Lund&Bendsen
developing developers

Communication

The Reactive Manifesto

Lund&Bendsen
developing developers

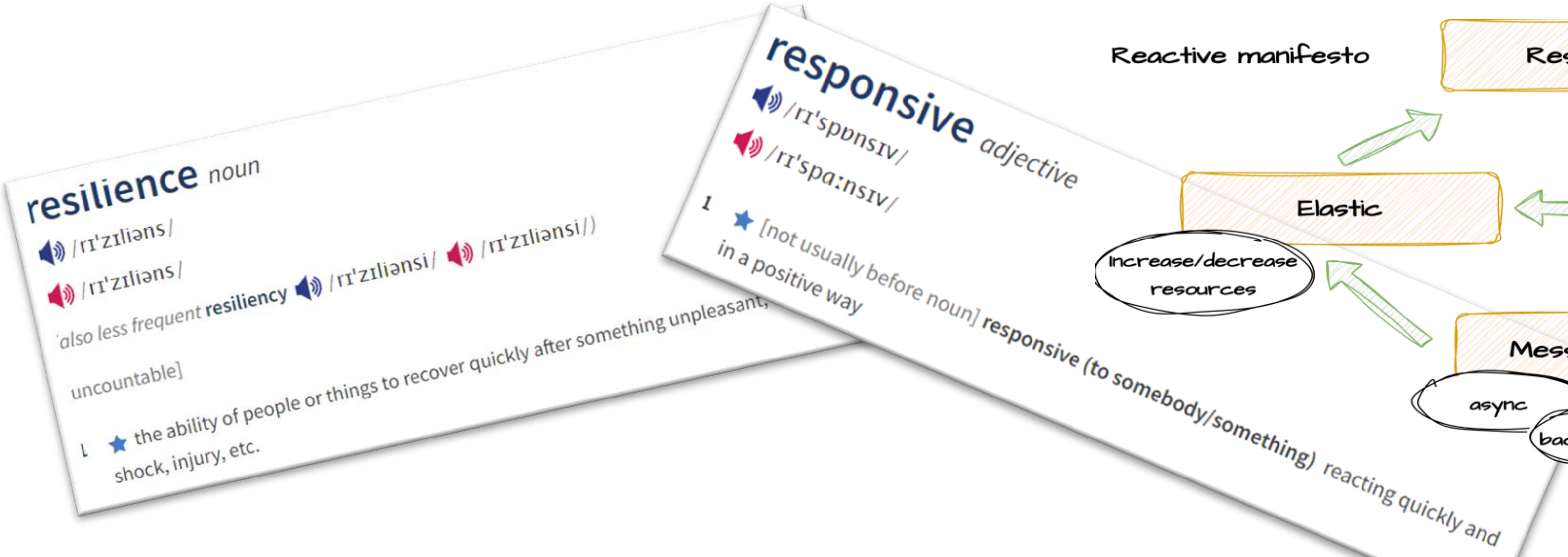
Published in 2014 by Jonas Bonér, ‘The Reactive Manifesto’ formulates several principles and best practices when building distributes systems. The emphasis lies on message-driven, making the components in the system loosely-coupled by favoring an asynchronous programming model. Having loosely coupled components brings several other qualities to the system.

Responsiveness: This principle emphasizes the need for systems to respond promptly to user requests and external events. The system must remain available even under heavy loads.

Resilience: Reactive systems are designed to detect and handle failures gracefully, with mechanisms such as fault tolerance, error recovery, and redundancy.

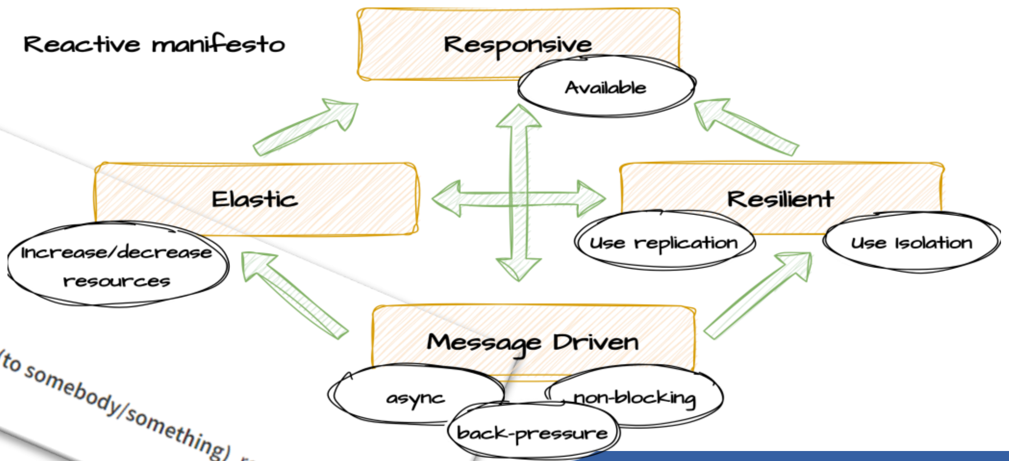
Elasticity: Elasticity refers to the capability of a system to scale its resources up or down dynamically in response to changing workloads

Message-Driven: Message-driven architecture involves using asynchronous message passing between components or services to decouple them and enable better handling of concurrent operations.



resilience *noun*
/rɪˈzɪliəns/
/rɪˈzɪliəns/
(*also less frequent* **resiliency** /rɪˈzɪliənsi/ /rɪˈzɪliənsi/)
[uncountable]
1 ★ the ability of people or things to recover quickly after something unpleasant, shock, injury, etc.

responsive *adjective*
/rɪˈspɒnsɪv/
/rɪˈspɑːnsɪv/
1 ★ [not usually before noun] **responsive (to somebody/something)** reacting quickly and
in a positive way

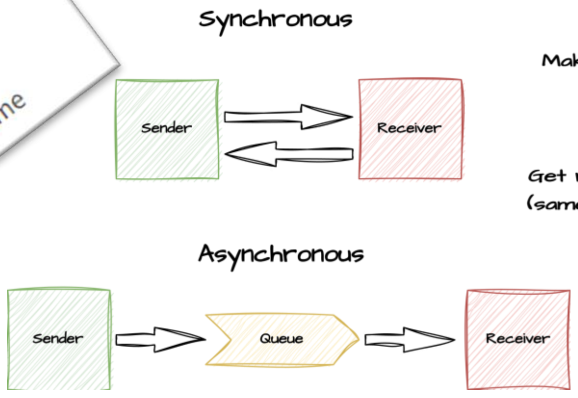


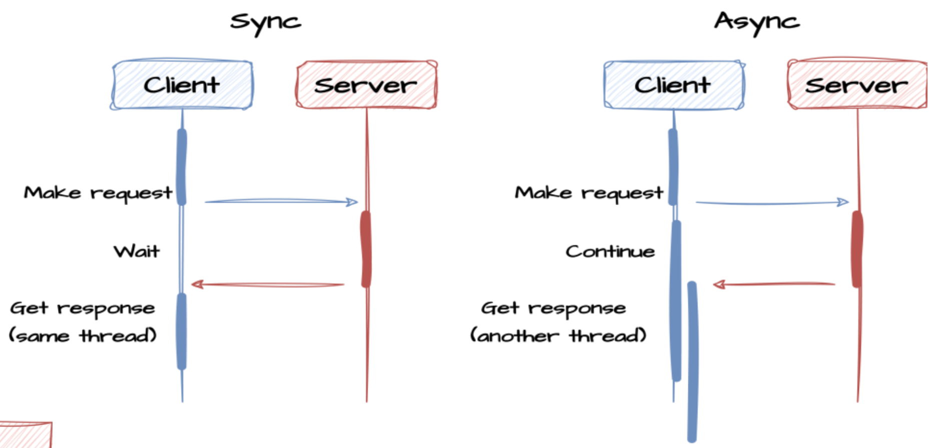
Sync vs Async

Sync Calls: Synchronous calls are blocking and occupies a thread while waiting for the response from the server to complete. This is suitable for simple, straightforward operations where blocking the caller's execution is acceptable and the response time is fast and predictable. Can be combined with various resiliency tools (circuit-breaker, retry and bulkhead) to reduce risk of thread starvation

Async Calls: Asynchronous calls are preferred for tasks that involve waiting for external resources, such as I/O operations (e.g., file I/O, network requests), user interactions in graphical user interfaces (GUIs), and tasks where responsiveness and scalability are critical. Async calling code is typically harder to comprehend as it relies on Futures / callbacks to communicate the result. Results are often handled on a different thread than the calling thread

synchronous *adjective*
/ˈsɪŋkrənəs/
/ˈsɪŋkrənəs/
(specialist)
★ happening or existing at the same time





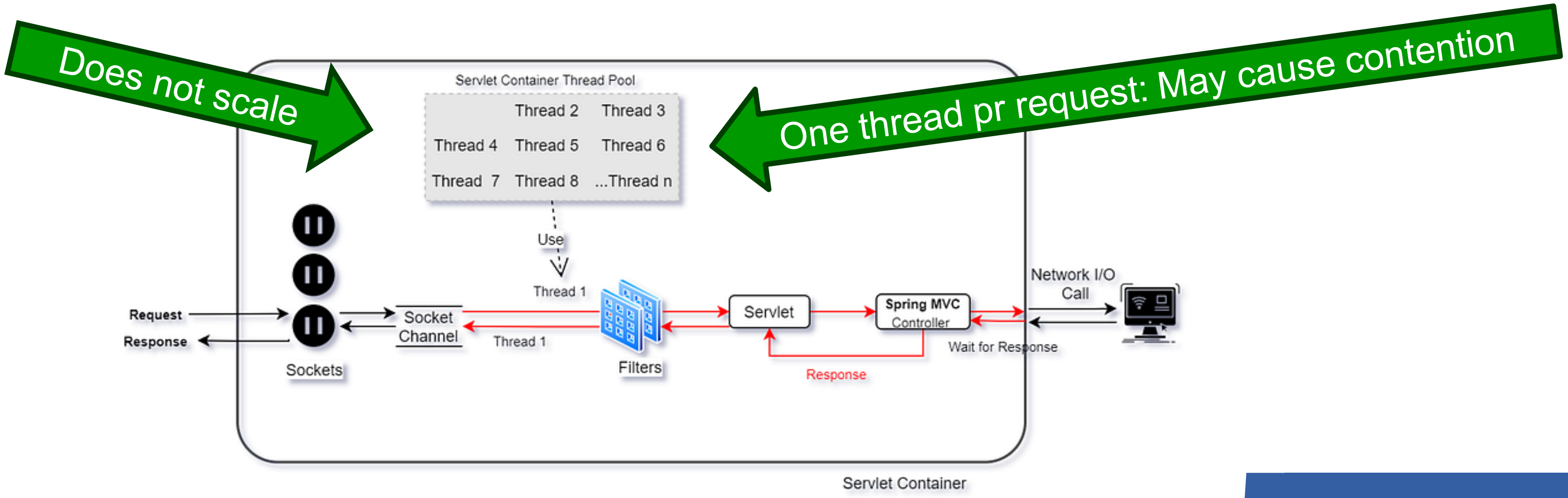
Synchronous/blocking

Blocking: In a synchronous call, the caller waits for the called function to complete before proceeding with further execution. The execution of the caller is blocked until the called function finishes its task. A timeout can often be specified to avoid endless waiting

Response Time: Since synchronous calls block the caller, they can lead to unforeseeably long processing time and impact other parts of the caller code. If synchronous calls are part of a transaction or other time-critical operations, these operations may time out.

Resource Utilization: Synchronous calls can be resource-intensive, as resources such as CPU and memory may be tied up while waiting for the called function to return.

Simplicity: Synchronous calls are often simpler to reason about and implement, as they follow a straightforward execution flow.



Asynchronous/non-blocking

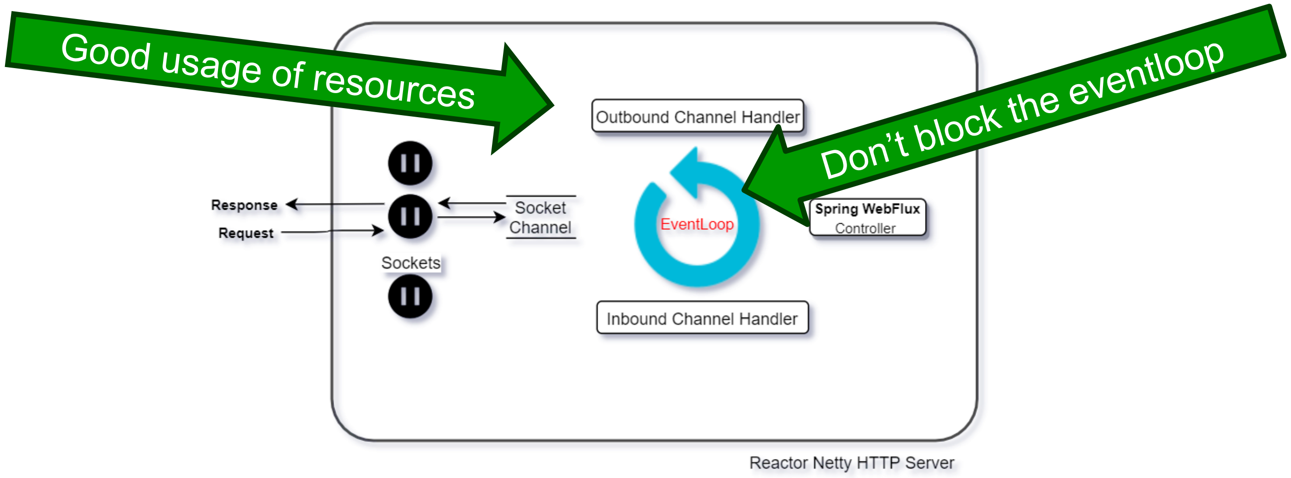
Non-Blocking: In an asynchronous call, the caller does not wait for the called function to complete. Instead, the caller continues with its execution, and the called function runs in the background or executes in a remote process and the response is handled in a separate process.

Resource Efficiency: Asynchronous calls are typically more resource-efficient because they do not block resources while waiting. This can result in better scalability and responsiveness.

Complexity: Implementing and managing asynchronous code can be more complex than synchronous code. It often involves handling callbacks, promises, or async/await constructs to manage the flow of execution.

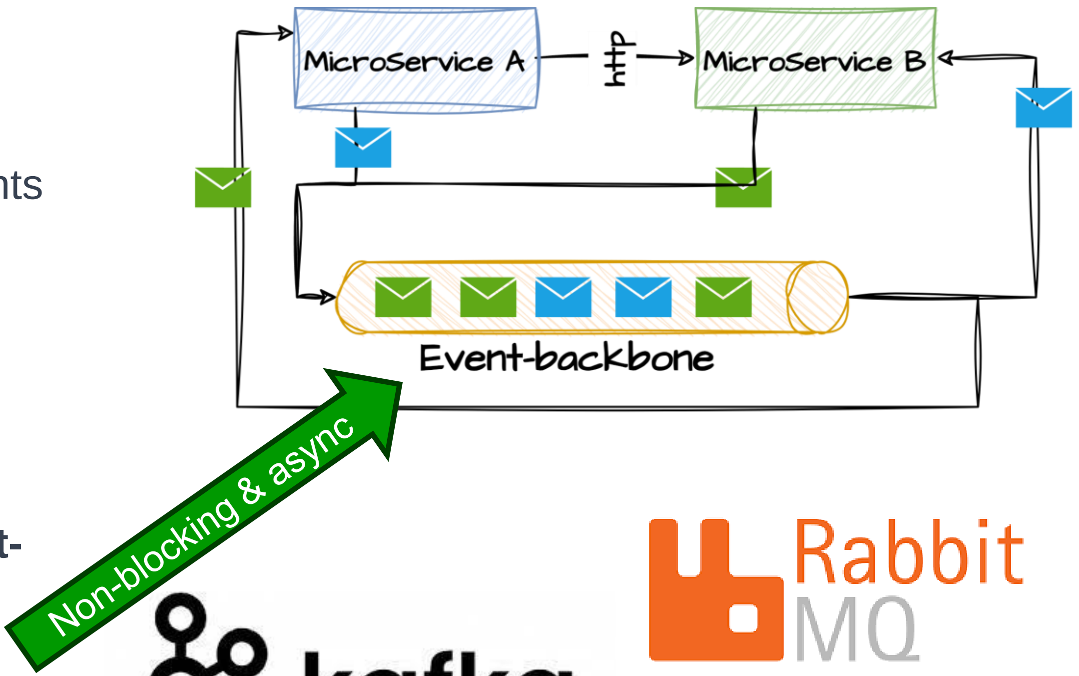
Concurrency Control: Asynchronous code may require additional attention to concurrency control to prevent issues like race conditions and data inconsistency when multiple tasks run concurrently.

Error Handling: Handling errors in asynchronous code can be more challenging due to the distributed nature of async operations. Proper error handling and debugging is crucial.

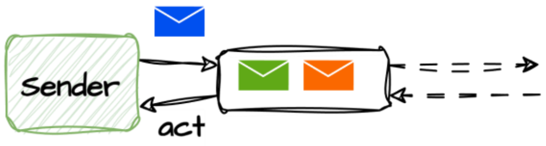


Event backbone (EB)

An event backbone, often referred to as an event bus or an event broker, is a core component of a distributed message driven architecture. It serves as the central infrastructure that facilitates the flow of events (i.e., discrete pieces of information about something that happened) within a software system or across multiple systems. The primary purpose of an event backbone is to ensure the reliable and efficient communication of events between event producers and event consumers. **The event-backbone may offer additional services such as pub/sub, durability, replay of messages, priority queues and advanced routing**.



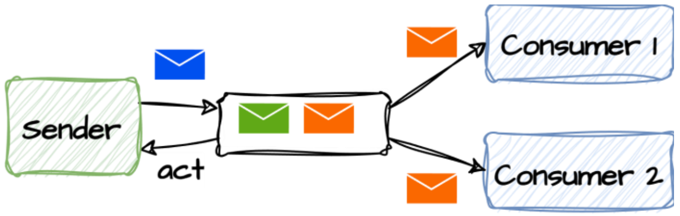
Why use an Event Backbone?



Reduce knowledge of systems

Senders and consumers are decoupled.

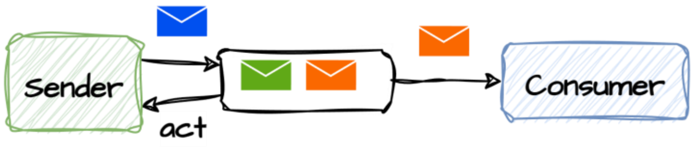
They don't know about each other



Parallel processing

With pub/sub notify multiple consumers

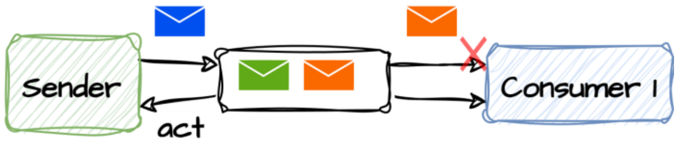
Can scale to run processes in parallel



Reduce pressure off consumers

Protect consumers from being overloaded

by buffering up messages



Prevent message being lost

Messages can be reprocced after failures

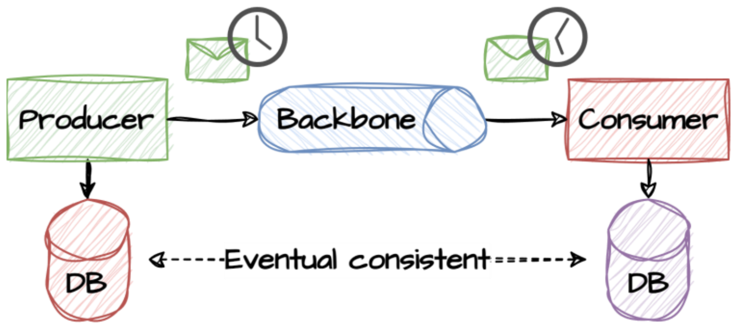
EB challenges

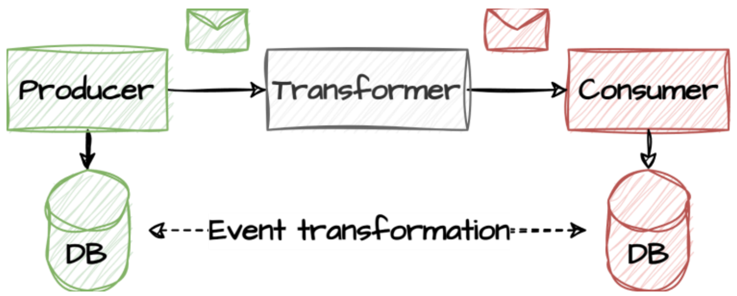
Data (eventual) Consistency:

Eventual consistency (or deferred execution) is a consistency model used in distributed computing to achieve high availability. If no new updates are made to a given data item, eventually all accesses to that item will return the last updated value. Common challenges is ‘read your own write’, which states that the system guarantees that, once an item has been updated, any attempt to read the record by the same client will return the updated value

Data Transformation and Validation:

Microservices often have different data requirements and formats. Ensuring that data is properly transformed and validated as it moves between services is essential to prevent data-related issues.





EB challenges

Data Ownership:

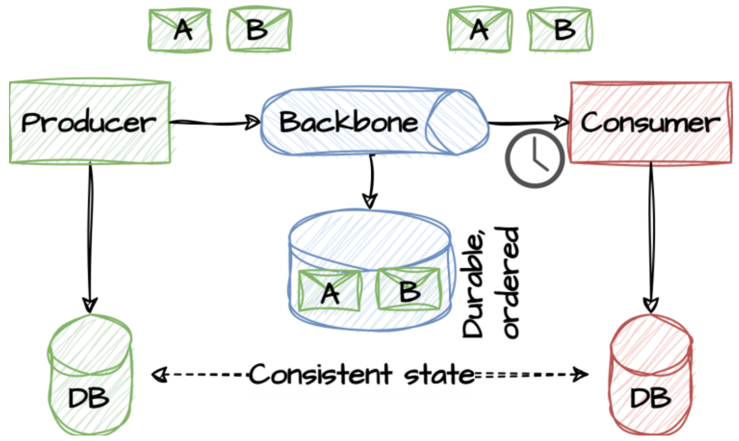
Determining which microservice is the authoritative source of certain data can be challenging. This is especially true in cases where multiple microservices may have their own copies of the same data.

Message Ordering:

Ensuring the correct order of messages or events is vital in many microservices applications. Guaranteeing that events are processed in the right sequence can be challenging, especially when dealing with distributed messaging systems.

Message Durability:

Guaranteeing the durability of messages or events is crucial. This may require using durable message queues or event stores.



EB challenges

Back Pressure:

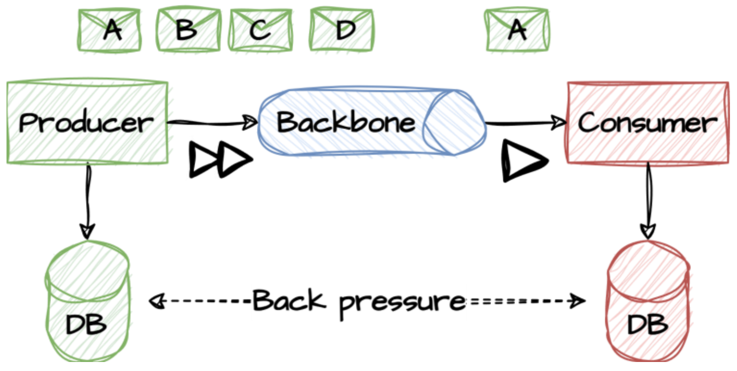
When data is flowing into a microservice faster than it can process, it may use back-pressure to make the producing component / EB slow down. Managing back pressure and ensuring that services can handle spikes in traffic is a challenge.

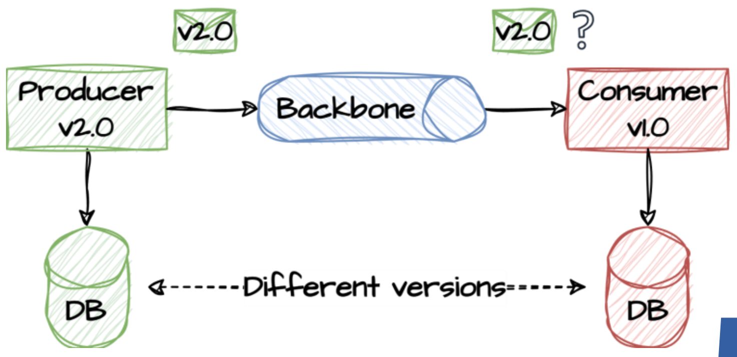
Data Privacy and Security:

Ensuring data privacy and security is a challenge when data flows between services. Implementing proper access controls, encryption, and data masking is crucial to protect sensitive information.

Data Versioning:

As microservices evolve independently, data schemas may change. Managing data versioning and handling backward compatibility is important to avoid breaking existing dataflows.





EB challenges

Error Handling:

Handling errors that occur during dataflow is essential. Microservices should be designed to handle failures gracefully, including retries, dead-letter queues, and error notifications.

Monitoring and Debugging:

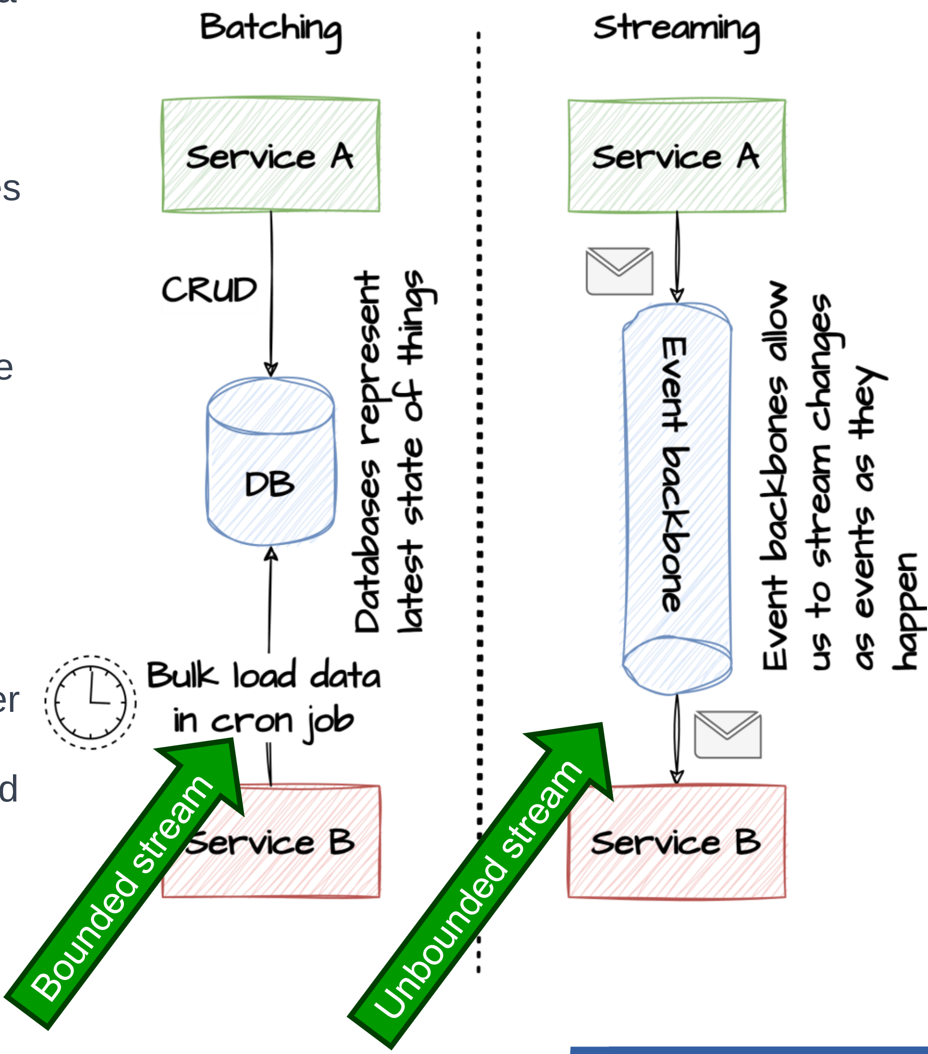
Understanding the flow of data between microservices and diagnosing issues can be difficult. Comprehensive monitoring and logging are necessary to trace the flow of data and troubleshoot problems.

Data at-rest or in-transit

With a message-driven architecture focus changes from data at-rest (stored in a data-store) to data in-transit (infinite stream of messages)

Database (at-rest): Data at rest is typically held in data-stores such as relational database tables. Often, these tables hold information about the current state of the entity that the table models. Data are written to the database using CRUD operations through a well-defined API. Consumers of the data are not notified about changes to the entities. Exchange of data between services happen through batch processing. This data can be considered a **bounded-stream** where computations happen at the end of the stream (when the batch runs)

Event backbone (in-transit): No central database of the domain entities exist. Instead, changes are communicated and exchanged as an **unbounded-stream** of messages over an event backbone. Computations happen on the stream of messages as they arrive. Consumers of messages may build up local state that project the entities in ways that are optimized for that consumer.


Batching
Service A
CRUD
DB
Databases represent latest state of things
Bulk load data in cron job
Service B
Bounded stream
Streaming
Service A
Event backbone
Event backbones allow us to stream changes as events as they happen
Service B
Unbounded stream