

Thick and Thin events

- ▶ A "thin" event only carries a small data package such as time of the event and some sort of ID to identify the event. The data payload is not sent with the event. This approach is used in "**Event Notification**" pattern
 - ▶ producer publishes an event indicating that something happened, maybe also an ID and timestamp
 - ▶ state, i.e., the data, requires consumers to query the producer to obtain the data, typically via an API call
- ▶ Thin event benefits include
 - ▶ less data transferred via the broker
 - ▶ consumer can ask for whatever data it needs from one or more APIs rather than take all of it
 - ▶ popular for legacy monolithic systems that don't support thick events but do have API functionality
- ▶ Challenges of thin events
 - ▶ Synchronous API call adds tight coupling so reduces resilience
 - ▶ Slower as an event is processed then the API is called
 - ▶ APIs can suffer due to concurrent burst-type requests from subscribers requesting data on receipt of the event
- ▶ Both patterns are valid depending on the use case. If consumers can tolerate synchronous dependence on a producer, thin messages or synchronous calls may be preferable; whereas fat events are better suited to asynchronous, decoupled integration where resilience and autonomy are prioritised e.g. distributed systems

Kafka

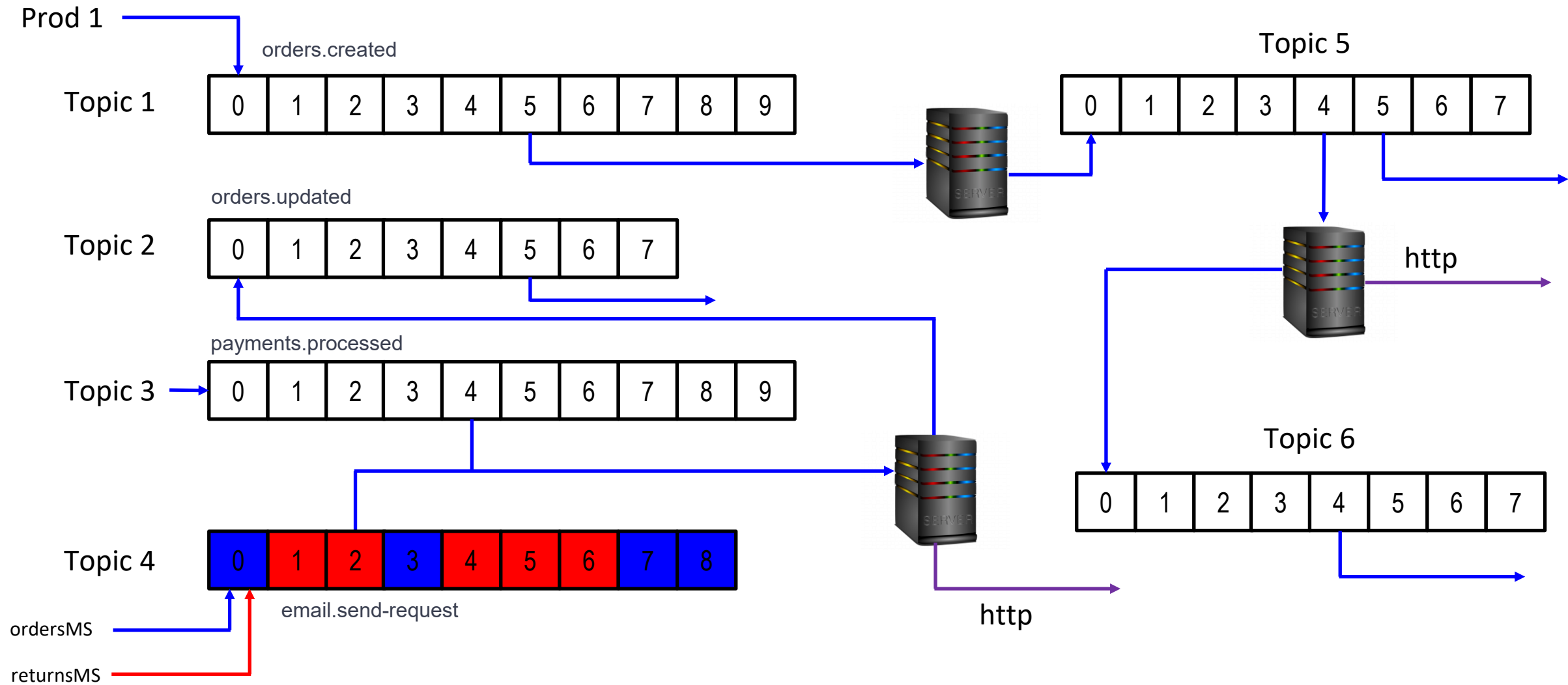
- ▶ Popular open source event-based pub / sub streaming application
 - ▶ Developed by LinkedIn to track user activity on its platform
 - ▶ It's now been made open source as part of the Apache foundation and managed by Confluent
 - ▶ Very popular in event driven microservice architectures
 - ▶ Used by Netflix, Spotify, Uber, LinkedIn, Twitter and many other well-known enterprises
 - ▶ As in RabbitMQ, Kafka uses producer/consumer model and supports prod/cons and pub/sub message (event) pattern
 - ▶ In principle it is very similar to RabbitMQ but is functionally different in that it uses persistent logs instead of queues
- ▶ It captures and stores events
 - ▶ Events such as IoT sensor data, Web-site click stream, track and monitor delivery vehicles etc., are captured in real-time into a log - i.e. append-only read-only storage so maintains sequence and can't be changed
 - ▶ As in MQ, there can be multiple producers and multiple consumers
 - ▶ Anyone interested in the Payments topic would subscribe to that topic
 - ▶ Events are stored however you want, text, xml, whatever as data is serialised like in MQ

▶ **Kafka is a distributed application**

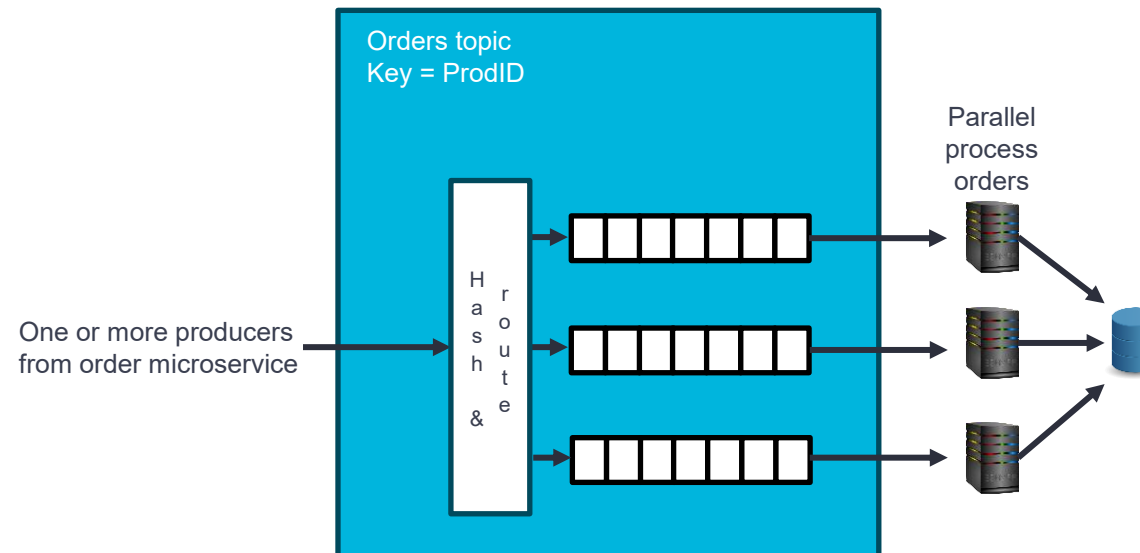
- ▶ Fault tolerant, horizontally scalable, enables parallel processing across multiple nodes

▶ **Kafka and Queue key differences**

- ▶ A basic queue has one producer. Kafka can have multiple producers as can MQ by using an exchange
- ▶ A basic queue has one consumer. The queued message is deleted once read. Kafka can have many subscribers (consumers in Kafka Speak) to the same topic because reading the data doesn't cause it to be deleted (removed from the queue) as data is not deleted once read; Kafka events are persisted. A bit like reading an array or file using a pointer for each consumer
- ▶ Kafka data retention can be set from minutes through to infinity, by default 1 week, or by storage size
- ▶ Unlike RabbitMQ which can be configured as a priority queue, Kafka stores events / messages as an immutable append log - i.e. in the order in which they come in is maintained
- ▶ Kafka consumers have an index into the log (basically a file pointer) which increments as they read messages. If they need to go back, they can change the read index on the log - basically like accessing and reading a log file
- ▶ Rather than replicating data in a one queue per subscriber, Kafka has one copy of the events and subscribers read the events using their own offsets into the event stream
- ▶ Kafka supports data streaming. e.g. data can be streamed in through a "Kafka connect" connector to a streaming data source such as a click stream, IoT, video etc. The stream can be processed by a consumer, then output to another topic which is streamed back out through "Kafka stream". e.g. IoT data streams in, the consumer processes it in real time, event by event, and streams the output through another topic to real time analytics service or whatever
- ▶ You can add whatever events you want to a topic from multiple producers, the consumers will filter on what they want to see based on a key of some sort, e.g., customerID



- ▶ The example showed a single topic partition - there are various options to improve performance and resilience
- ▶ When a cluster is created you can specify the number of partitions (previous discussion was one), say we want 3
- ▶ The producer will provide a key, say a user ID. Kafka will hash that to choose the partition: $p = \text{hash}(\text{key}) \% 3$
- ▶ This ensures strict ordering of users or products or whatever based on a key this enables parallel processing
- ▶ A Kafka cluster can support any number of topics. LinkedIn has 10s of thousands of topics, millions of partitions



- ▶ Events are not pushed by the broker as in RabbitMQ, a consumer polls the partition - this sounds inefficient but ...
- ▶ A subscriber tries to fetch data. If there is none, it can deliberately block until there is data then it is returned
- ▶ It can also request a return only when a specified number of events have landed
- ▶ So we need to use JavaScript async calls or Java threads

MQ and Kafka basic comparison

Feature	Kafka	RabbitMQ
Primary Role	Distributed event streaming platform	Message broker for reliable messaging
Message Storage	Persistent log (messages retained for a configurable time)	Transient - messages deleted
Ordering	Guaranteed. No prioritisation	Guaranteed but can be config for priority queue - prod can set a priority level
Replay	Supported (consumers can re-read old events)	Not supported after ack
Throughput	Extremely high (millions of messages/sec)	Lower than Kafka, optimized for reliability (uses ack for pub and sub)
Consumer Scaling	Consumer groups to partitions for parallel processing	Multiple queues or competing consumers on one queue - e.g. round robin
Protocol	Custom (binary)	AMQP (Advanced Message Queuing Protocol)
Serialization	Opaque byte arrays	Opaque byte arrays

Summary of patterns

Messaging and architectural patterns

- ▶ **Request–Response:** Synchronous. Caller waits for a reply. Tight coupling. Not formally a message pattern
 - ▶ Typical tech: HTTP, REST, gRPC
- ▶ **Point-to-Point Asynchronous Messaging:** One message, one consumer. Asynchronous. Message lives in a queue. Loose coupling
 - ▶ Typical tech: RabbitMQ queue, Azure Service Bus Queue, AWS Simple Queueing Service (SQS)
- ▶ **Publish–Subscribe Messaging:** Typically, one message many subscribers. Asynchronous. Loose coupling. Subscribers receive independent message copies
 - ▶ Typical tech: Kafka topics, RabbitMQ fanout/topic exchanges
- ▶ **Event Notification:** Event signals that something happened. Carries minimal data (e.g. ID only - thin event). Consumers typically calls synchronous API for state so creates tight coupling
- ▶ **Event Streaming:** Continuous sequence of events. Replayable so ordering matters
 - ▶ Typical tech: Kafka, Amazon Event Hubs, AWS Kinesis
- ▶ **Event-Carried State Transfer:** How a distributed system propagates state between services without creating runtime coupling. Typically uses pub-sub pattern but not a requirement. Event carries the state (thick / fat events) - no callback needed
- ▶ **Event-driven architecture:** Components react to events, are loosely coupled, asynchronous, usually using pub/sub messaging
- ▶ Good Kafka intro video here: <https://youtu.be/QkdkLdMBuL0>

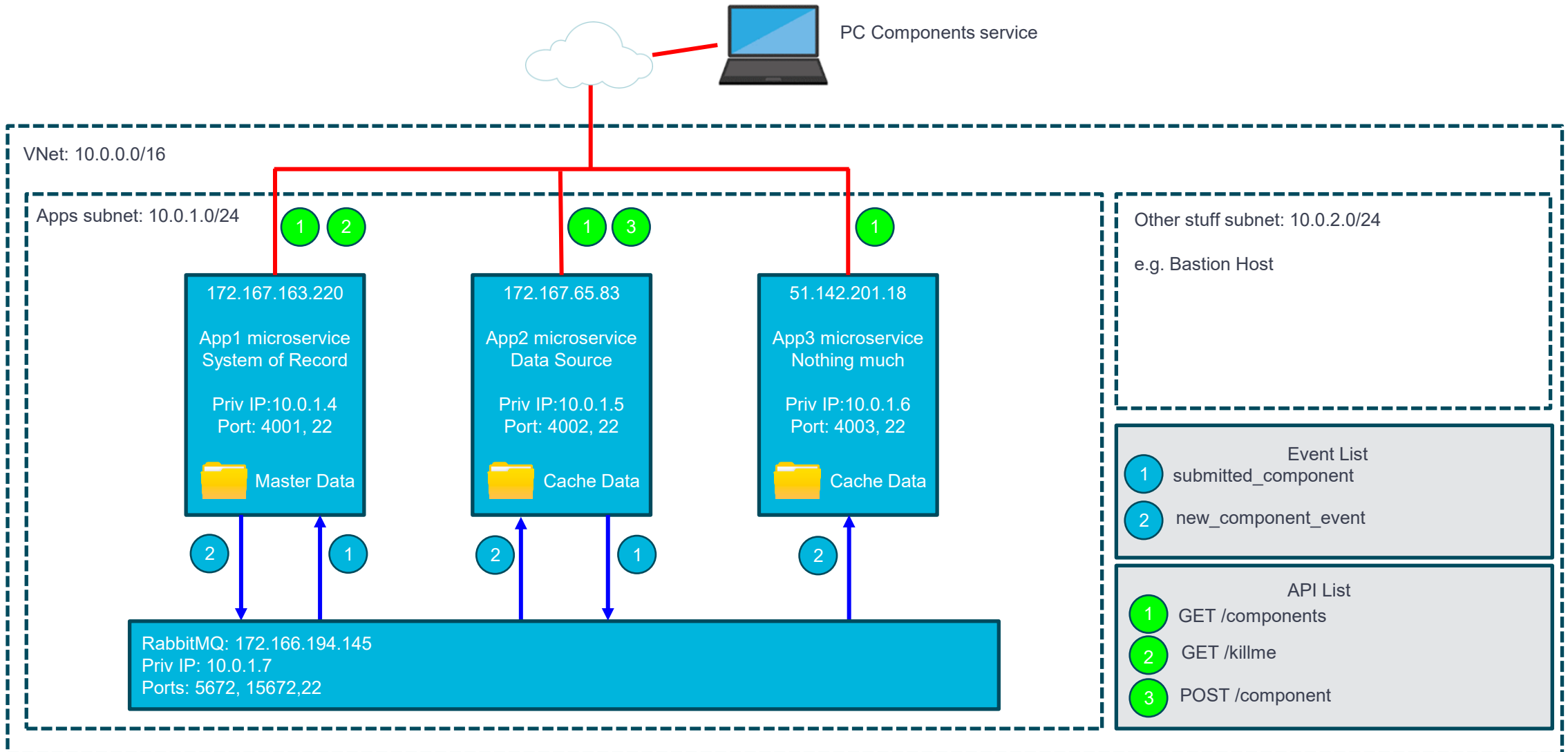


Subnets & Bastion Host

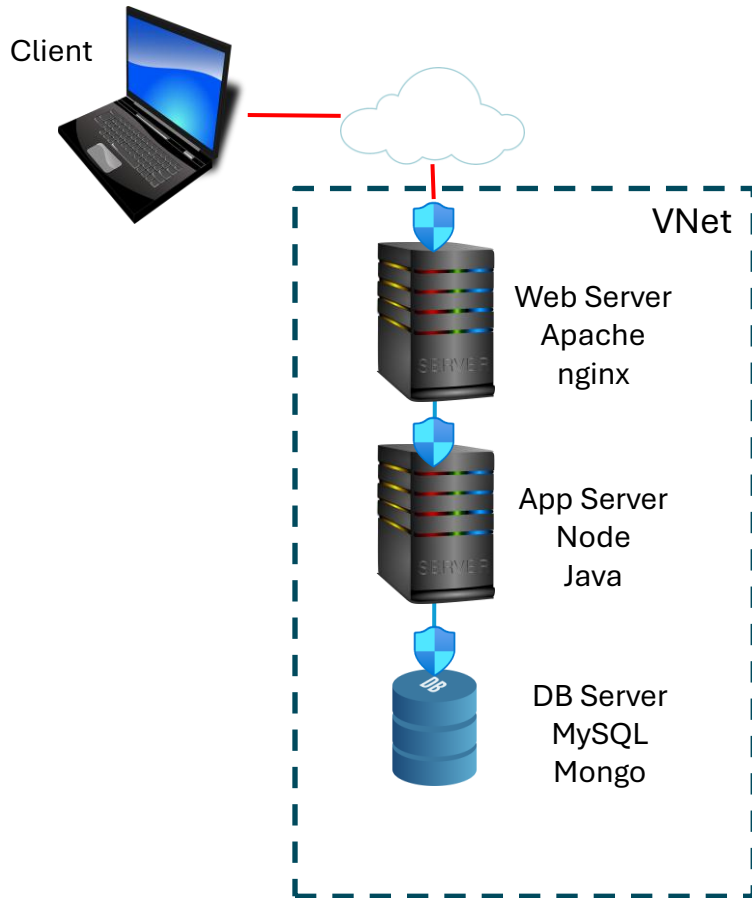
Learning objectives

- ▶ Upon successfully completing this section, you will be able to:
 - ▶ Create subnetworks
 - ▶ Explain the benefit of subnetworks
 - ▶ Explain an architectural need for subnetworks
 - ▶ Create subnetwork security rules
 - ▶ Create a bastion host in the Azure portal to work with last week's demo or any VMs to enable admin access
 - ▶ Explain the purpose and properties of a Bastion host
 - ▶ SSH into any VM using redirects from the SSH config file
 - ▶ Explain virtual network peering
 - ▶ Peer two virtual networks across different regions

Could use Kafka here

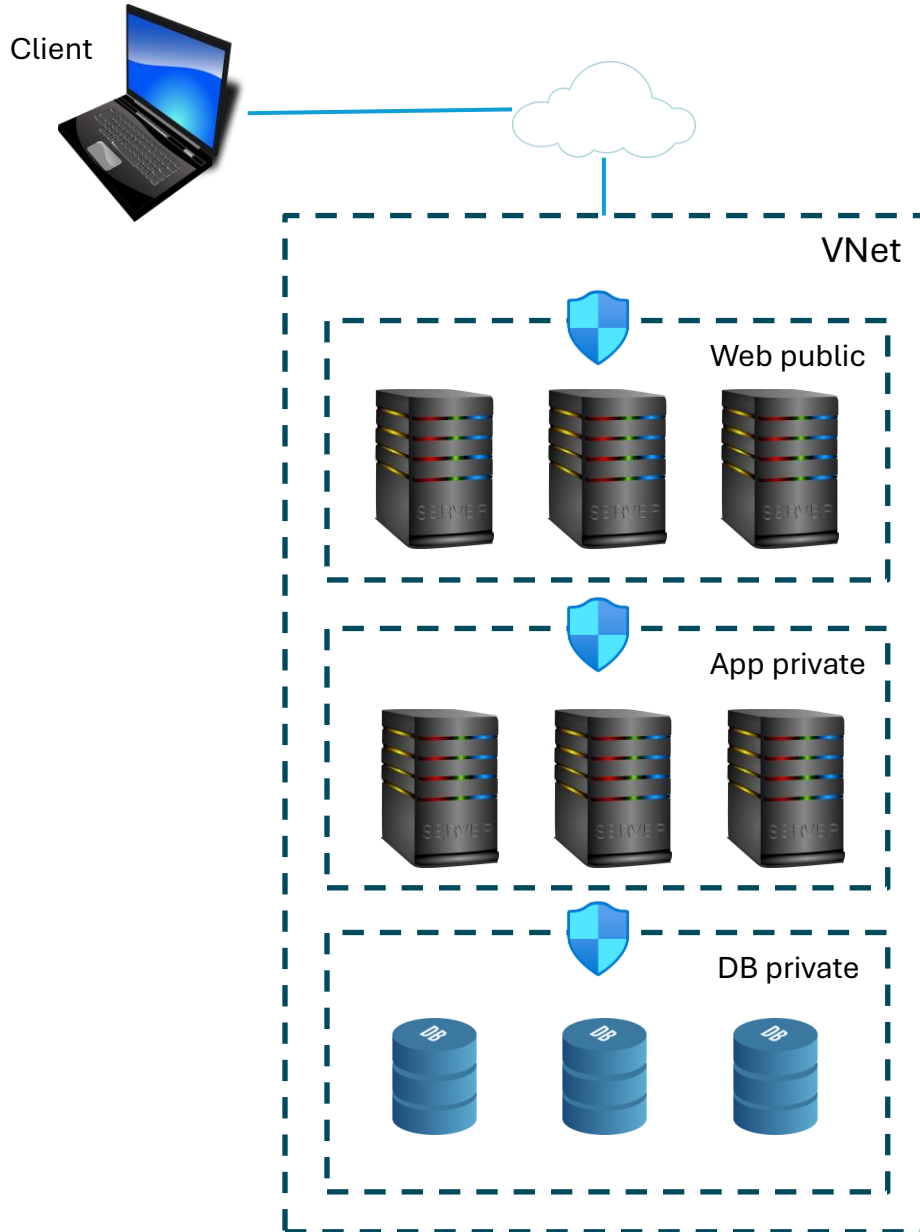


3-tier architecture



- ▶ If each tier runs on a VM, in Azure the default is:
 - ▶ Put all VMs in the same VNet - typically 10.0.0.0/16
 - ▶ Allocate public and private IP addresses to all VMs
 - ▶ Allocate NSG to each VM
- ▶ All are exposed to the internet via IP & network so easy targets
 - ▶ So remove the public IPs of the app server and database
 - ▶ But now we can't access them for OS updates etc
 - ▶ If a hacker gets through a vulnerability in the web server, they can access the app server and DB by guessing private IP addresses, but the "gold" is in the database tier
 - ▶ Each VM has an NSG so could limit access. e.g. App server only allows http traffic from private IP 10.0.0.5 and only open port 3000
 - ▶ That works, but if we scale either tier, we have to deal with IPS

subnet isolation in 3-tier architecture

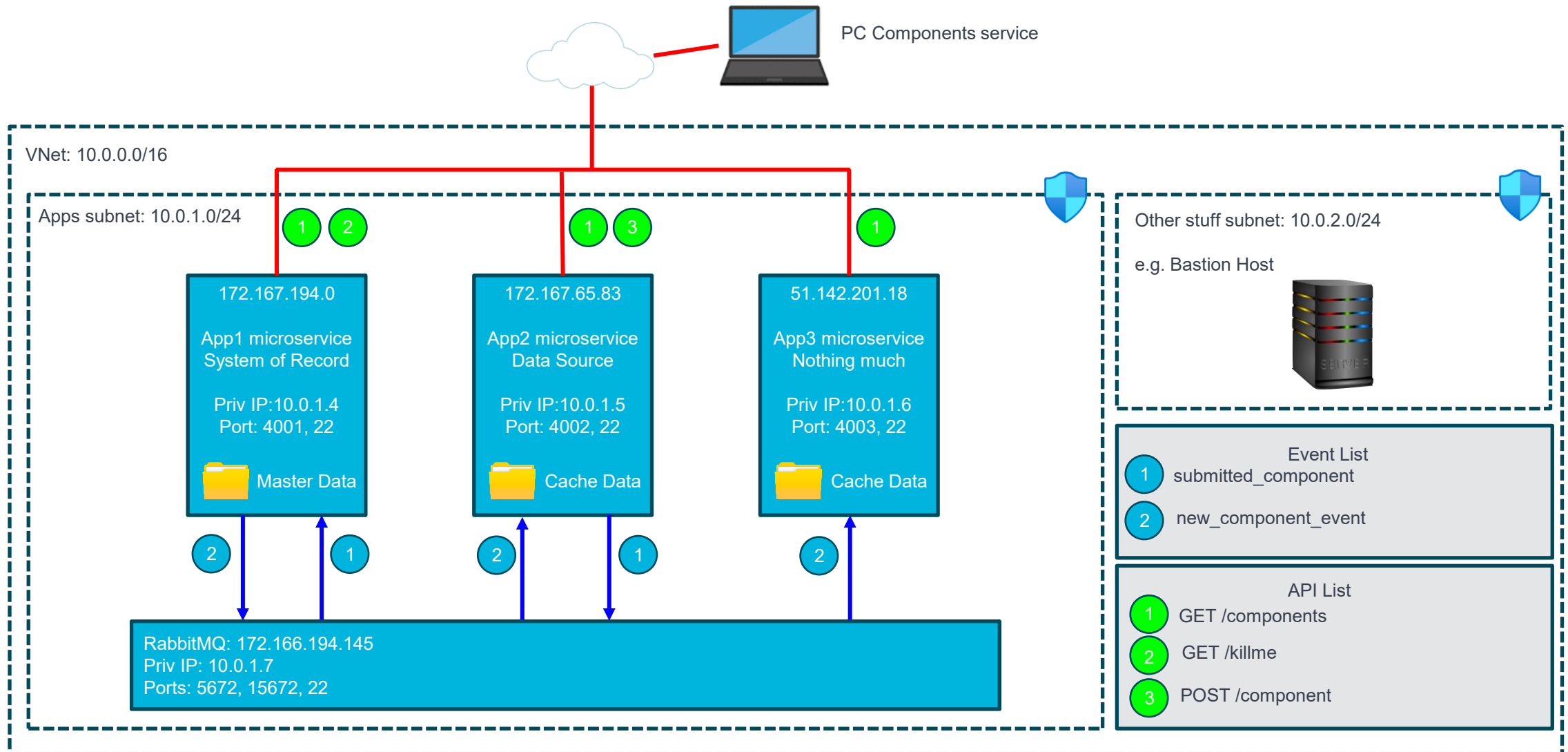


- ▶ We may have many VMs if we scale the solution so many NSG rules need to be set if left at the VM level
- ▶ Use sub-networks and attach an NSG to the sub-net
 - ▶ All rules apply to all resources in the subnet
 - ▶ Set rules to only allow traffic from specific subnet
- ▶ Great. But we still can't access the VMs for admins in the App or DB tier as there are no public IPs
- ▶ And, we don't really want admin access via the web tier even through port 22 as anyone in the world can reach it and have a go at trying to get through
- ▶ Although a microservice is not strictly a 3-tier architecture, it is composed of multiple tiers, API routing, business logic and data so can be protected in the same way using subnets and NSGs

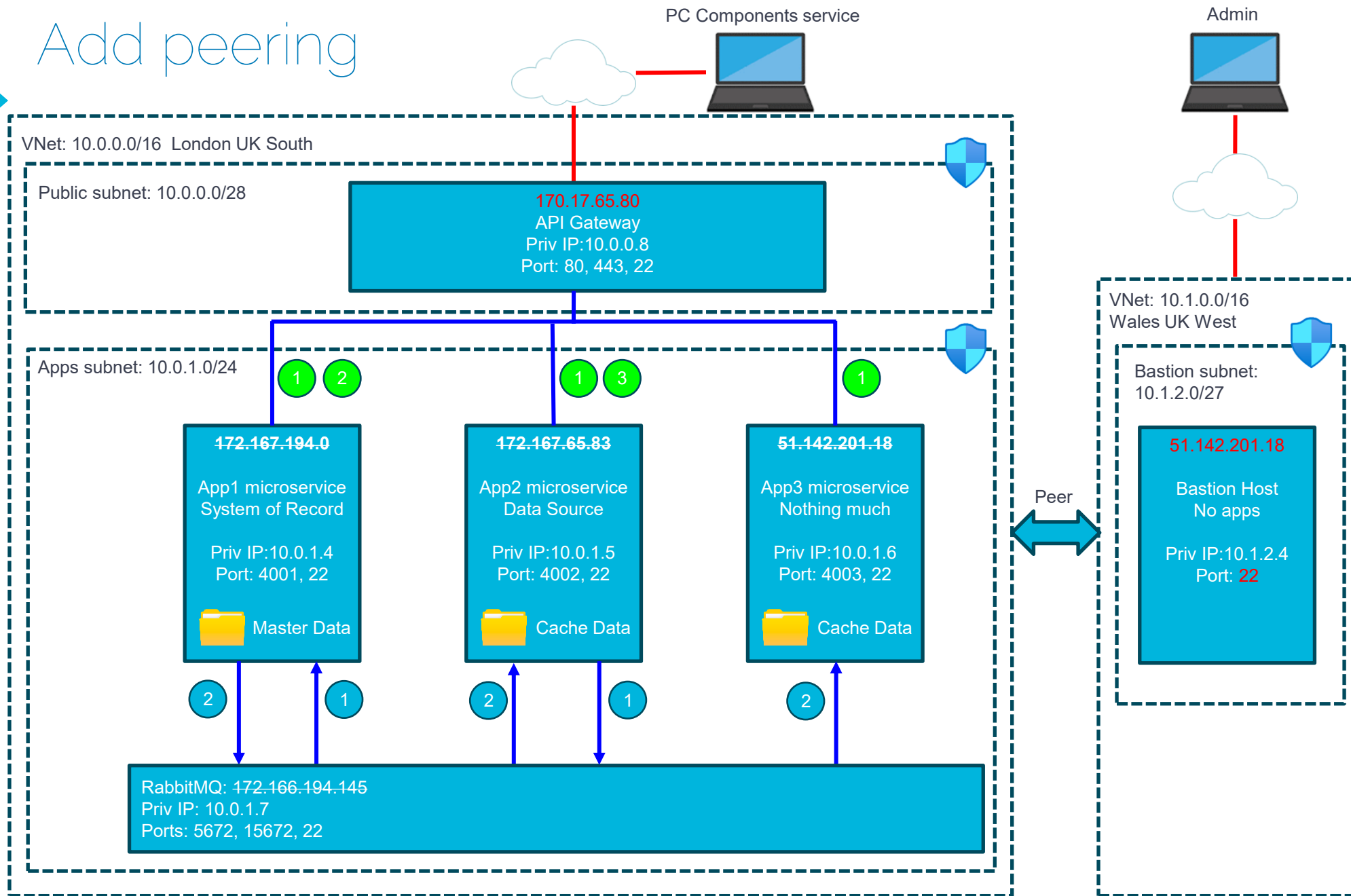
Bastion Host (Jump host or Jump box)

- ▶ We need a solution to securely provide admin access to our VMs
- ▶ A bastion or jump box is used by system admins and maybe engineers to access VMs on a private network
- ▶ Modern fully cloud-native and zero-trust architectures are diverging from traditional bastion hosts, but bastion hosts remain widely used and are still popular in many cloud and hybrid environments and very much for on-prem
- ▶ The Bastion is exposed to the internet as a single entry-point for administered machines
- ▶ It is the only system component accessible to SSH (or RDP) from outside the private network
- ▶ The assumption is that this VM WILL be attacked so it is hardened, monitored and limited functionality
 - ▶ Hardened means, removing non-essential software, including in the OS, closing all ports other than SSH, all non-essential services, no root login via SSH, limited users, password authentication disabled (must use key), admin IP whitelist
 - ▶ Basically, hardening is the deliberate removal of trust, functionality, and exposure until only what is essential remains
- ▶ All VMs not needing internet access (e.g. web server, API Gateway do) do not have public IP addresses
- ▶ Firewall / NSG limit access only from the Bastion host for admin traffic e.g. ssh on port 22
- ▶ Bastion is placed in its own subnet
- ▶ Reduces the attack surface but does provide a "Once in you can access all" so needs protection
 - ▶ Intrusion detection
 - ▶ Suspicious behaviour alerts
 - ▶ Command and user access auditing
 - ▶ Does not store private keys - simply forwards SSH to the target system

Add a Bastion



Add peering



NSG rules

Bastion NSG

Source: public IP (x.x.x.x/32)
Protocol: TCP
Destination port: 22
Action: Allow
Public admin -> bastion
and
Source: 10.1.2.0/27 (**Bastion**)
Protocol: TCP
Destination port: 22
Action: Allow
Bastion -> any allowed port 22

API Gateway NSG

Source: Internet (0.0.0.0/0)
Protocol: TCP
Dest ports: 80, 443
Action: Allow
anywhere -> Kong
and **Bastion**

Apps subnet NSG

Source: 10.0.0.0/28 (API GW)
Protocol: TCP
Dest ports: 4001, 4002, 4003
Action: Allow
Kong -> Microservices
and **Bastion**