Damilah

We build great software

...with the best people

Building Resilient Systems: Leveraging Event Driven Design for Fault Tolerance and Adaptability

MSc. Konstantin Bogdanoski

What is EDD?

also known as Event Driven Architecture - EDA

Modern architecture

Event Driven Design is a software architecture pattern, built from small, decoupled services, that publish, consume (listen) or route events

Events

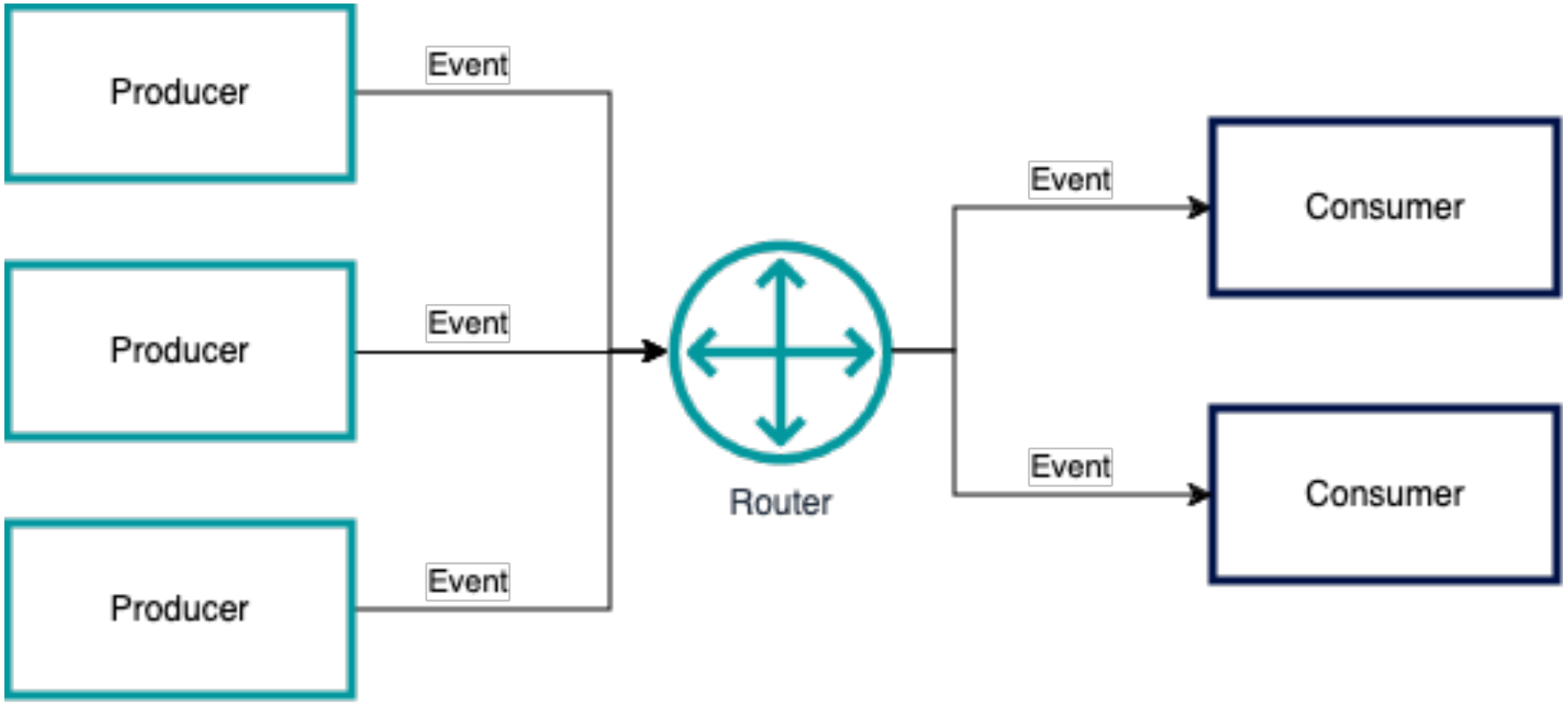
Events represent changes in a service's state, updates included. Examples: adding files to a shopping cart, an order becoming ready to ship.

Important note

EDA promotes loose coupling between services in the microservice system, thus making it easier to scale, update, and independently deploy separate components of the system

What is EDD?

also known as Event Driven Architecture - EDA



Damilah

Loose coupling

decoupled architecture

General

Decoupled architecture is an approach which allows components of a complex system to be completely autonomous and unaware of each other. Cloud computing is sometimes said to have a decoupled architecture because the cloud provider manages the physical infrastructure, but not the applications or data hosted on it

Software

Loose coupling is an approach to connect services in a microservice system, or connect components in a network, so that those services/components depend on each other in the least extent practicable. Coupling refers to the degree of direct knowledge that one element has of another

Loose coupling

loosely coupled vs tightly coupled systems

Tightly coupled systems

Various components are linked closely to each other, such that one change in one component, likely affects the others. All the components work together to generate an output – thus the whole system acts as one unit. In a data centric application, tight coupling might perform better than a loosely coupled architecture, since any change to any component, likely affects the overall output.

Loosely coupled systems

Interaction between services/components is smaller compared to a tightly coupled system. Change to a service is unlikely to affect other services. Nodes in a distributed computer system is an example of loose coupling, where each machine has its' own processing power, memory, and they only communicate when necessary

Loose coupling

loosely coupled vs tightly coupled systems

Damilah

Tight Coupling

1. **Simplicity** – Tight coupling can simplify certain aspects of development by reducing the need for explicit interfaces and abstractions. In simple systems or small projects, tightly coupled components may lead to faster initial development.
2. **Performance** – In some cases, tight coupling can lead to better performance due to reduced overhead from communication between components. Direct method calls or shared memory access can be more efficient than indirect communication mechanisms used in loosely coupled architectures.
3. **Control** – Tight coupling provides more direct control over the interactions between components, making it easier to manage state and enforce business rules. This level of control can be advantageous in systems where predictability and deterministic behavior are critical.
4. **Synchronization** – Tight coupling ensures that related components are synchronized and operate in lockstep, which can be beneficial for real-time systems or scenarios where consistency is paramount.
5. **Simplified Deployment** – In tightly coupled architectures, deploying the entire system as a single unit can simplify deployment and configuration management. This can be advantageous in certain deployment environments or for monolithic applications.

Loose Coupling

1. **Modularity** – Loose coupling promotes modular design, allowing components to be developed, tested, and maintained independently. This enhances code organization and reusability.
2. **Flexibility** – Changes to one component have minimal impact on other components, making the system more adaptable to evolving requirements. This enables easier upgrades, extensions, and modifications without causing widespread disruptions.
3. **Scalability** – With loosely coupled components, it's easier to scale parts of the system independently. This allows for better resource utilization and performance optimization, especially in distributed or cloud-based architectures.
4. **Maintainability** – Loose coupling reduces the risk of unintended side effects when making changes to the system. Developers can modify or replace individual components without affecting the overall stability or functionality of the system.
5. **Parallel Development** – Teams can work on different components concurrently without causing conflicts or dependencies. This speeds up development cycles and fosters collaboration among team members.

EDD Benefits

Damilah

Scale/Fail independently

Decoupling the components, allows scaling them up or down, depending on the need – as the scale of events changes.
Decoupling the components, allows the components to fail independently to each other – one failed component will not affect the runtime of others.

Developer agility

With event-driven architecture and event routers, developers no longer need to write custom code to poll, filter, and route events. An event router automatically filters and pushes events to consumers. The router also removes the need for heavy coordination between producer and consumer services, thus increasing developer agility.

EDD Benefits

Damilah

Extensibility

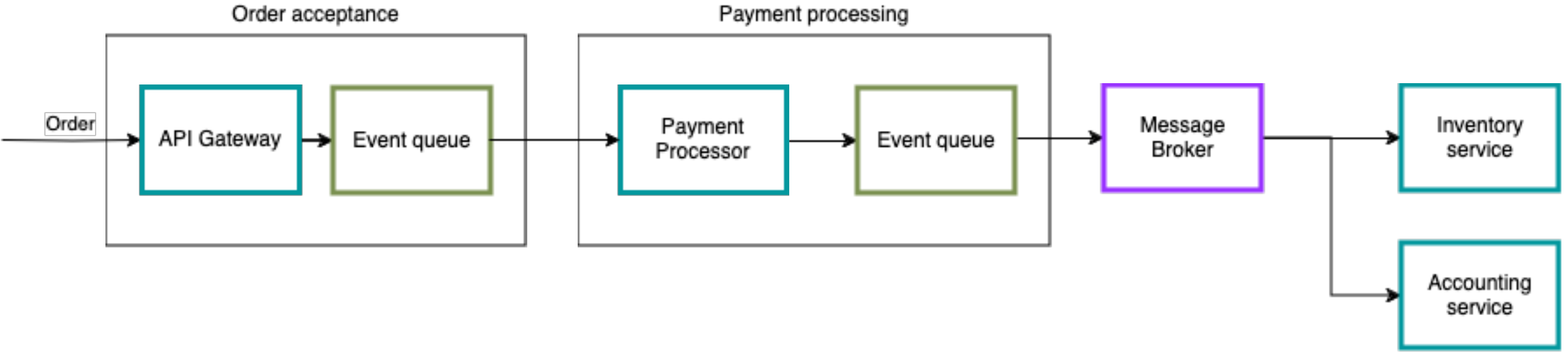
Different teams can extend features and functionalities without impacting already existing services. By publishing events, the service can be integrated to listen to other events, and produce them, so future applications can easily integrate as consumers

Complexity

Microservices enable developers and architects to decompose complex workflows. A workload that might be complex to manage and orchestrate in a monolith becomes a series of simple, decoupled services that are managed independently and communicate asynchronously through event messages.

Example

Damilah



EDD Benefits

Damilah

Audit

An event router in an event-driven architecture acts as a centralized location to audit your application and define policies. These policies can restrict who can publish and subscribe to a router and control which users and resources have permission to access your data. You can also encrypt your events both in transit and at rest.

Costs

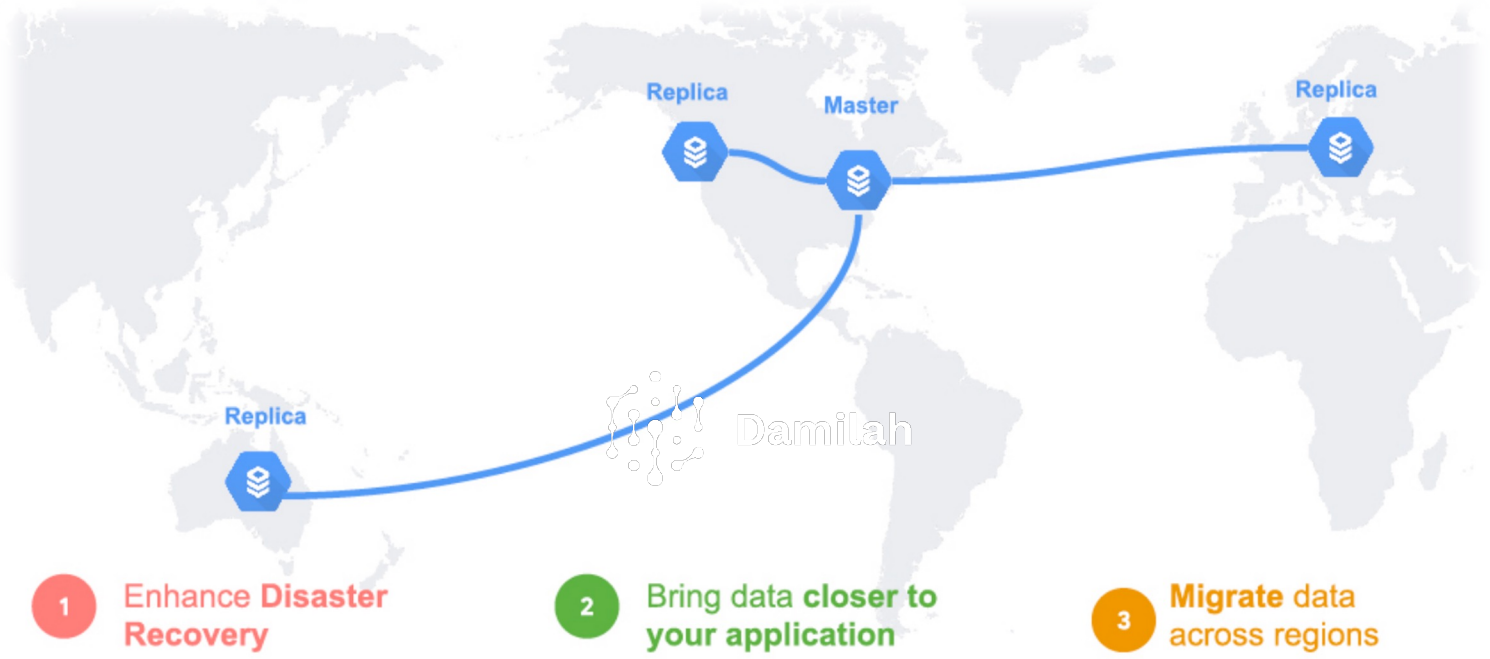
EDAs are push-based, so everything happens on demand as the event presents itself in the router. This way, you're not paying for continuous polling to check for an event. This means less network bandwidth consumption, less CPU utilization, less idle fleet capacity, and fewer SSL/TLS handshakes.

When to use EDD

Damilah

Cross-account and Cross-region data replication

When there is a need to coordinate systems between teams operating in and deploying across different regions and accounts. By using an event router to transfer data between systems, you can develop, scale, and deploy services independently from other teams.

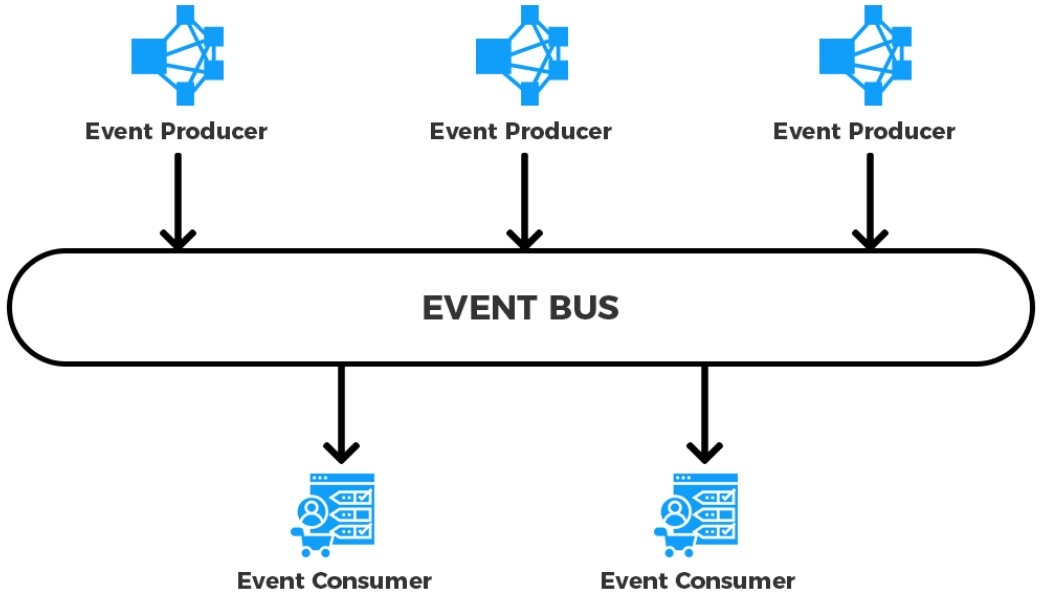


When to use EDD

Damilah

Parallel processing

If your systems need to operate in response to a certain event, an EDD is a good choice, since there isn't a need to write custom code for each consumer, because the router will push the event to all systems, which can process it in parallel, each with a different purpose.



When to use EDD

Damilah

State monitoring

Rather than continuously checking on your resources, you can use an event-driven architecture to monitor and receive alerts on any anomalies, changes, and updates. These resources can include storage buckets, database tables, serverless functions, compute nodes, and more.

ATERA

Grafana

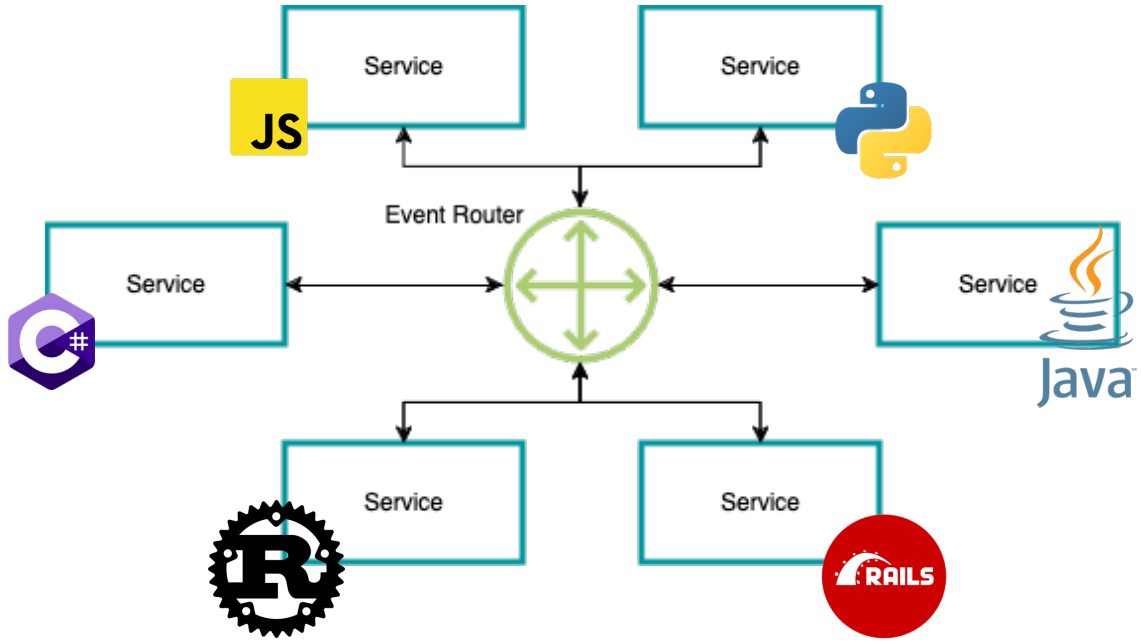
DATADOG

When to use EDD

Damilah

Integration of heterogeneous systems

If you have systems running on different stacks, you can use an event-driven architecture to share information between them without coupling. The event router establishes indirection and interoperability among the systems, so they can exchange messages and data while remaining agnostic.



Should you use EDD?

Damilah

Event-driven architectures are ideal for improving agility and moving quickly. They're commonly found in modern applications that use microservices, or any application that has decoupled components. When adopting an event-driven architecture, you may need to rethink the way you view your application design. To set yourself up for success, consider the following:

- **The durability of your event source**. Your event source should be reliable and guarantee delivery if you need to process every single event.
- **Your performance control requirements**. Your application should be able to handle the asynchronous nature of event routers.
- **Your event flow tracking**. The indirection introduced by an event-driven architecture allows for dynamic tracking via monitoring services, but not static tracking via code analysis.
- **The data in your event source**. If you need to rebuild state, your event source should be deduplicated and ordered.

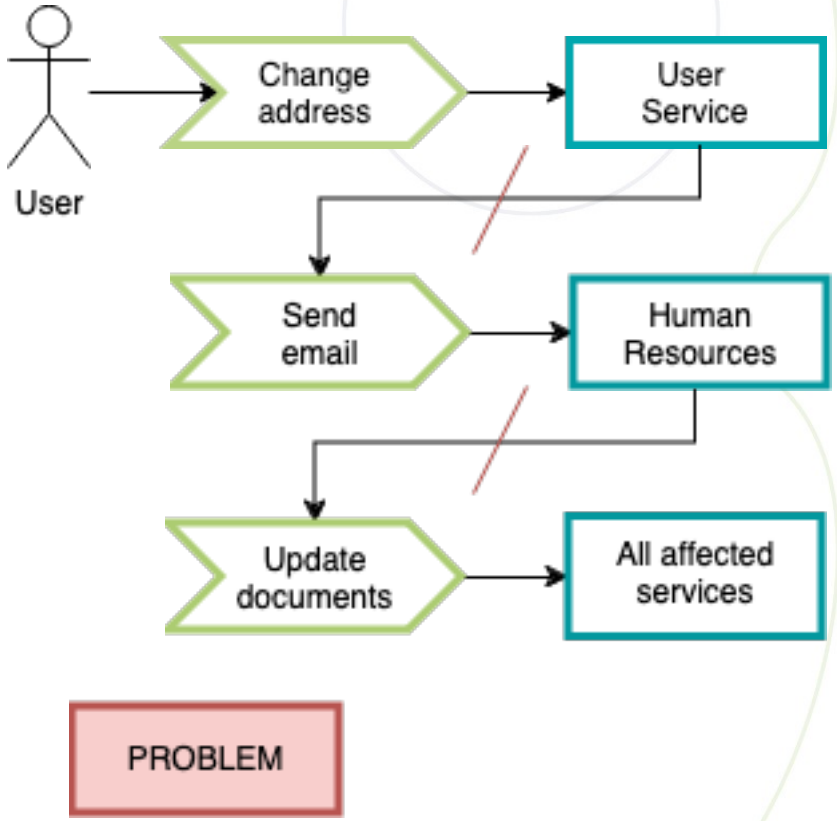
Event Driven Patterns

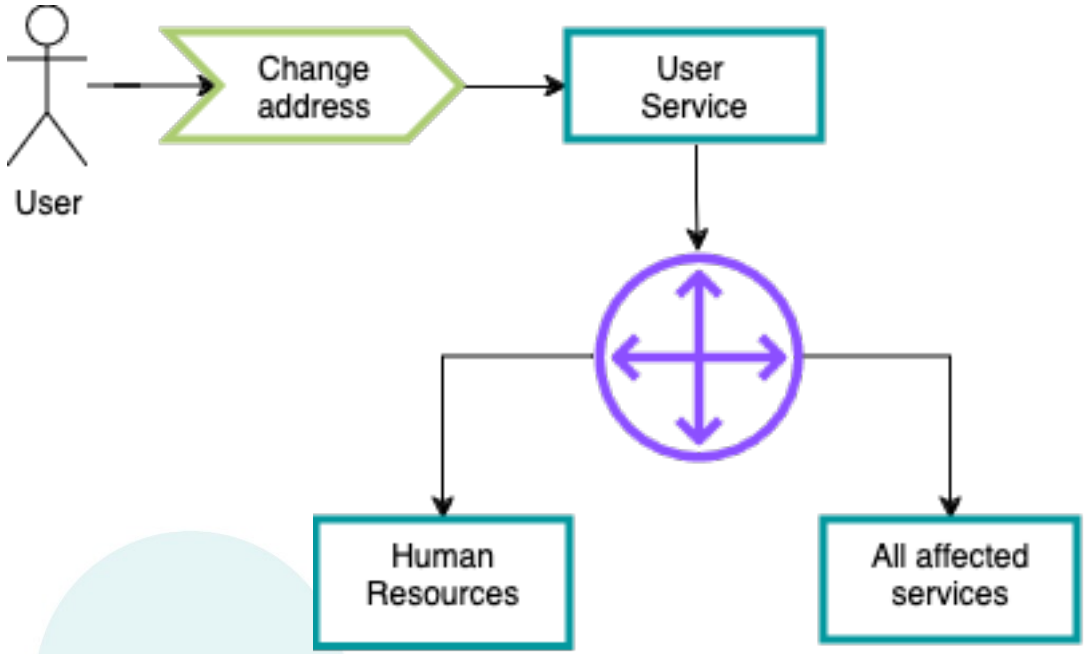
commonly used EDA patterns

Damilah

Event Notification	Event Carried State Transfer	Event Sourcing	CQRS

Event Notification





Event Notification

Damilah

Events are used to notify other microservices in the system that an interesting change has occurred. The notification event is small and concise as it only contains a reference to the state that was changed. This is likely some unique identifier, supported with some helpful metadata that can help consumers determine if the change is relevant to their responsibility.

This provides loose coupling, with having the complexity of the order microservice is also greatly reduced because explicit awareness about other domains and components in the system is no longer needed.

This pattern provides **high availability** of the system.

Events or Commands.

- Events describe a change in the system's state or a notable occurrence
- Commands represent an intention or instruction for the system to perform an action

Event Notification

Damilah

cons

1. Complexity: Implementing event-driven architectures can introduce complexity, especially when managing event propagation, handling event consistency, and ensuring reliable delivery.

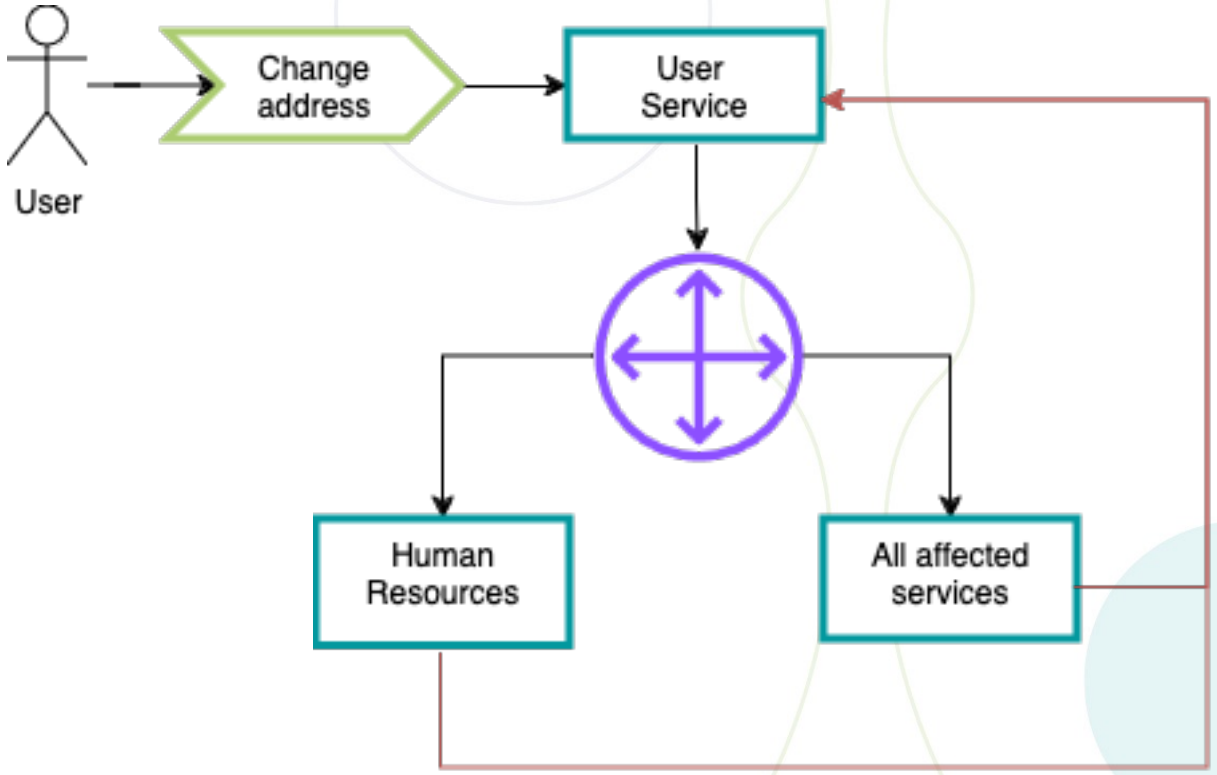
2. Debugging: Debugging event-driven systems can be more challenging compared to traditional request-response architectures. Tracking the flow of events and understanding the sequence of actions can be complex.

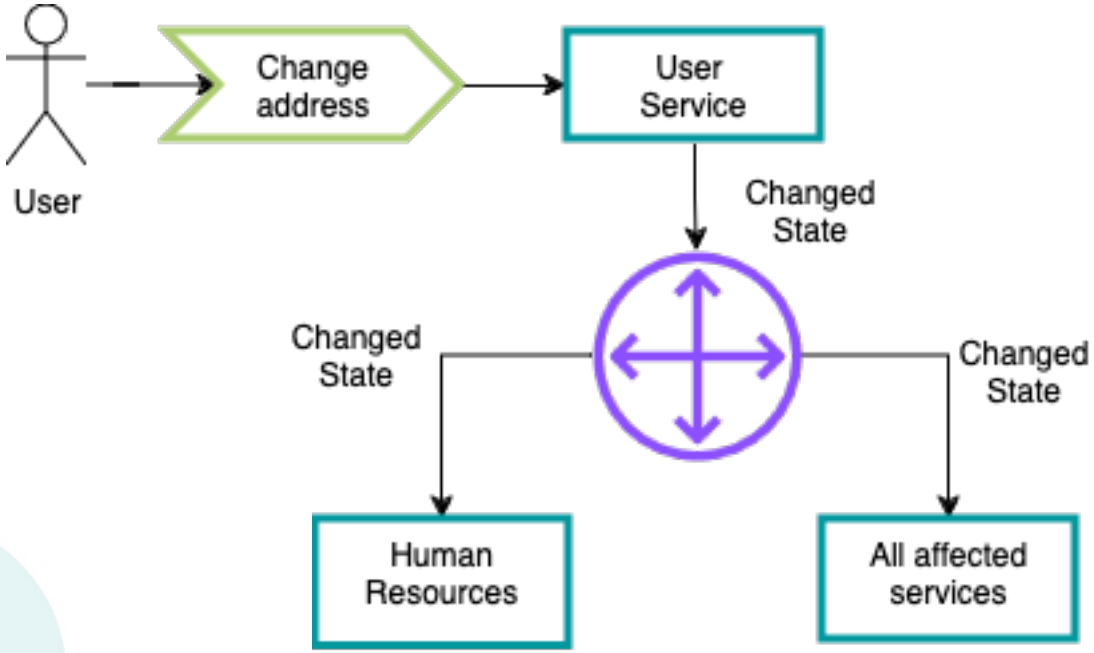
3. Eventual Consistency: Event-driven architectures often prioritize eventual consistency over immediate consistency.

4. Error Handling: Handling errors and failures in event-driven systems requires careful consideration. Components must be resilient to failures such as network partitions, service outages, or unexpected behaviors of downstream systems.

Event Carried State Transfer

Damilah





Event Carried State Transfer

Damilah

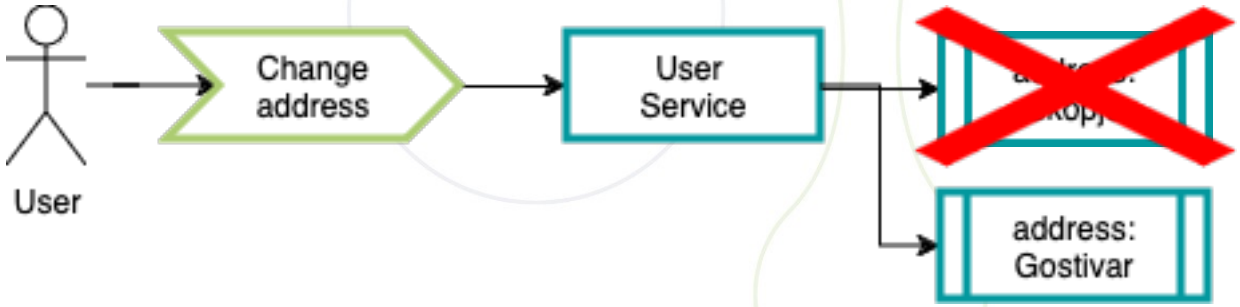
This pattern facilitates updating clients of a system without requiring them to directly contact the source system for further work. For instance, a customer management system might emit events whenever customer details change, allowing recipients to update their own copies of customer data without needing to communicate with the main system.

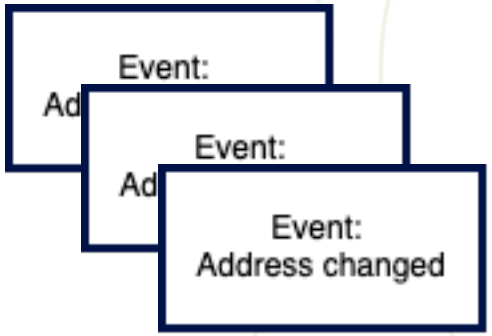
While this approach involves increased data duplication and complexity on the recipient's end, it offers benefits such as **greater resilience**, **reduced latency**, and **alleviated load on the source system**.

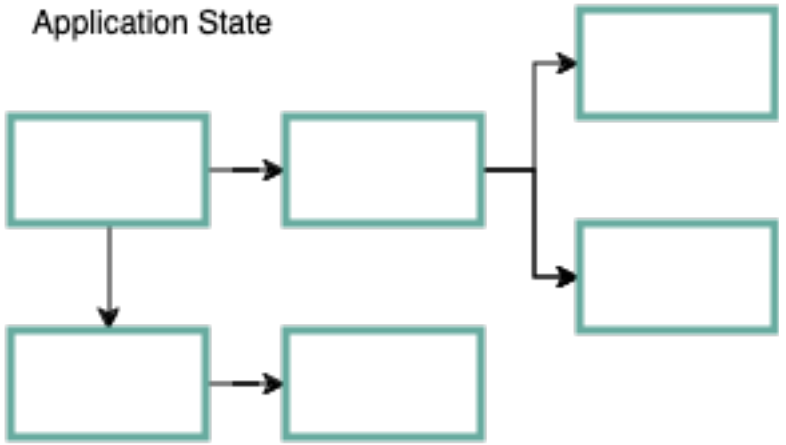
However, managing state on the recipient side becomes more intricate compared to simply querying the sender for information when necessary.

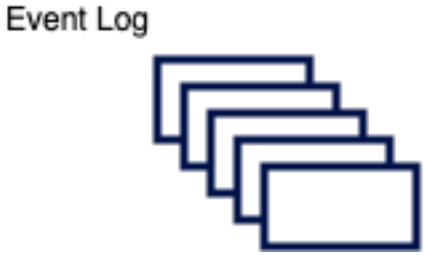
As the previous pattern, this pattern also offers **high availability**, due to the decoupling of the services.

Event Sourcing









Event Sourcing

Damilah

The Event Sourcing pattern defines an approach to handling operations on data that's driven by a sequence of events, each of which is recorded in an append-only store.

Application code sends a series of events that imperatively describe each action that has occurred on the data to the event store, where they're persisted.

Each event represents a set of changes to the data.

Event Sourcing

Damilah

- Events are immutable and can be stored using an append-only operation.
- Events are simple objects that describe some action that occurred, together with any associated data that's required to describe the action represented by the event. Events don't directly update a data store. They're simply recorded for handling at the appropriate time.
- Events typically have meaning for a domain expert, whereas object-relational impedance mismatch can make complex database tables hard to understand. Tables are artificial constructs that represent the current state of the system, not the events that occurred.
- Event sourcing can help prevent concurrent updates from causing conflicts because it avoids the requirement to directly update objects in the data store. However, the domain model must still be designed to protect itself from requests that might result in an inconsistent state.

Event Sourcing

Damilah

- The append-only storage of events provides an audit trail that can be used to monitor actions taken against a data store. It can regenerate the current state as materialized views or projections by replaying the events at any time, and it can assist in testing and debugging the system.

- The event store raises events, and tasks perform operations in response to those events. This decoupling of the tasks from the events provides flexibility and extensibility.

Event Sourcing

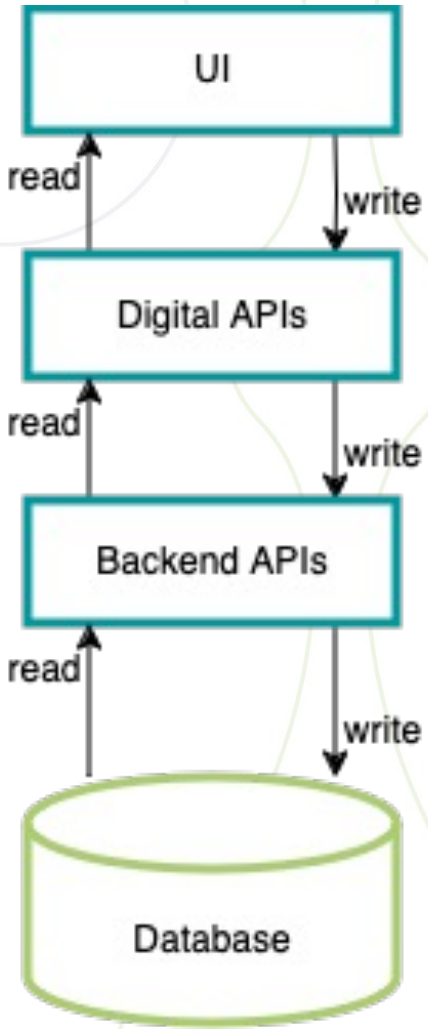
Damilah

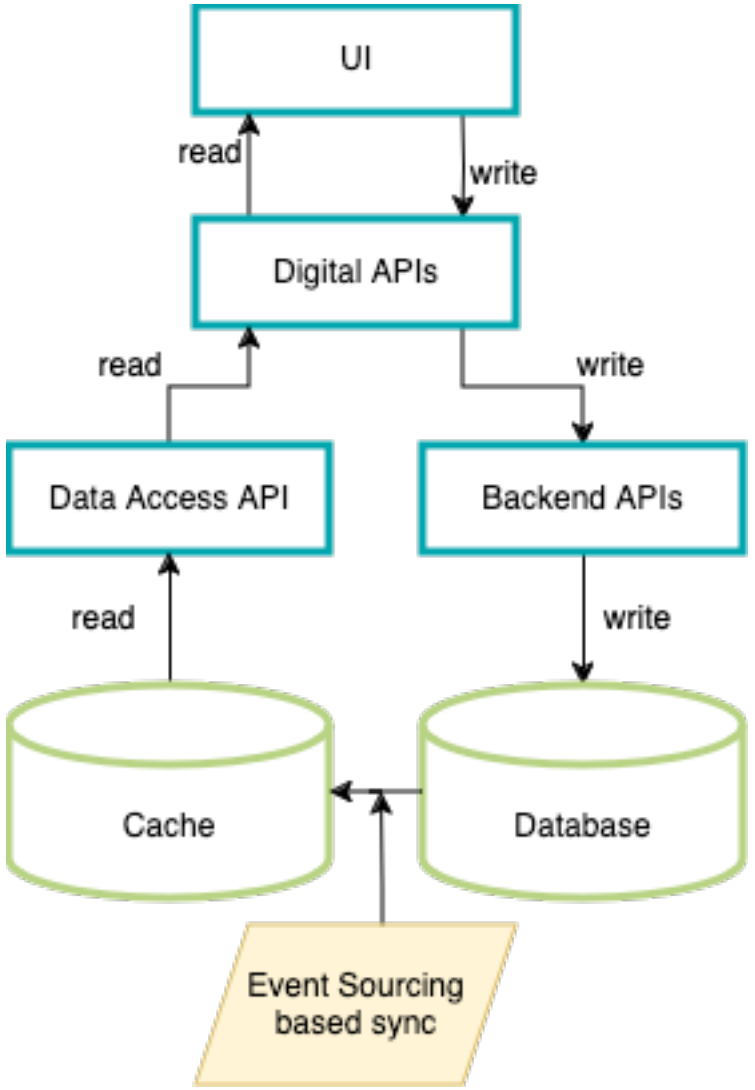
cons

- Requires an extremely efficient network infrastructure given the inherent time sensitivity of the pattern and the susceptibility to potential latency issues.
- Requires a reliable way to control message formats, for example, using a schema registry.
- Different events will contain different payloads. There needs to be a single source of truth for defining and determining message formats for a particular event.
- Code changes cause previous events to not be used for restoring the system's state

CQRS

Damilah





CQRS

Damilah

CQRS stands for **Command** and **Query Responsibility Segregation**, a pattern that separates read and update operations for a data store.

Physical separation of the read data from the write data. In that case, the read database can use its own data schema that is optimized for queries. For example, it can store a materialized view of the data, to avoid complex joins or complex ORM mappings. It might even use a different type of data store. For example, the write database might be relational, while the read database is a document database.

If separate read and write databases are used, they must be kept in sync. Typically, this is accomplished by having the write model publish an event whenever it updates the database.

CQRS

pros

- **Independent scaling**. CQRS allows the read and write workloads to scale independently and may result in fewer lock contentions.
- **Optimized data schemas**. The read side can use a schema that is optimized for queries, while the write side uses a schema that is optimized for updates.
- **Security**. It's easier to ensure that only the right domain entities are performing writes on the data.
- **Separation of concerns**. Segregating the read and write sides can result in models that are more maintainable and flexible. Most of the complex business logic goes into the write model. The read model can be relatively simple.
- **Simpler queries**. By storing a materialized view in the read database, the application can avoid complex joins when querying.

CQRS

cons

- **Complexity**. The basic idea of CQRS is simple. But it can lead to a more complex application design, especially when using the Event Sourcing pattern.

- **Messaging**. Although CQRS does not require messaging, it's common to use messaging to process commands and publish update events. In that case, the application must handle message failures or duplicate messages.

- **Eventual consistency**. If you separate the read and write databases, the read data may be stale. The read model store must be updated to reflect changes to the write model store, and it can be difficult to detect when a user has issued a request based on stale read data.

Additional note: *CQRS is often improperly used!*

Damilah

Publish - Subscribe

Point-to-Point

Request-Reply

Event Streaming

Event Driven Communication Patterns

Publish-Subscribe

Damilah

Publish-subscribe is a communication pattern that decouples applications by having them publish messages to an intermediary broker rather than communicating directly with consumers (as in point-to-point). This approach introduces an asynchronous mode of communication between publishers and subscribers, offering increased scalability, improved reliability, and the ability to defer processing with the use of a queue on the broker.

The publish-subscribe pattern is often used in event-driven architecture for building event-driven microservices architectures and real-time streaming. It can help decouple different parts of an application, enable flexible scaling, and support real-time processing of large volumes of data.

Point-to-Point

Damilah

If you need to deliver a message to a specific recipient, the publish-subscribe mode isn't necessary. In such cases, the point-to-point message exchange pattern is used to deliver the message to a single recipient in what's called a "one-to-one" exchange.

It is also possible that several senders send messages to the same recipient, a "*many-to-one*" exchange. In these situations, the endpoint typically takes the form of a named *queue*. If several receivers are connected to the queue, only one of them will receive the message. **Apache ActiveMQ** (Message Broker) is a software which implements this pattern (same as RabbitMQ, AmazonMQ, etc.)

Request-Reply

Damilah

While both publish-subscribe and point-to-point communication modes facilitate one-way communication, a request-reply pattern is necessary when your goal is specifically to get a response from the recipient. This pattern ensures that the consumer sends a response to each consumed message, with the implementation varying from the use of a return address to sending a response to a separate endpoint.

In essence, we are examining two distinct messaging operations: one where the requestor transmits a message using either a point-to-point (**deliver**) or publish-subscribe (**broadcast**) mode, and the other where the receiver replies to the request in a point-to-point manner directed to the sender. It is important to note that the reply exchange is always point-to-point, as the intent is solely to return the response to the requestor.

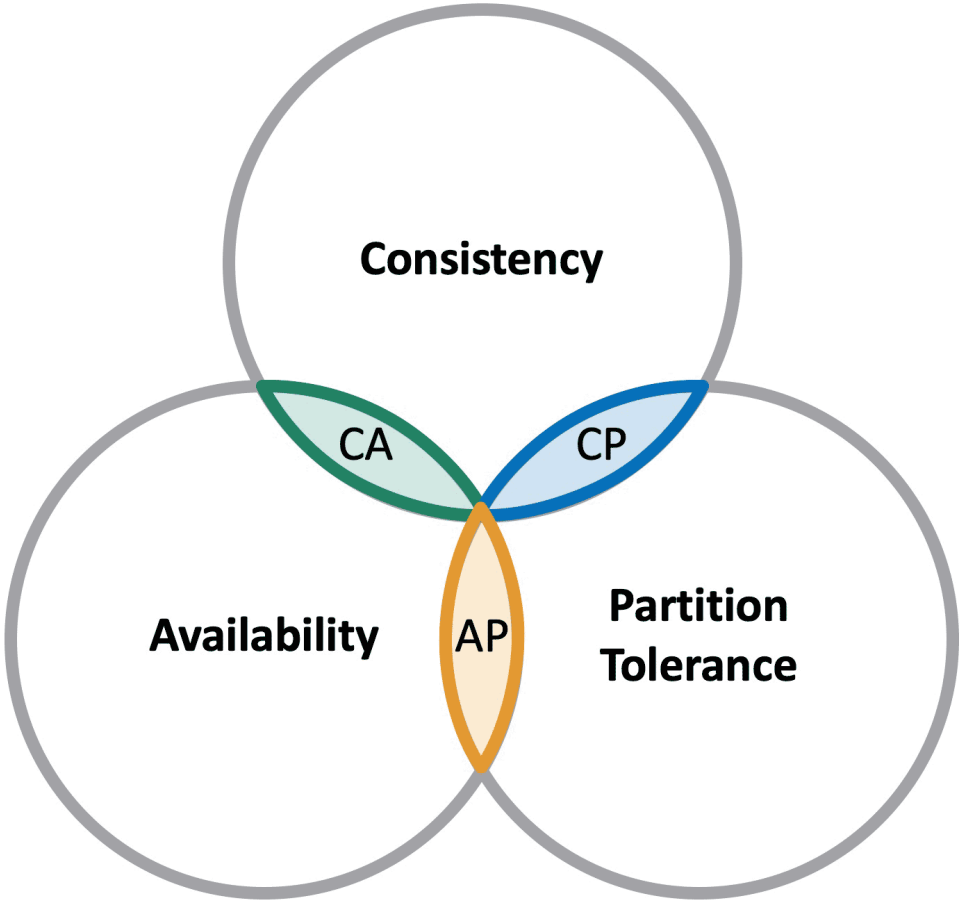
Event Streaming

Damilah

Event streaming allows for the continuous delivery of events to interested parties. It is often used in event-driven architecture to decouple applications and services and to enable real-time processing of events. In this pattern, data is continuously ingested from various sources and streamed in real-time, enabling the ability to build instant insights and take immediate actions.

The event streaming pattern enables the creation of highly responsive and real-time systems that can analyze and act on data as it arrives. This pattern is often used in applications that require real-time monitoring, such as **fraud detection**, **predictive maintenance**, and **IoT systems**.

CAP Theorem



CAP Theorem

Damilah

The CAP theorem is a concept in distributed systems that states a system can only guarantee two out of the following three properties: Consistency, Availability, and Partition Tolerance.

Consistency
Every read receives the most recent write or an error. In other words, all nodes see the same data at the same time. This means that any data written to the system will be immediately visible to all subsequent reads.

Availability
Every request receives a non-error response, without the guarantee that it contains the most recent write. This means that even if the system cannot find the most current data, it still responds to requests instead of failing.

Partition Tolerance
The system continues to operate despite arbitrary partitioning due to network failures. This means that the system remains operational even if some of the nodes are unable to communicate with others.

Q&A

Thank you

PhD. Kostadin Mishev

"Ss. Cyril and Methodius" University in Skopje
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Resources

- https://martinfowler.com/articles/201701-event-driven.html
- https://medium.com/@seetharamugn/the-complete-guide-to-event-driven-architecture-b25226594227
- https://aws.amazon.com/what-is/eda/
- https://solace.com/what-is-event-driven-architecture/

Let's link up

LinkedIn

Damilah