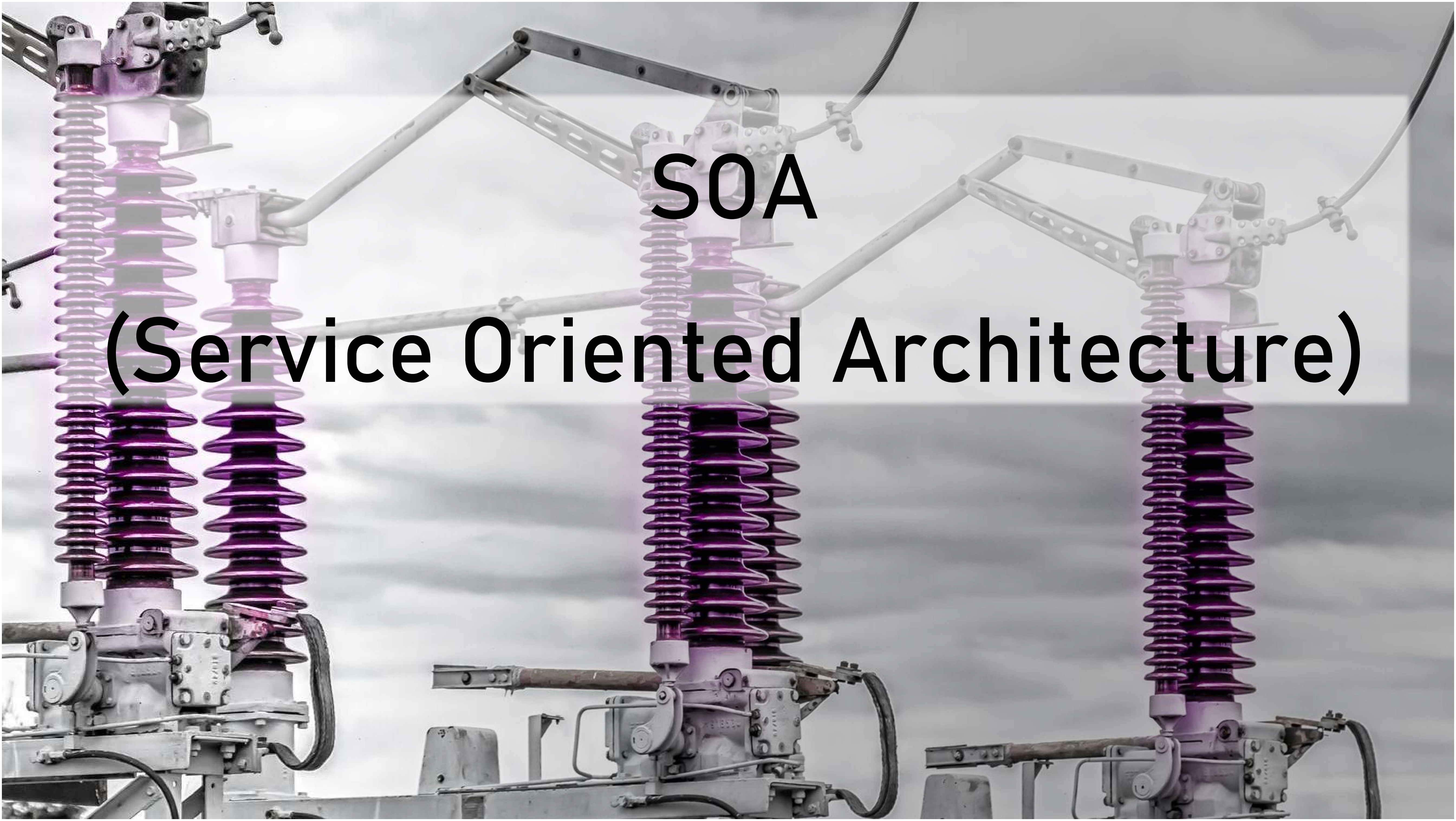
Events

- The cornerstone of event driven architecture
- Require a well-defined definition
- Evolved from other architectures

Microservices Architecture

- Based on loosely-coupled services
- Each service in its own process
- Lightweight communication protocols
- Polyglot
    - No platform dependency between services
- Replaces two legacy architectures

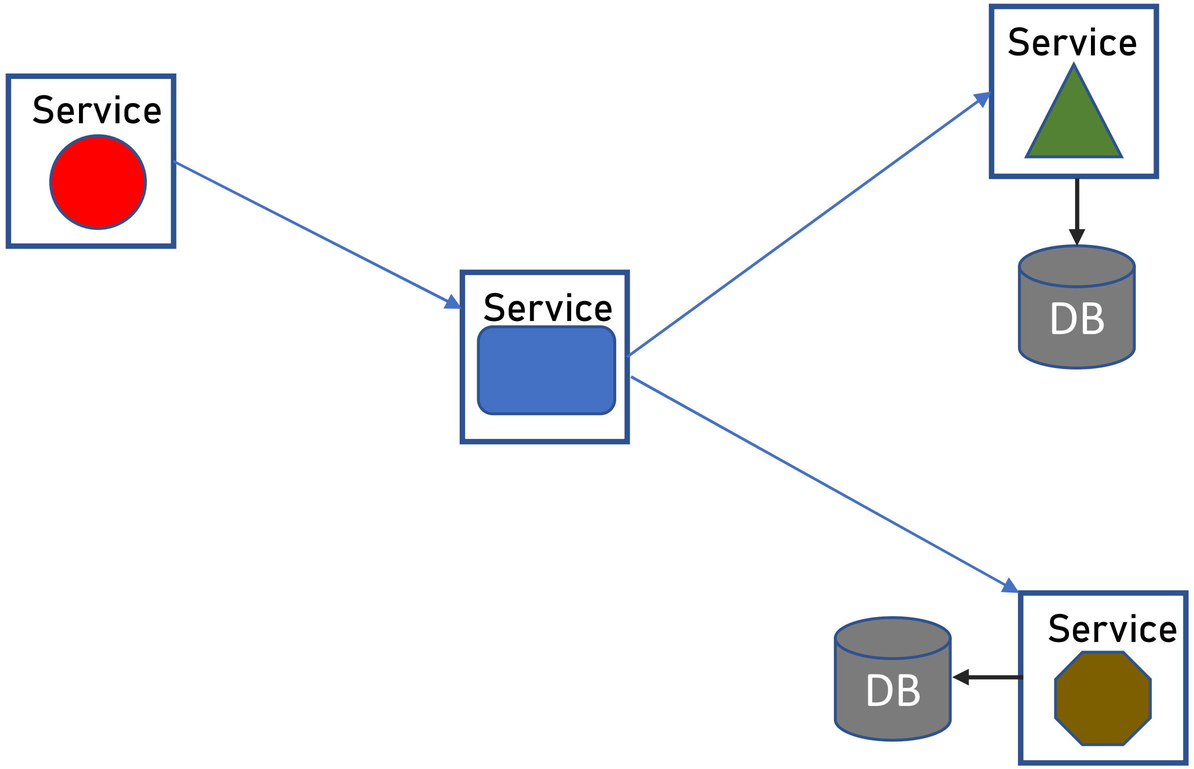
Monolith



SOA

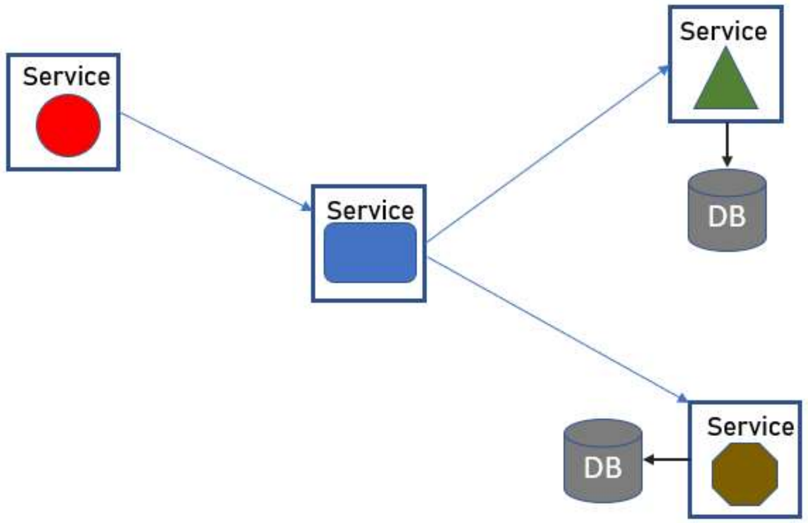
(Service Oriented Architecture)

Typical Microservices System



Microservices Communication

- Perhaps the most important part in microservices architecture
- Dictates performance, scalability, implementation and more
- Event Driven Architecture handles the communication part

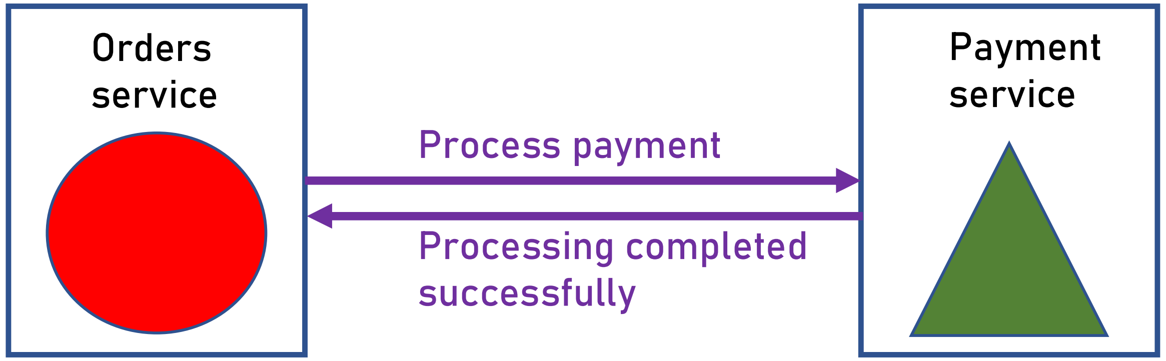


Command and Query

- The classic communication between services
- Services either:
    - Send command
    - Query for data

Command

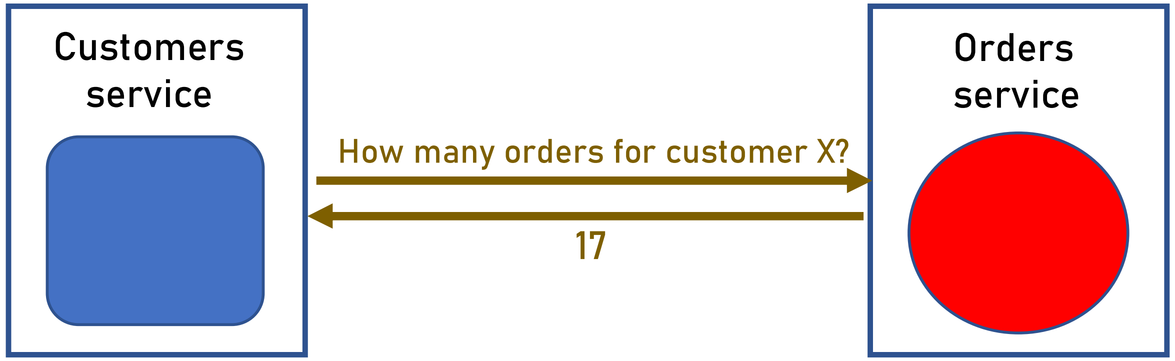
- Service asks another service to do something



- There might be a response to the command, usually a success or failure indicator

Query

- Service asks another service for data



- There's always a response to the query, containing the data

Command and Query

- Main characteristics:

Command

- Do something
- Usually synchronous
- Sometimes returns a response
- Calling service needs to know who handles the command

Query

- Retrieve data
- Almost always synchronous
- Always returns a response
- Calling service needs to know who handles the query

Problems with Command and Query

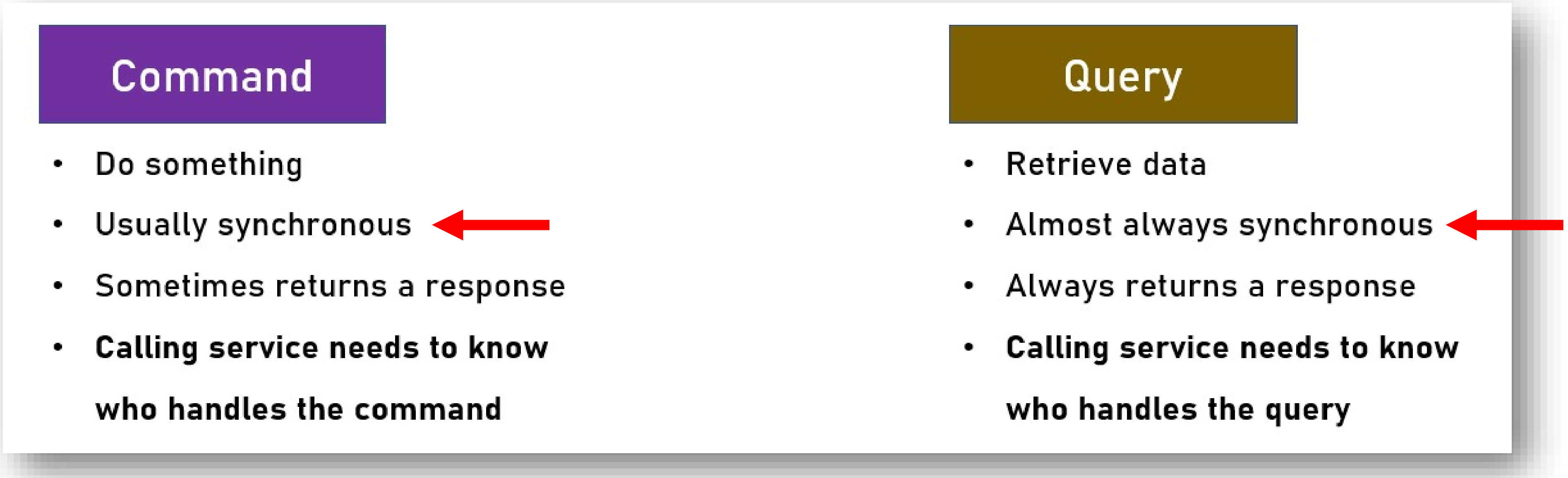
- Three major problems with command and query:

Performance

Coupling

Scalability

Performance



Command

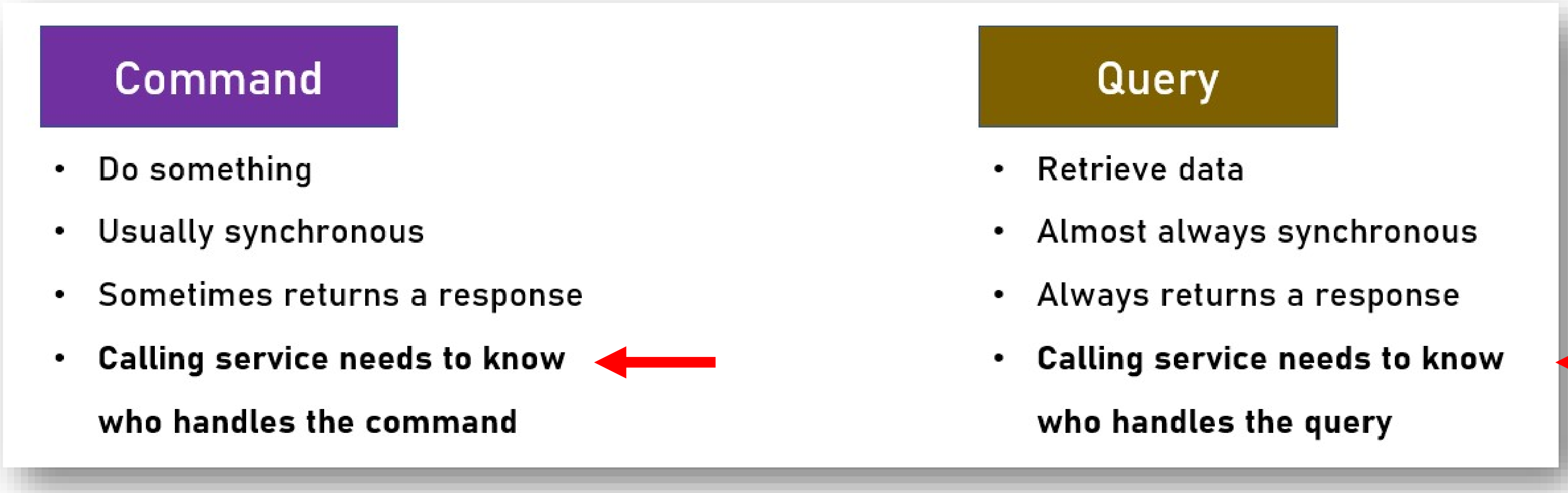
- Do something
- Usually synchronous ←
- Sometimes returns a response
- **Calling service needs to know who handles the command**

Query

- Retrieve data
- Almost always synchronous ←
- Always returns a response
- **Calling service needs to know who handles the query**

- Synchronous = the calling service waits for the command / query to complete
- Potential for performance hit

Coupling



Command

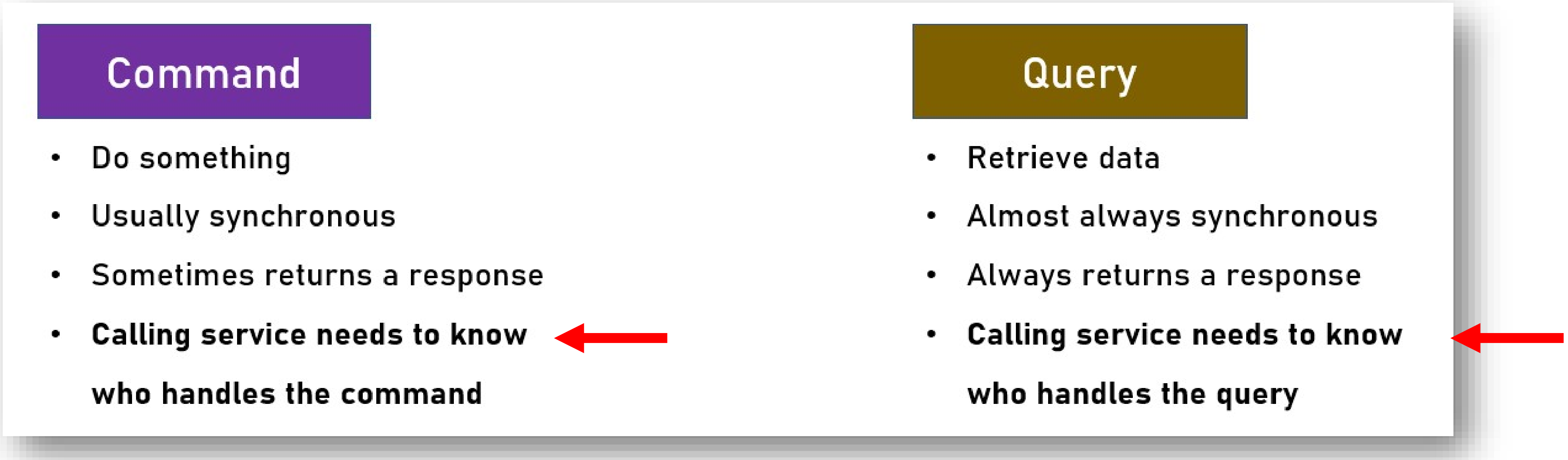
- Do something
- Usually synchronous
- Sometimes returns a response
- **Calling service needs to know who handles the command**

Query

- Retrieve data
- Almost always synchronous
- Always returns a response
- **Calling service needs to know who handles the query**

- The calling service calls a specific service
- If the called service changes – the calling service has to change too
- More work, more maintenance

Scalability



Command

- Do something
- Usually synchronous
- Sometimes returns a response
- **Calling service needs to know who handles the command** ←

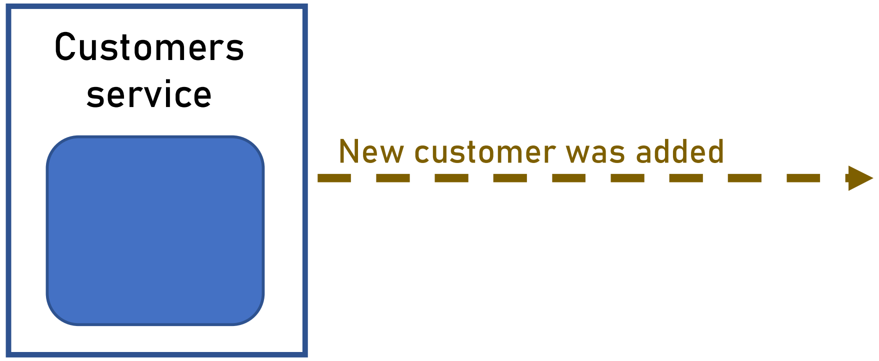
Query

- Retrieve data
- Almost always synchronous
- Always returns a response
- **Calling service needs to know who handles the query** ←

- The calling service calls a single instance of a service
- If this instance is busy – there's a performance hit
- Adding another instances is possible, but difficult
  - Add load balancer, configure probes etc.

Event

- Indicates that something happened in the system



- There's never a response to the event

Event

- Main characteristics:

Event

- Something happened
- Asynchronous
- Never returns a response
- Calling service has no idea who handles the event

Contents of Event

- Two types of event data:

Complete

- Contains all the relevant data
- Usually entity data
- No additional data is required for the event processing
- Example:

```
event_type: CustomerCreated
customer_id: 17
first_name: David
last_name: Jones
join_date: 2022-03-15
```

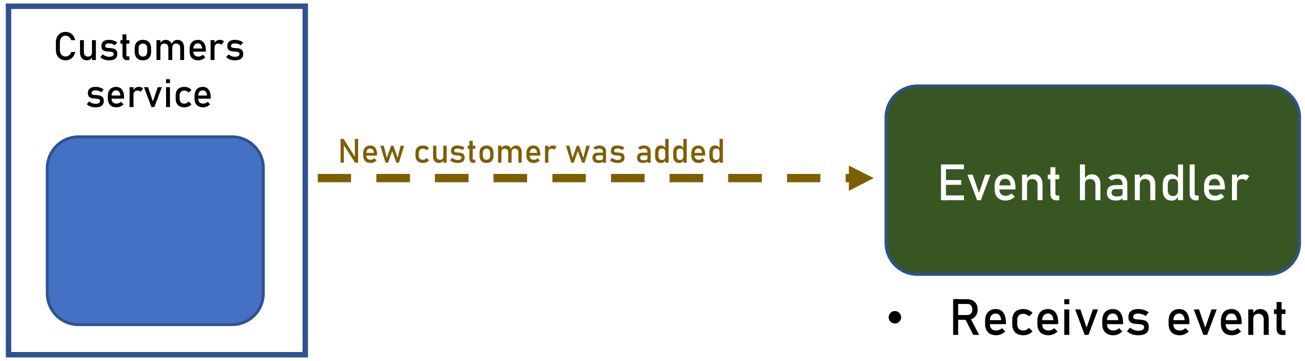
Pointer

- Contains pointer to the complete data of the entity
- Complete data usually stored in a database
- Event handler needs to access the database to retrieve complete data
- Example:

```
event_type: CustomerCreated
customer_id: 17
```

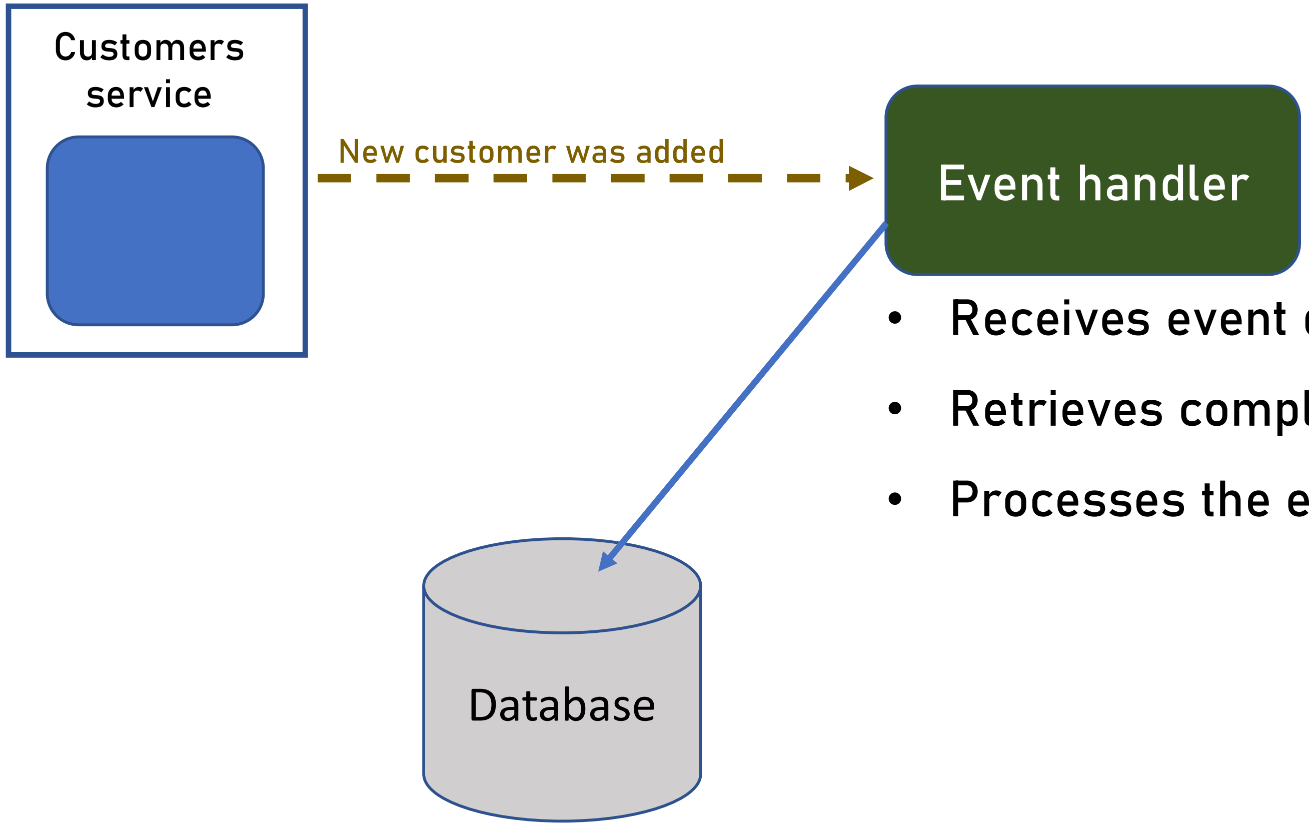
Pointer

Flow of Complete Event Handling



- Receives event data
- Processes the event

Flow of Pointer Event Handling



- Receives event data
- Retrieves complete data from database
- Processes the event

Complete vs Pointer

- When to use which?

Complete

- The better approach
- Makes the event completely autonomous
- Can get out of the system boundaries

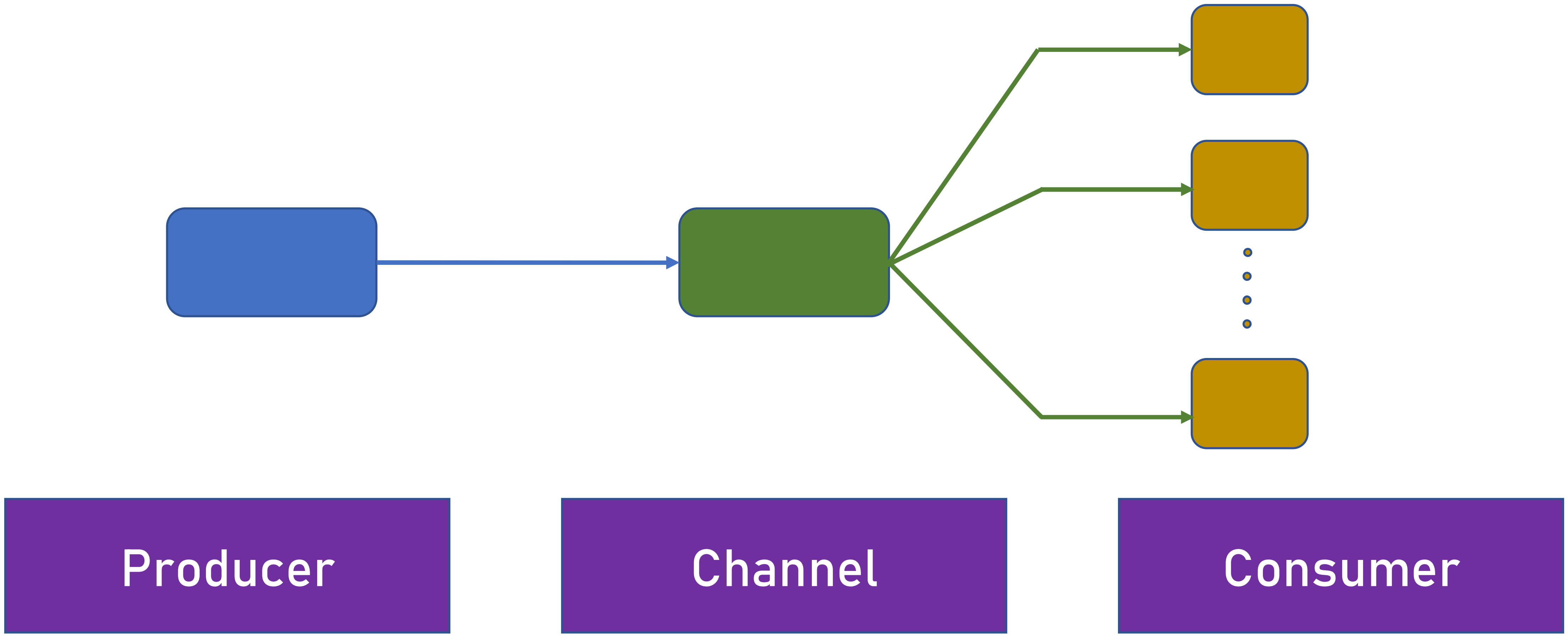
Pointer

- Use when:
  - Data is large
  - Need to ensure data is up-to-date
    - Assuming database is a single-source-of-truth

Event Driven Architecture

- A software architecture paradigm that uses events as the mean of communication between services
- Often called EDA
- Has three main components

Event Driven Architecture Components

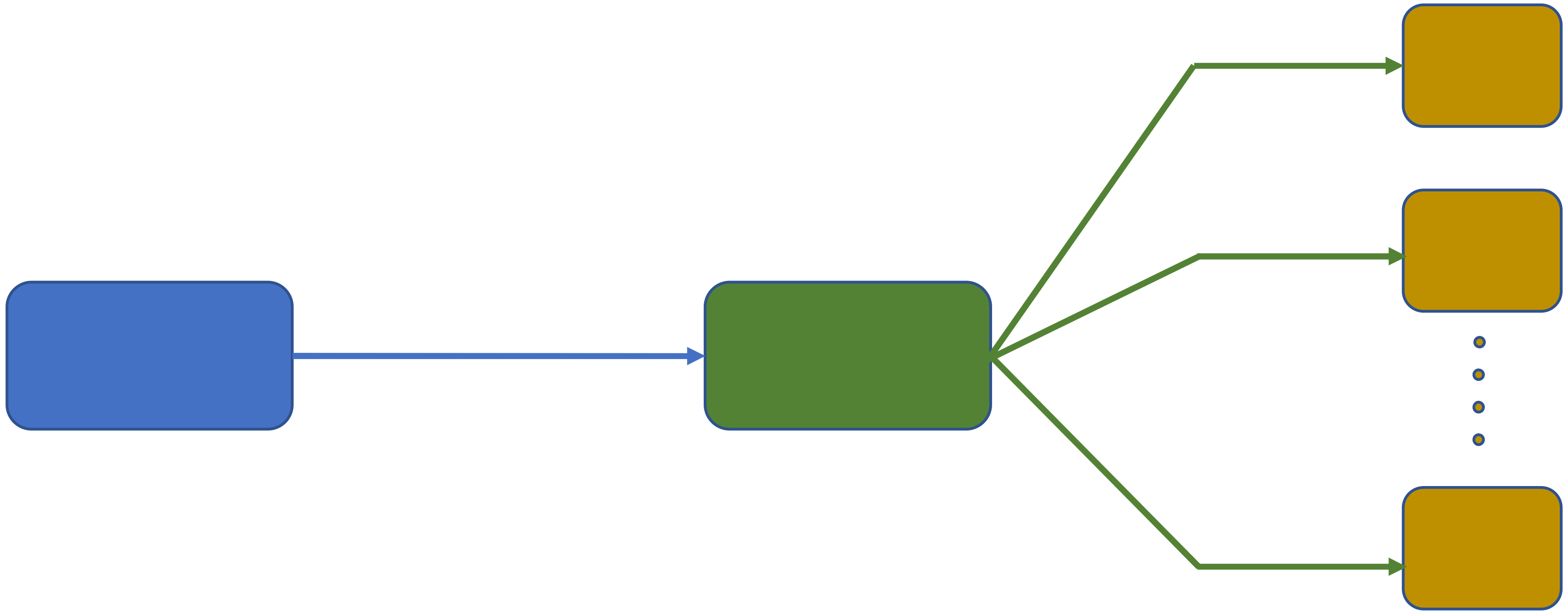


Producer

Channel

Consumer

Producer



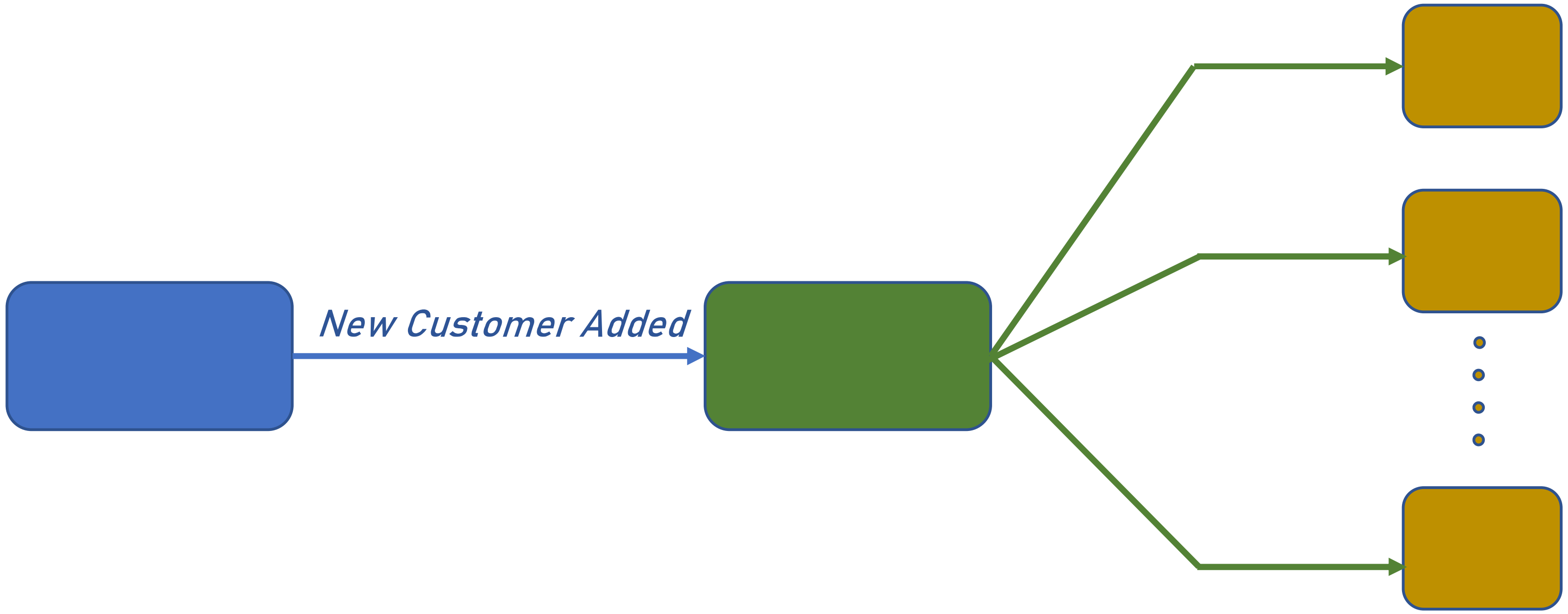
Producer

Producer

- The component / service sending the event
- Often called Publisher
- Usually sends event reporting something the component done
- Examples:
    - Customer service -> *New Customer Added* event
    - Inventory service -> *Item Sold Out* event

Producer

- The producer sends the event to the Channel



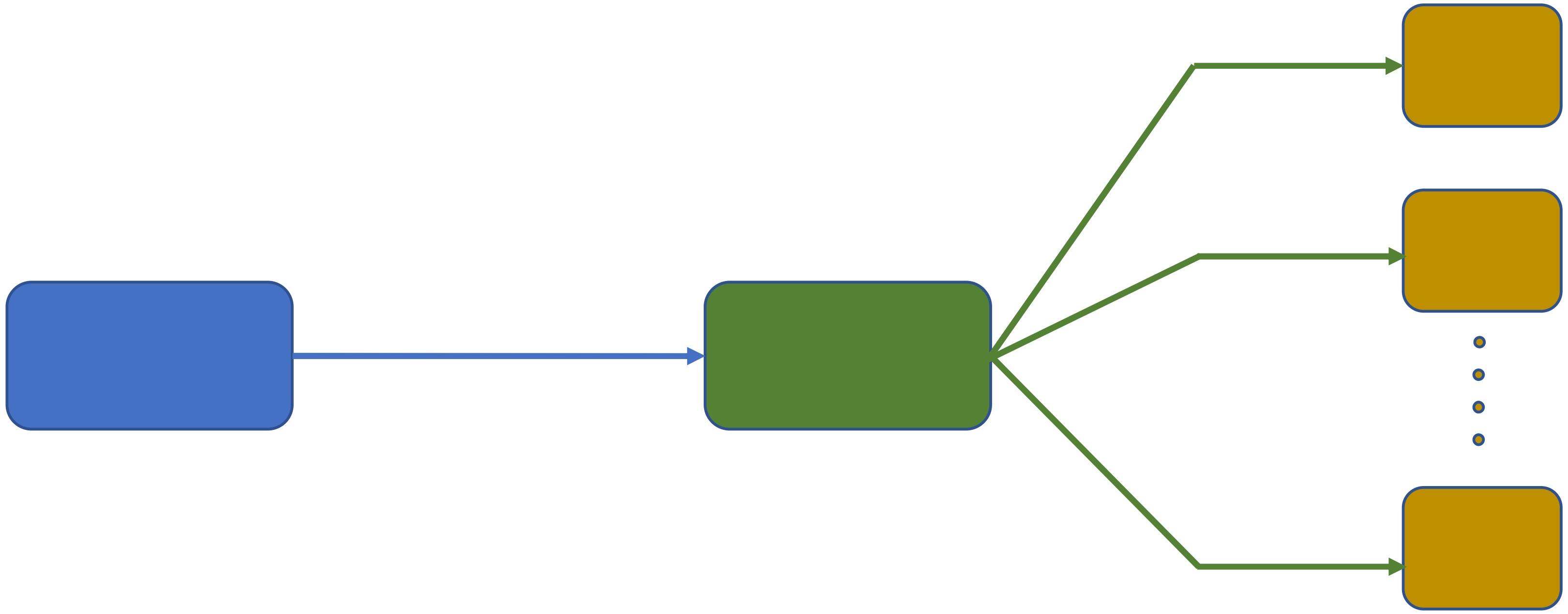
Producer

Channel

Producer

- Exact method of calling the channel depends on the channel
- Usually using a dedicated SDK developed by the channel vendor
- Utilizes some kind of network call, usually with specialized ports and proprietary protocol
- I.e.: RabbitMQ listens on port 5672 and uses the AMQP protocol
- Producer can be developed using any development language

Channel



Channel

Channel

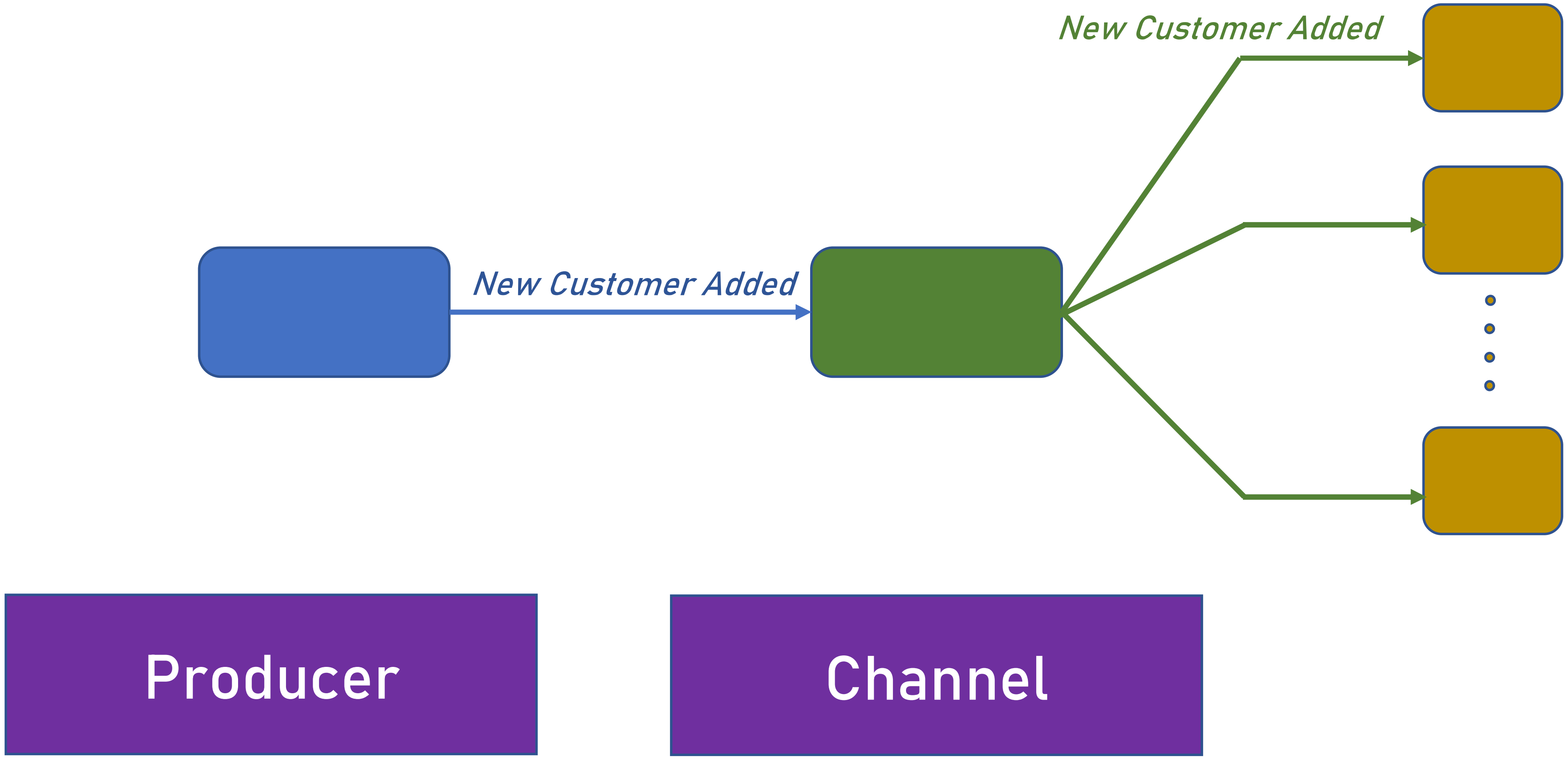
- The most important component in the Event Driven Architecture
- Responsible for distributing the events to the relevant parties
- The channel places the event in a specialized queue, often called Topic or Fanout
- Consumers listen to this queue and grab the event

Channel

- Note:
    - Implementation details vary wildly between channels
    - RabbitMQ works differently than Kafka that works differently than WebHooks etc
    - Always dive deep into the docs of the channel you're using
    - We'll use RabbitMQ and SignalR in the implementation section

Channel

- The Channel distributes the event to the Consumers



New Customer Added

New Customer Added

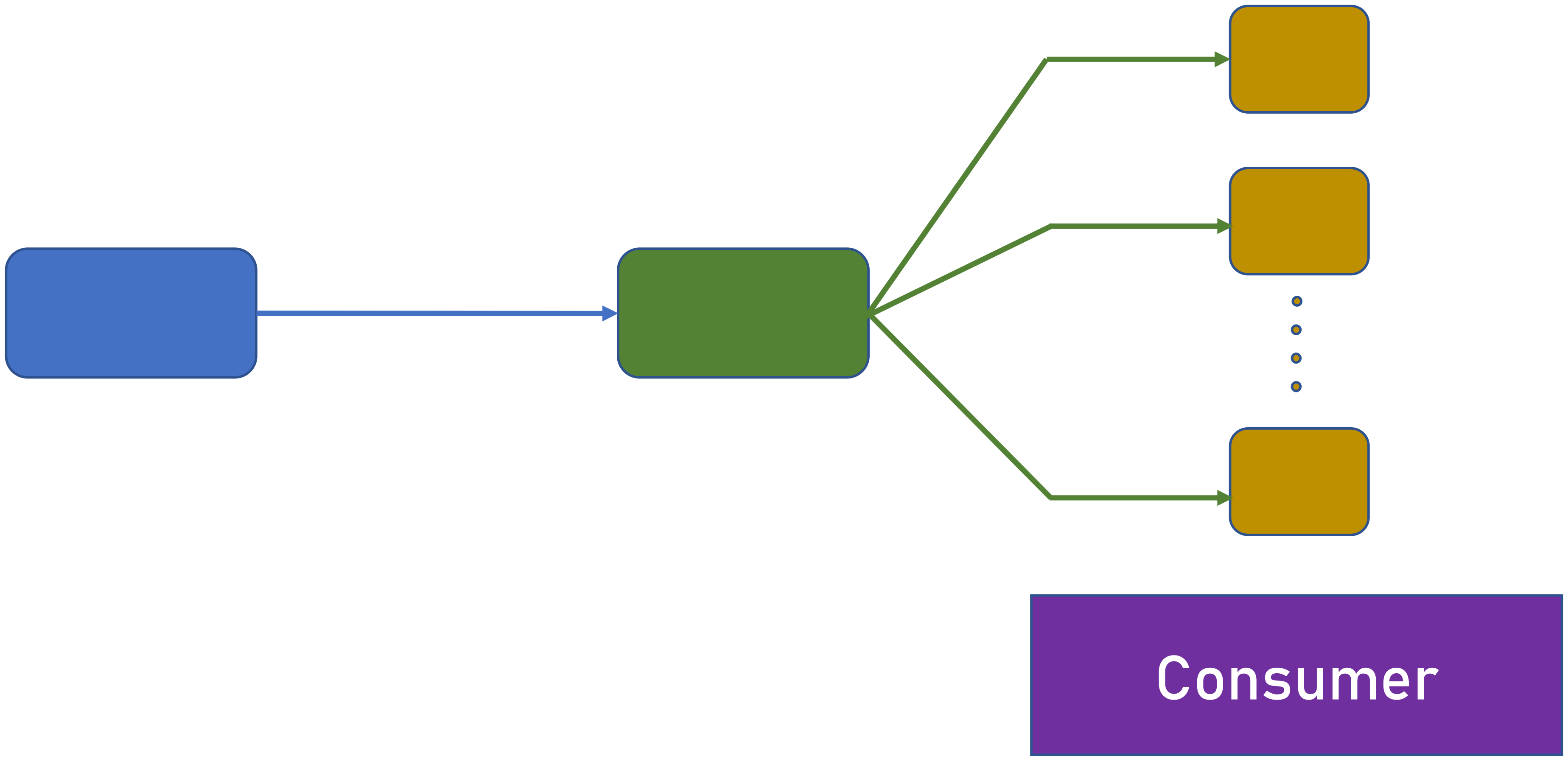
Producer

Channel

Channel

- The channel’s method of distribution varies between channels
- Can be:
  - Queue
  - REST API call
  - Proprietary listener

Consumer

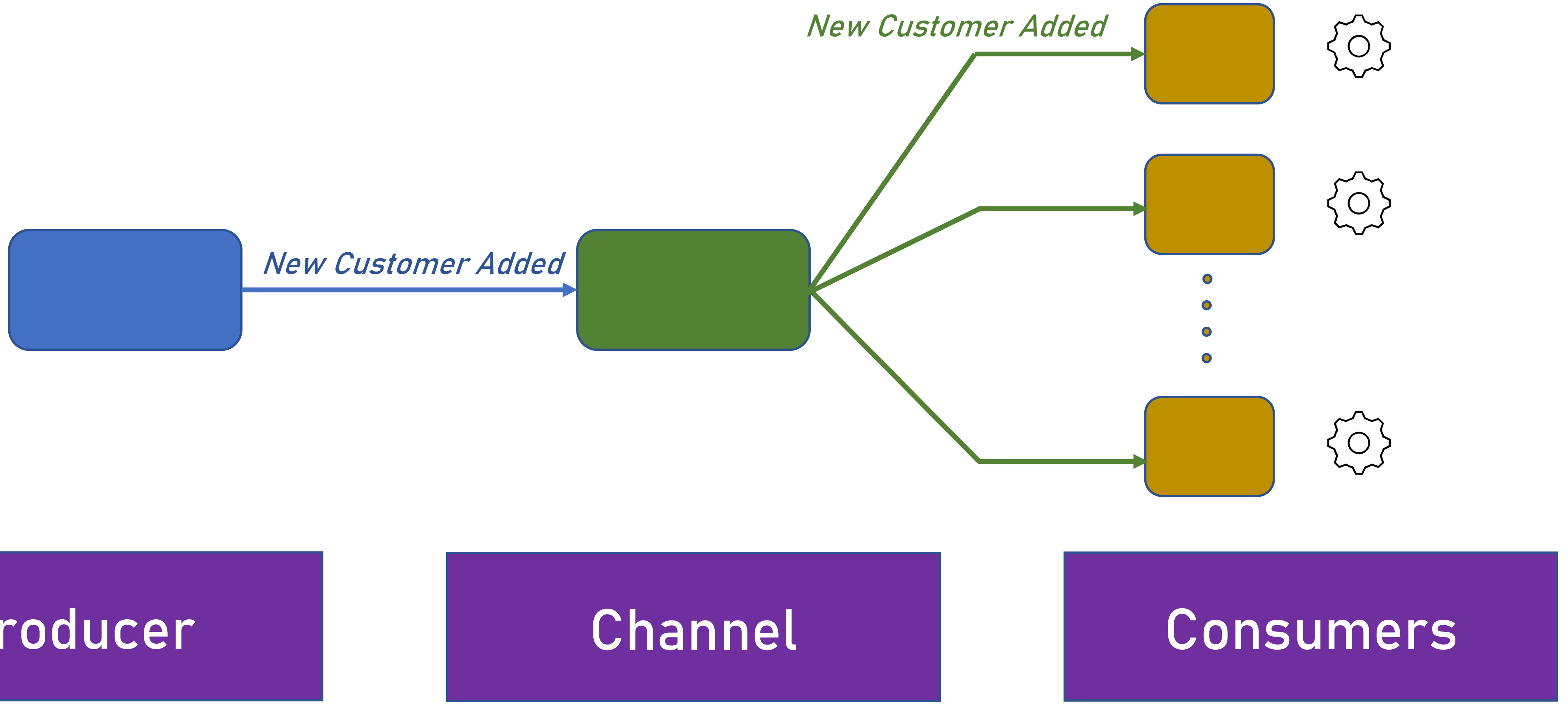


Consumer

- The component that receives the event sent by the Producer and distributed by the Channel
- Can be developed in any development language compatible with the Channel's libraries (if any)
- Processes the event
- Sometimes – reports back when processing is complete (Ack)

Consumer

- The Consumer receives and processes the event

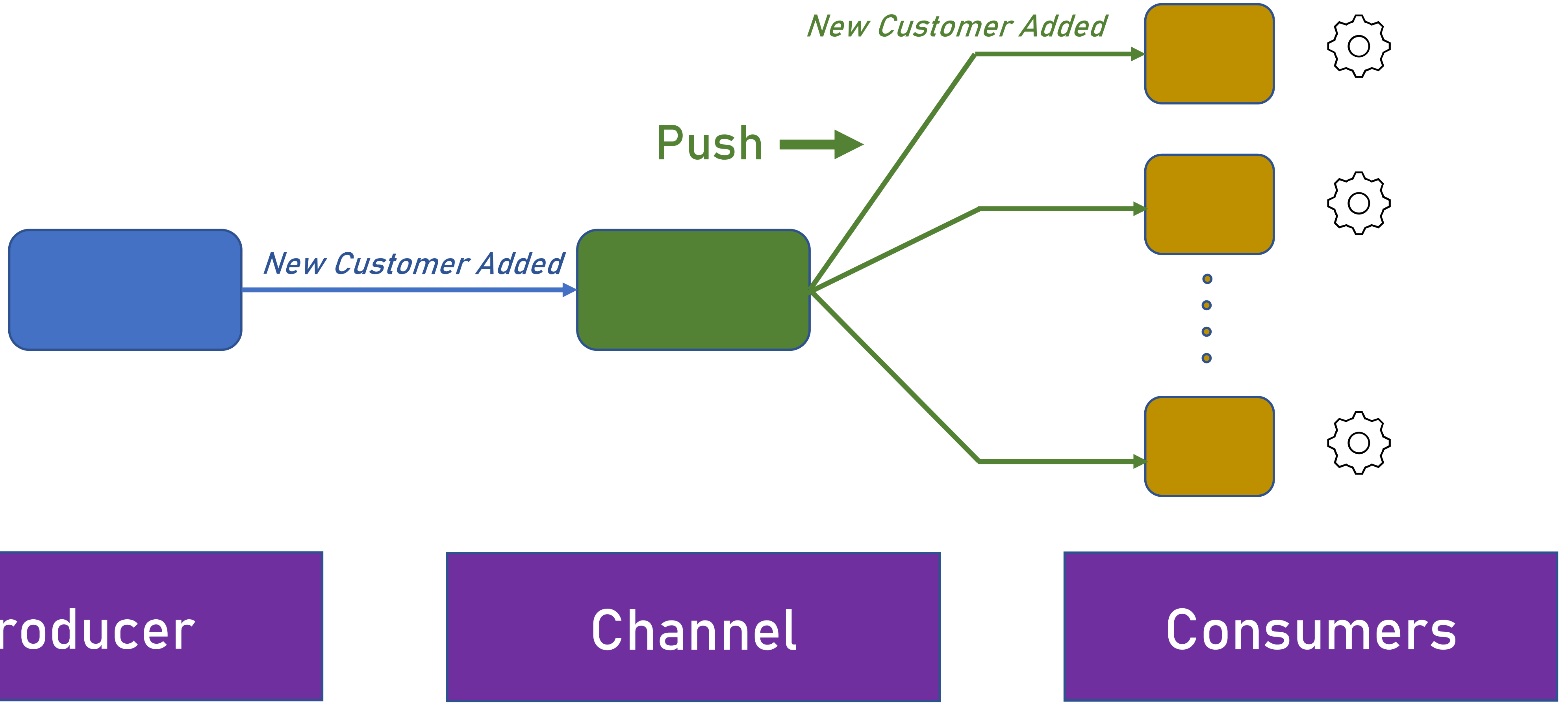


Consumer

- Consumer gets the event using either:
  - Push
  - Pull
- The method depends on the channel

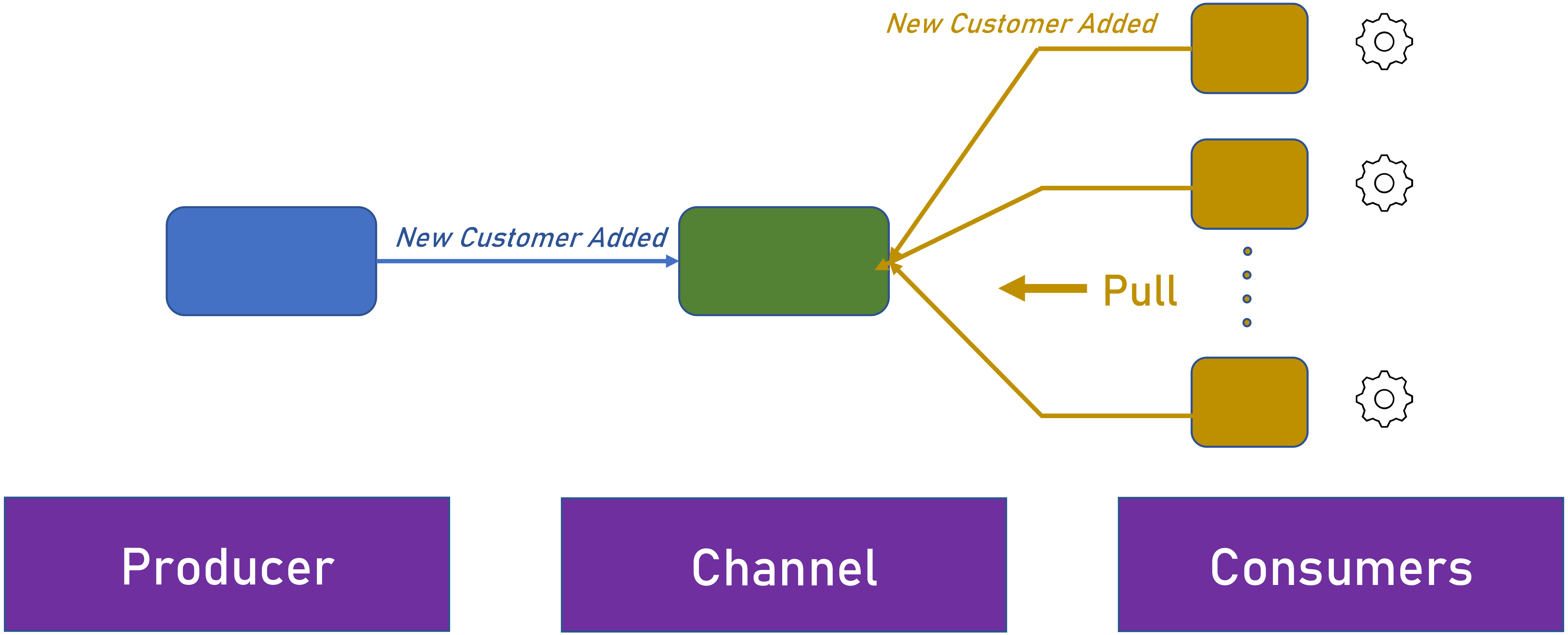
Push

- The Channel pushes the event to the Consumers



Pull

- The Consumers poll the Channel for new events



Advantages of EDA

- Event Driven Architecture has a lot of advantages over other architecture paradigms
- As a quick refresher…

Problems with Command and Query

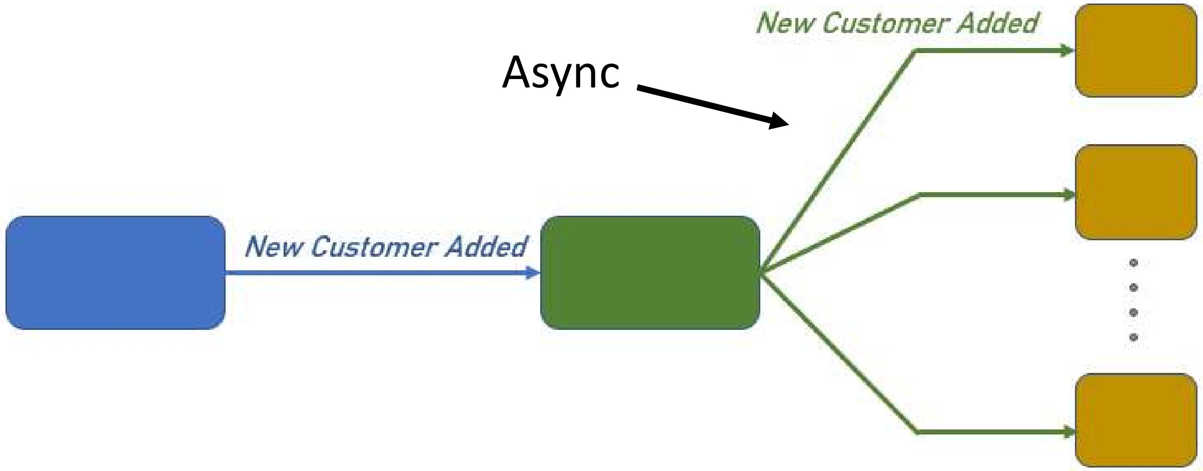
- Three major problems with command and query:

Performance

Coupling

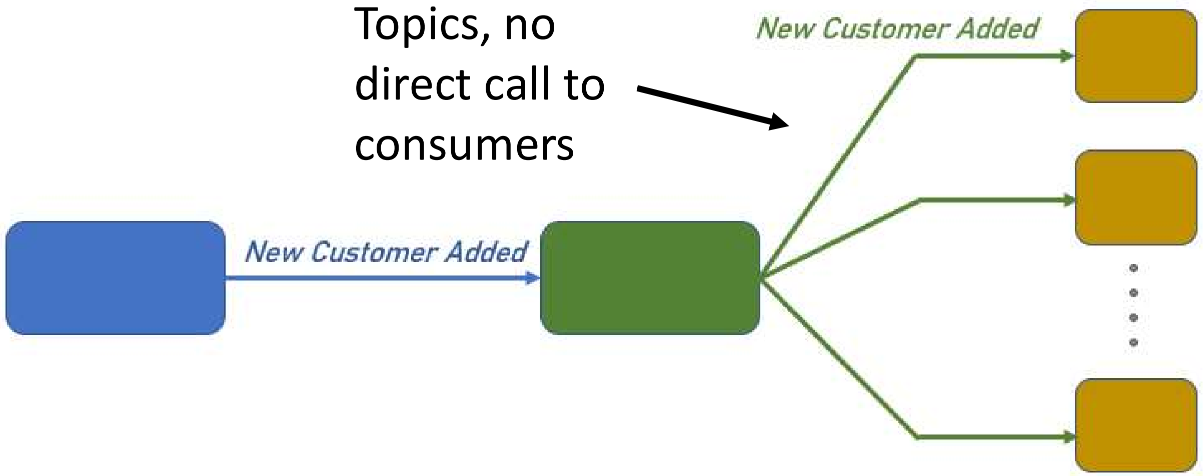
Scalability

Performance



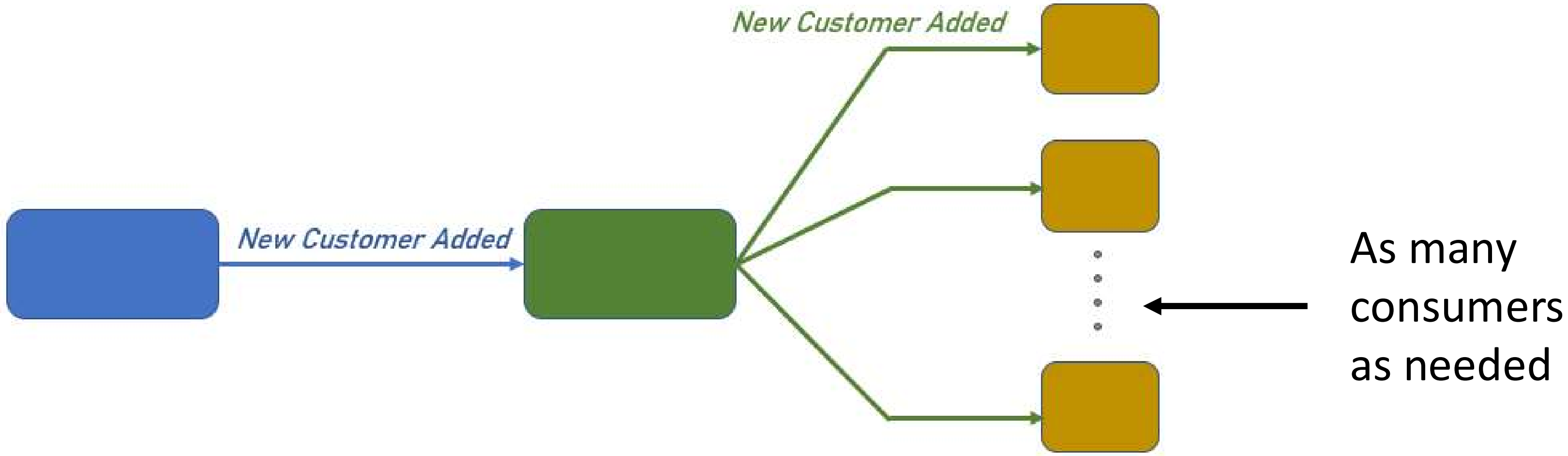
- EDA is an asynchronous architecture
- The Channel does not wait for response from consumer
- No performance bottlenecks

Coupling



- The producer sends events to the channel
- The channel distributes events to topics / queues
- Both have no idea who’s listening to the event (except in WebHooks)
- No coupling

Scalability



As many consumers as needed

- Many consumers can listen to events from channel
- More can be added as needed
- Channel doesn’t care, producer doesn’t know
- Fully scalable

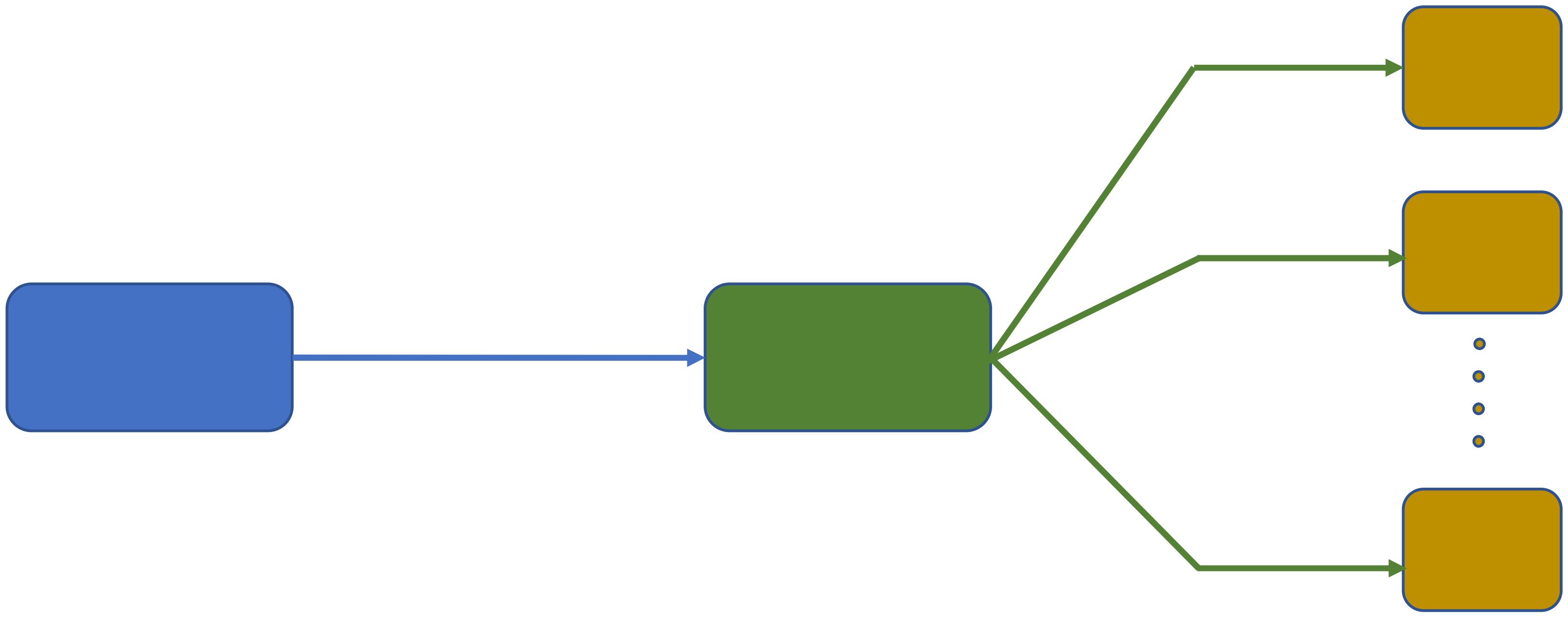
EDA and Pub/Sub

- Event Driven Architecture is often mentioned with Pub/Sub

- Pub/Sub = Publish and Subscribe

- A messaging pattern used by Event Driven Architecture

Components of Pub/Sub



Publisher

Broker

Subscriber

EDA and Pub/Sub

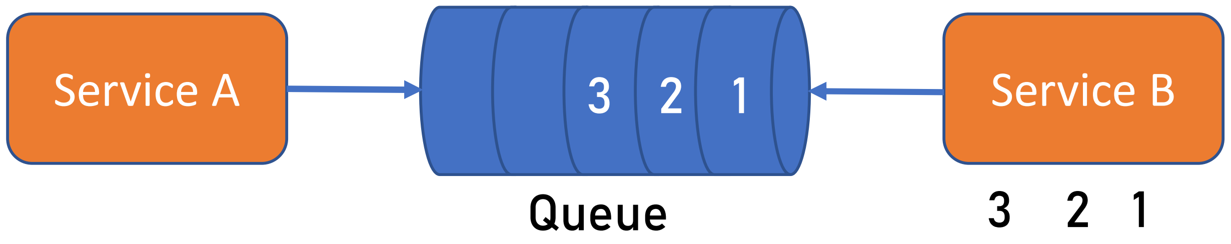
- Event Driven Architecture and Pub/Sub are extremely similar
- Main difference:
  - EDA describes the whole architecture of the system
  - Pub/Sub is a messaging pattern used by the system
    - Not exclusively!

EDA and Pub/Sub

- For example:
  - *“My Event Driven Architecture uses mainly Pub/Sub for inter-service communication, but I do have some REST APIs for synchronous queries.”*

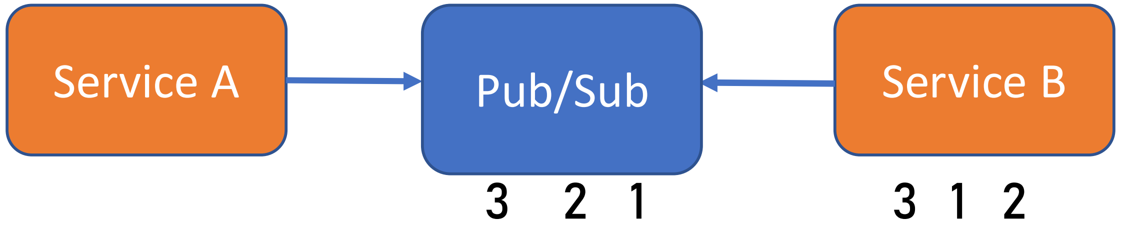
Ordering in EDA

- Messaging engines often guarantee the order of the messages
- Popular mainly in traditional queues



Ordering in EDA

- With Event Driven Architecture (especially with Pub/Sub) ordering is not always guaranteed
- Ordering might be affected by consumer latency, code performance and more



Ordering in EDA

- If ordering is important, make sure to select a channel that supports this capability
- Examples:
    - RabbitMQ supports it
    - SignalR does not
- We’ll use both in the case study section

Orchestration and Choreography

- Event Driven Architecture usually employs one of two architectural styles

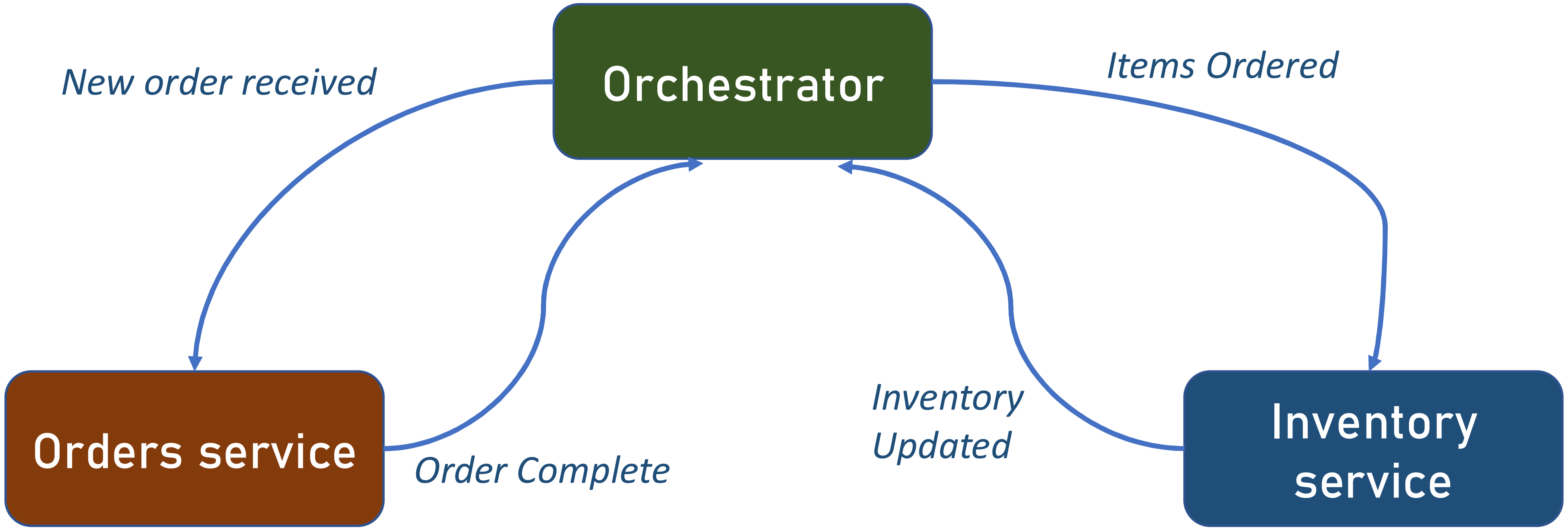
Orchestration

Choreography

Orchestration

- Flow of events in the system is determined by a central orchestrator
- Orchestrator receives output from components and calls the next component in the flow
- The next component sends the output back to the orchestrator etc.

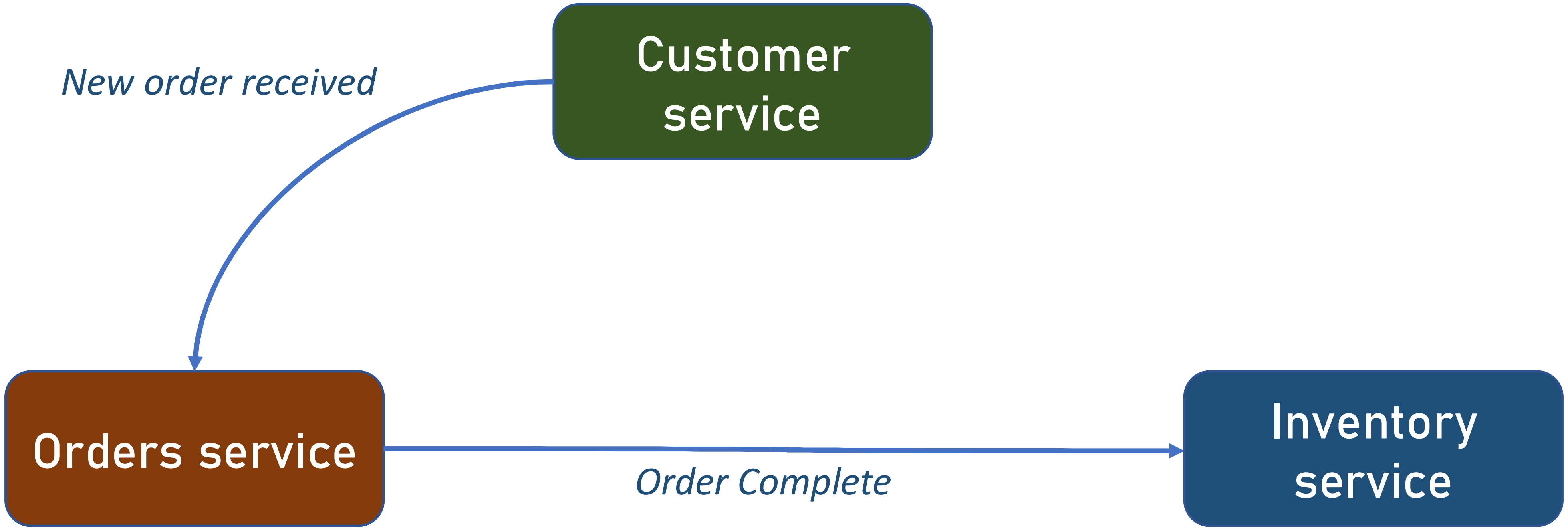
Orchestration



Choreography

- No central “knowing all” component
- Each component notifies about the status of events
- Other components listen to the events and act accordingly

Choreography



Customer service

New order received

Orders service

Order Complete

Inventory service

Orchestration and Choreography

Orchestration

- Logic is defined in a single place – easier to maintain
- Central traffic gateway – easier monitoring and logging

Choreography

- Performance – no middleman
- Reliability – if one component fails, the rest still work

Orchestration and Choreography

- Not constrained to EDA only
- Can be used with other types of communication
- Became popular with EDA

Event Sourcing and CQRS

- Event Driven Architecture is mainly about services
- Events can be used as the basic building blocks of data too
- Event Sourcing and CQRS offer a pattern to store data as events

Problems with Traditional DBs

- Traditional databases hold data about current state of entity
  - This is true for SQL and NoSQL databases

Problems with Traditional DBs

- Example: Employees table

emp_id	first_name	last_name	address	role	date_join
1	John	Smith	Beverly Hills 90210	Development Manager	2009-04-23
2	Sarah	Jones	42nd st. NYC	Sales	2019-01-30
3	Britney	Flyn	Marigold Lane, Boca Raton	HR	2022-05-19

Problems with Traditional DBs

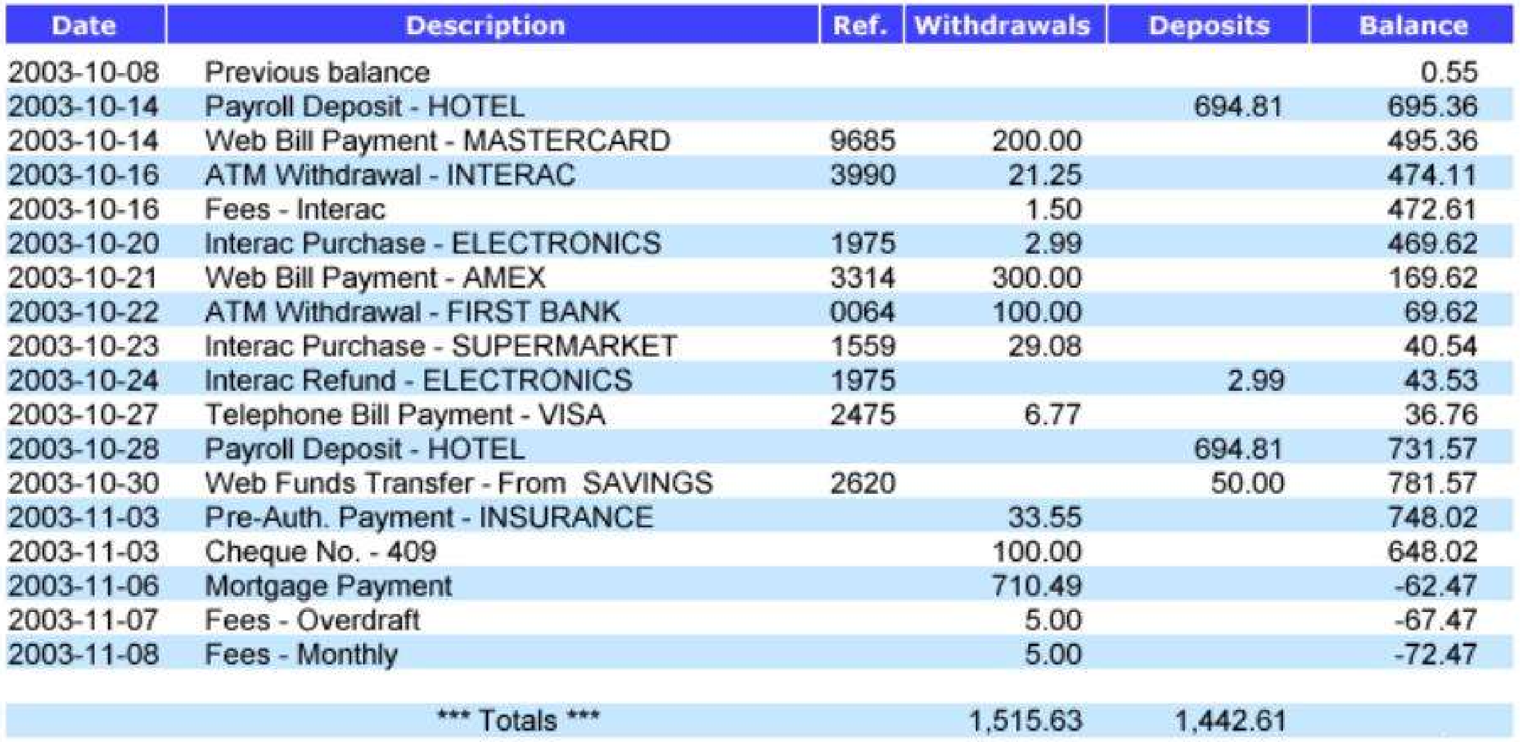
emp_id	first_name	last_name	address	role	date_join
1	John	Smith	Beverly Hills 90210	Development Manager	2009-04-23
2	Sarah	Jones	42nd st. NYC	Sales	2019-01-30
3	Britney	Flyn	Marigold Lane, Boca Raton	HR	2022-05-19

- This table doesn’t tell us:
  - What was John’s previous role?
  - When did Sarah move to NYC?
  - Did any of the employees change his/her name?

Problems with Traditional DBs

- Traditional databases hold data about current state of entity
- There is no way to see historical data of entities
- Data is a “snapshot” of a point in time
- Especially problematic with…

Problems with Traditional DBs



Date	Description	Ref.	Withdrawals	Deposits	Balance
2003-10-08	Previous balance				0.55
2003-10-14	Payroll Deposit - HOTEL			694.81	695.36
2003-10-14	Web Bill Payment - MASTERCARD	9685	200.00		495.36
2003-10-16	ATM Withdrawal - INTERAC	3990	21.25		474.11
2003-10-16	Fees - Interac		1.50		472.61
2003-10-20	Interac Purchase - ELECTRONICS	1975	2.99		469.62
2003-10-21	Web Bill Payment - AMEX	3314	300.00		169.62
2003-10-22	ATM Withdrawal - FIRST BANK	0064	100.00		69.62
2003-10-23	Interac Purchase - SUPERMARKET	1559	29.08		40.54
2003-10-24	Interac Refund - ELECTRONICS	1975		2.99	43.53
2003-10-27	Telephone Bill Payment - VISA	2475	6.77		36.76
2003-10-28	Payroll Deposit - HOTEL			694.81	731.57
2003-10-30	Web Funds Transfer - From SAVINGS	2620		50.00	781.57
2003-11-03	Pre-Auth. Payment - INSURANCE		33.55		748.02
2003-11-03	Cheque No. - 409		100.00		648.02
2003-11-06	Mortgage Payment		710.49		-62.47
2003-11-07	Fees - Overdraft		5.00		-67.47
2003-11-08	Fees - Monthly		5.00		-72.47
	*** Totals ***		1,515.63	1,442.61	

- Event Sourcing and CQRS try to solve this problem

Event Sourcing

- A data store pattern in which every change in the data is captured and saved
- Database stores list of changes for the entity, not the entity itself
- No updates or deletes, just inserts
- Every row documents a change in a property/ies of the entity
- In this pattern, the database is called Event Store

Event Sourcing

- Instead of this:

emp_id	first_name	last_name	address	role	date_join
1	John	Smith	Beverly Hills 90210	Development Manager	2009-04-23
2	Sarah	Jones	42nd st. NYC	Sales	2019-01-30
3	Britney	Flyn	Marigold Lane, Boca Raton	HR	2022-05-19

Event Sourcing

- We have this:

event_id	timestamp	event
1	2009-04-23	Employee John Smith joined
2	2009-04-23	Address of John Smith updated to Hott Street, Clinton
3	2009-04-23	Role of John Smith updated to Junior Developer
4	2013-05-22	Address of John Smith updated to Beverly Hills 90210
5	2017-09-12	Role of John Smith updated to Development Manager
6	2019-01-30	Employee Sarah Jones joined
7	2019-01-30	Role of Sarah Jones updated to Sales
8	2021-07-05	David Richer left the company

- Specific columns are up to you, depends on the system requirements
- Note there's a lot more information than in the regular table

Event Sourcing

- How can we view the current state of an entity?
- By replaying the events

rel

eve

event_id	timestamp	event
1	2009-04-23	Employee John Smith joined
2	2009-04-23	Address of John Smith updated to Hott Street, Clinton
		verly Hills
5	2017-09-12	Role of John Smith updated to Development Manager
6	2019-01-30	Employee Sarah Jones joined
7	2019-01-30	Role of Sarah Jones updated to Sales
8	2021-07-05	David Richer left the company

emp_id	first_name	last_name	address	role	date_join
1	John	Smith	Beverly Hills 90210	Development Manager	2009-04-23

Event Sourcing

Pros

- Extremely easy to view historical data
- Simple database structure
- Simple database operations (no updates, no concurrency)
- Very fast inserts

Cons

- Viewing current entity state is cumbersome and slow
- Large database capacity (many records per entity)

CQRS to the Rescue!

CQRS

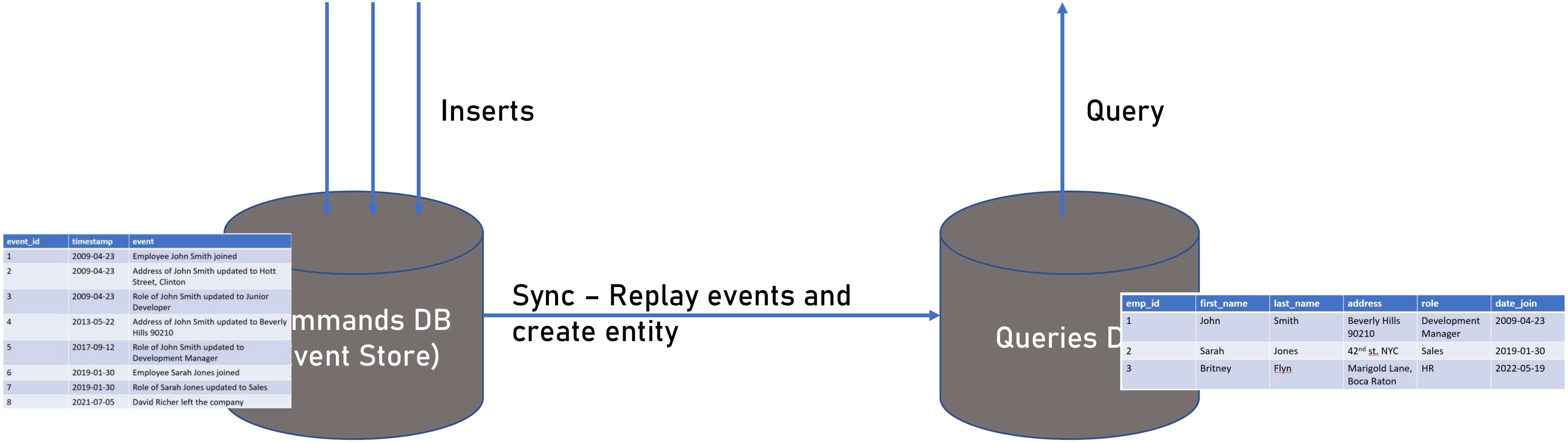
- Stands for:

Command and Query Responsibility Segregation

CQRS

- Means:
  - Separating the commands (updates / inserts / deletes) from the queries
  - Each one of them in a separate database
  - Commands database is implemented as Event Store to improve performance and simplicity
  - Queries database stores entities
  - Database are synced using a central synchronization mechanism

CQRS



event_id	timestamp	event
1	2009-04-23	Employee John Smith joined
2	2009-04-23	Address of John Smith updated to Hott Street, Clinton
3	2009-04-23	Role of John Smith updated to Junior Developer
4	2013-05-22	Address of John Smith updated to Beverly Hills 90210
5	2017-09-12	Role of John Smith updated to Development Manager
6	2019-01-30	Employee Sarah Jones joined
7	2019-01-30	Role of Sarah Jones updated to Sales
8	2021-07-05	David Richer left the company

emp_id	first_name	last_name	address	role	date_join
1	John	Smith	Beverly Hills 90210	Development Manager	2009-04-23
2	Sarah	Jones	42nd st. NYC	Sales	2019-01-30
3	Britney	Flyn	Marigold Lane, Boca Raton	HR	2022-05-19

CQRS

Pros

- Combines Event Sourcing pros with traditional entity query
- No performance hit when querying entities

Cons

- Entity data is not updated in real-time
- Difficult to set-up and maintain

When to Use Event Sourcing & CQRS

- When access to historical data is extremely important
    - Regulation, finance, healthcare etc.
- When data is large and replaying events is not feasible
- When performance is critical (inserts or queries)

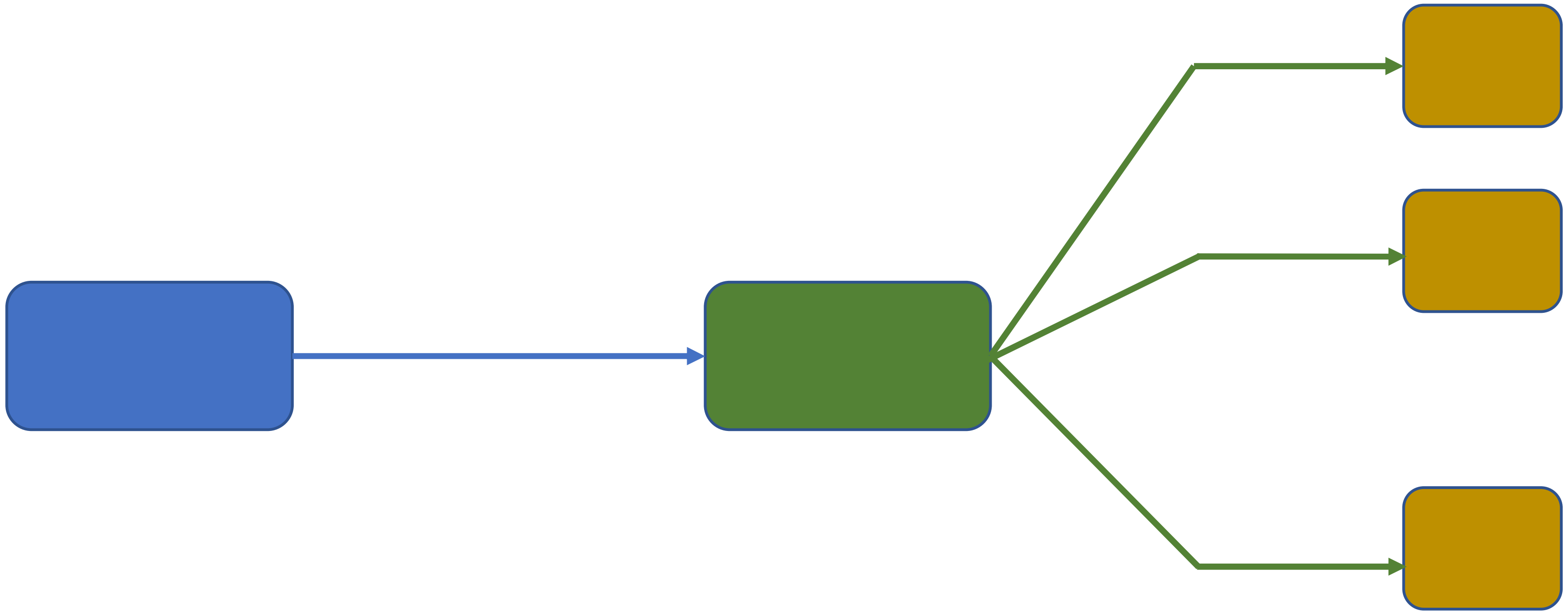
When to Use Event Driven Architecture

- Event Driven Architecture is not easy to implement
- Requires setting up and configuring channels
- Not trivial logging and monitoring
- Be sure to use it when needed

Scalability

- Scalability is a non-issue in EDA
- New consumers can be added as needed with no changes to the architecture
- Great for fluctuating load

Scalability



Asynchronous

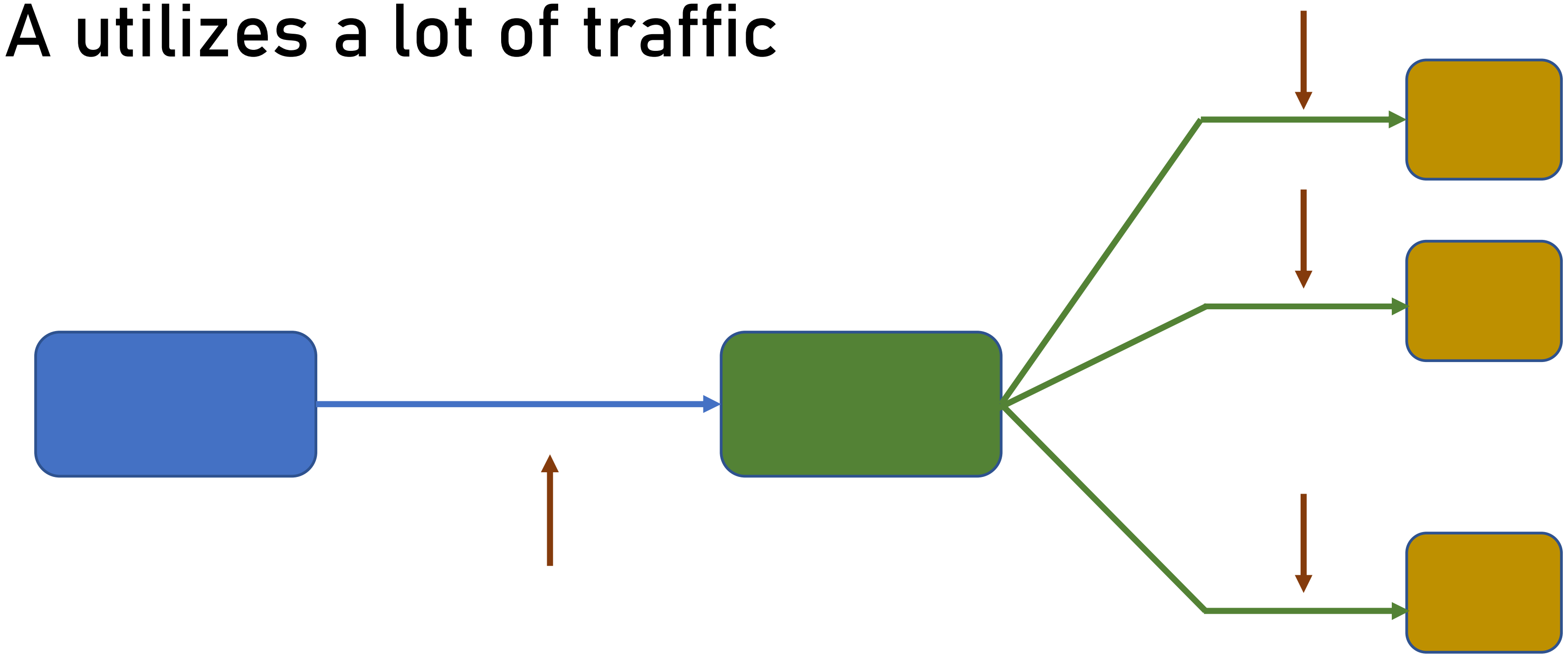
- If inter-service communication can be asynchronous, consider EDA
- Remember: EDA is async by nature
- Examples:
  - Send instructions to perform payment
  - Write to log

Asynchronous

- Check how many synchronous interactions there are
- Usually mainly queries
- The more synchronous calls – the less EDA is relevant

Reliable Network

- EDA utilizes a lot of traffic



- Network should be reliable or performance will be slow

When not to Use EDA

- EDA is not suitable for:
    - Small systems with a few services
    - Synchronous-oriented systems
        - ie. Information system serving mainly queries from end users

Stateless vs Stateful EDA

- There are two main patters in implementing EDA
- Stateless and Stateful
- Related to the consumers behavior
- Both are legitimate, but make sure to select the right one for your scenario

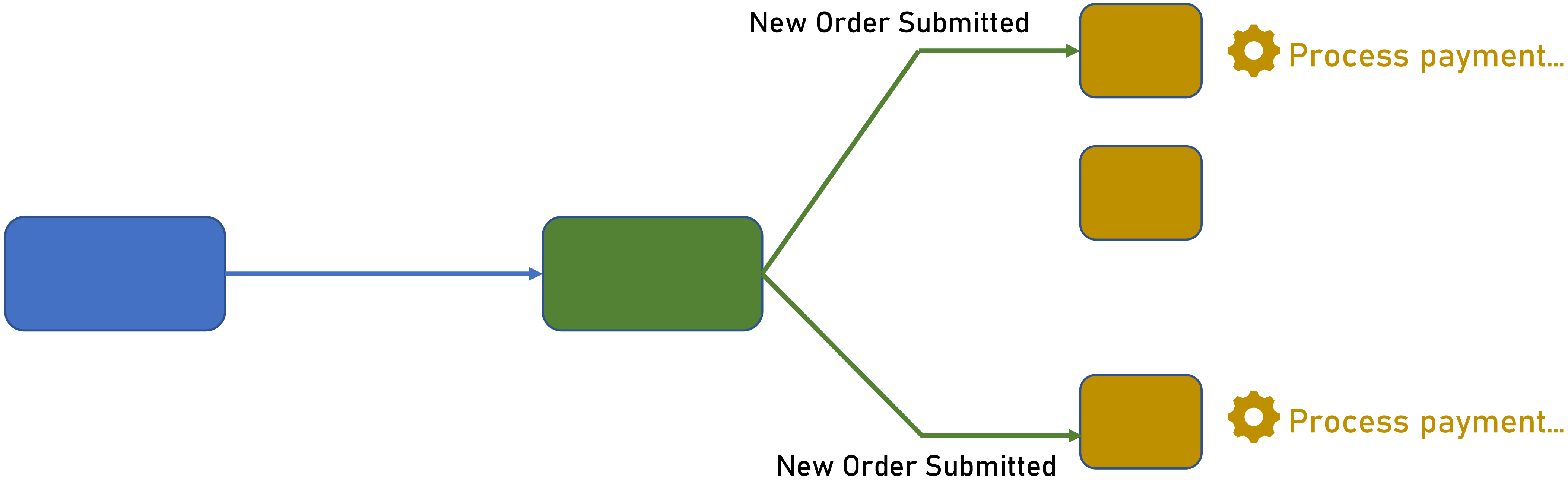
Stateless vs Stateful EDA

- In software architecture there's also the stateless vs stateful debate
- While the concepts are similar, the reasoning is different
- With software architecture it's often said that:
  - "Stateful is bad"
- This is not necessarily the case with EDA

Stateless EDA

- Each event handled by a consumer is completely autonomous and is not related to past / future events
- Should be used when the event is an independent unit with its own outcomes

Stateless EDA



Payment processing consumers

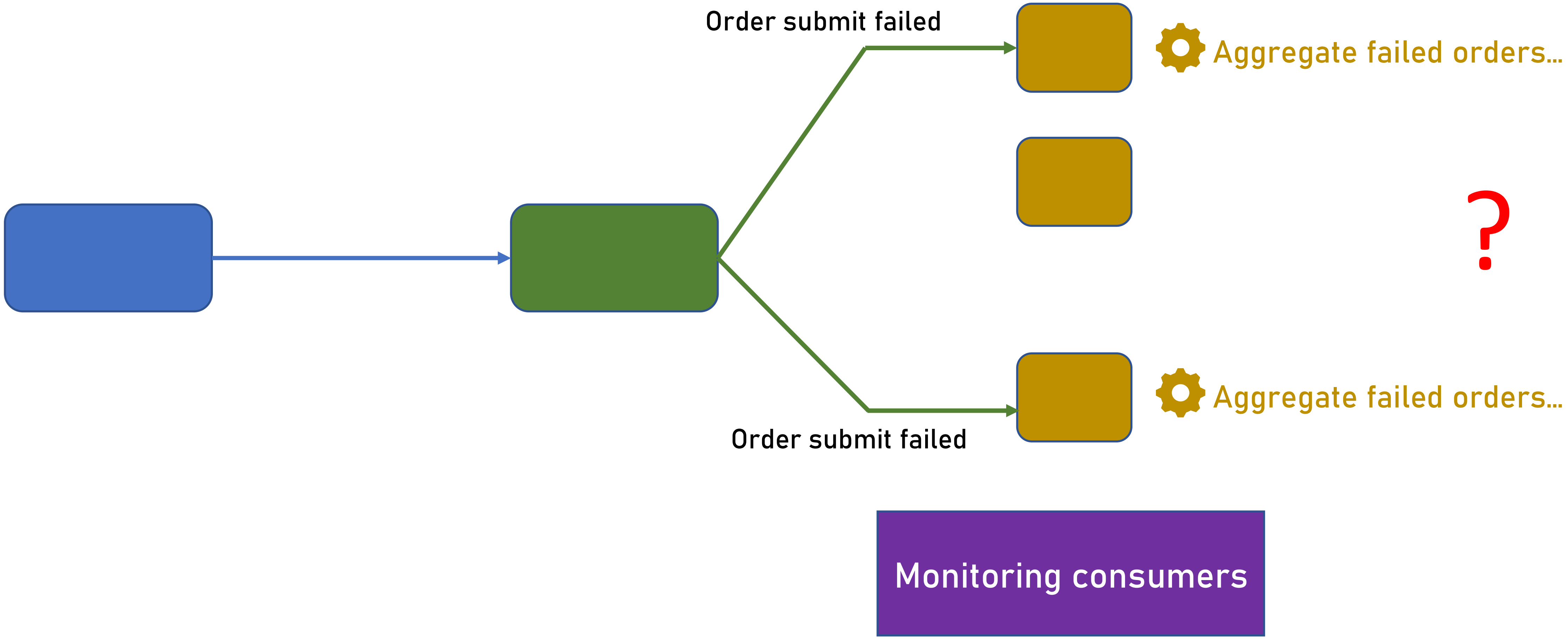
Stateless EDA

- It doesn’t matter which consumer is handling the event
- The outcome is always the same
- Should be used when each event is autonomous
- Note: Has nothing to do with the question of what data is contained in the event and whether a call to a DB is required

Stateful EDA

- Events might be related to past / future events
- Should be used mainly for aggregators and time-related events
- Examples:
  - Send an email if more than 5 failure events were received in a single minute
  - Calculate the amount of orders submitted in an hour

Stateful EDA



Stateful EDA

- It’s extremely important which consumer handles the event
- Current state is stored in specific consumer(s)
- Should be used when events are part of a chain of events

Problems with Stateful EDA

- Stateful EDA presents some problems that should be taken care of

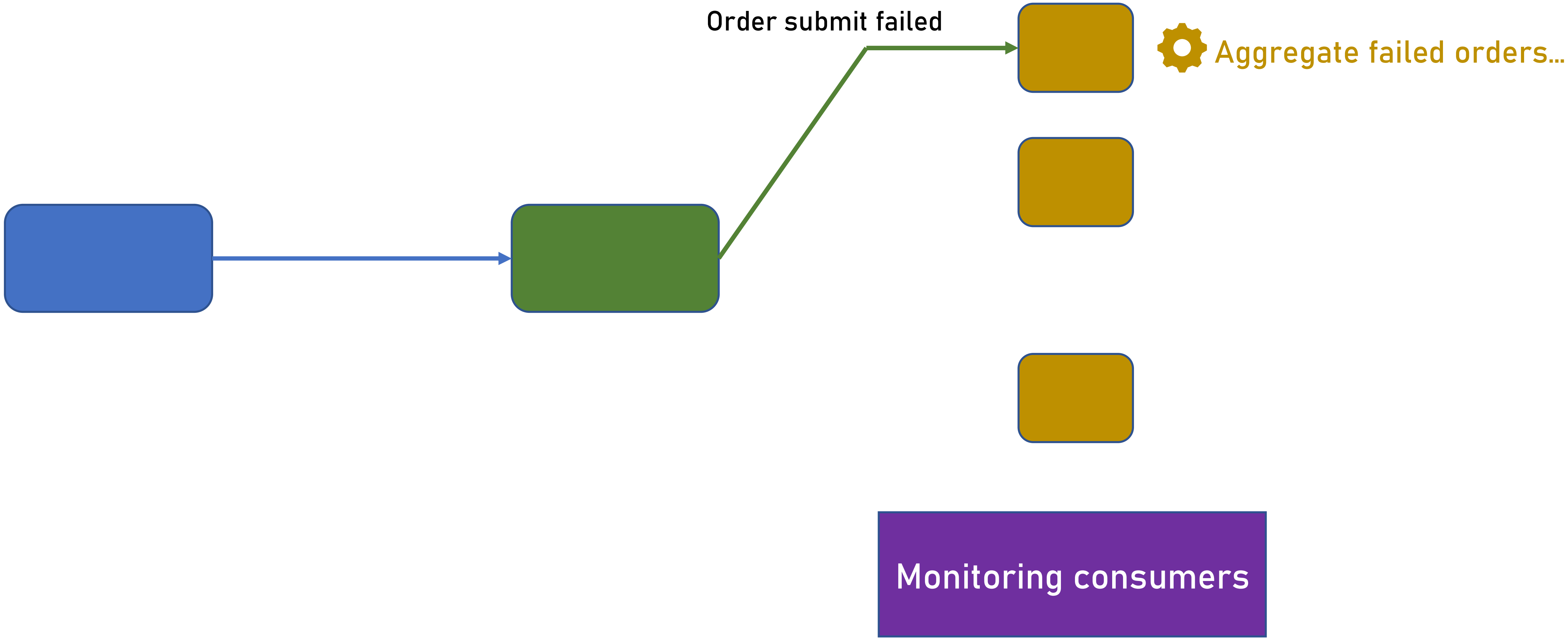
Load balancing

Scalability

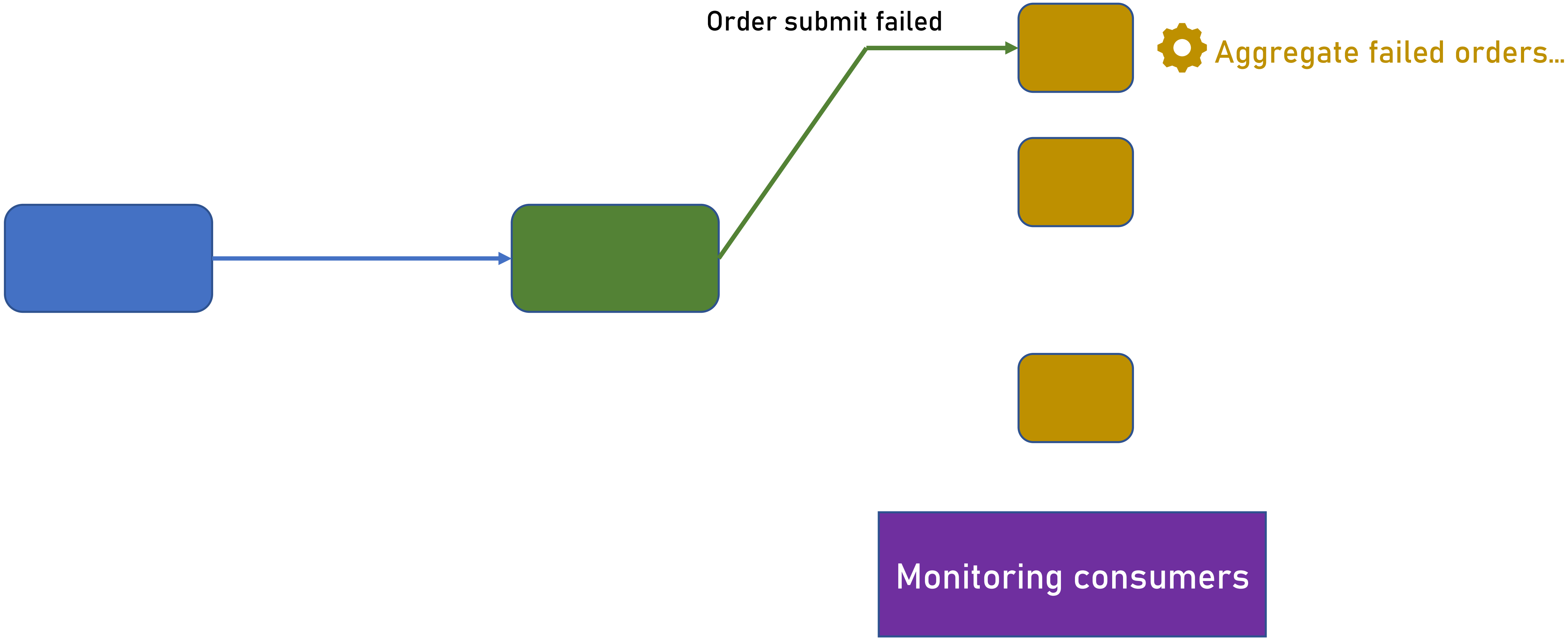
Load Balancing

- Since the state is stored in a specific consumer, subsequent events must be routed to the same consumer
- No load balancing is possible

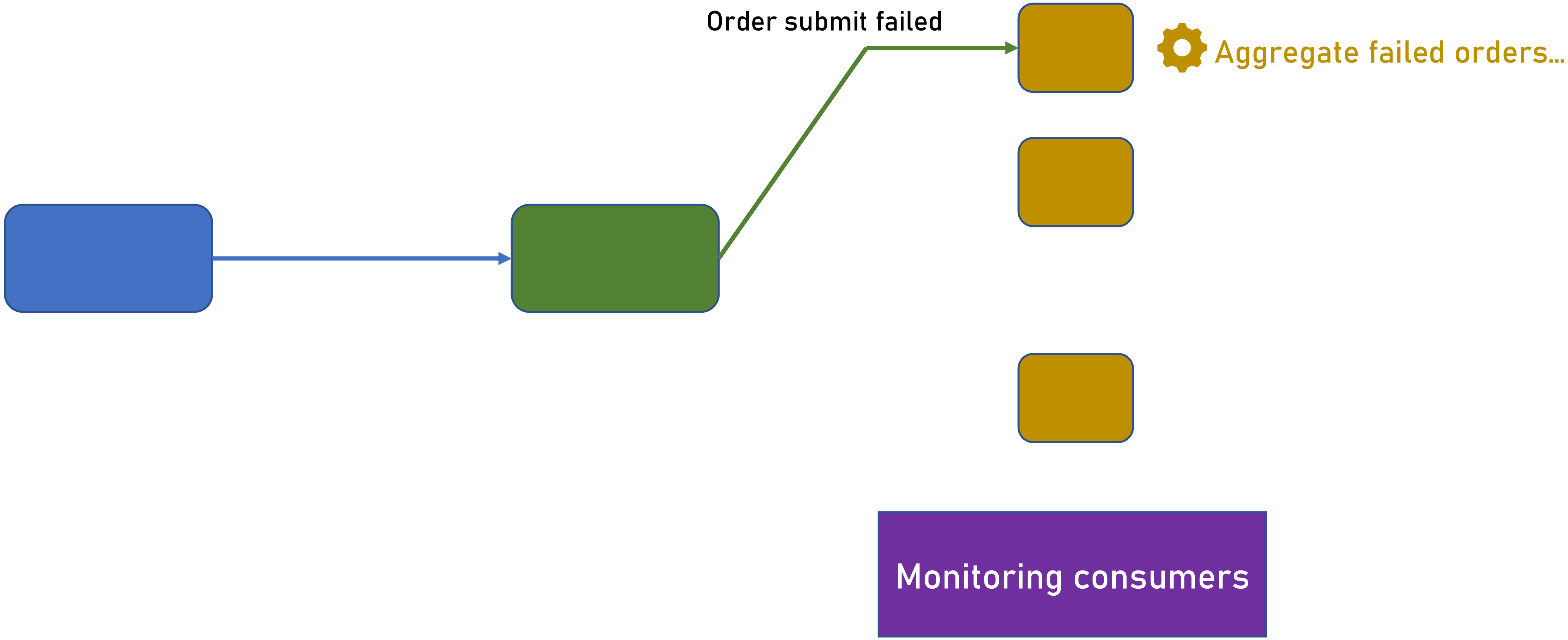
Load Balancing



Load Balancing



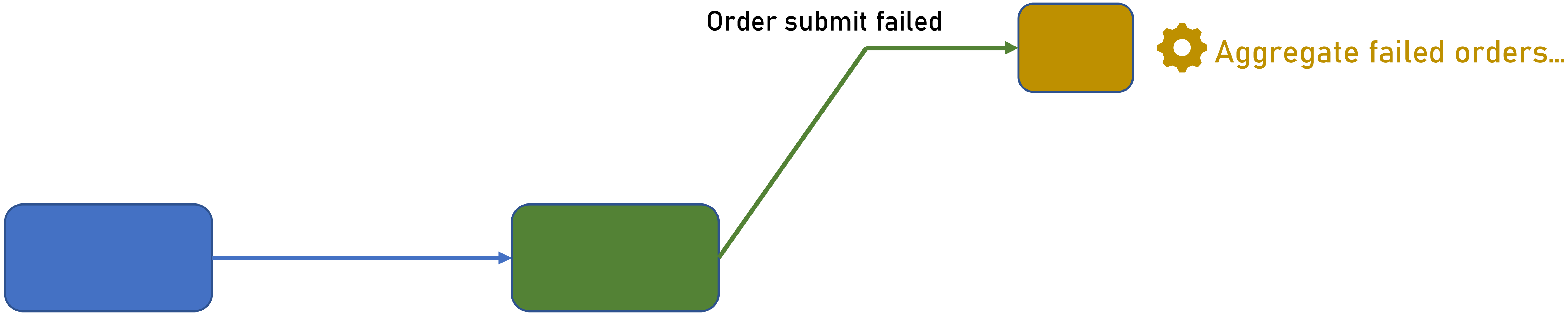
Load Balancing



Scalability

- Since the state is stored in a specific consumer, additional consumers cannot be added to handle the events

Scalability

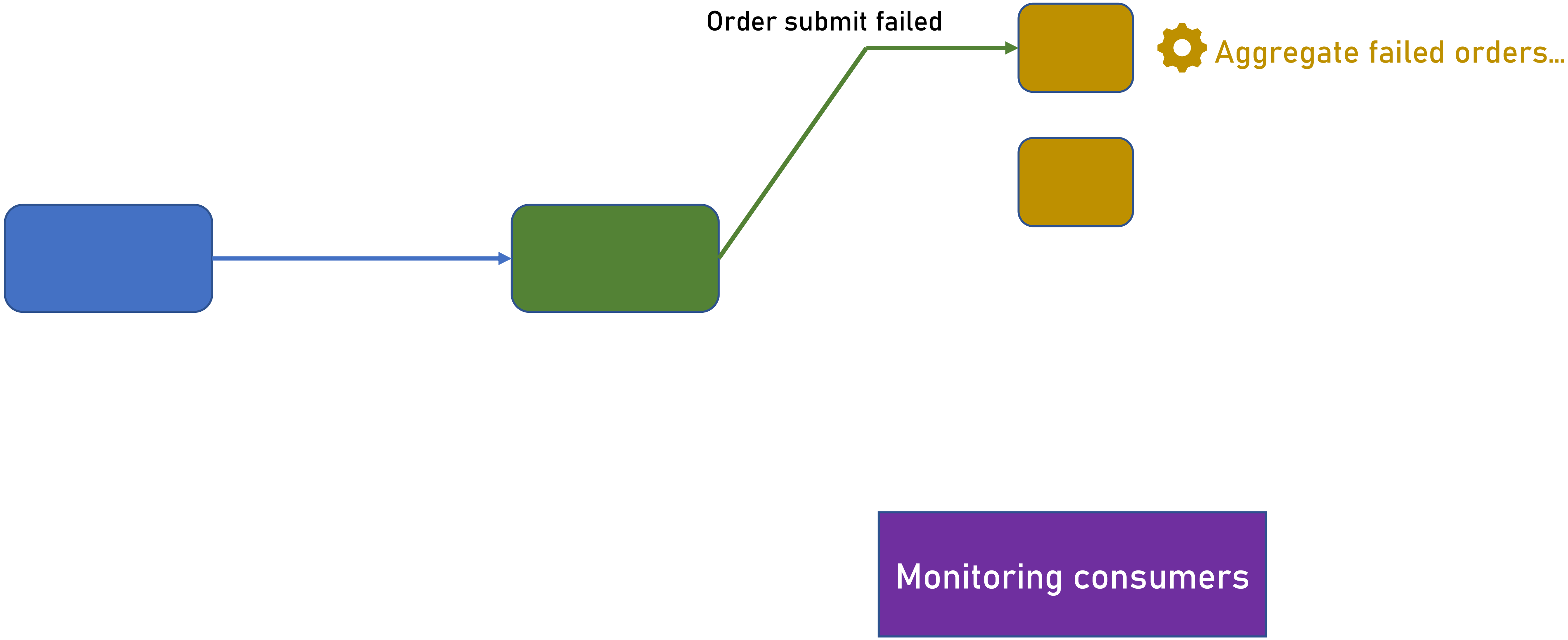


Order submit failed

Aggregate failed orders…

Monitoring consumers

Scalability



Stateless vs Stateful

- Rule of thumb:
  - Use stateless EDA unless the business requirements force you to use stateful

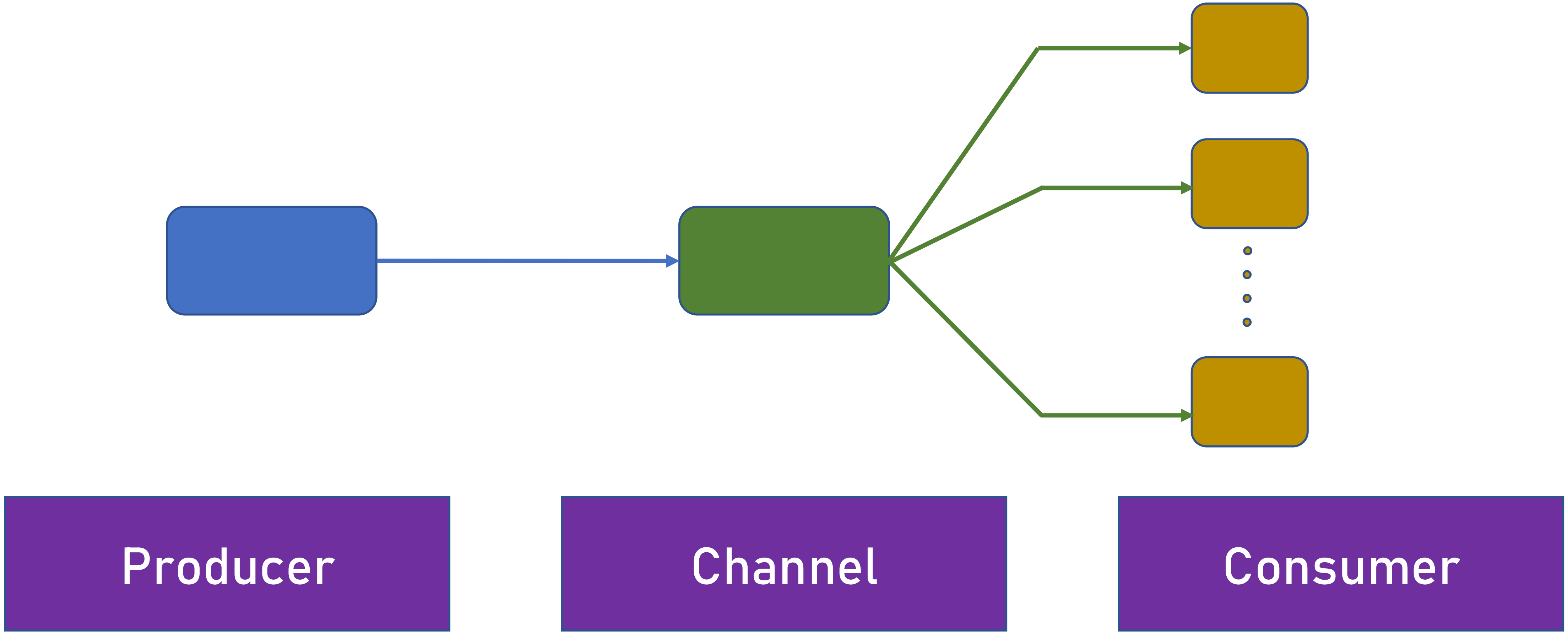
Event Streaming

- So far we talked about Event Driven Architecture
  - Something happened
  - An event was created
  - Someone listened to the event and handled it
- Event Streaming is another event-oriented pattern
- These are not the same, but share similar characteristics

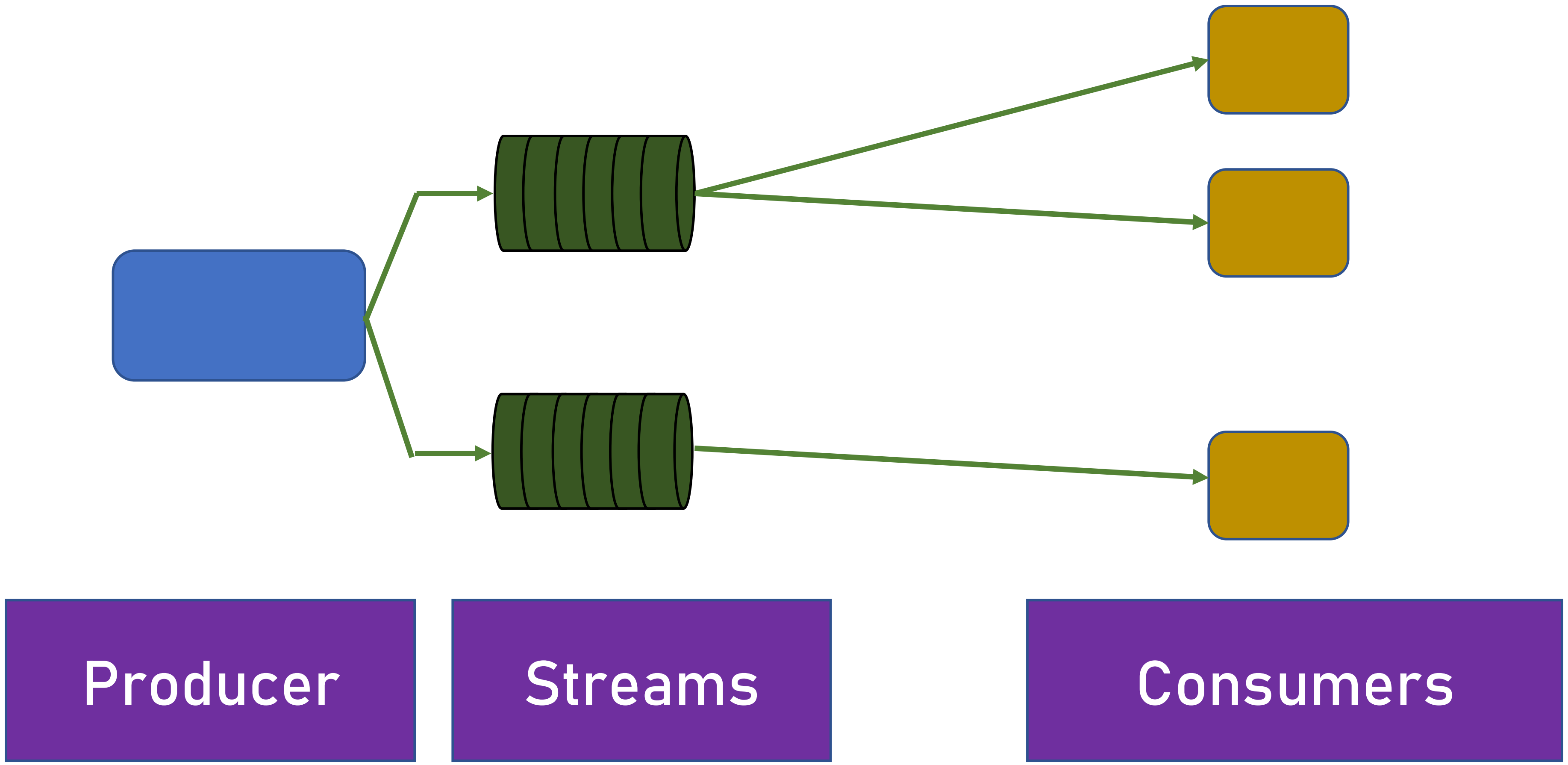
What is Event Streaming

- Event Streaming engines publish stream of events
  - E.g. Telemetry from sensors, system logs etc.
  - The events are published to a “stream”
  - Consumers subscribe to specific stream
  - Events are retained in a stream for a specified amount of time

Regular EDA



Event Streaming



Producer

Streams

Consumers

Event Streaming

- Consumers can retrieve events that were sent in the past (usually up to a few days)
- Streaming Engine can be used as a central database
  - A single source of truth
- Not all events are necessarily handled
  - Some might be not relevant

Event Streaming vs EDA

Event Streaming

- Usually used for events generated outside of the system
- Events are retained
- Not all events are handled
- High load

EDA

- Usually used for events happening inside the system
- Events are not retained
- All events are handled
- No high load

When to Use Event Streaming

- When the system needs to handle stream of events from the outside
  - E.g. Sensor data, logs, etc.
- When events should be retained for future use
- When high load is expected

Implementing Event Driven Architecture

- Mainly 4 things to consider:

Events Approach

Implementing the Channel

Implementing the Producer

Implementing the Consumer

Events Approach

- Two main approaches for implementing events:

Events are retained

Events are not retained

Retaining Events

- The channel retains the event for future handling
- A retention period is defined which after it expires – the event is removed
- Great for streaming events and when the channel is the source of truth

Not Retaining Events

- The channel publishes the events and does not store them
- If a consumer missed an event – it can’t be replayed
- Used mainly for in-system events

Implementing the Channel

- Depends on the events approach

Events are retained

- Use a messaging / queue engine
- Common engines:
  - RabbitMQ
  - Kafka

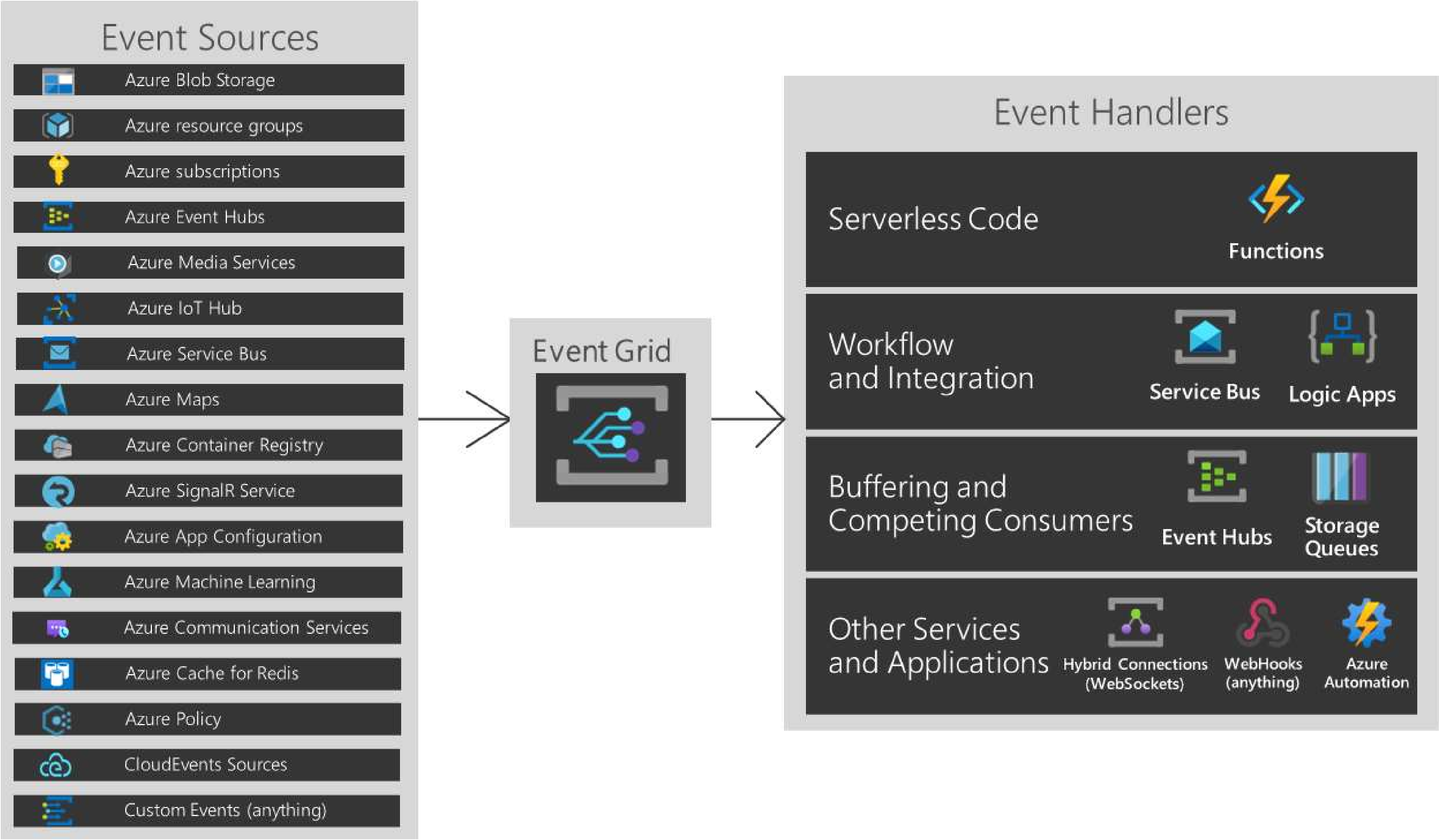
Events are not retained

- Use an event publisher
- Specific engine depends on platform used, types of interfaces and more
- Let's see some examples…

Azure Event Grid

- Events publisher in the cloud
- Fully hosted in the Azure cloud, no installation required
- Great integration with a lot of event sources and handlers
- Can deal with thousands of events / sec

Azure Event Grid

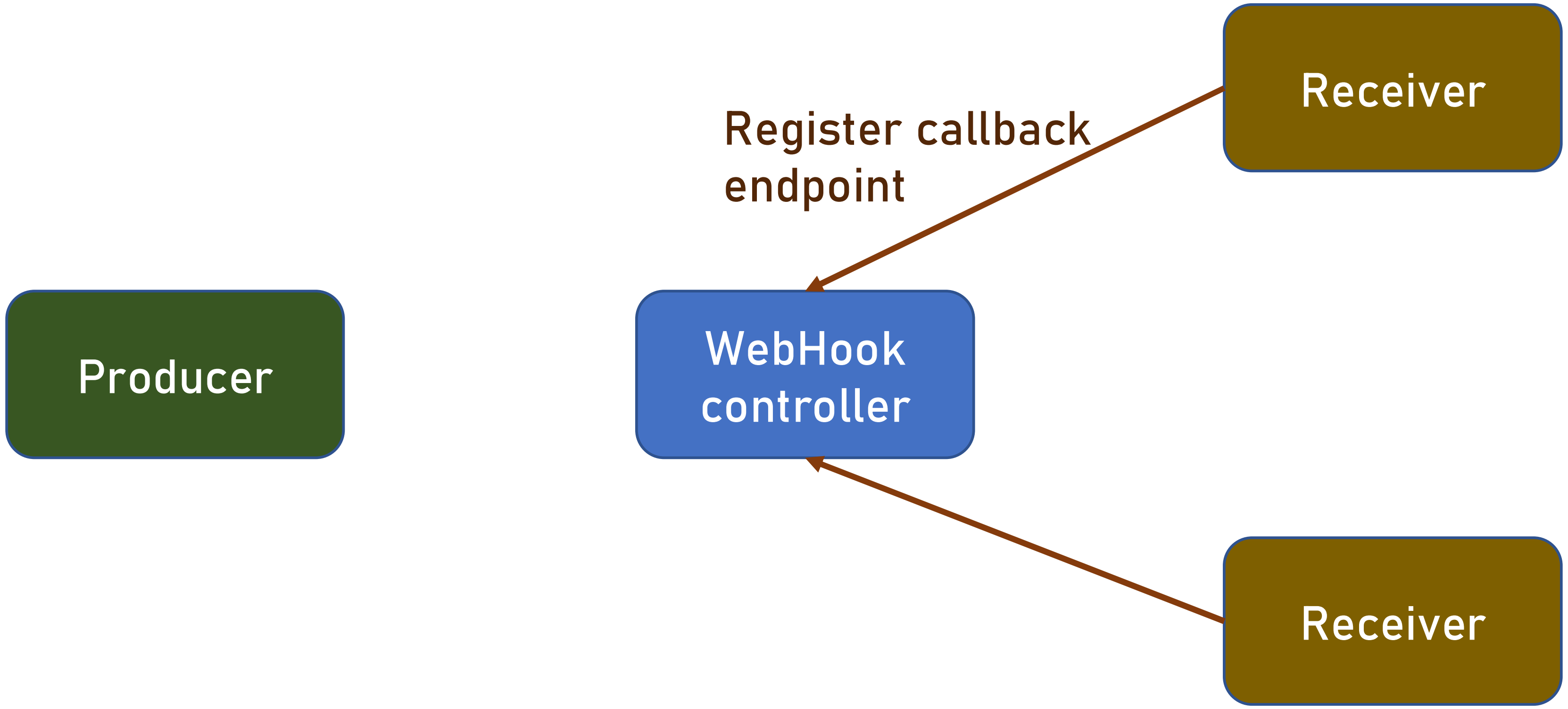


Source: https://docs.microsoft.com/en-us/azure/event-grid/overview

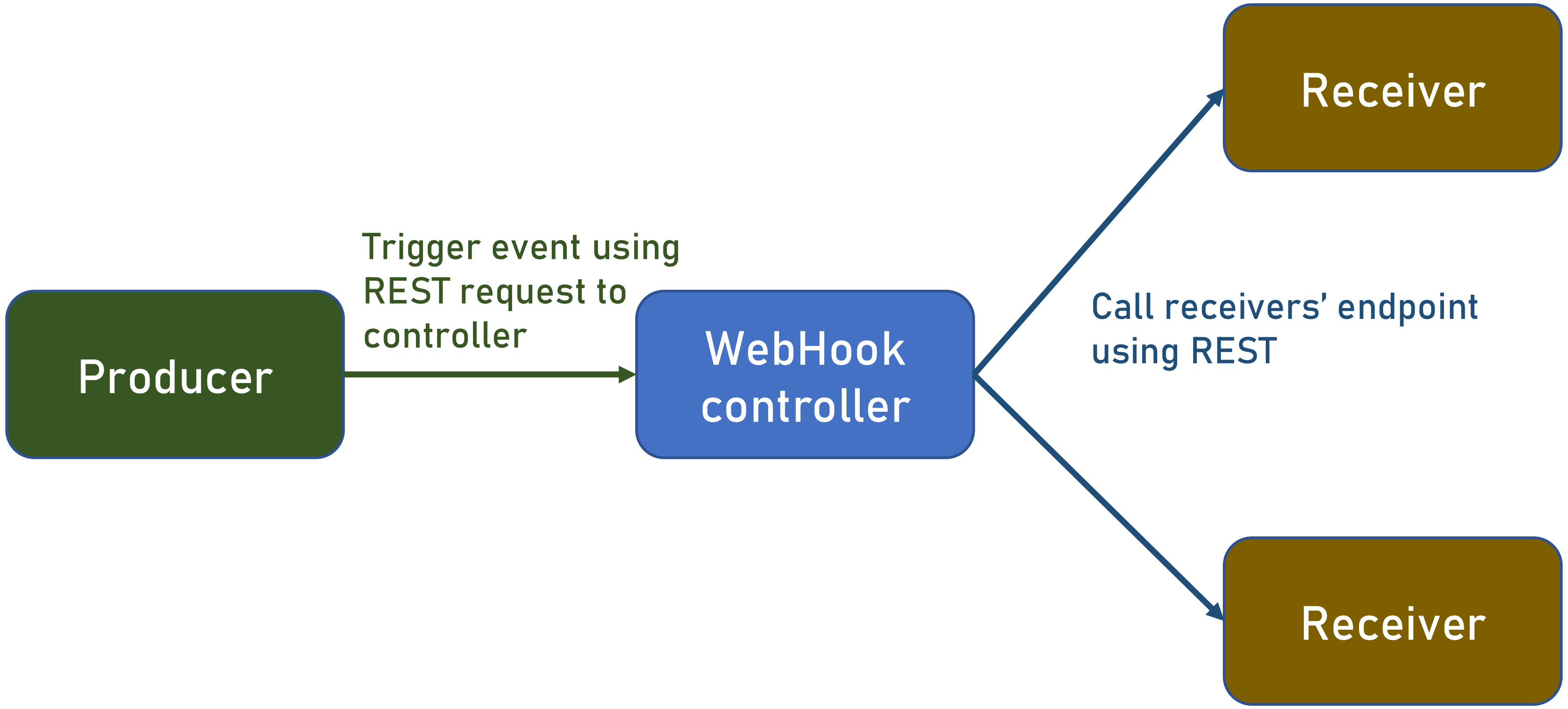
WebHooks

- A standard for publishing events using REST API
- Consumers subscribe to the WebHook engine and register a REST API endpoint that will be called when an event occurs
- The WebHook will call the endpoints when event is triggered
- Easy to implement
- Supported by GitHub, DropBox, PayPal, Stripe and more

WebHooks Flow



WebHooks Flow


Trigger event using REST request to controller
Producer
WebHook controller
Call receivers' endpoint using REST
Receiver
Receiver

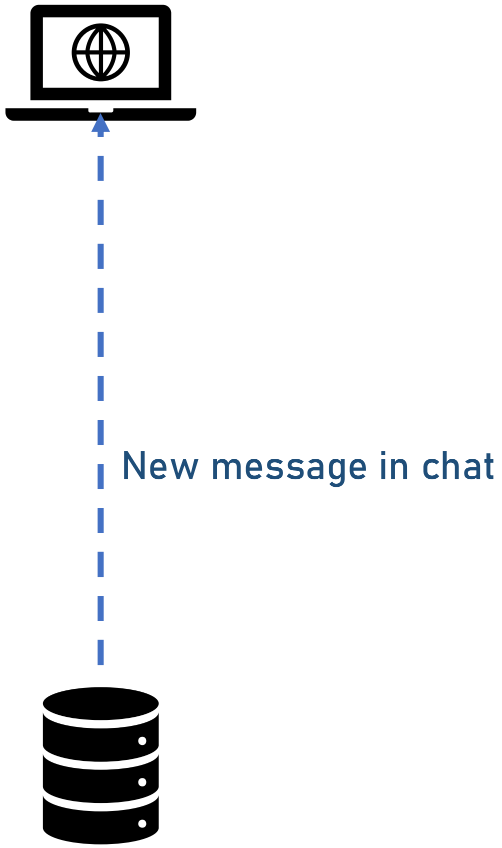
WebHooks Implementation

- Libraries for implementing WebHooks in various platforms
  - E.g. ASP.NET WebHooks
- Websites offering WebHooks:
  - Zapier
  - Ifttt
  - HostedHooks
  - And more…

HTTP Push Notification

- Send events from the server to client(s)
- Great for chats, message notification and more

HTTP Push Notifications



Implementing Push Notifications

- Quite a lot of libraries and frameworks:
  - SignalR
  - Socket.IO
  - gRPC
  - And more…

How to Choose Event Publisher

Use…	When…
Azure Event Grid or similar	- Hosted in the cloud - Need strong integration between backend services
WebHooks	- Receivers expose REST API - Need something simple and quick
HTTP Push Notification	- Need to notify the end user

Remember…

NEVER develop your own channel

Implementing the Producer

- Can be based on any platform
- Needs to be able to communicate with the channel
- Depends on the channel implementation
- Let’s see some examples…

Implementing the Producer

- RabbitMQ
  - Use the RabbitMQ client library for your platform
  - There's one for almost every platform
  - Choose from:
    - https://www.rabbitmq.com/devtools.html

Implementing the Producer

- SignalR
  - Install the SignalR library
  - Configure the Hub
  - Allow connections from clients
  - Define functions that will send messages to clients
  - Optional – create groups to filter messaging

Implementing the Consumer

- Can be based on any platform
- Needs to be able to communicate with the channel
- Depends on the channel implementation
- Let’s see some examples…

Implementing the Consumer

- RabbitMQ
  - Use the RabbitMQ client library for your platform
  - There’s one for almost every platform
  - Choose from:
    - https://www.rabbitmq.com/devtools.html

Implementing the Consumer

- WebHooks
  - Register the consumer using the WebHook REST API
  - Expose REST API that will be called by the WebHooks

Our System

- Introducing:

NOP

The NOise Processing system

NOP

- A system for receiving and processing noise data from external sensors
- The system should:
  - Receive the telemetry
  - Validate it
  - Notify clients on new data

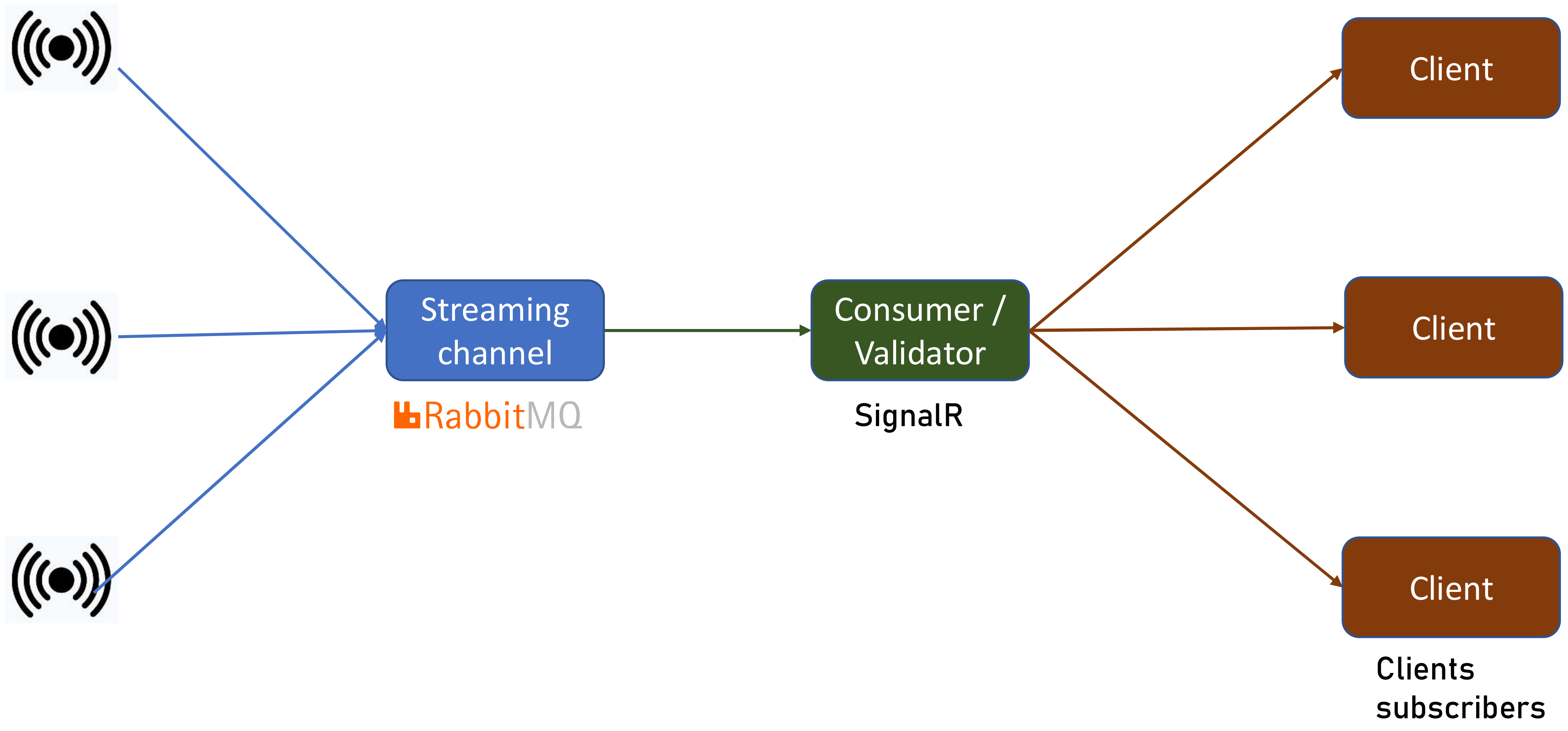
NOP

- The data is a number representing the decibels recorded
- Every sensor sends the data every 30 secs
- …That means that if there are a lot of sensors, there’s quite a lot of data…
- E.g. 1000 sensors => 33 msgs / sec

NOP Design Requirements

Handle load	Streaming engine should be used. Processors pull from the stream when possible
Validate the data	The first thing that should happen after receiving the data
Unknown number of clients	Classic events requirement
No sync users' commands	No synchronous actions required

NOP Event Driven Architecture


Streaming channel
RabbitMQ
Consumer / Validator
SignalR
Client
Client
Client
Clients subscribers