



Event driven architecture
RabbitMQ and Kafka

Learning objectives

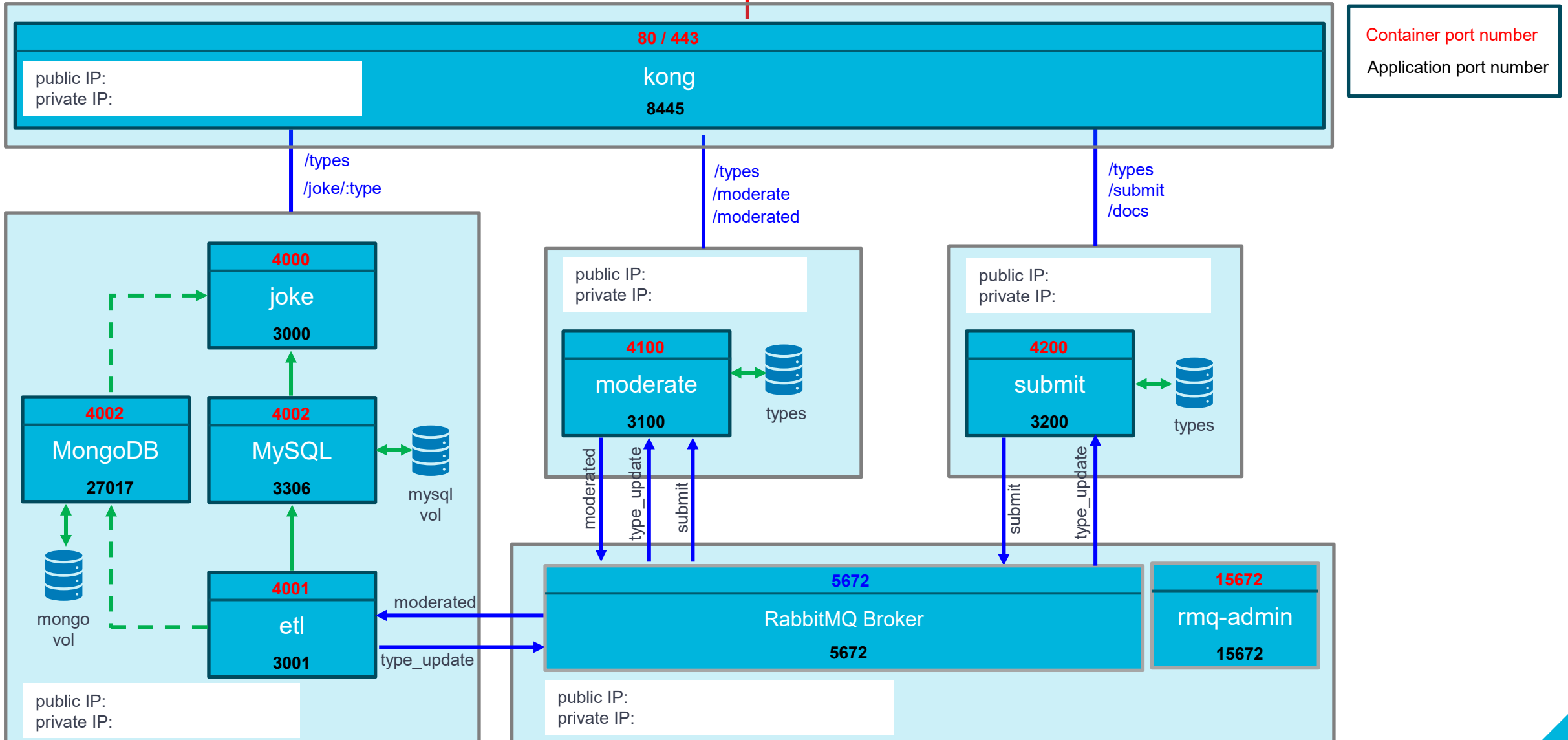
- ▶ Upon successfully completing this section, you should be able to:
 - ▶ Explain the benefits and limitations of various messaging and architectural patterns including:
 - ▶ request-response
 - ▶ point-to-point asynchronous
 - ▶ publisher-subscriber
 - ▶ event notification
 - ▶ event streaming
 - ▶ event carried state transfer (ECST)
 - ▶ Design and draw and explain a basic event-driven architecture
 - ▶ Differentiate between "Thick" and "Thin" events and explain the advantages and disadvantages
 - ▶ Explain the properties and basic operation of Kafka
 - ▶ Compare an event driven architecture to a message driven architecture
 - ▶ Compare the key properties of Kafka to RabbitMQ
 - ▶ Explain how Kafka partitions enable parallel processing

Event-driven



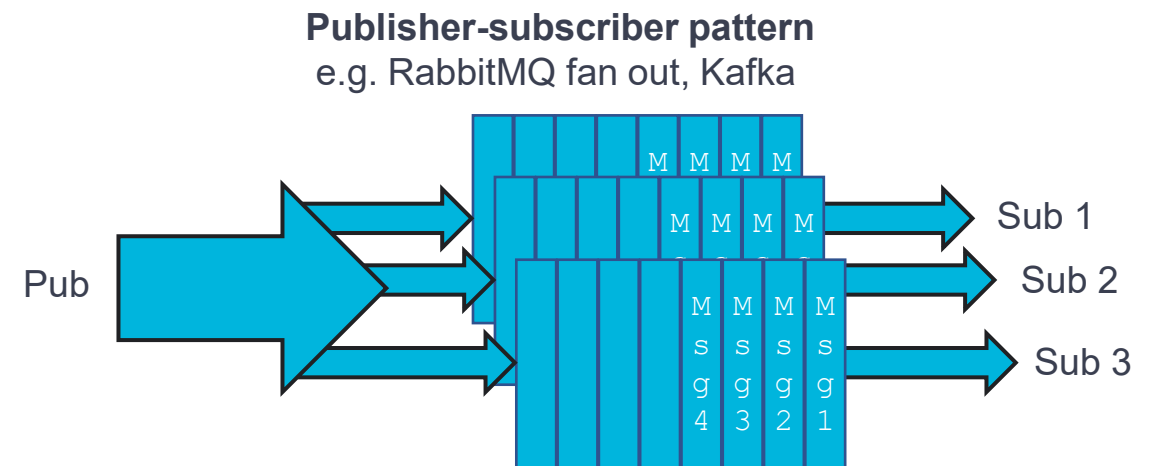
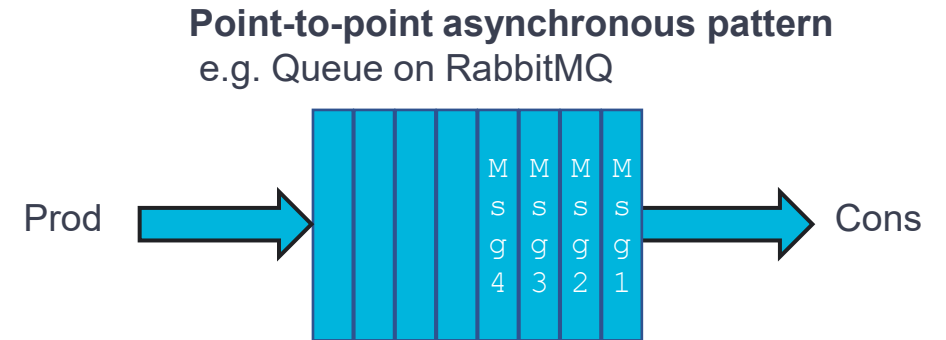
Public internet
Azure private network
Docker private network

Virtual machine
Application container
Rabbitmq combined container



Inter-microservice messaging

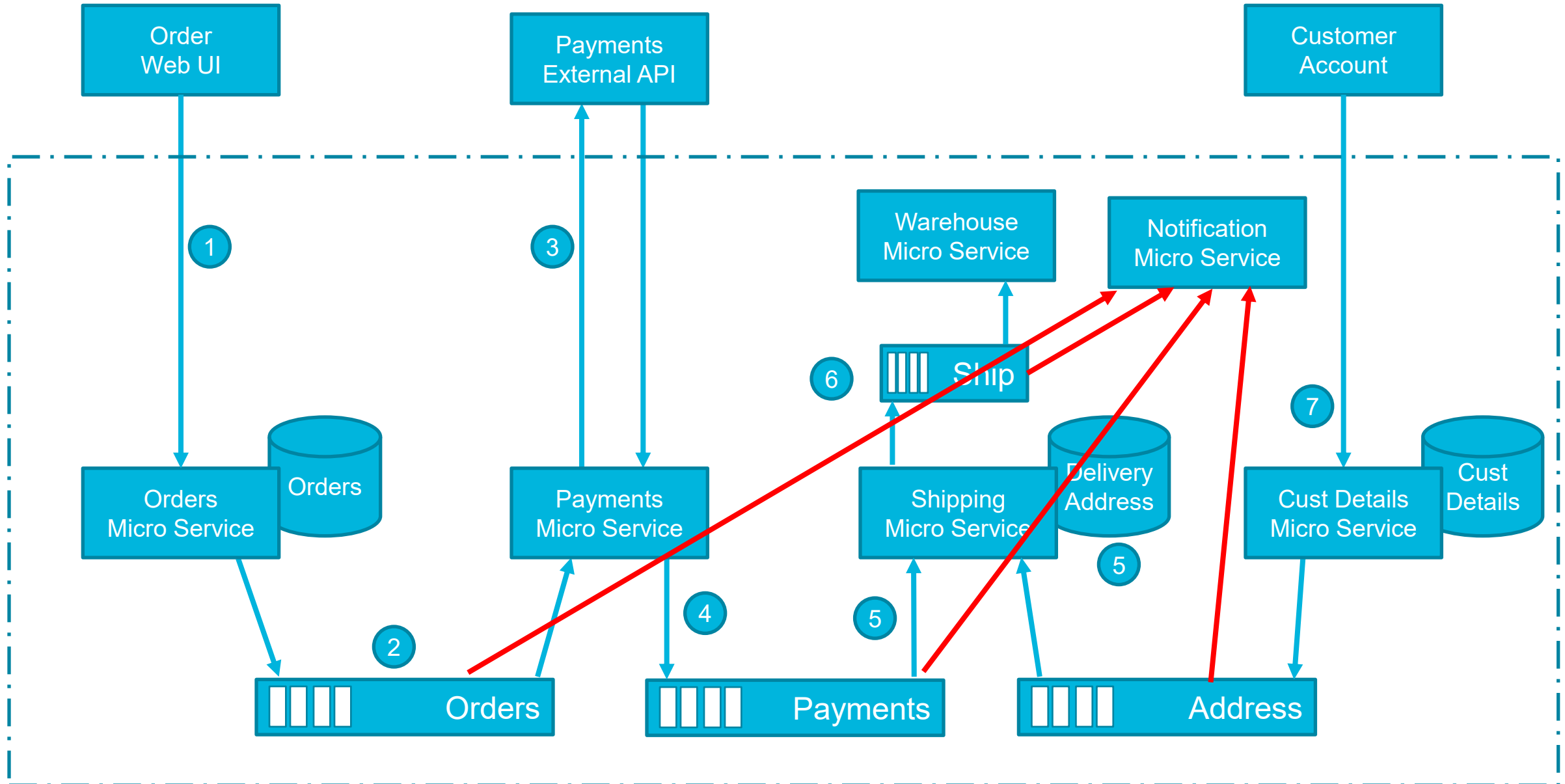
- ▶ As a reminder of what we have already done
- ▶ **Request-response pattern.** http is the simplest messaging pattern, and given there is no intermediary isn't usually even classified as a messaging pattern. More of an interaction pattern. Tight coupling.
- ▶ Point-to-point (p2p) asynchronous message pattern. Typically implemented with a queue. Single producer and single consumer. Loose coupling.
- ▶ Publisher-subscriber pattern. Typically shown as one publisher and many subscribers but also supports many publishers if required. Often implemented using RabbitMQ in a topic configuration if we want subscribers to subscribe to specific topics. The exchange will filter on topic, e.g. PR* in a postcode all go to queue1 for example.
- ▶ Alternatively, RabbitMQ can be configured with a fanout exchange, where every message published to the exchange is delivered to every bound queue, and therefore to every subscriber. Unlike topic exchanges, no routing key or filtering is applied.
- ▶ This reminder is because this is a popular exam topic, and I will use RabbitMQ fanout in pub-sub mode for event processing

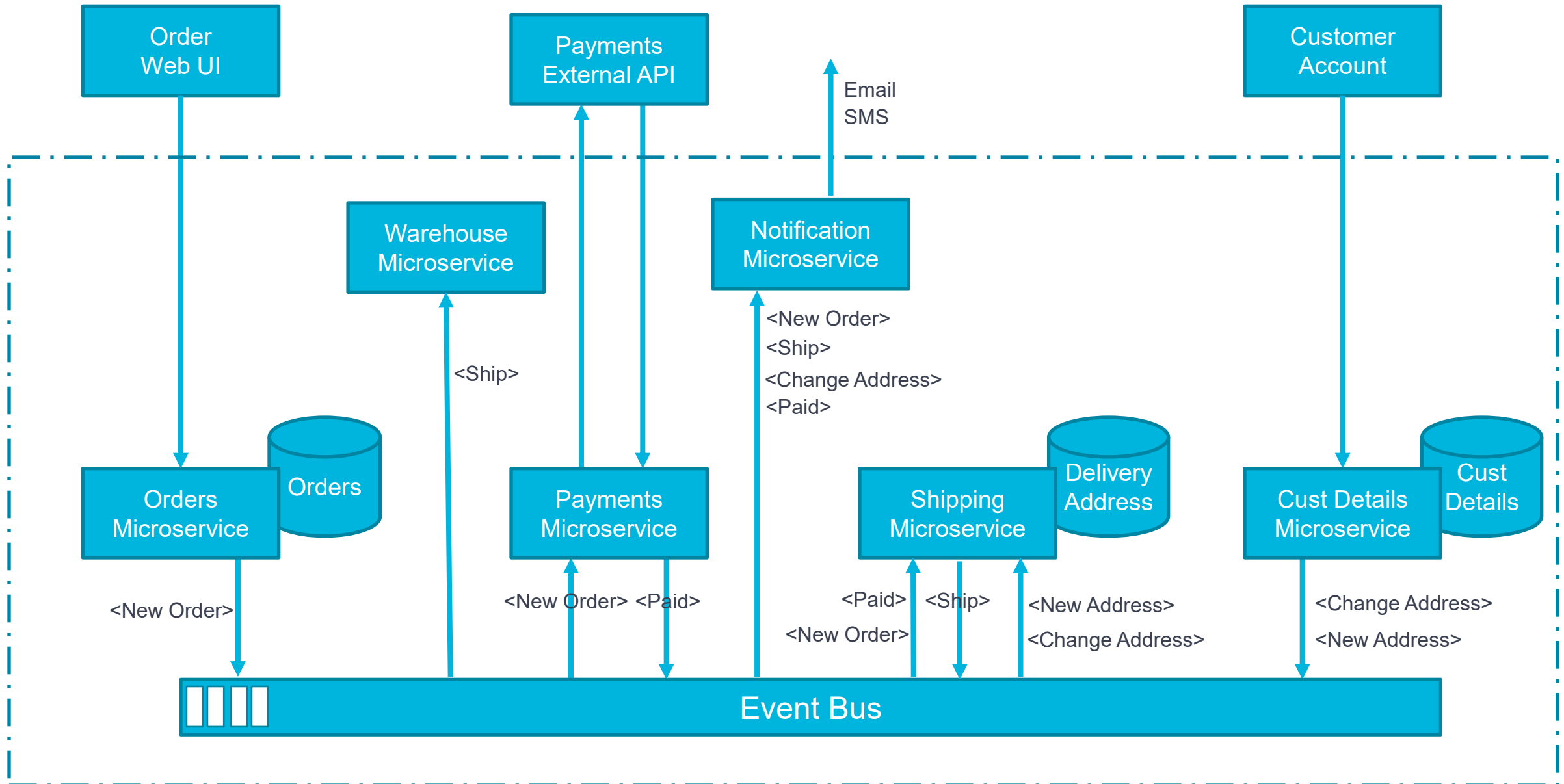


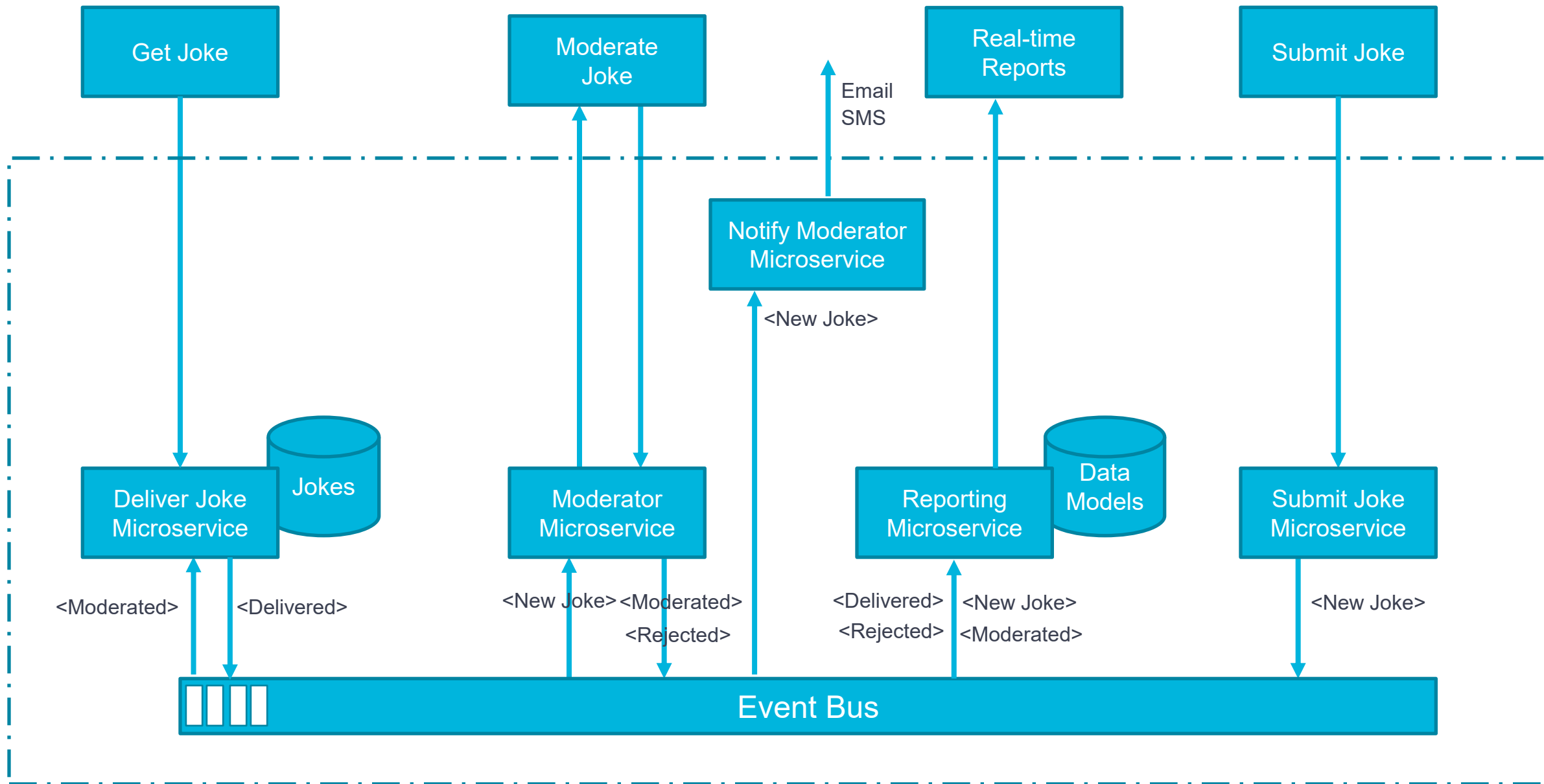
What is an event

- ▶ A significant change in state or occurrence within a system that is captured and communicated as a discrete message to notify other components
 - ▶ Represents state change: e.g., "OrderPlaced", "PaymentProcessed", or "UserLoggedIn"
 - ▶ Immutable: An event is a statement of fact - we don't change it
 - ▶ Context: Data describes the change. e.g., orderID, userDetails, timestamp or whatever
 - ▶ Asynchronous: generally published to an event bus or broker (e.g. Kafka, RabbitMQ)
- ▶ Use case:
 - ▶ In an event-based microservice architecture, messaging tends to act like a state machine. A change in one service may need to be known by others
 - ▶ For example, services tend to have a master data store - often called a System of Record (SoR) - single source of truth
 - ▶ In a distributed system such as microservices, other microservices may need access to that SoR data
 - ▶ We'll look now at a basic event-driven architecture and discuss event-messaging

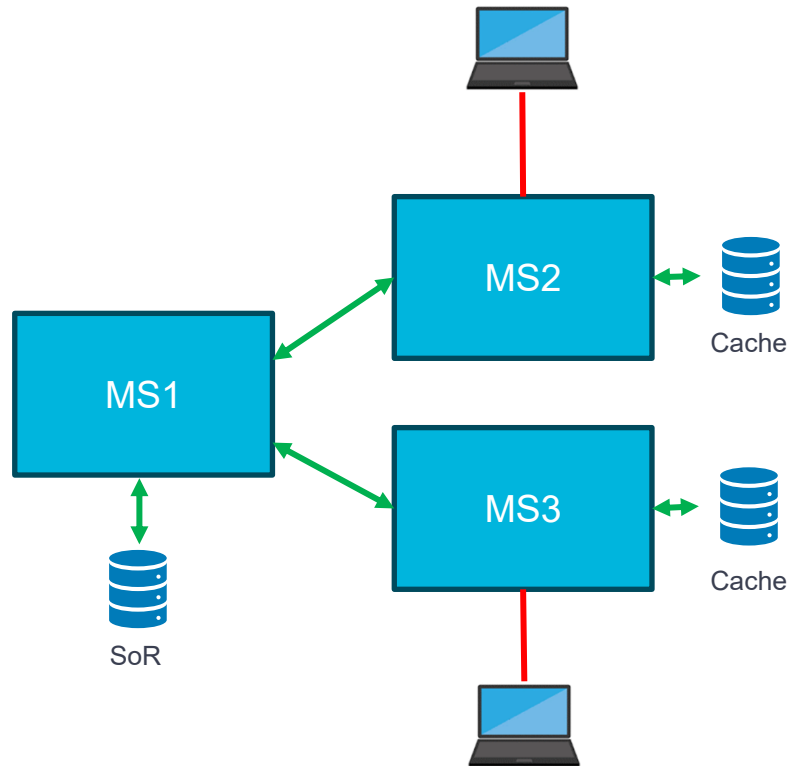
Event driven architecture







Point to point synchronous messaging

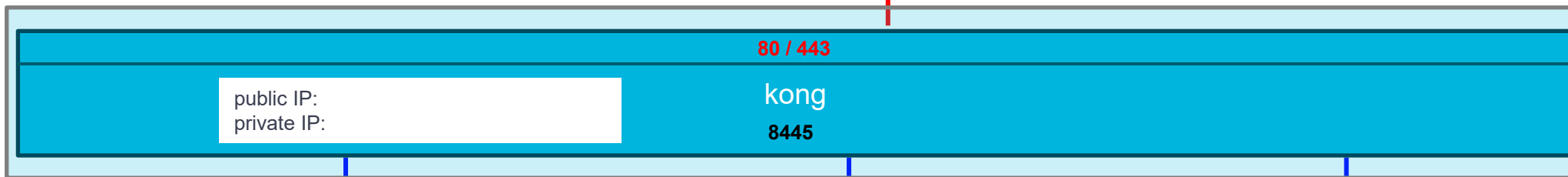


- ▶ User submits data to MS2
- ▶ MS2 writes data to the SoR. If it needs access to the data, it reads from the SoR
- ▶ MS3 writes data to the SoR. If it needs access to the data, it reads from the SoR
- ▶ If a user needs data, MS1 or MS2 need to access the SoR. This is inefficient and tightly coupled - if the SoR fails, the whole system fails
- ▶ MS2 and MS3 could keep a copy of the SoR for resilience and performance but it would need to regularly poll the SoR for changes. e.g. MS3 writes to the SoR, MS2 doesn't know about it until it polls. This is a bit more resilient to SoR failure
- ▶ This is the pattern used in a previous assignment

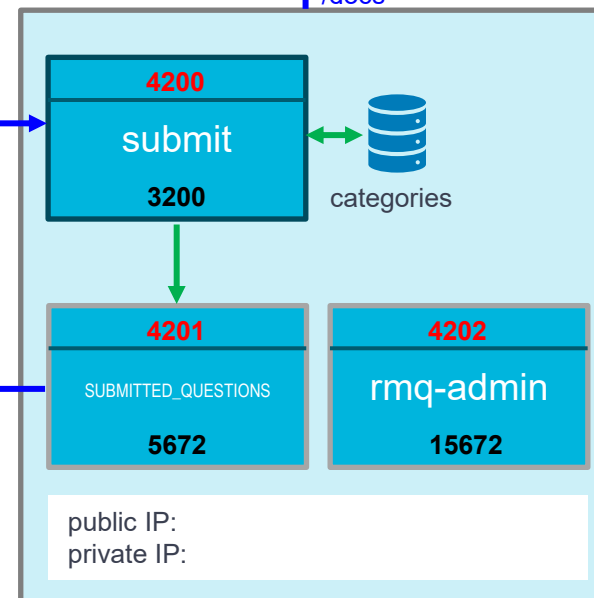
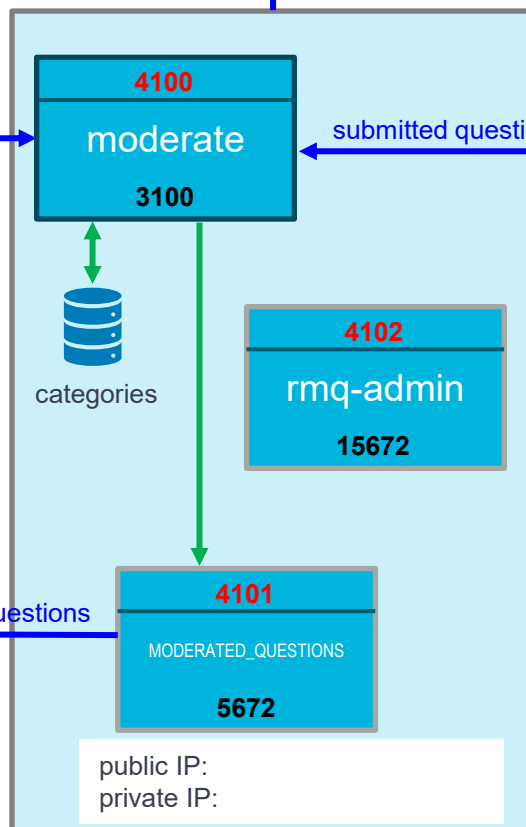
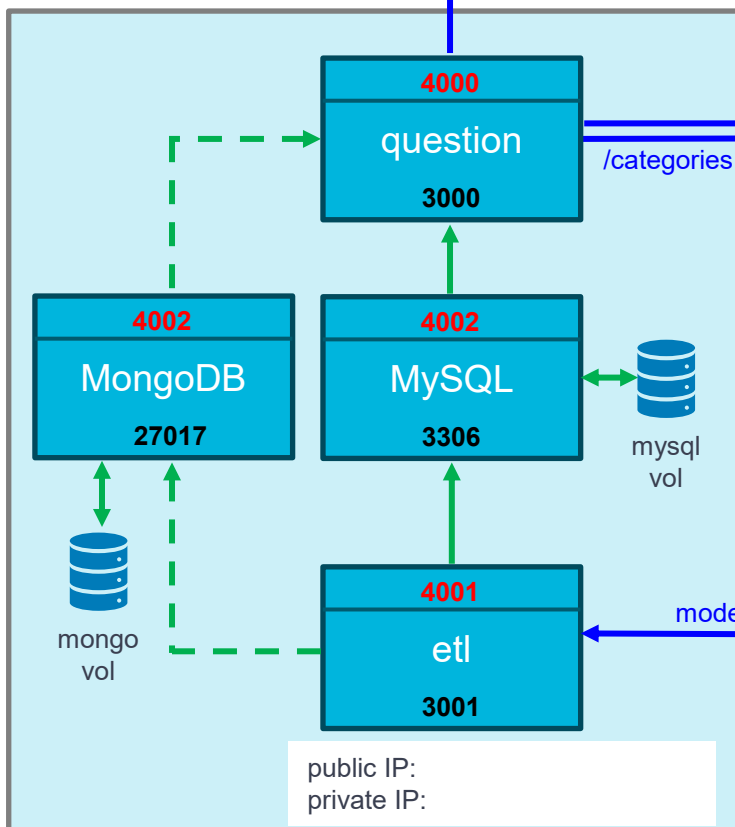


- Public internet
- Azure private network
- Docker private network

- Virtual machine
- Application container
- Rabbitmq combined container



Container port number
Application port number

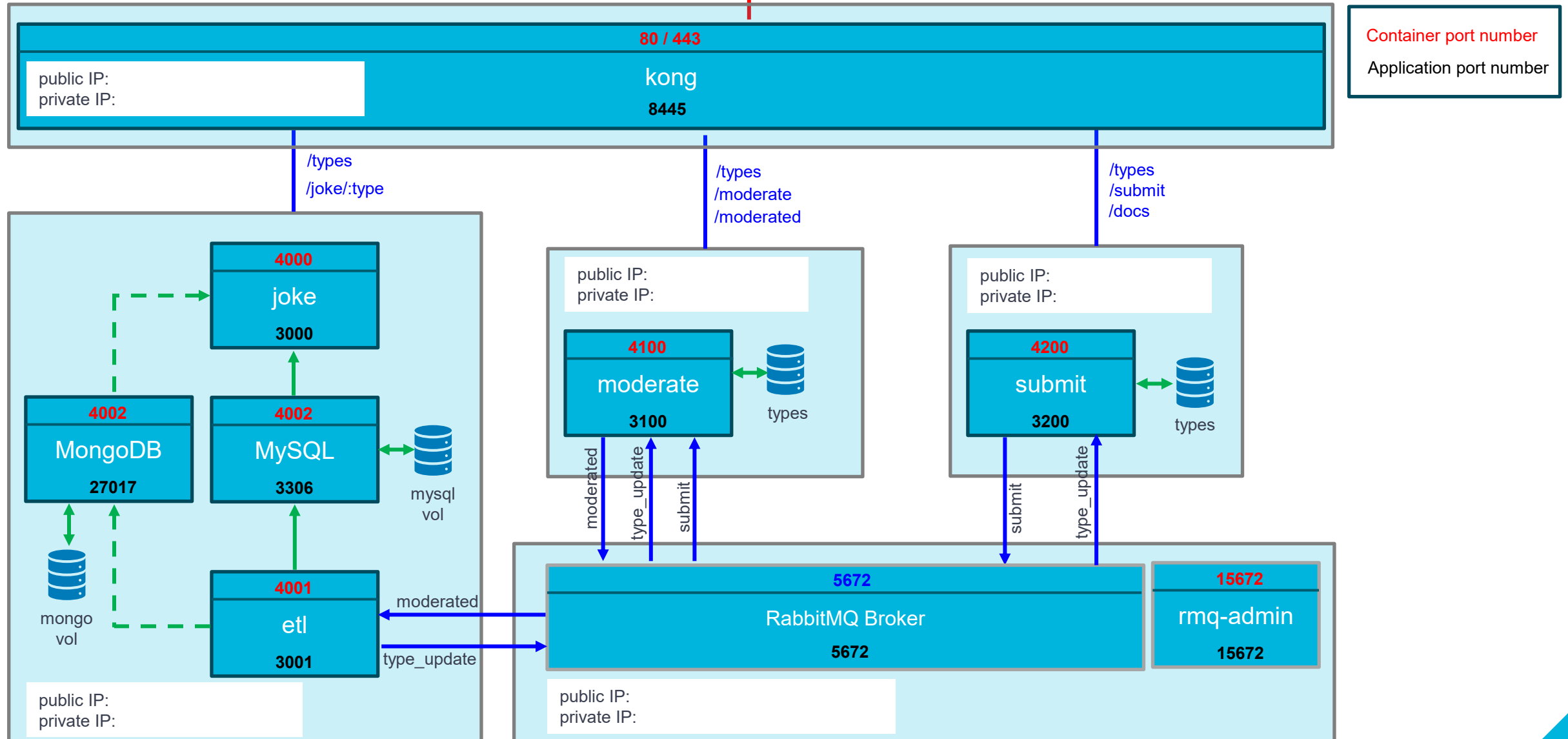


Event-driven



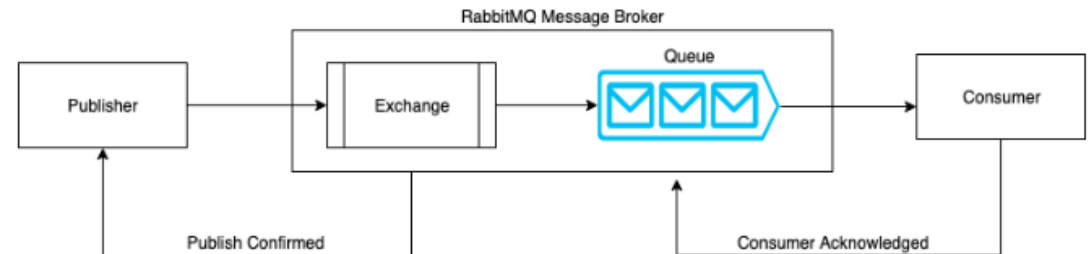
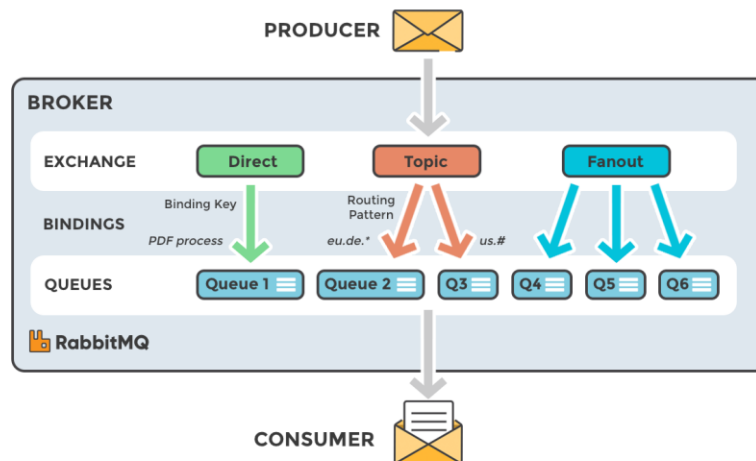
Public internet
Azure private network
Docker private network

Virtual machine
Application container
Rabbitmq combined container



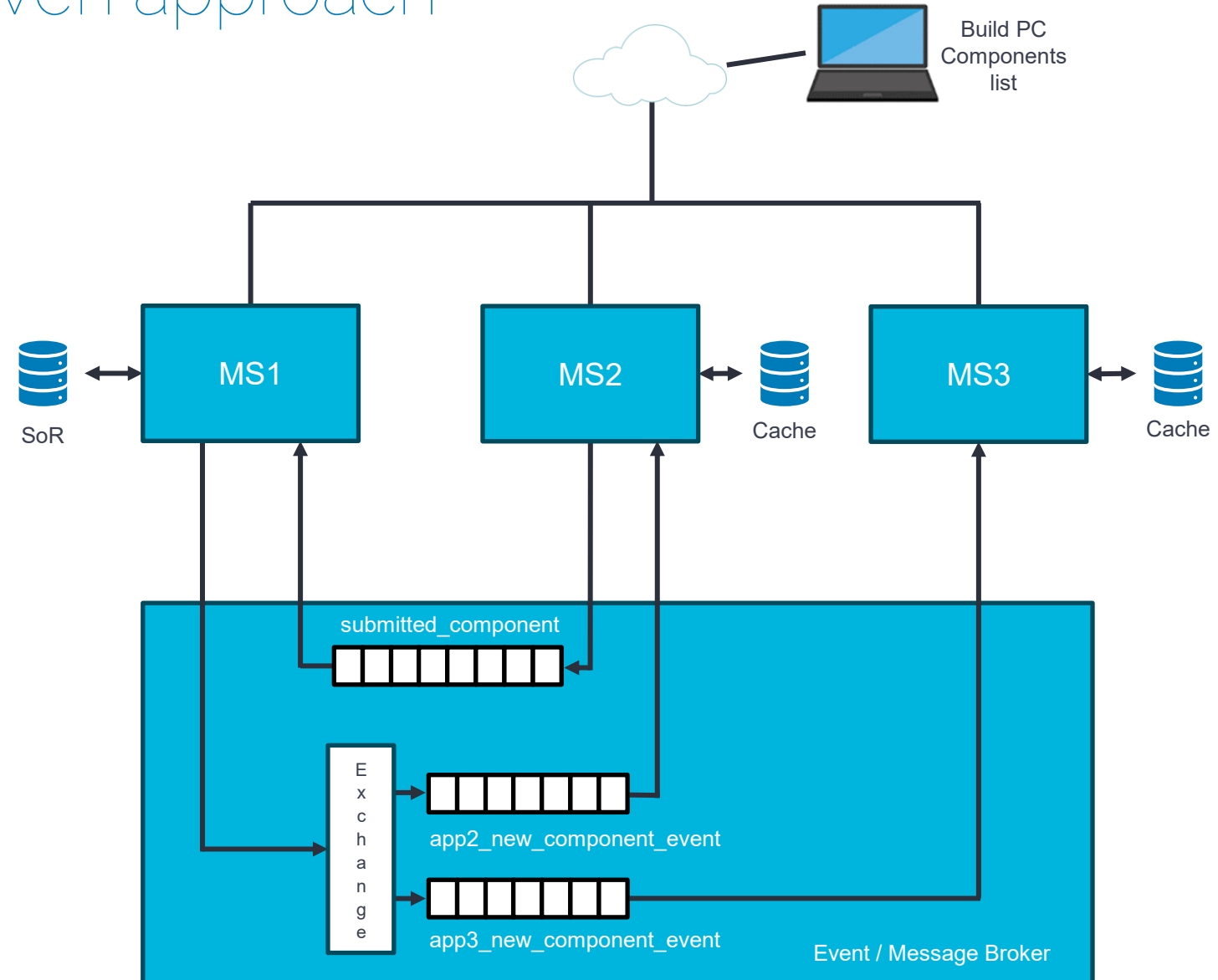
RabbitMQ message broker recap

- ▶ Let's create an event driven architecture using RabbitMQ
- ▶ A broker is an intermediary between the message producer and message consumer - like post-office
 - ▶ Asynchronous pattern - producer doesn't wait for the consumer to acknowledge. Contract is with the broker
 - ▶ Loose coupling between producer and consumer
 - ▶ Consumer can consume from the queue in its own time - queuing smooths spikes to prevent consumer overload
 - ▶ Supports single queue one producer to one consumer
 - ▶ Supports publisher / subscriber message distribution with multiple subscribers to the same messages
 - ▶ Generally pushes data from the broker to the consumer
 - ▶ Generally message deleted from the queue when the consumer acknowledges receipt of the message
 - ▶ RabbitMQ is resilient as can run a cluster of brokers so caters for failure of one or more nodes

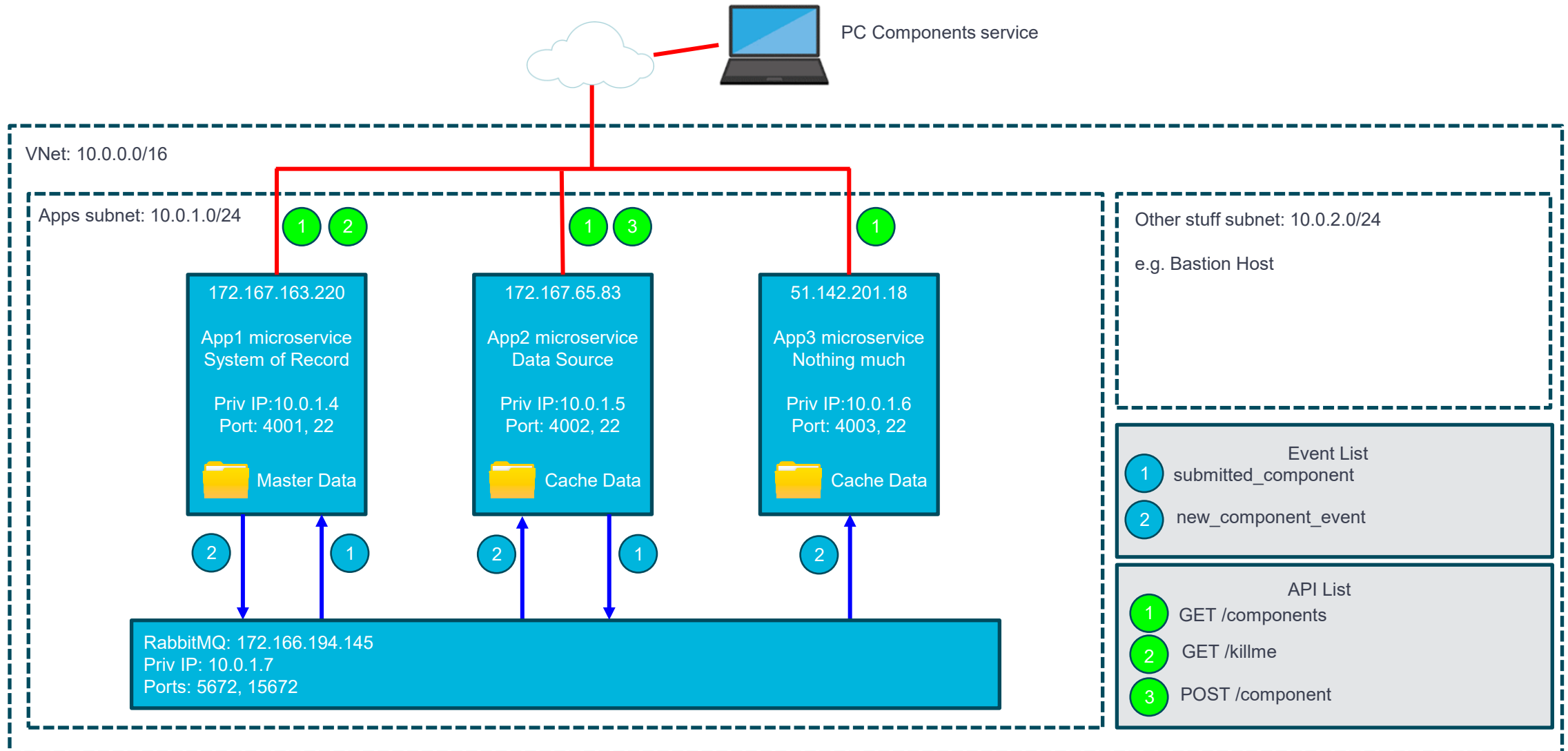


Asynchronous Event-driven approach

- ▶ Simple use-case: add components to a list with no duplicates. All services informed of new entries
- ▶ User sends data to MS2
- ▶ MS2 updates the SoR (MS1) via a queue
- ▶ Message pattern is Asynchronous point-to-point
- ▶ MS1 decides if it should update the SoR and if so, creates an update event with the data
- ▶ Event is published (to an exchange in RabbitMQ)
- ▶ MS2 and 3 subscribe to their queue
- ▶ Queues push the event to each of MS2 and MS3
- ▶ They update their local cache with event data
- ▶ The data reaches eventual consistency
- ▶ This arch pattern is called "**Event-Carried State Transfer**" (ECST) using pub-sub messaging
- ▶ Users access data from cache so quicker
- ▶ System is resilient if SoR or any microservice fails
- ▶ Data transfer loosely coupled in both directions
- ▶ An alternative to RabbitMQ is Kafka
- ▶ Clearly more complex than point to point but more resilient to failure of one or more microservices



Detailed architectural diagram



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. We can easily detect a change in the database and emit an event, but the system is more likely to provide data via an API
 - ▶ producer remains the sole copy of the truth
- ▶ 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

- ▶ Thick or fat events, carry all the data (state). Used in "Event-Carried State Transfer" architectural pattern to update subscribers to the event and its state
- ▶ Thick event benefits include
 - ▶ Potentially lower latency as no additional API call
 - ▶ Asynchronous so provides loose coupling and hence more resilience to producer failure
 - ▶ Producer does not need to scale as more consumers are added
- ▶ Challenges
 - Likely not a practical solution when integrating to legacy without lots of nugatory legacy improvement
 - All data is always sent even if not all subscribers want all the data so network and broker load can be increased
 - Load on the API producer is decreased but now moved to the broker instead so that needs to be considered

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