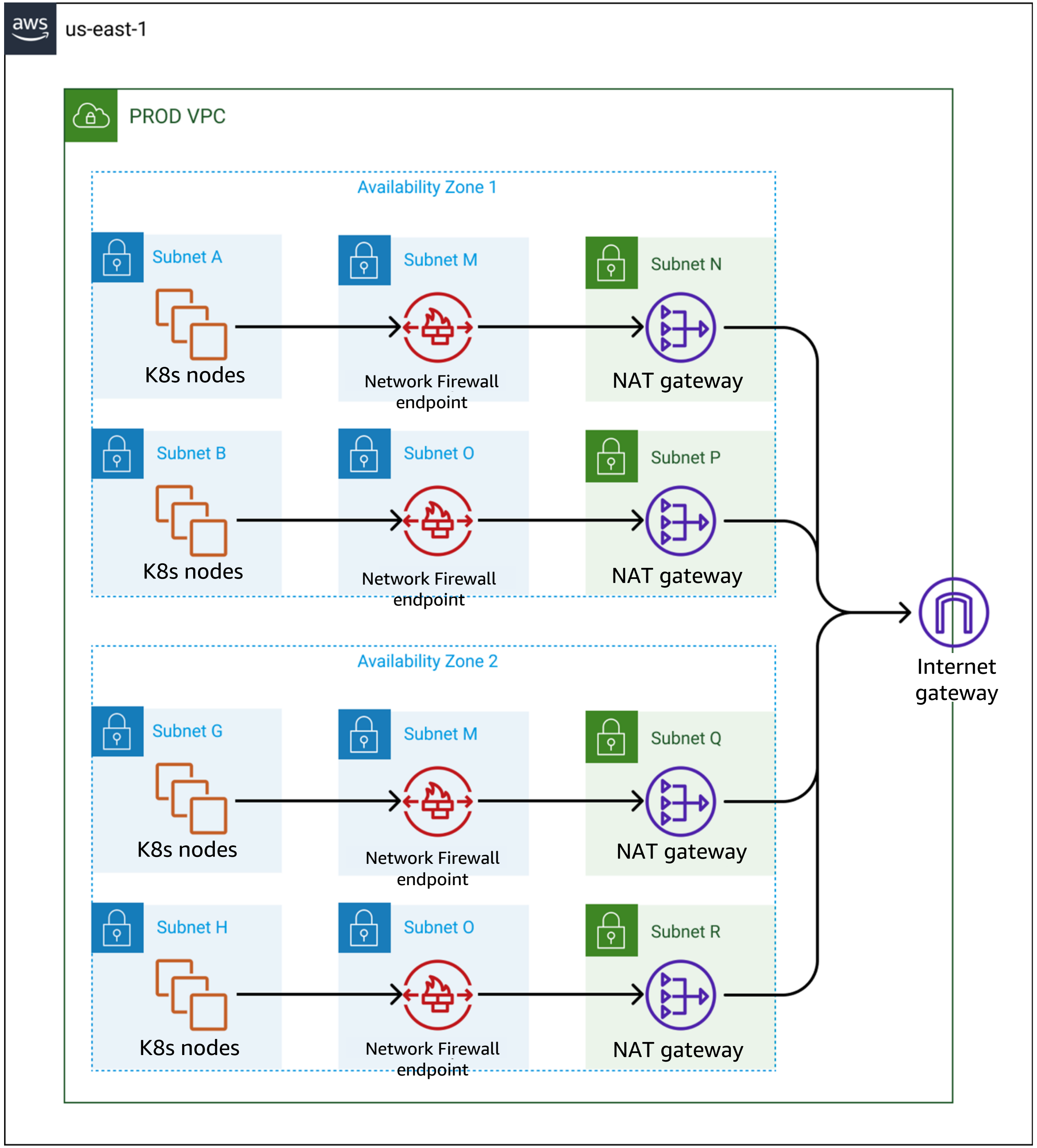
Network Firewall endpoint – 100 Gbps

Robinhood

We almost maxed 45 Gbps NAT limit

Now, each K8s cluster has its own NAT per AZ

Centralized model does not meet our scaling needs, but in most cases it is a good fit



Our infrastructure

ROBINHOOD WITHOUT THE FIREWALL

Example of our Kubernetes cluster

Public subnets: NAT gateways and load balancers
Private subnets: K8s nodes (Amazon EC2 instances)

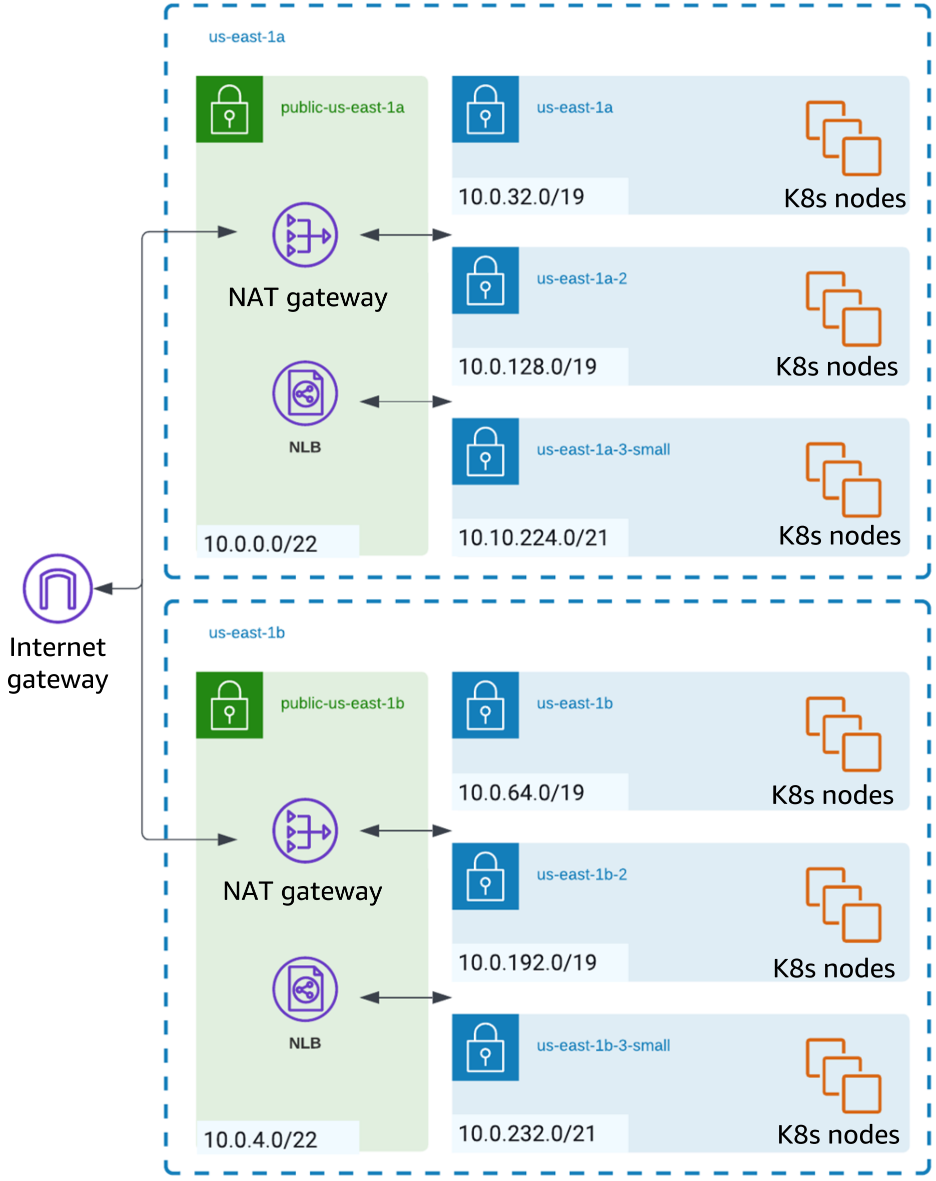
Some key details

One NAT gateway per Availability Zone

Public LBs and the NAT gateway live in same subnets

Public subnets route directly to IGW

Private subnets route through NATs



Destination	Route
10.0.0.0/16	local
0.0.0.0	igw

Destination	Route
10.0.0.0/16	local
0.0.0.0	nat-1a

Robinhood and Network Firewall

ROBINHOOD WITH THE FIREWALL

One Network Firewall endpoint per Availability Zone

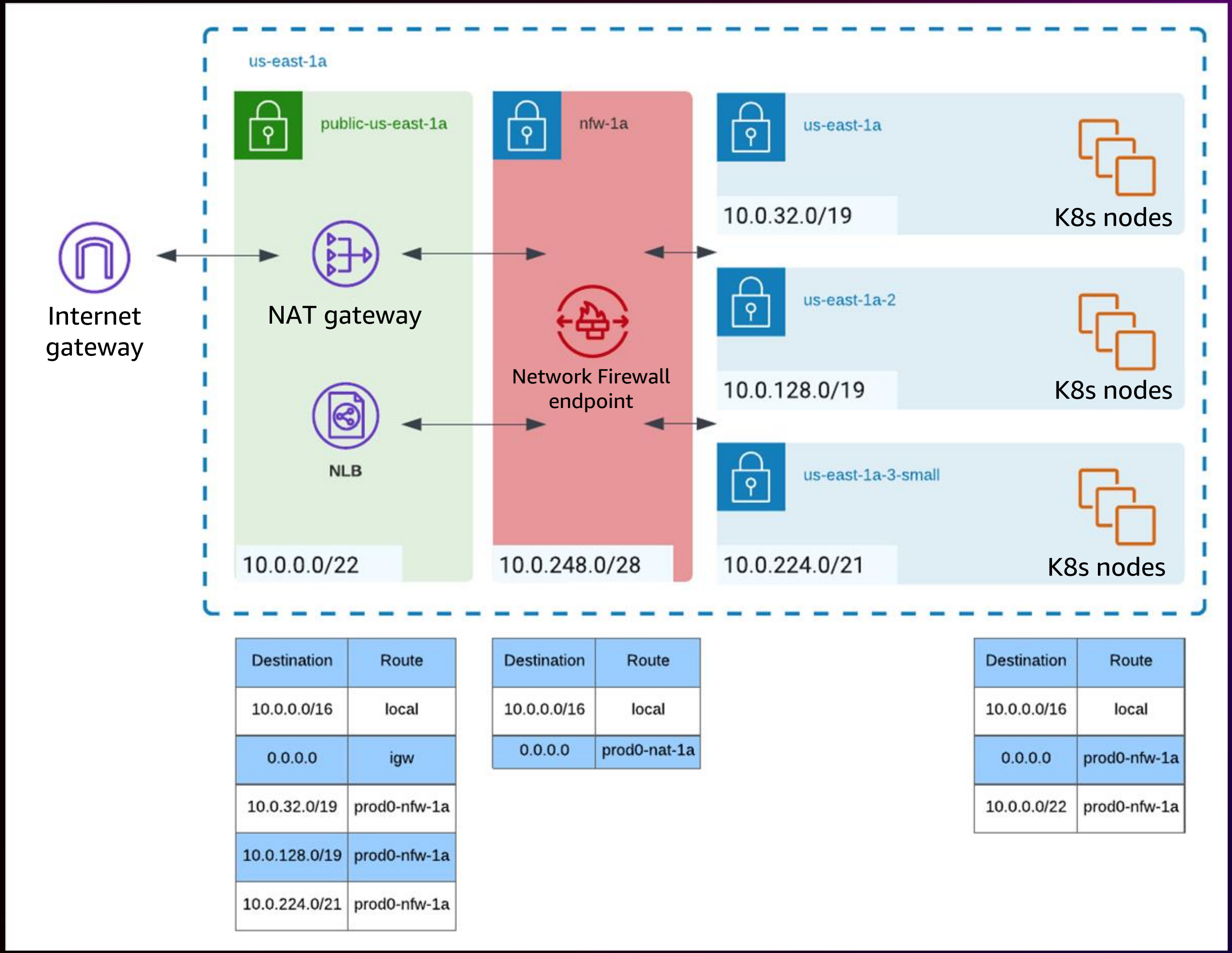
- Aligns with our scaling needs of
  1 NAT → 1 Network Firewall endpoint

Due to NAT and LBs sharing a subnet, firewall sits in between public and private subnets

- Captures some internal traffic within same AZ heading to NLB

Why not place the firewall after NAT (before IGW)?

- Makes the routing simpler but . . .
- The source IP of the K8s nodes would be masked by the NAT
- Logs would simply display the NAT IP as the source



Destination	Route
10.0.0.0/16	local
0.0.0.0	igw
10.0.32.0/19	prod0-nfw-1a
10.0.128.0/19	prod0-nfw-1a
10.0.224.0/21	prod0-nfw-1a

Destination	Route
10.0.0.0/16	local
0.0.0.0	prod0-nat-1a

Destination	Route
10.0.0.0/16	local
0.0.0.0	prod0-nfw-1a
10.0.0.0/22	prod0-nfw-1a

Network symmetry

ASYMMETRIC ROUTE = BAD DAY

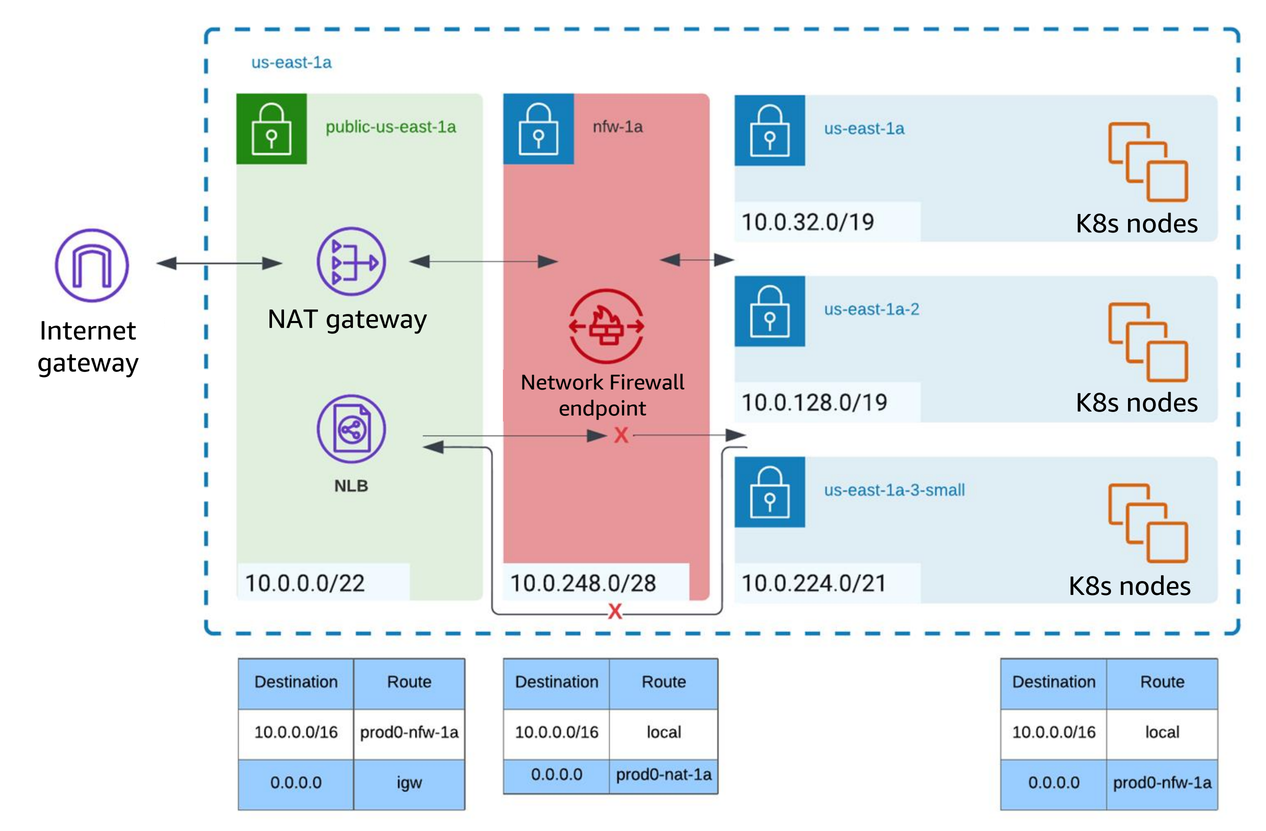
The return path **must** be symmetric

If traffic in one direction passes through a firewall endpoint, then the return traffic must also pass through the same firewall endpoint

Otherwise, firewall simply drops the packet

Example on the right

Public Subnet ← Firewall ← Private Subnet
Public Subnet → Firewall → Private Subnet



Destination	Route
10.0.0.0/16	prod0-nfw-1a
0.0.0.0	igw

Destination	Route
10.0.0.0/16	local
0.0.0.0	prod0-nat-1a

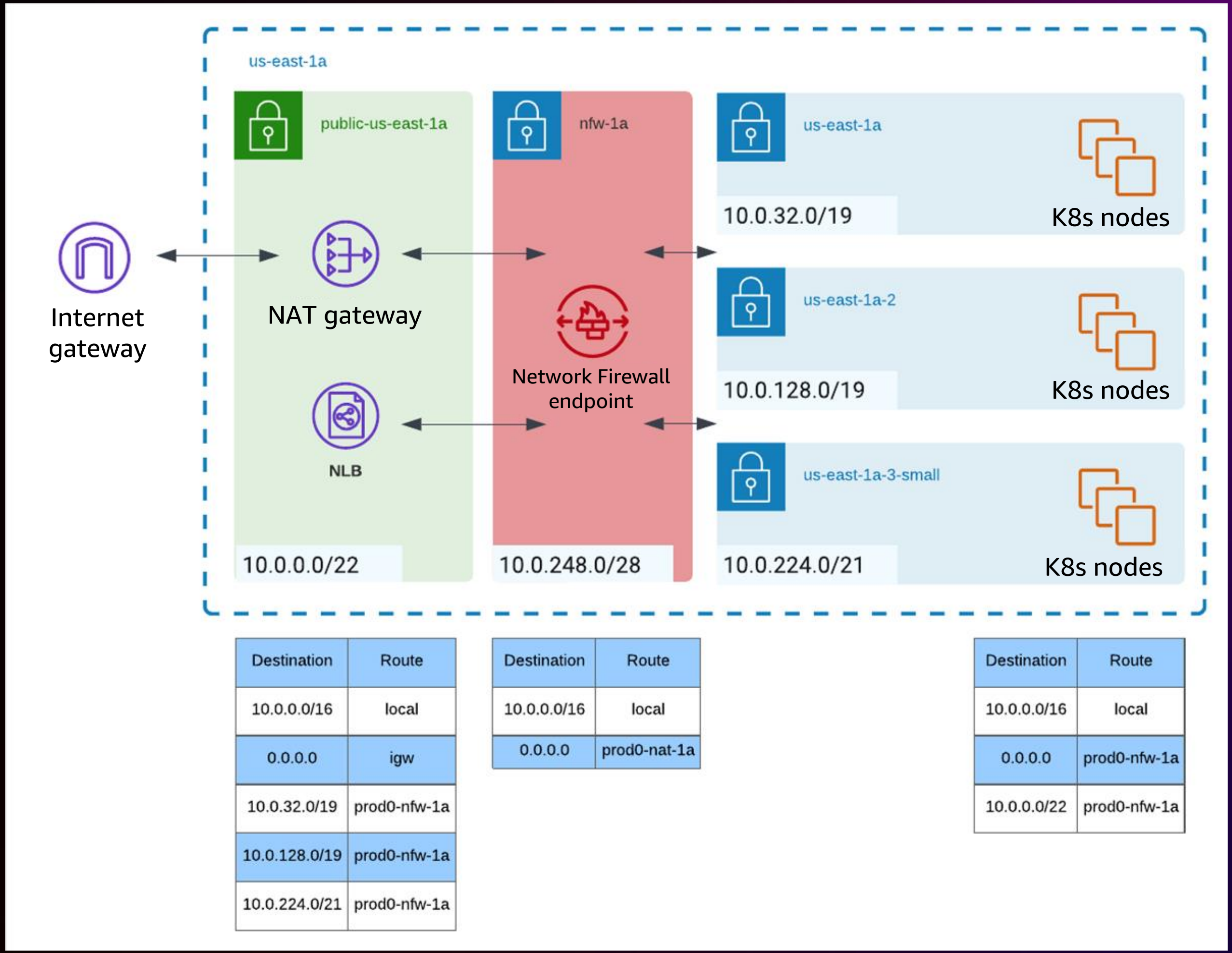
Destination	Route
10.0.0.0/16	local
0.0.0.0	prod0-nfw-1a

Network symmetry

GRANULAR ROUTES SAVE THE DAY

Solution – more granular routes

Use "More Specific Route" to explicitly define target subnets CIDR range and desired target into the route tables



Destination	Route
10.0.0.0/16	local
0.0.0.0	igw
10.0.32.0/19	prod0-nfw-1a
10.0.128.0/19	prod0-nfw-1a
10.0.224.0/21	prod0-nfw-1a

Destination	Route
10.0.0.0/16	local
0.0.0.0	prod0-nat-1a

Destination	Route
10.0.0.0/16	local
0.0.0.0	prod0-nfw-1a
10.0.0.0/22	prod0-nfw-1a

Deployment

REPLICATING DEPLOYMENTS ACROSS ENVIRONMENTS

HashiCorp Terraform

One Terraform module to rule all deployment

Automatically creates

- Subnets for the firewall endpoints
- New route tables for **public**, **private**, **firewall** subnets
- Proper routes to NAT gateways
- Logging rules and destination

Network Firewall – pay for what you use

With a conditional flag
Only deploy in environments where it is needed

Why create new set of route tables for both public and private subnets? 🤔

Why duplicate route tables just to add firewall routes?

```
module "firewall" {
  # Create conditionally
  count  = var.firewall_enabled ? 1 : 0
  source = "modules/network_firewall"
  is_k8s = true  # For resource naming convention

  name        = var.name
  vpc_id      = var.vpc_id
  target_cidr = var.cidr_block

  firewall_subnets = local.firewall_subnets
  public_subnets   = local.public_subnets
  private_subnets  = local.private_subnets

  tags = {
    "kubernetes.io/cluster/${var.name}" = "firewall"
    KubernetesCluster                    = var.name
  }
}
```

Deployment

DEPLOY WITH ZERO PRODUCTION DOWNTIME

HashiCorp Terraform

Inserting a firewall into a live production system is tricky

One mistake in the route table
can bring the production down

How did we solve this?

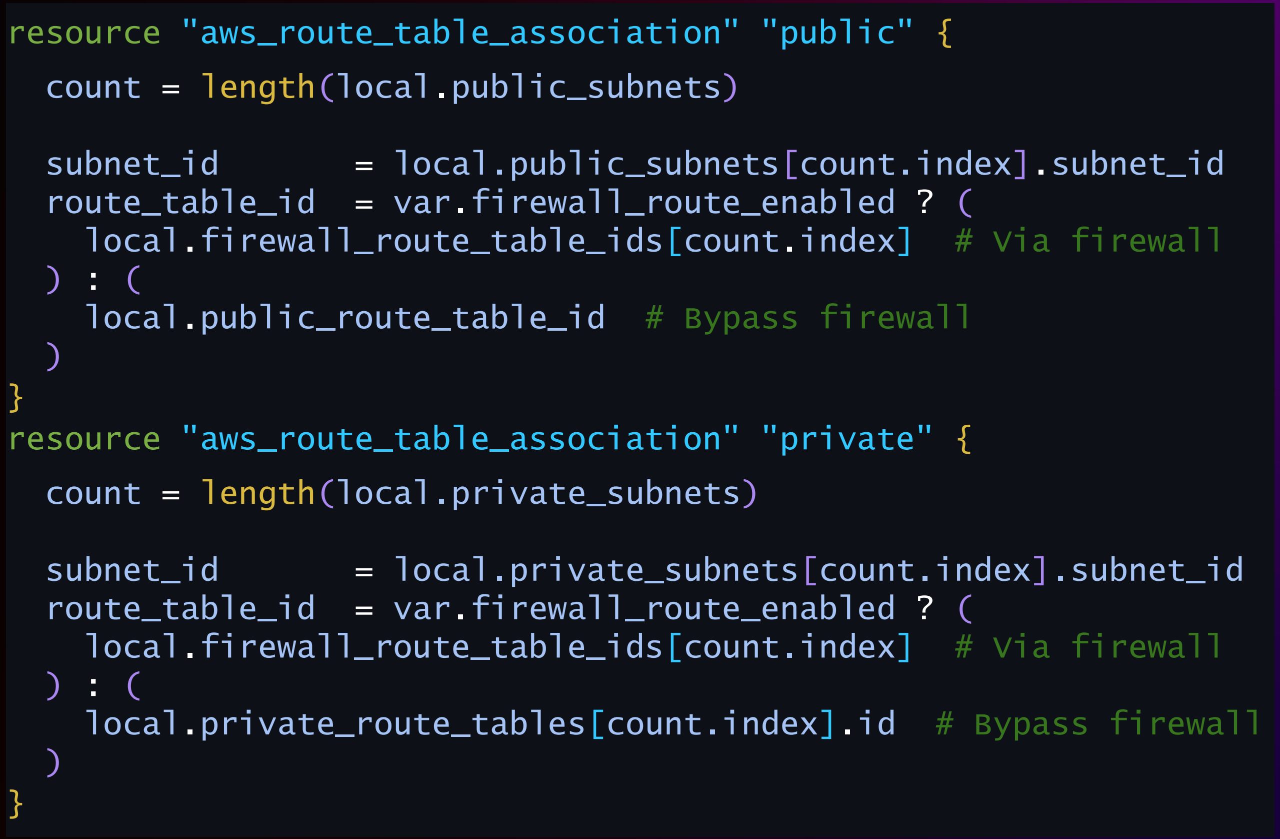
Create all necessary resources **in advance**

Duplicate all routes and route tables

This allows us to verify once more
before taking the system live

Once ready, set the Boolean switch to **true**
Then, execute a final **terraform apply**

This updates route table associations of all
affected subnets simultaneously



```
resource "aws_route_table_association" "public" {
  count = length(local.public_subnets)

  subnet_id        = local.public_subnets[count.index].subnet_id
  route_table_id   = var.firewall_route_enabled ? (
    local.firewall_route_table_ids[count.index]  # Via firewall
  ) : (
    local.public_route_table_id  # Bypass firewall
  )
}
resource "aws_route_table_association" "private" {
  count = length(local.private_subnets)

  subnet_id        = local.private_subnets[count.index].subnet_id
  route_table_id   = var.firewall_route_enabled ? (
    local.firewall_route_table_ids[count.index]  # Via firewall
  ) : (
    local.private_route_tables[count.index].id  # Bypass firewall
  )
}
```

Fail-safe

WHAT HAPPENS IF THE FIREWALL FAILS?

Imagine a rare outage scenario . . .
where an AWS shared cell with our firewall endpoint fails

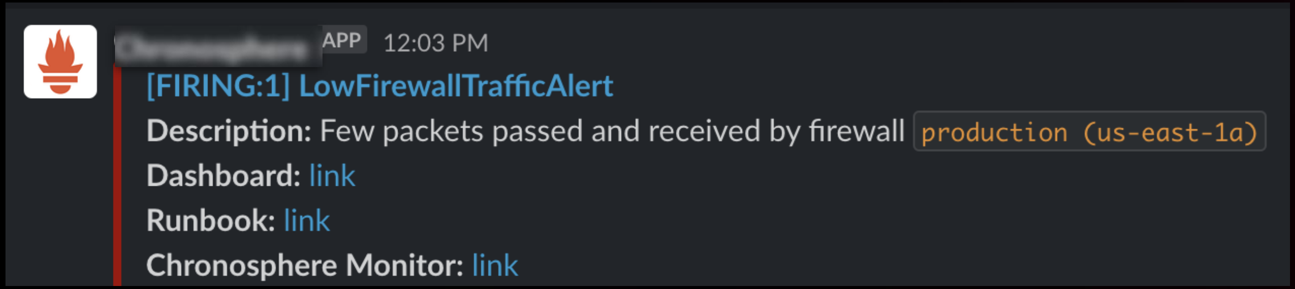
The same switch mechanism doubles as a fail-safe

- A simple Terraform Boolean value change (**true** → **false**)
- Reverts route table associations back to the original
- Bypasses the firewall in matter of seconds

Monitoring and alerting

KEEPING AN EYE ON THE FIREWALL

CloudWatch provides the necessary operational metrics



[redacted] APP 12:03 PM

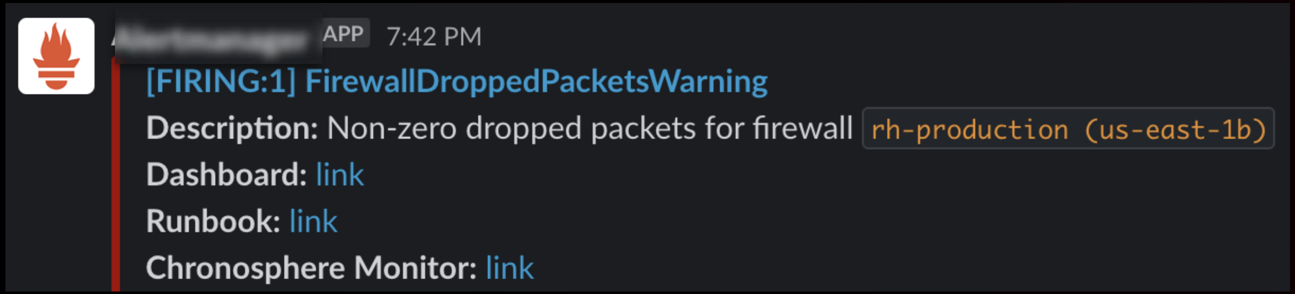
[FIRING:1] LowFirewallTrafficAlert

Description: Few packets passed and received by firewall `production (us-east-1a)`

Dashboard: link

Runbook: link

Chronosphere Monitor: link



[redacted] APP 7:42 PM

[FIRING:1] FirewallDroppedPacketsWarning

Description: Non-zero dropped packets for firewall `rh-production (us-east-1b)`

Dashboard: link

Runbook: link

Chronosphere Monitor: link

If the firewall traffic drops significantly due to being

- Disabled
- Bypassed

If there is a surge in dropped packets due to one of the following

- Hitting bandwidth limits
- Firing blocking rule

We would get alerted on Slack and paged on our phones

Logging and visibility

CAPTURING THE VALUABLE DATA

Valuable application layer insights

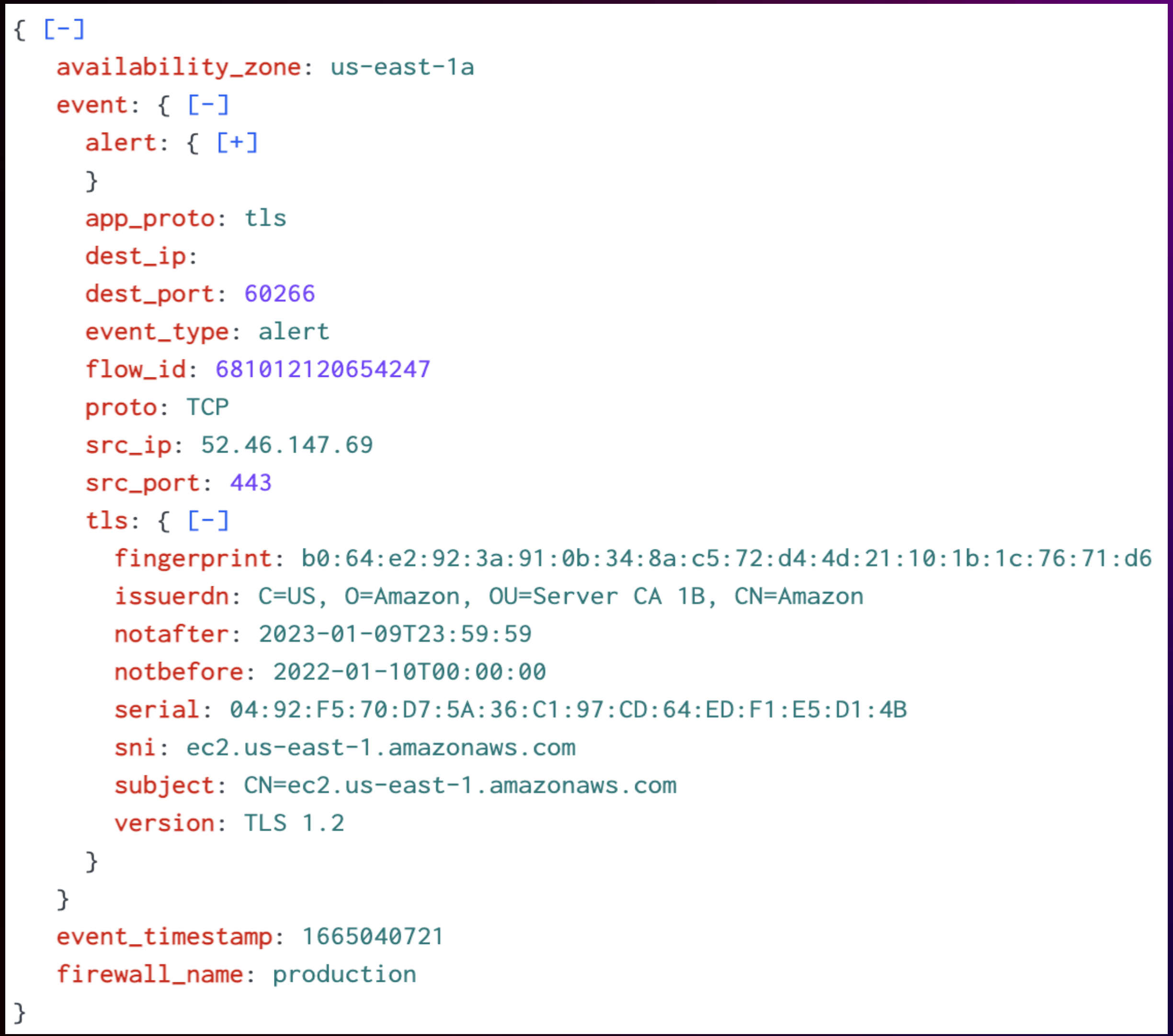
For example,

TLS information provides additional context into the network traffic

Whereas just an IP address is difficult to understand and investigate

Can reverse stack-rank JA3 hashes or clients to find outliers

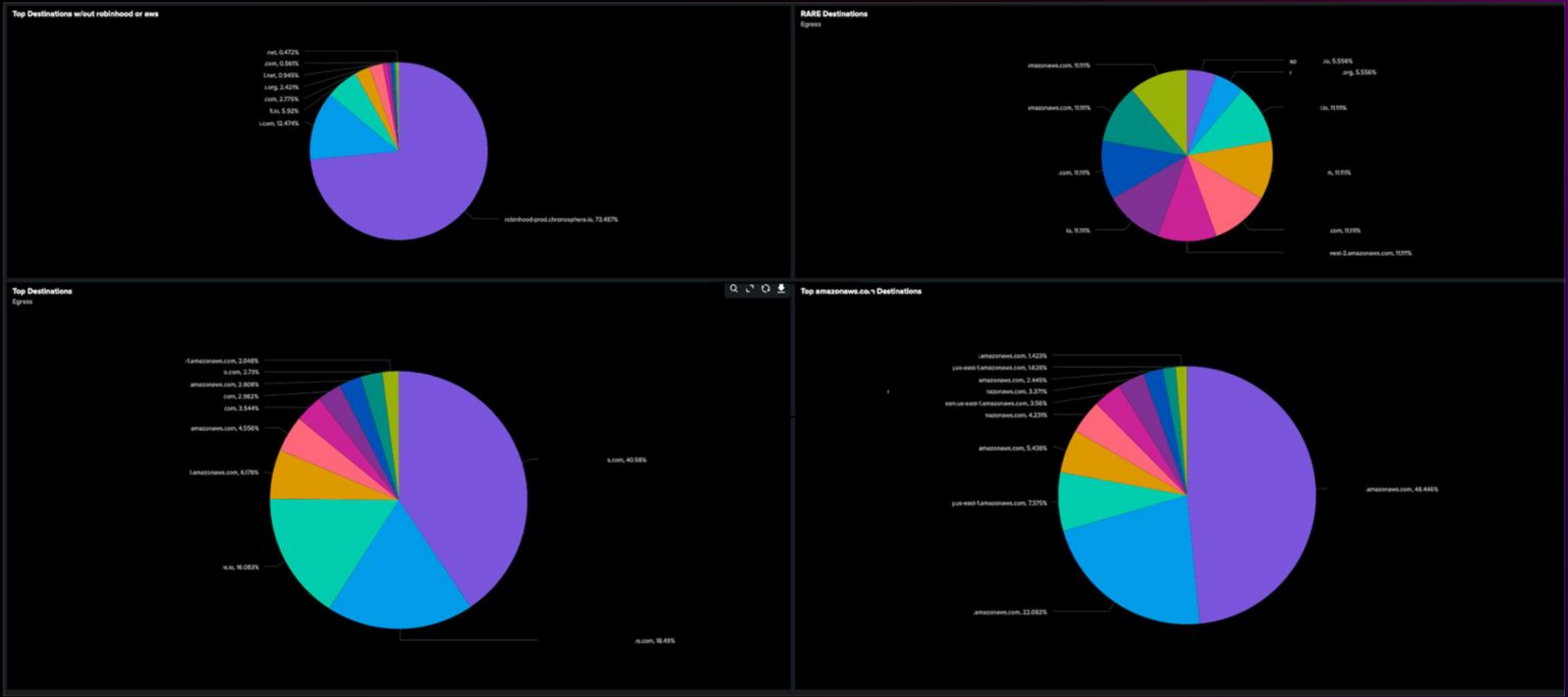
Built-in flow logs without need for VPC Flow Logs



```
{ [-]
   availability_zone: us-east-1a
   event: { [-]
     alert: { [+]
     }
     app_proto: tls
     dest_ip:
     dest_port: 60266
     event_type: alert
     flow_id: 681012120654247
     proto: TCP
     src_ip: 52.46.147.69
     src_port: 443
     tls: { [-]
       fingerprint: b0:64:e2:92:3a:91:0b:34:8a:c5:72:d4:4d:21:10:1b:1c:76:71:d6
       issuerdn: C=US, O=Amazon, OU=Server CA 1B, CN=Amazon
       notafter: 2023-01-09T23:59:59
       notbefore: 2022-01-10T00:00:00
       serial: 04:92:F5:70:D7:5A:36:C1:97:CD:64:ED:F1:E5:D1:4B
       sni: ec2.us-east-1.amazonaws.com
       subject: CN=ec2.us-east-1.amazonaws.com
       version: TLS 1.2
     }
   }
   event_timestamp: 1665040721
   firewall_name: production
}
```

Logging and visibility

CAPTURING THE VALUABLE DATA



Interesting discoveries

1. Lots of things going to the internet that aren’t malicious but are unnecessary
   a. Quickly identified VPC endpoint opportunities for AWS traffic
   b. Robinhood internal traffic routing to the internet

2. Measure the top and bottom
   a. Early stages: look for highest count domains, accessed by highest count clients, to add to AllowList
   b. Identify and investigate rare and outlier domains; could be malicious, could be misconfig
   c. Remove outliers to form a baseline; is there a middle?

3. Deeper segmentation opportunities
   a. Some applications may not fit a positive security model

Further improvements

Move NAT gateways into own subnets

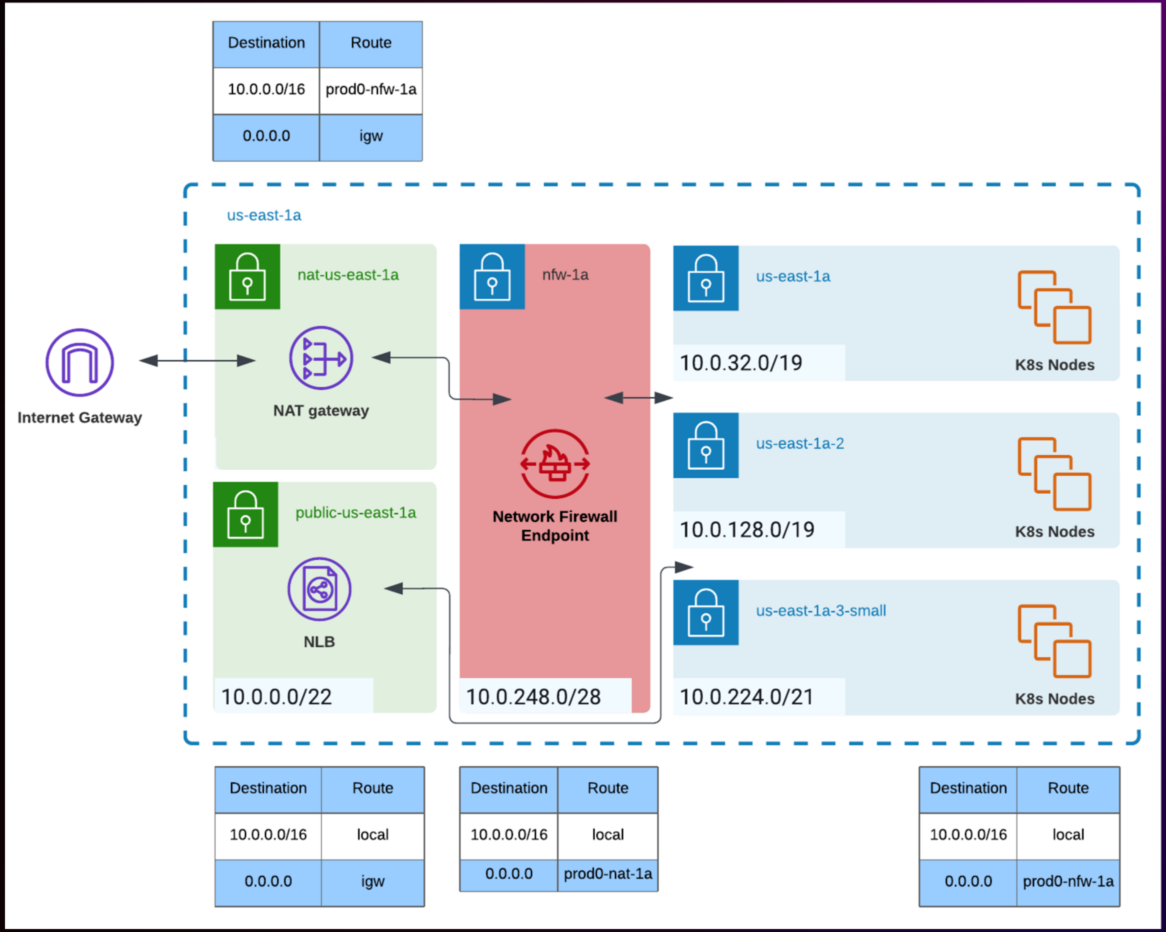
Recall where firewall subnet is sandwiched between private and public subnets

This captures some internal to internal traffic, which

- Does not aid in egress control
- Adds noise to our logging capabilities
- Takes up firewall bandwidth and adds cost

Simplifies routes in route tables

Doable with zero production downtime



Destination	Route
10.0.0.0/16	prod0-nfw-1a
0.0.0.0	igw

Destination	Route
10.0.0.0/16	local
0.0.0.0	igw

Destination	Route
10.0.0.0/16	local
0.0.0.0	prod0-nat-1a

Destination	Route
10.0.0.0/16	local
0.0.0.0	prod0-nfw-1a

Thank you!

Please complete the session survey in the **mobile app**